Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Virtualization Techniques on Embedded Systems with further application on satellites

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

Νικολέτας-Μαρκέλας Ηλιακοπούλου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2022

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

Virtualization Techniques on Embedded Systems with further application on satellites

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

Νικολέτας-Μαρκέλας Ηλιακοπούλου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 12η Ιουλίου, 2022.

.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξύδης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Χ.Π.Α.

Αθήνα, Ιούλιος 2022

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

.....

Νικολετα-Μαρκελα Ηλιακοπουλου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός
και Μηχανικός Υπολογιστών Ε.Μ.Π.

στη μαμά μου

Περίληψη

Η πρόσφατη ανάπτυξη στη διαστημική βιομηχανία έχει επιστήσει την προσοχή στο Δορυφόρο ως Υπηρεσία(ΔωΥ). Ο πρωταρχικός στόχος του ΔωΥ είναι να μεγιστοποιήσει τη χρήση των πόρων σε τροχιά εισάγοντας παράλληλα νέες έννοιες, όπως η ιδέα της επεξεργασίας δεδομένων επάνω σε έναν δορυφόρο. Οι πρόσφατες τάσεις της αγοράς προτείνουν νέες τεχνολογίες, όπως Deep/Machine Learning ή/και Τεχνητή Νοημοσύνη, για τη διαστημική βιομηχανία. Η κρίσιμη πτυχή είναι η φιλοξενία και η εκτέλεση διαφορετικών λογισμικών σε μια αφηρημένη πλατφόρμα υλικού, η οποία θα αναδιαμορφώνεται εκ νέου τακτικά. Βασικό ρόλο στην επίτευξη του στόχου αυτού έχει η εικονικοποίηση.

Σε αυτή τη Διπλωματική Εργασία, διερευνούμε δύο στρατηγικές εικονικοποίησης ή αλλιώς δύο πιθανούς υποψηφίους,τον Jailhouse Hypervisor και τα Docker Containers. Χρησιμοποιώντας ένα Raspberry Pi και Linux, κατασκευάζουμε και διαμορφώνουμε ένα πλήρως λειτουργικό Jailhouse "οικοσύστημα" προκειμένου να αξιολογήσουμε την επίδραση του Jailhouse Hypervisor σε αυτό και να συγκρίνουμε εν μέρει την απόδοσή του με αυτή των Docker Containers. Η μελέτη του Jailhouse Hypervisor στο Raspberry Pi μας οδήγησε στο συμπέρασμα ότι τα Docker containers προσθέτουν μικρή ποσότητα επιβάρυνσης συστήματος και μπορούν να συνδυαστούν με τον Jailhouse για να παρέχουν ένα απομονωμένο και ασφαλές περιβάλλον με την ευελιξία που παρέχει η τεχνική της χρήσης container. Όσον αφορά την ντετερμινιστική συμπεριφορά του συστήματος, τα πρώτα ευρήματά μας σχετικά με την επίδραση στην απόδοση πραγματικού χρόνου είναι αρκετά ικανοποιητικά και ενθαρρυντικά. Επιπλέον, επιβεβαιώσαμε το ισχυρό χαρτί του Jailhouse, την απομόνωση, αφού δεν δημιουργήθηκε πρόβλημα σε κανένα από τα σενάρια εκτέλεσής μας, καθώς και την αδυναμία του, τη μείωση της απόδοσης της εφαρμογής όταν επικοινωνούν τα κελιά μεταξύ τους ή η κίνηση του διαύλου συστήματος αυξάνεται.

Λέξεις Κλειδιά — Εικονικοποίηση, Jailhouse Hypervisor, επεξεργασία στο δορυφόρο, Δορυφόρος ως Υπηρεσία, Docker Containers, Raspberry Pi, διαστημική βιομηχανία, απομόνωση, απόδοση πραγματικού χρόνου

Abstract

Recent growth in the space industry has drawn attention to Satellite as a Service (SaaS). The primary goal of SaaS is to maximize the use of orbital resources while introducing novel concepts, such as the idea of data processing onboard a satellite. Recent market trends suggest novel technologies, such as Deep/Machine Learning and/or Artificial Intelligence, for the space industry. The crucial aspect is hosting and executing diverse software across an abstract hardware platform, which will be re-instantiated regularly. The key component to this objective is virtualization.

In this Diploma Thesis, we investigate two virtualization strategies or we could say two potential candidates, namely the Jailhouse Hypervisor and Docker Containers. Using a Raspberry Pi and Linux we build and configure a fully functional Jailhouse "ecosystem" in order to evaluate the effect of the Jailhouse hypervisor on it and partially compare its performance to that of Docker containers. The study of the Jailhouse Hypervisor on the Raspberry Pi led us to the conclusion that Docker containers add a small amount of system overhead and can be combined with Jailhouse to provide an isolated and secure environment with the flexibility provided by the containerization technique. In terms of observed deterministic behavior, our preliminary findings regarding the effect on real-time performance are quite satisfying. In addition, we confirmed Jailhouse's strength, isolation, which did not pose a problem in any of our execution scenarios, as well as its weakness, the decrease in application performance when cells communicate or system bus traffic increases.

Keywords — Virtualization, Jailhouse Hypervisor, onboard data processing, Satellite as a Service, Docker Containers, Raspberry Pi, Space Industry, isolation, real-time performance

Ευχαριστίες

Αρχικά, θα ηθελα να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντα Καθηγητή Δημήτριο Σούντρη για την εμπιστοσύνη που μου έδειξε με την ανάθεση αυτής της διπλωματικής εργασίας, αλλά και για τις πολύτιμες συμβουλές και καθοδήγησή του όσον αφορά τα μετέπειτα επαγγελματικά μου βήματα. Επίσης, ευχαριστώ θερμά τον μεταδιδακτορικό ερευνητή Γιώργο Λεντάρη και τον υποψήφιο διδάκτορα Δημοσθένη Μασούρο για τις πολύτιμες κατευθυντήριες γραμμές και συμβουλές που μου προσέφεραν κατά τη διάρκεια εκπόνησης της διπλωματικής αυτής, καθώς και για την άψογη συνεργασία μας. Δε θα μπορούσα να παραλείψω να αναφερθώ στην οικογένεια της OHB-Hellas και ιδιαίτερα στους Mathieu Bernou και Σίμονα Βέλλα, που με έκαναν να νιώσω από την πρώτη στιγμή μέρος της.

Ένα μεγάλο ευχαριστώ στους συμφοιτητές μου και ιδιαίτερα στους Σπύρο Π., Χριστίνα Σ., Αθηνά Τ., Γεωργία Σ., Βαλεντίνα Κ., Έθελ Γ., Στέφανο Α. για όλες τις στιγμές που περάσαμε και για τις ατελείωτες συζητήσεις που κάναμε. Επιπλέον, θα ήθελα να ευχαριστήσω τον Γιάννη Σ. για την πολυτιμή στήριξή του κατά τη διάρκεια συγγραφής της διπλωματικής αυτής καθώς και για την πολύτιμη βοήθειά του στη χάραξη της μετέπειτα επαγγελματικής μου πορείας.

Τέλος, το μεγαλύτερο ευχαριστώ ανήκει δικαιωματικά στην οικογενειά μου και ιδιαίτερα στους γονείς μου Ηλία και Αθηνά και στα αδέρφια μου Νίκο και Χρήστο που στέκονται πάντα στο πλευρό μου και με έκαναν το άτομο που είμαι σήμερα.

Contents

Figure List

Table List

Εκτεταμένη Ελληνική Περίληψη

Εισαγωγή

Ζούμε σε μια εποχή όπου η συλλογή και η επεξεργασία δεδομένων διαδραματίζουν ουσιαστικό ρόλο στην καθημερινή ζωή των ανθρώπων. Οι επιχειρήσεις, οι κυβερνήσεις, οι βιομηχανίες, οι μη κερδοσκοπικοί οργανισμοί και η επιστημονική έρευνα παράγουν και μοιράζονται έναν άνευ προηγουμένου όγκο δεδομένων. Ο διαστημικός τομέας διαδραματίζει καίριο ρόλο στη λήψη αυτών των δεδομένων, ποσοτικών ή ποιοτικών, αφού μπορούμε και συλλέγουμε ζωτικά δεδομένα από δορυφόρους που επιτρέπουν την έγκαιρη ανίχνευση περιβαλλοντικών και κλιματικών αλλαγών.

Σε αντίθεση με τις τελευταίες τάσεις της αγοράς του διαστημικού τομέα, οι παραδοσιακοί δορυφόροι επικεντρώνονται κυρίως στην εκτέλεση συγκεκριμένων μακροπρόθεσμων αποστολών στο διάστημα. Έχουν σχεδιαστεί για να εκτελούν μόνο μία συγκεκριμένη εργασία κατά τη διάρκεια της αποστολής τους και η διάρκεια ζωής τους είναι προκαθορισμένη Επιπρόσθετα, ο χρονοβόρα ανάπτυξη και δοκιμή του φανερώνει το υψηλό κόστος σχεδιασμού και ανάπτυξης που κρύβει ένας δορυφόρος. Αυτός είναι ο λόγος που η διαστημική αγορά είναι περιορισμένη. Η άποψη που κυριαρχεί στον διαστημικό τομέα είναι να δοθεί προτεραιότητα στην ντετερμινιστική συμπεριφορά του δορυφόρου. Έτσι, η τεχνολογία που συνηθίζεται να χρησιμοποιείται μέχρι και σήμερα για τη δημιουργία και την εκτόξευση του δορυφόρου δεν είναι η πιο ενημερωμένη. Αντί να χρησιμοποιηθούν νέες και επαναστατικές τεχνολογίες, προτιμούνται τεχνικές που εφευρέθηκαν πριν χρόνια καθώς έχουν δοκιμαστεί επανειλημμένα. με Έτσι, η όλη ιδέα δεν ανταποκρίνεται στις πραγματικές ανάγκες του κόσμου μας.

Παράλληλα, αρκετά συχνά η ποσότητα των ακατέργαστων δεδομένων που παράγονται από σύγχρονα όργανα επάνω στους δορυφόρους είναι πολύ μεγαλυτερη από αυτή που μπορεί να μεταδοθεί στο έδαφος. Αυτό καθιστά απαραίτητη τη χρήση διαφόρων τεχνικών συμπίεσης των δεδομένων για τη μείωση του όγκου τους. Η όλη διαδικασία απόκτησης ακατέργαστων δεδομένων, μεταφοράς και αποθηκευσής τους στο δορυφόρο με σκοπό τη συμπίεση και μετάδοση τους στο έδαφος, όπου εν τέλει θα γίνει η επεξεργασία τους, είναι αρκετά πολύπλοκη και χρονοβόρα.

Τα τελευταία χρόνια, ο διαστημικός τομέας συνεχώς επεκτείνεται, φέρνοντας στο προσκήνιο την έννοια του Δορυφόρου ως Υπηρεσία. Πρωταρχικός του στόχος είναι να μεγιστοποιήσει τη χρήση των πόρων που βρίσκονται σε τροχιά συστήνοντας παράλληλα νέες ιδέες και τεχνικές, όπως η έννοια της επεξεργασίας δεδομένων επάνω στο δορυφόρο.

Οι τάσεις της αγοράς μάλιστα, προτείνουν την επέκταση του αριθμού των δορυφόρων που υιοθετούν αυτή την ιδέα, ενώ εισάγουν τεχνολογίες μέχρι τώρα άγνωστες στη διαστημική βιομηχανία, όπως η Βαθιά/Μηχανική Μάθηση και/ή η Τεχνητή Νοημοσύνη. Οι τεχνολογίες αυτές παρουσιάζουν μια καινοτόμο, αποτελεσματική μέθοδο υπολογισμού των απαραίτητων αποτελεσμάτων, που μειώνονει την τελική μεταφορά δεδομένων στο έδαφος αποσυμφωρίζοντας τις δικτυακές γραμμές ανάμεσα σε γη και διάστημα.

Είναι ολοφάνερο από τα παραπάνω ότι ο Δορυφόρος ως Υπηρεσία οδηγεί στη χρήση καινοτόμων τεχνολογιών. Οι παλιοί, συμβατικοί και ειδικά κατασκευασμένοι δορυφόροι πρέπει να αλλάξουν ή να αντικατασταθούν. Κάθε δορυφόρος οφείλει πλέον να εξυπηρετεί πολυάριθμες εφαρμογές και αποστολές, απαιτώντας μια ενιαία αφηρημένη πλατφόρμα υλικού για την υποστήριξη και εκτέλεση πολλαπλών διαφορετικών λογισμικων. Το βασικό συστατικό αυτής της μετάβασης είναι η εικονικοποίηση.

Προκλήσεις και Συνεισφορά Διπλωματικής Εργασίας

Η εικονικοποίηση είναι μια ευρεία έννοια με πολλές εξειδικευμένες επεκτάσεις και τεχνικές. Προκειμένου να καθοριστεί ποια τεχνολογία είναι πιο συμβατή με τις ανάγκες μας, είναι απαραίτητο να αντιμετωπίσουμε ορισμένες προκλήσεις με βάση τις γενικές απαιτήσεις και περιορισμούς των δορυφόρων για την εκτέλεση λογισμικού μικτής κρισιμότητας.

Συνοπτικά οι προκλήσεις αυτές είναι

1. Απομόνωση – Ανάκτηση
2. Απόδοση Πραγματικού Χρόνου
3. Επαναχρησιμοποίηση
4. Απλότητα - Ευελιξία
5. Φιλικότητα προς τον χρήστη

Σε αυτή τη διπλωματική εργασία, σε μια προσπάθεια να αντιμετωπίσουμε τις προαναφερθείσες προκλήσεις, μελετάμε το Jailhouse Hypervisor εγκατεστημένο σε μια ενσωματωμένη συσκευή. Στόχος μας είναι να εξετάσουμε και να αξιολογήσουμε τον αντίκτυπο που έχει ο hypervisor στο σύστημα στο πλαίσιο της εκτέλεσης εφαρμογών μικτής κρισιμότητας, συγκρίνοντας αυτήν την τεχνική εικονικοποίησης με τη δημιουργία κοντέινερ και, πιο συγκεκριμένα, τα Docker κοντέινερ. Αρχικά, δημιουργούμε ένα πλήρως λειτουργικό σύστημα που αποτελείται από μια πλατφόρμα υλικού που τρέχει Linux, το Jailhouse Hypervisor και ένα Linux non-root κελί, το οποίο προσαρμόσαμε ανάλογα με τις ανάγκες μας. Αυτό το σύστημα, το οποίο αναφέρουμε ως «Jailhouse οικοσύστημα», θα χρησιμοποιηθεί ως εργαλείο όπου εξετάζουμε και αξιολογούμε την επίδραση του Jailhouse Hypervisor στο Jailhouse οικοσύστημά μας δημιουργώντας και εκτελώντας διάφορα σενάρια εκτέλεσης.

Σχετική Βιβλιογραφία

Η ενότητα αυτή εξετάζει τρέχουσες αντιπροσωπευτικές περιπτώσεις λύσεων εικονικοποίησης και σχετικές τεχνικές που ήδη χρησιμοποιούνται ή έχουν τη δυνατότητα να χρησιμοποιηθούν στον διαστημικό τομέα, λαμβάνοντας υπόψη τις τρέχουσες τάσεις.

Τα παραδείγματα που θα αναφερθούν επιλέγονται από τέσσερις κατηγορίες που αντιστοιχούν σε τέσσερις τάσεις στην τρέχουσα/μελλοντική υιοθέτηση της εικονικοποίησης στη βιομηχανία[1]:

- Λύσεις βασισμένες σε διαχωρισμό πυρήνα και μικροπυρήνα που δημιουργήθηκαν κυρίως για τη βιομηχανία και τον κλάδο των ενσωματωμένων συστημάτων. Αναφορικά κάποιες απο αυτές είναι τα λογισμικά PikeOS, Xtratum Hypervisor, Bao Hypervisor, Quest-V και Jailhouse Hypervisor ο οποίος θα αναλυθεί μετέπειτα.
- Λύσεις που προσπαθούν να αξιοποιήσουν τα πλεονεκτήματα των υπαρχόντων hypervisor γενικού σκοπού, όπως ο Xen και ο KVM, και να τους προσαρμόσουν στις απαιτήσεις του κλάδου.
- Χρησιμοποιώντας τις πιο πρόσφατες δυνατότητες απομόνωσης στο υλικό (π.χ. ARM TrustZone) για την επίτευξη επιπέδων απομόνωσης που απαιτούνται από τα βιομηχανικά πρότυπα.
- Σε σύγκριση με τις συμβατικές μεθόδους εικονικοποίησης, λύσεις που χρησιμοποιούν ελαφριά εικονικοποίηση, όπως κοντέινερ ή unikernels, σε μια προσπάθεια να μειωθεί το αποτύπωμα στη μνήμη και να αυξηθεί η ευελιξία

Αξίζει να σημειωθεί η δημιουργία μιας νέα πλατφόρμας που στοχεύει να εισέλθει στη διαστημική κοινότητα και να αλλάξει τον τρόπο που αντιμετωπίζονταν μέχρι τώρα τα κρίσιμα για την ασφάλεια συστήματα υψηλής απόδοσης. Πιο αναλυτικά, το SELENE [2, 3] είναι μια αυτο-ελεγχόμενη αξιόπιστη πλατφόρμα για συστήματα κρίσιμης σημασίας για την ασφάλεια, βασισμένη σε ένα σύνολο κρίσιμων για την ασφάλεια υπολογιστικών πλατφορμών που βασίζονται σε στοιχεία ανοιχτού κώδικα, όπως η αρχιτεκτονική συνόλου εντολών RISC-V, το GNU/Linux, και ο Jailhouse hypervisor, που θα μελετήσουμε σε αυτή τη διπλωματική εργασία.

Αυτή η εξελιγμένη υπολογιστική πλατφόρμα στοχεύει να δημιουργήσει ένα αφηρημένο σύστημα και να το προσαρμόσει στις εκάστοτε απαιτήσεις και περιορισμούς διαφορετικών εφαρμογών αλλάζοντας τις ρυθμίσεις του συστήματος. Ταυτόχρονα, εγγυάται λειτουργικές ιδιότητες απομόνωσης, επιτρέποντας τη συνύπαρξη λογισμικών μικτής κρισιμότητας και μικτών απαιτήσεων απόδοσης, στην ίδια πλατφόρμα υλικού. Τέλος, ακόμα ένας σκοπός είναι η αποτελεσματική εκτέλεση εφαρμογών λογισμικού υψηλού υπολογιστικού φορτίου με τη βοήθεια συγκεκριμένων επιταχυντών.

Ανάλυση τεχνικών εικονικοποίησης

Η ενότητα αυτή περιγράφει λεπτομερώς τις βασικές τεχνολογίες που διέπουν αυτήν τη μελέτη, συμπεριλαμβανομένης της έννοιας της εικονικοποίησης και των διαφόρων ειδών

της καθώς και του μηχανισμού του hypervisor και των διάφορων τύπων του, ανοίγοντας το δρόμο για το κεφάλαιο του Jailhouse Hypervisor.

Εικονικοποίηση

Η εικονικοποίηση είναι ένας ευρέως χρησιμοποιούμενος όρος που μπορεί να ερμηνευτεί και να οριστεί με διάφορους τρόπους με βάση το πλαίσιο και τις τεχνολογίες που χρησιμοποιούνται. Αν προσπαθήσουμε να δώσουμε έναν περιεκτικό αλλά μάλλον αφηρημένο ορισμό της λέξης «εικονικοποίηση» θα μπορούσαμε να πούμε ότι είναι μια τεχνολογία που μας επιτρέπει να πάρουμε ένα ενιαίο, φυσικό σύστημα υλικού και διαιρώντας το να δημιουργήσουμε πολλαπλά εικονικά περιβάλλοντα, που συνήθως ονομάζονται εικονικές μηχανές. Αυτά τα περιβάλλοντα μπορούν να λειτουργούν ταυτόχρονα και ανεξάρτητα το ένα από το άλλο. Για τον προσδιορισμό ενός συγκεκριμένου τύπου εικονικοποίησης χρησιμοποιούνται μία ή περισσότερες τεχνικές, όπως η κατάτμηση υλικού και λογισμικού, η κοινή χρήση χρόνου, η μερική ή πλήρης προσομοίωση μηχανήματος, η εξομοίωση, η ποιότητα της υπηρεσίας κ.α. [4].

Γιατί εικονικοποίηση;

Η εικονικοποίηση μπορεί να θεωρηθεί μια από τις πιο ελκυστικές αρχιτεκτονικές στρατηγικές για την υλοποίηση συστημάτων μικτής κρισιμότητας, δηλαδή την ενσωμάτωση διαφορετικών στοιχείων λογισμικού με διαφορετικά επίπεδα κρισιμότητας σε μια κοινόχρηστη πλατφόρμα υλικού.

Οι δορυφόροι αποτελούνται από συστήματα με μικτή κρισιμότητα. Με άλλα λόγια, συχνά έχουν τουλάχιστον ένα στοιχείο πραγματικού χρόνου στο οποίο είναι ζωτικής σημασίας να ολοκληρωθούν συγκεκριμένες δραστηριότητες μέσα σε ένα ντετερμινιστικό, εγγυημένο χρονικό πλαίσιο. Μπορεί επίσης να προστεθεί ένα στοιχείο μη πραγματικού χρόνου, το οποίο συνήθως χρησιμοποιείται για πληροφορίες και επεξεργασία δεδομένων σε πραγματικό χρόνο, διαχείριση ή διαμόρφωση συστήματος.[5]

Τα στοιχεία πραγματικού χρόνου μπορούν να επηρεαστούν από προγράμματα μη πραγματικού χρόνου χωρίς την τεχνική της εικονικοποίησης, αφού απαιτείται συχνά διαχωρισμός και εκτέλεση των διαφορετικών προγραμμάτων σε μια ξεχωριστή φυσική CPU. Χρησιμοποιώντας την εικονικοποίηση, διάφορα στοιχεία μπορούν να συγχωνευθούν σε μια ενιαία πλατφόρμα διατηρώντας παράλληλα την πραγματικού χρόνου ακεραιότητα του συστήματος .[5]

Τεχνικές εικονικοποίησης

Υπάρχουν διαφορετικές προσεγγίσεις για την εφαρμογή εικονικοποίησης, κατάλληλες για διαφορετικές καταστάσεις. Τεχνικές όπως η παγίδα και μίμηση(trap and emulate), η πλήρης εικονικοποίηση και η παραεικονικοποίηση είναι αρκετά δημοφιλείς και συνιστούν πολλές φορές κατάλληλη λύση. Παρ' όλα αυτά, στην περίπτωσή μας θέλουμε να εφαρμόσουμε την εικονικοποίηση σε ένα ενσωματωμένο σύστημα, όπου η απλότητα συνιστά αναπόσπαστο στοιχείο. Η ρύθμιση και η διαμόρφωση των προαναφερόμενων διαδικασιών

μπορεί να αποβεί αρκετά δύσκολη, ιδιαίτερα για ενσωματωμένες συσκευές. Επομένως, για την προσομοίωση ανεξάρτητων συστημάτων, κατάλληλη τεχνική θεωρείται η στατική κατάτμηση (static partitioning) που απομονώνει προγράμματα ή εργασίες σε συγκεκριμένα κομμάτια του τρέχοντος υλικού. Τα διαχωρισμένα περιβάλλοντα σε επίπεδο λογισμικού συνδέονται με διαχωρισμένα περιβάλλοντας σε επίπεδο υλικού, περιορίζοντας τον αριθμό των λειτουργικών συστημάτων ανά φυσικό πυρήνα σε ένα.

Όλες οι προσεγγίσεις επιτρέπουν τον διαχωρισμό μεμονωμένων και ζωτικών εργασιών από τις λιγότερο κρίσιμες. Ωστόσο, η ευελιξία που προσφέρει η πλήρης εικονικοποίηση αντισταθμίζεται από την εγγύηση ντετερμινιστικής συμπεριφοράς από τη στατική κατάτμηση. Δεδομένου ότι διαφορετικές στρατηγικές εικονικοποίησης αντιδρούν διαφορετικά στο υλικό, καθεμία έχει εγγενή πλεονεκτήματα για συγκεκριμένες εφαρμογές.

Σε μια προσέγγιση στατικής κατάτμησης, οι φυσικοί πόροι της πλατφόρμας περιορίζουν τον αριθμό των σε λειτουργία εικονικών περιβαλλόντων. τα συστήματα με στατική κατάτμηση προσφέρουν τα ίδια πλεονεκτήματα για τον διαχωρισμό των εργασιών, είτε αυτές οι εργασίες μικτής κρίσιμης σημασίας σχετίζονται με την ασφάλεια είτε με τη λειτουργία πραγματικού χρόνου. Ωστόσο, ο προγραμματισμός των εργασιών επηρεάζεται λιγότερο, επειδή οι φυσικοί πόροι συνδέονται πιο στενά και άμεσα με τα εικονικά περιβάλλοντα. Αυτό καθιστά την περιγραφηθείσα τεχνική καταλληλότερη για ενσωματωμένα συστήματα με περιορισμένους υπολογιστικούς πόρους.

Η έννοια του hypervisor

Ανεξάρτητα από το ποια τεχνική εικονικοποίησης θα εφαρμοσθεί, οι περισσότερες έχουν σαν βασικό συστατικό τους τον hypervisor. Ένας hypervisor είναι ένα στρώμα λογισμικού που διαχωρίζει πόρους υλικού προκειμένου να τρέξει πολλαπλές εικονικές μηχανές στην ίδια φυσική μηχανή. Δεδομένου ότι οι εικονικές μηχανές είναι ανεξάρτητες από το υλικό του φυσικού μηχανήματος, οι hypervisors καθιστούν δυνατή την καλύτερη και πιο αποδοτική χρήση του συστήματος και των διαδέσιμων πόρων. Επειδή ένας hypervisor επιτρέπει σε πολλαπλές εικονικές μηχανές να εκτελούνται σε ένα μόνο φυσικό μηχάνημα, μειώνει τις απαιτήσεις ενέργειας, χώρου και συντήρησης.

Υπάρχουν δύο κύριοι τύποι hypervisor. "Τύπος 1" ή "bare-metal" και "Τύπος 2" ή "hosted". Ένας hypervisor τύπου 1 συμπεριφέρεται ως ένα ελαφρύ λειτουργικό σύστημα και εκτελείται απευθείας στο υλικό του κεντρικού υπολογιστή, ενώ ένας hypervisor τύπου 2 εκτελείται ως επιπλέον επίπεδο λογισμικού σε ένα λειτουργικό σύστημα, παρόμοια με άλλες εφαρμογές.

Jailhouse Hypervisor

Η ενότητα αυτή είναι αφιερωμένη στον Jailhouse Hypervisor, τον πρωταγωνιστή αυτής της διπλωματικής εργασίας. Αναλύουμε την έννοια και τη λειτουργία του Jailhouse ο οποίος πρόκειται να εξερευνηθεί σε βάθος, να μελετηθεί και να αξιολογηθεί.

Ο Jailhouse [6] [7] είναι ένας hypervisor στατικής κατάτμησης που βασίζεται σε Linux και

ξεκίνησε από τον Jan Kiszka, επικεφαλή προγραμματιστή στη Siemens, AG. Προσφέρει υψηλό επίπεδο απομόνωσης μεταξύ των κατατμήσεων του, οι οποίες μπορούν να εκτελούν είτε εφαρμογές γυμνού μετάλλου (bare-metal) είτε να φιλοξενούν ολόκληρα λειτουργικά συστήματα. Η τρέχουσα έκδοση του Jailhouse είναι η 0.12 και έχει ενεργή υποστήριξη προγραμματιστών και κοινότητας. Υποστηρίζει αρχιτεκτονικές ARM, ARM64 και x86 και πρόσφατα έκανε τα πρώτα του βήματα προς την υποστήριξη της RISC-V αρχιτεκτονικής [8] [2].

Ο Jailhouse hypervisor δεν μπορεί να ταξινομηθεί με την αυστηρή του όρου έννοια σε μια εκ των δύο κατηγορίες (Τύπου 1 ή Τύπου 2)[9]. Αντίθετα, αποτελεί έναν υβριδικό Τύπου 1 και Τύπου 2 hypervisor [7]. Μόλις ενεργοποιηθεί, ο jailhouse εκτελείται εγγενώς σε υλικό όπως ένας hypervisor τύπου 1 χωρίς να παρεμβένει κάποιο ενδιάμεσο επίπεδο και χωρίς καμία ανάγκη εξωτερικής υποστήριξης. Ωστόσο, απαιτείται Linux για τη φόρτωση και τη διαμόρφωση του Jailhouse. Με άλλα λόγια, το Linux χρησιμοποιείται ως φορτωτής (bootloader), αλλά όχι ως κεντρικό λειτουργικό σύστημα που φιλοξενεί τον hypervisor.

Ο Jailhouse προτιμά την απλότητα έναντι των πολλών και πολύπλοκων λειτουργιών. Αντί να χρησιμοποιεί πολύπλοκα και χρονοβόρες τεχνικές (παρα-)εικονικοποίησης, όπως κάνουν οι Xen και KVM, το Jailhouse παρέχει μόνο κατάτμηση υλικού (εκμεταλλευόμενος τις επεκτάσεις εικονικοποίησης της εκάστοτε πλατφόρμας), αλλά σκόπιμα δεν παρέχει ούτε "scheduler" ούτε εικονικές CPU. Μόνο (λίγοι) πόροι που δεν μπορούν, ανάλογα με την υποστήριξη υλικού, να χωριστούν με αυτόν τον τρόπο, εικονικοποιούνται σε επίπεδο λογισμικού.

Μετατρέπει τα συμμετρικά συστήματα πολλαπλής επεξεργασίας (SMP) σε συστήματα ασύμμετρης πολλαπλής επεξεργασίας (AMP) εισάγοντας «εικονικά εμπόδια» μεταξύ του συστήματος και του διαύλου I/O. Το σύστημα χωρίζεται σε μεμονωμένα περιβάλλοντα που ονομάζονται "κελιά". Κάθε κελί φιλοξενεί έναν επισκέπτη και έχει ένα σύνολο εκχωρημένων πόρων (CPU, περιοχές μνήμης, συσκευές PCI) τους οποίους ελέγχει πλήρως. Η δουλειά του hypervisor είναι να διαχειρίζεται τα κελιά και να διατηρεί την απομόνωση μεταξύ τους, διασφαλίζοντας ότι τα κελιά δεν θα παρεμβαίνουν μεταξύ τους με μη αποδεκτό τρόπο. Από την οπτική του υλικού, ο δίαυλος συστήματος εξακολουθεί να είναι κοινόχρηστος, ενώ το λογισμικό είναι φυλακισμένο σε κελιά και μπορεί να έχει πρόσβαση μόνο σε ένα υποσύνολο φυσικού υλικού, αυτό που τους έχει εκχωρηθεί. Αυτή η προσέγγιση είναι πιο χρήσιμη για εικονικοποίηση εργασιών που απαιτούν πλήρη έλεγχο της CPU. Τα παραδείγματα περιλαμβάνουν εργασίες ελέγχου πραγματικού χρόνου και μακροχρόνιες εργασίες υπολογιστών υψηλής απόδοσης.

Ανάπτυξη

Η ενότητα αυτή αφορά την έρευνα και ανάπτυξη του συστήματος μας, του "Jailhouse οικοσυστήματος", το οποίο αποτελείται από μια πραγματική πλατφόρμα υλικού, συγκεκριμένα μια ενσωματωμένη συσκευή, τον Jailhouse Hypervisor και ένα πλήρως λειτουργικό Linux non-root κελί προσαρμοσμένο στις ανάγκες μας. Αναλύουμε τη διαδικασία κατασκευής του συστήματος το οποίο θα χρησιμοποιηθεί για την εκτενή αξιολόγηση και εκτίμηση του Jailhouse Hypervisor.

Μετά από κάποια συζήτηση και συγκρίσεις μεταξύ πιθανών υποψηφίων, καταλήξαμε στο συμπέρασμα ότι το Raspberry Pi 4 Model B είναι η πιο κατάλληλη επιλογή ενσωματωμένης συσκευής για αυτή τη μελέτη.

Για να εγκαταστήσουμε και να τρέξουμε με επιτυχία το Jailhouse στην πλακέτα rpi4 θα πρέπει να κάνουμε τα εξής:

- Παίρνουμε το Linux distro της επιλογής μας
- Κλωνοποιούμε το αποθετήριο του Jailhouse από το git
- Ομοίως κλωνοποιούμε ένα Linux δέντρο με κατάλληλες προσαρμογές ώστε να υποστηρίζει τον Jailhouse κι έτσι χτίζουμε και εκκινούμε το δικό μας προσαρμοσμένο πυρήνα
- Χτίζουμε και εκτελούμε τον Jailhouse Hypervisor

Ο Jailhouse πρέπει να μεταγλωττιστεί με αντικείμενα πυρήνα. Επομένως, χρειαζόμαστε πρώτα ένα αντίγραφο του πηγαίου κώδικα του πυρήνα και να το μεταγλωττίσουμε. Το Jailhouse απαιτεί έκδοση πυρήνα >=4.7. Εδώ, θα χρησιμοποιήσουμε την έκδοση Kernel 5.10.31 ή/και 5.10.27-rt που παρέχεται από τη Siemens/Jan Kiszka [10].

Η ετερομεταγλώττιση (cross-compilation) σε ένα μηχάνημα x86 είναι σίγουρα πιο γρήγορη, ωστόσο είναι πιο επιρρεπής σε λάθη σε σύγκριση με την μεταγλώττιση στην πλακέτα. Κάποιος μπορεί να επιλέξει την ενότητα που αντιστοιχεί στην κατάστασή του. είτε τα εγγενή χτισίματα είτε την ετερομεταγλώττιση. Αν και υπάρχουν πολλά κοινά βήματα μεταξύ των δύο, υπάρχουν επίσης σημαντικές διαφορές.

Πρέπει να δεσμεύσουμε εκ των προτέρων τη μνήμη για Jailhouse Hypervisor και για τα περαιτέρω κελιά και το κάνουμε μέσω του device tree overlay, χαρακτηριστικό του Raspberry Pi. Επίσης, μεταγλωττίζουμε και προσθέτουμε το εκτελέσιμο του Arm-Trusted-Firmware, απαραίτητο στοιχείο για την ομαλή λειτουργία του Jailhouse.

Ανάπτυξη του Linux non-root κελιού

Σε αυτήν την υποενότητα θα περιγράψουμε τα βήματα που ακολουθήσαμε για να αναπτύξουμε το Linux non-root κελί μας.

Αρχικά δοκιμάσαμε τη λειτουργικότητα του non-root κελιού φορτώνοντας τον πυρήνα και ένα ελάχιστο .cpio αρχείο, παρμένο από το απωθετήριο jailhouse-images[11]. Χρησιμοποιούμε την υπάρχουσα διαμόρφωση για την αρχιτεκτονική arm64 ως αναφορά για τη δημιουργία του προσαρμοσμένου πυρήνα και εικόνας μας.

Δημιουργήσαμε τον προσαρμοσμένο Linux πυρήνα και εικόνα μας με το Buildroot[12]. Πρόκειται για ένα εργαλείο που απλοποιεί και αυτοματοποιεί τη διαδικασία κατασκευής ενός πλήρους και εκκινήσιμου περιβάλλοντος Linux για ένα ενσωματωμένο σύστημα, αξιοποιώντας την ετερομεταγλώττιση για να καταστεί δυνατή η κατασκευή διαφόρων πλατφορμών στόχων (hardware target platforms) σε μια ενιαία μηχανή ανάπτυξης που βασίζεται σε Linux. Το Buildroot μπορεί να δημιουργήσει την απαραίτητη αλυσίδα εργαλείων πολλαπλής μεταγλώττισης (toolchain), να δημιουργήσει ένα ριζικό σύστημα αρ-

χείων(root filesystem), να μεταγλωττίσει μια εικόνα πυρήνα Linux και να δημιουργήσει έναν φορτωτή εκκίνησης(bootloader)για ένα ενσωματωμένο σύστημα ή μπορεί να εκτελέσει οποιονδήποτε ανεξάρτητο συνδυασμό αυτών των εργασιών. Για παράδειγμα, μπορεί να χρησιμοποιηθεί μια ανεξάρτητα εγκατεστημένη αλυσίδα εργαλείων πολλαπλής μεταγλώττισης(toolchain), ενώ το Buildroot απλώς να κατασκευάσει το ριζικό σύστημα αρχείων[13].

Κατεβάζουμε την έκδοση του Buildroot με την οποία θέλουμε να δουλέψουμε. Στην περίπτωσή μας είναι η 2022-02-03. Χρησιμοποιούμε ως αναφορά το αρχείο διαμόρφωσης για την εικόνα του rootfs που δημιουργείται από το Jailhouse-Images και στη συνέχεια εφαρμόζουμε τις απαραίτητες αλλαγές σύμφωνα με τις προτιμήσεις μας.

Χρησιμοποιούμε το προσαρμοσμένο αποθετήριο git που χρησιμοποιήσαμε πρωτίτερα για να δημιουργήσουμε τον πυρήνα μας, μαζί με την ίδια διαμόρφωση. Με αυτόν τον τρόπο, έχουμε την ίδια έκδοση πυρήνα, επομένως η αξιολόγηση της απόδοσης του συστήματος και οι διάφορες συγκρίσεις που θα προκύψουν είναι έγκυρες. Δημιουργούμε τον πυρήνα πραγματικού χρόνου 5.10.27-rt36. Βεβαιωνόμαστε ότι ο πυρήνας έχει τον οδηγό ivshmem-net, προκειμένου να δημιουργηθεί μια σύνδεση εικονικού δικτύου μεταξύ του root και του non-root κελιού.

Το Buildroot δεν παρέχει διαχειριστή πακέτων. Είναι στη φιλοσοφία του εργαλείου ότι χτίζουμε την εικόνα μας με όλα τα πακέτα που χρειαζόμαστε εκ των προτέρων. Σε αυτό το πλαίσιο, για την εικόνα του συστήματος αρχείων root επιλέγουμε τα απαραίτητα πακέτα για να υποστηρίξουμε την εκτέλεση των σεναρίων μας. Κάνουμε τις εξής επιλογές:

- Αντί να δημιουργήσουμε ένα .cpio rootfs, χτίζουμε ένα RAM σύστημα αρχείων, συνδεδεμένο στον πυρήνα του linux. Ο λόγος είναι η περιορισμένη μνήμη που διαθέτουμε για το Linux non-root κελί μας.

- Επιλέγουμε τα πακέτα της python, του docker και των rt-tests που θα μας χρειαστούν στα μετέπειτα πειράματά μας.

Μετά τη δημιουργία του πυρήνα και της εικόνας που δημιουργήσαμε με το Buildroot, μεταφέρουμε τα αρχεία στο Raspberry Pi για να "σηκώσουμε" το Linux non-root κελί μας.

Αν και τα δύο κελιά (το root και το non-root) μπορούν να συνδεθούν και να επικοινωνήσουν μεταξύ τους μέσω πρωτοκόλλων δικτύωσης, και λόγω του ivshmem-net οδηγού, ουσιαστικά έχουμε ένα απομονωμένο δίκτυο μεταξύ των δύο κελιών. Για να συνδεθεί επιτυχώς το inmate κελί στο διαδίκτυο, το κελί ρίζα (root cell) θα πρέπει να γίνει δρομολογητής δικτύου. Μια απλή λύση που εφαρμόζουμε είναι η δημιουργία μιας γέφυρας δικτύου όπου εμείς επισυνάπτουμε στο κελί ρίζα την πραγματική φυσική διεπαφή δικτύου και αργότερα την εικονική. Τέλος κάνουμε κάποιες τροποποιήσεις στον πίνακα διαδρομών (routing table) του non-root κελιού μας και ρυθμίζουμε το Docker να τρέχει στη RAM, αφού ουσιαστικά και το Linux non-root κελί τρέχει στη μνήμη. Πλέον το Jailhouse οικοσύστημά μας είναι έτοιμο και πλήρως λειτουργικό για την εκτέλεση διαφόρων σεναρίων για τη μελέτη και εκτίμηση του Jailhouse Hypervisor.

Αξιολόγηση

Η ενότητα αυτή αφορά τα πειράματα που πραγματοποιήθηκαν για τη μέτρηση της επίδρασης του Jailhouse Hypervisor στην απόδοση εφαρμογών λογισμικού που έτρεξαν στο σύστημά μας, καθώς και στην αξιολόγηση ορισμένων μετρικών για τον προσδιορισμό της συνολικής συμπεριφοράς του hypervisor.

Χρόνος εκκίνησης & χρόνος ανάκτησης

Σε αυτήν την υποενότητα, θα μετρήσουμε τον χρόνο εκκίνησης του Jailhouse οικοσυστήματος. Ο χρόνος εκκίνησης είναι μια σημαντική πτυχή της απόδοσης του συστήματος, επειδή οι χρήστες πρέπει να περιμένουν την εκκίνηση της συσκευής πριν τη χρησιμοποιήσουν. Σε περιπτώσεις αποτυχίας-ανάκτησης, ο χρόνος (επαν)εκκίνησης ή ανάκτησης είναι επίσης κρίσιμος. Στην περίπτωσή μας, έχουμε ένα σύστημα με ένα εικονικό επίπεδο που μπορεί να φιλοξενήσει τόσο το κρίσιμο λογισμικό του δορυφόρου όσο και διάφορες άλλες επιθυμητές εφαρμογές. Είναι καίριο:

1. να είμαστε σε θέση να διαμορφώνουμε εκ νέου και ουσιαστικά να επαναφέρουμε το σύστημα ώστε μπορεί να φιλοξενεί διαφορετικές εφαρμογές (πραγματικού χρόνου)

2. σε περίπτωση μερικής αποτυχίας μιας εφαρμογής λογισμικού, το σύστημα να επανεκκινήσει χωρίς να επηρεάσει τη συνολική απόδοση των εφαρμογών.

Έτσι, ο χρόνος εκκίνησης του συστήματος θεωρείται κρίσιμος.

Μετράμε τον χρόνο εκκίνησης του Jailhouse οικοσυστήματος, τον χρόνο επανεκκίνησης του non-root κελιού και τον χρόνο επανεκκίνησης του Jailhouse οικοσυστήματος. Για κάθε μετρική κάναμε συνολικά 50 μετρήσεις. Τα αποτελέσματα συνοψίζονται στον Πίνακα 1 και στην Εικόνα 0.0.1.

Εάν συγκρίνουμε τους μέσους-medians κάθε γραφικής παράστασης κουτιού (box plot), είναι προφανές ότι υπάρχει διαφορά μεταξύ της τρίτης ομάδας (ο χρόνος επανεκκίνησης του οικοσυστήματος Jailhouse) και των άλλων δύο, καθώς η διάμεση γραμμή του box plot της τρίτης ομάδας βρίσκεται έξω από το πλαίσιο των γραφικών του πλαισίου σύγκρισης. Αυτό εξηγείται εύκολα. Για την τελευταία ομαδική μέτρηση χρησιμοποιήσαμε μια επιπλέον εντολή του jailhouse, την **jailhouse disable**. Αυτή η εντολή προσθέτει περίπου 0,4 δευτερόλεπτα επιπλέον καθυστέρηση στο συνολικό χρόνο (επαν)εκκίνησης σε σύγκριση με τις άλλες δύο ομάδες.

Συγκρίνοντας τις δύο πρώτες ομάδες, ο αριθμός των εντολών που εκτελούνται είναι πανομοιότυπος. Η διάκριση αφορά τους τύπους εντολών που χρησιμοποιούνται. Στο πρώτο σενάριο, εκτελείται η εντολή **jailhouse enable**, ενώ στο δεύτερο σενάριο χρησιμοποιείται η **jailhouse cell destroy**. Αυτό μπορεί να ευθύνεται για τη διαφορά μεταξύ του εύρους διατεταρτημορίων, καθώς τα δεδομένα που αναφέρονται για τη δεύτερη ομάδα είναι πιο διασκορπισμένα και διάσπαρτα.

Συνολικά, τα αποτελέσματα υποδεικνύουν έναν προβλέψιμο χρόνο (επανα)εκκίνησης. Ωστόσο, η απόκλιση (outlier) στο box plot του χρόνου επανεκκίνησης του οικοσυστή-

ματος Jailhouse μας ωθεί να αναζητήσουμε μεγαλύτερο άνω φράγμα για το σύνολο των χρόνων (επανα)εκκίνησης.

Σύστημα-Δράση	**Μέσος-Mean**	**Μέσος-Median**	**Τ. Απόκλιση**
JH οικοσύστημα εκκίνηση	1.4518s	1.4515s	0.0189s
JH non-root κελί επανεκκίνηση	1.5072s	1.4697s	0.1306s
JH οικοσύστημα επανεκκίνηση	1.8419s	1.8232s	0.0753s

Table 1: Μέσος-Mean, Μέσος-Median, and τυπική απόκλιση των χρόνων (επαν)εκκίνησης του Jailhouse οικοσυστήματος

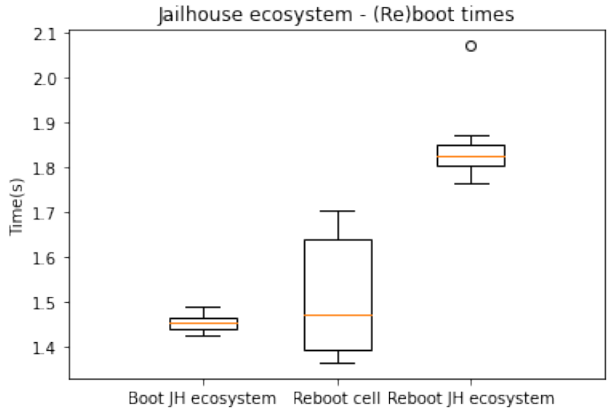


Figure 0.0.1: Χρόνοι (επαν)εκκίνησης του Jailhouse οικοσυστήματος

Συγκριτική Αξιολόγηση Απόδοσης: Πρόβλημα των N Βασιλισσών

Σε αυτή την υποενότητα θα επικεντρωθούμε στη συγκριτική αξιολόγηση απόδοσης. Θα μετρήσουμε τον χρόνο εκτέλεσης μιας εφαρμογής για να συγκρίνουμε τον αντίκτυπο του Jailhouse Hypervisor με αυτόν των Docker κοντέινερ.

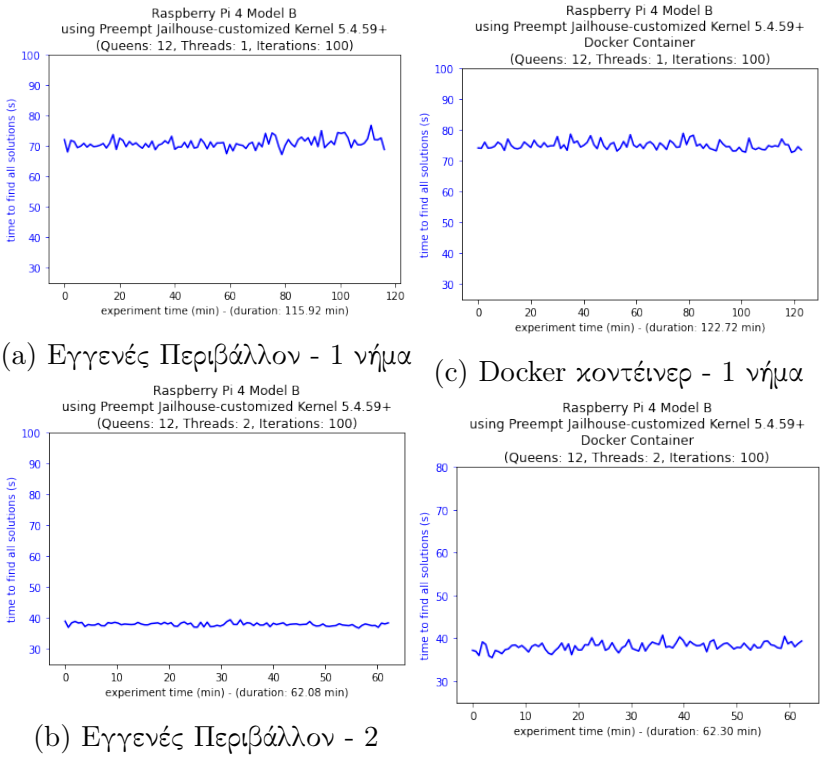
Το πρόβλημα των N Βασιλισσών είναι μια γενίκευση του γνωστού προβλήματος των 8 βασιλισσών, σύμφωνα με το οποίο πρέπει να βρούμε έναν τρόπο να βάλουμε 8 βασίλισσες σε μια κλασική σκακιέρα 8x8, ώστε καμία βασίλισσα να μην μπορεί να επιτεθεί ή να δεχθεί επίθεση από άλλη.

Βρήκαμε έναν κώδικα python στο github [14]. Υπολογίζει όλες τις λύσεις χρησιμοποιώντας τη μέθοδο του backtracking και έχει χρονική πολυπλοκότητα $O(N^2)$. Εκμεταλλεύεται επίσης τον παραλληλισμό, ώστε να μπορούμε να εκτελέσουμε τον κώδικα για διαφορετικούς αριθμούς νημάτων.

Εκτέλεση στο Linux Root κελι με έκδοση 5.4.59+

Διαλέγουμε την έκδοση 5.4.59+ για τον πυρήνα μας λόγω αστοχίας και αποκλίσεων των αποτελεσμάτων από την εκτέλεση στον 5.10.27-rt36. Εκτελούμε τον κώδικα προβλήματος

των N βασιλισσών για N=12, 100 επαναλήψεις και για 1 και 2 νήματα, τόσο σε εγγενές περιβάλλον όσο και σε ένα κοντέινερ Docker. Τα αποτελέσματα παρουσιάζονται στην Εικόνα 0.0.2. Ο χρόνος εκτέλεσης στο εγγενές περιβάλλον είναι λογικός. Για την εκτέλεση μέσα σε Docker κοντέινερ, ο μέσος χρόνος για την εκτέλεση με 1 νήμα είναι 5,8% μεγαλύτερος σε σχέση με το εγγενές περιβάλλον, ενώ για την εκτέλεση με 2 νήματα, η αύξηση είναι 0,3%.



(a) Εγγενές Περιβάλλον - 1 νήμα

(c) Docker κοντέινερ - 1 νήμα

(b) Εγγενές Περιβάλλον - 2 νήματα

(d) Docker κοντέινερ - 2 νήματα

Figure 0.0.2: Εκτέλεση του προβλήματος των N βασιλισσών στο root κελί

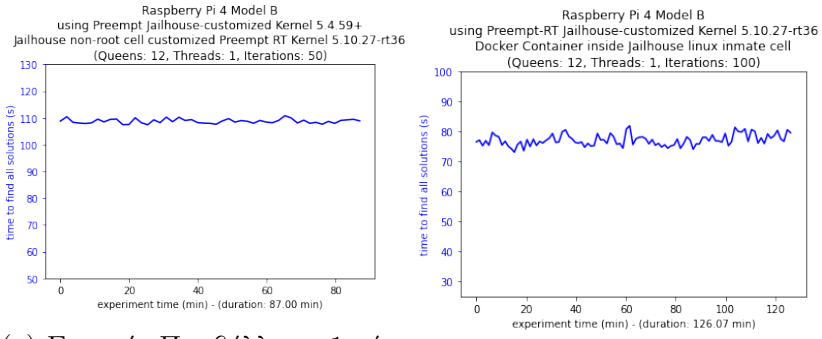
Εκτέλεση στο Linux non-root κελι

Εκτελούμε το μετροπρόγραμμα των N βασιλισσών στο Linux non-root κελί που δημιουργήσαμε και θέσαμε σε λειτουργία. Θα πρέπει να τονίσουμε, αν και αναφέρθηκε προηγουμένως, ότι η έκδοση του πυρήνα είναι 5.10.27-rt36. Η εγγενής εκτέλεση για 1 και 2 νήματα διαρκεί κατά μέσο όρο 69,55 και 37,25 δευτερόλεπτα, αντίστοιχα. Η σύγκριση αυτών των ευρημάτων με τους χρόνους εκτέλεσης στο root κελί αποκαλύπτει μια αύξηση 50,1% και 42,3% στον χρόνο εκτέλεσης, αντίστοιχα. Ομοίως, ο χρόνος εκτέλεσης μέσα σε ένα Docker κοντέινερ αυξάνεται κατά 2,7% και 3,5% για ένα και δύο νήματα, αντίστοιχα.

Όσον αφορά την εγγενή εκτέλεση απευθείας στο non-root κελί, η επιρροή του Jailhouse Hypervisor δεν μπορεί να δικαιολογήσει την τόσο μεγάλη αύξηση του χρόνου. Επιπλέον, οι διάρκειες των επαναλήψεων που μετρήθηκαν είναι σημαντικά μεγαλύτερες από αυτές για την εκτέλεση εντός ενός κοντέινερ docker. Αυτό υποδηλώνει ότι το ζήτημα σχετίζεται με τα ίδια τα λειτουργικά συστημάτα στα root και non-root κελια. Θα πρέπει να αναφέρουμε

ότι οι πυρήνες και οι εικόνες που χρησιμοποιούμε έχουν προσαρμοστεί από τους δημιουργούς και συνεργάτες του πρότζεκτ του Jailhouse Hypervisor. Επιπλέον, ο πυρήνας 5.10 είναι διαμορφωμένος και προσαρμοσμένος για πλατφόρμες γενικά arm αρχιτεκτονικής, ενώ ο πυρήνας 5.4 είναι προσαρμοσμένος ειδικά για το raspberry pi, γεγονός που πιθανώς εξηγεί την τεράστια διαφορά στους χρόνους εκτέλεσης.

Εάν παραβλέψουμε την απροσδόκητη υποβάθμιση της απόδοσης στην εγγενή εκτέλεση και εστιάσουμε στα αποτελέσματα από την εκτέλεση σε κοντέινερ docker, η ελάχιστη αύξηση του χρόνου εκτέλεσης δείχνει ότι ο Jailhouse Hypervisor ασκεί αμελητέα επίδραση στην απόδοση των εφαρμογών λογισμικού. Στην πραγματικότητα, οι εξαιρετικά μικρές διακυμάνσεις που απεικονίζονται στην Εικόνα 0.0.3 καταδεικνύουν την προβλέψιμη συμπεριφορά του οικοσυστήματος μας



(a) Εγγενές Περιβάλλον - 1 νήμα

(c) Docker κοντέινερ - 1 νήμα

(b) Εγγενές Περιβάλλον - 2 νήματα

(d) Docker κοντέινερ - 2 νήματα

Figure 0.0.3: Εκτέλεση του προβλήματος των N βασιλισσών στο non-root κελί

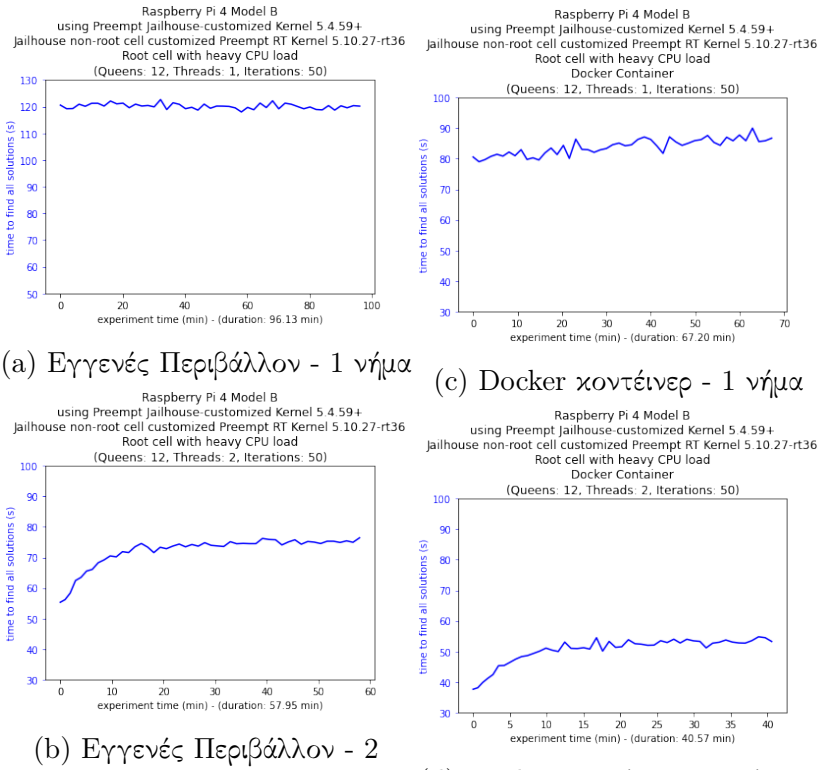
Εκτέλεση στο Linux non-root κελι - Προσομοίωση φορτίου συστήματος

Μέχρι στιγμής, έχουμε εκτελέσει τον κώδικα των N βασιλισσών τόσο στο root κελί ρίζας όσο και στο non-root κελί, αξιολογώντας τον αντίκτυπο του Jailhouse Hypervisor στην απόδοση της εφαρμογής. Εκτός από την εκτέλεση του κώδικα το σύστημα δεν είχε «σημαντικές» εργασίες να ολοκληρώσει, επομένως αυτό το σενάριο δεν θα μπορούσε να θεωρηθεί ρεαλιστικό. Στην πραγματικότητα, το σύστημα προορίζεται να εκτελεί πολλαπλά προγράμματα ταυτόχρονα, συνήθως σε ξεχωριστά κελιά. Αυτές οι εφαρμογές ενδέχεται να περιορίζονται ή να μην περιορίζονται από περιορισμούς πραγματικού χρόνου και να

αποτελούνται από λογισμικό χρηστών ή λογισμικό κρίσιμης σημασίας για την αποστολή του δορυφόρου.

Επομένως, θα επαναλάβουμε το πείραμα εκτελώντας τον κώδικα σε ένα Linux non-root κελί, προσομοιώνοντας σημαντικό φορτίο στο Linux root κελί.

Η εγγενής εκτέλεση σε ένα Linux root κελί για 1 και 2 νήματα διαρκεί κατά μέσο όρο 115,40 και 69,54 δευτερόλεπτα, αντίστοιχα. Η σύγκριση αυτών των ευρημάτων με τα ευρήματα από την εκτέλεση στο non-root κελί χωρίς επιπλέον φορτίο αποκαλύπτει μια αύξηση 10,5% και 33,1% στον χρόνο εκτέλεσης, αντίστοιχα. Ομοίως, ο χρόνος εκτέλεσης στην περίπτωση των Docker κοντέινερ αυξάνεται κατά 6,6% και 25,8% για ένα και δύο νήματα, αντίστοιχα.



(a) Εγγενές Περιβάλλον - 1 νήμα

(c) Docker κοντέινερ - 1 νήμα

(b) Εγγενές Περιβάλλον - 2 νήματα

(d) Docker κοντέινερ - 2 νήματα

Figure 0.0.4: Εκτέλεση του προβλήματος των N βασιλισσών στο non-root κελί με προσομοίωση σημαντικού φορτίου

Σημειώνουμε ότι η ποσοστιαία αύξηση του μέσου χρόνου εκτέλεσης από ένα σε δύο νήματα είναι σημαντική και στα δύο περιβάλλοντα εκτέλεσης. Αυτό το αποτέλεσμα είναι αναμενόμενο, καθώς το μετροπρόγραμμα που εκτελούμε απαιτεί πολλές προσβάσεις στη μνήμη (ανάγνωση και εγγραφή). Δεδομένων των περιορισμένων πόρων της κρυφής μνήμης που διαθέτουμε, είναι βέβαιο ότι θα αντιμετωπίσουμε αρκετές αστοχίες κρυφής μνήμης (cache misses) που απαιτούν περαιτέρω αναζήτηση στη μνήμη RAM. Ωστόσο, έχουμε εκχωρήσει στατικά τους πόρους στο non-root κελί μας και δεν μπορεί να ζητήσει περισσότερα. Με περιορισμένους πόρους και μνήμη RAM που διατίθεται για το Linux non-root κελί, είναι

αναπόφευκτο να προκύψουν πολλά σφάλματα σελίδας, με αποτέλεσμα το πρόγραμμα να αναζητήσει τη σελίδα στον αποθηκευτικό χώρο. Η εφαρμογή **perf** του Linux επιβεβαιώνει τον μεγάλο αριθμό σφαλμάτων σελίδας του προγράμματος.

Ο δίαυλος επικοινωνίας (bus) είναι κοινός για τα κελιά στο Jailhouse οικοσύστημά μας, όπως αναφέρθηκε προηγουμένως. Όλες αυτές οι αστοχίες της κρυφής μνήμης και τα σφάλματα σελίδας σε συνδυασμό με το μεγάλο φορτίο στο Linux root κελί, το οποίο χρησιμοποιεί σε μεγάλο βαθμό το δίαυλο, έχουν ως αποτέλεσμα αυξημένη επισκεψιμότητα (η οποία αυξάνεται με τον αριθμό των νημάτων) και ως εκ τούτου σημαντικές καθυστερήσεις.

Η μελέτη μας επιβεβαιώνει το μειονέκτημα του Jailhouse Hypervisor στην περίπτωση αυξημένης κίνησης στο δίαυλο επικοινωνίας και, κατά συνέπεια, την αρνητική επίδραση στην απόδοση των εφαρμογών.

Καθυστέρηση Προγραμματισμού σε Συστήματα Πραγματικού Χρόνου: cyclictest

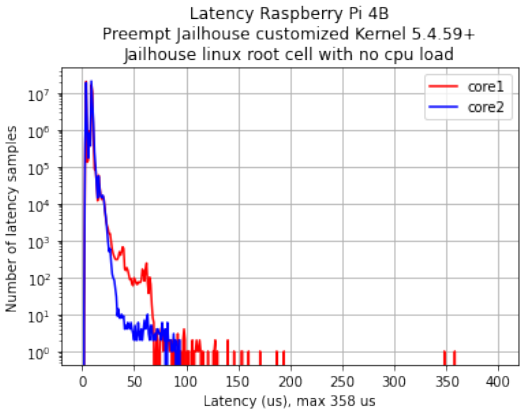
Το επόμενο βήμα μας στην αξιολόγηση του Jailhouse οικοσυστήματος είναι να ελέγξουμε την απόδοσή πραγματικού χρόνου του. Οι εργασίες πραγματικού ενεργοποιούνται συνήθως από εξωτερικά συμβάντα ή την περιοδική λήξη ενός χρονοδιακόπτη. Αυτές με την υψηλότερη προτεραιότητα θα πρέπει να προγραμματίζονται αμέσως μετά την ενεργοποίηση, ωστόσο στην πραγματικότητα, υπάρχει μια καθυστέρηση μεταξύ της στιγμής που λαμβάνει χώρα το συμβάν ενεργοποίησης και της στιγμής που αρχίζει να εκτελείται η εργασία[15]. Αυτή η καθυστέρηση, γνωστή ως καθυστέρηση προγραμματισμού(scheduling latency), επηρεάζει τους χρόνους αντίδρασης όλων των εργασιών και επιβάλλει ένα κάτω φράγμα στην ικανότητα του συστήματος να τηρεί τις προθεσμίες. Επομένως, η καθυστέρηση προγραμματισμού πρέπει να λαμβάνεται υπόψη όταν αποφασίζεται εάν ένα σύστημα μπορεί να παρέχει τις κατάλληλες χρονικές διαβεβαιώσεις. Για μια εμπειρική αξιολόγηση της καθυστέρησης προγραμματισμού, χρησιμοποιούμε το cyclictest, ένα πρόγραμμα ανίχνευσης που αντιμετωπίζει τον πυρήνα ως μαύρο κουτί και αναφέρει απευθείας την καθυστέρηση προγραμματισμού.

Όπως αναφέρεται στο εγχειρίδιό του, το cyclictest[16] μετρά με ακρίβεια και επανειλημμένα τη διαφορά μεταξύ της προβλεπόμενης ώρας αφύπνισης ενός νήματος και της ώρας κατά την οποία ξυπνά πραγματικά, προκειμένου να παρέχει στατιστικά στοιχεία σχετικά με τις καθυστερήσεις του συστήματος. Μπορεί να μετρήσει καθυστερήσεις σε συστήματα πραγματικού χρόνου που προκαλούνται από το υλικό, το υλικολογισμικό και το λειτουργικό σύστημα.

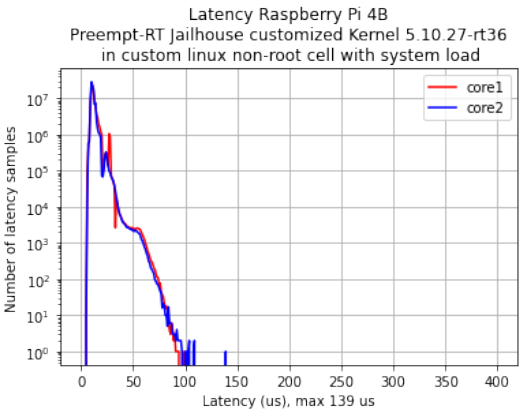
Η σουίτα δοκιμής μας βασίζεται στην παρακάτω εντολή.

```
sudo cyclictest -l100000000 -m -S -p90 -i200 -h400 -q  output.txt
```

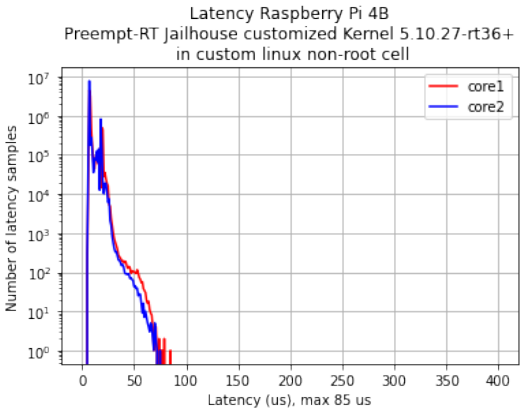
Εκτελούμε την εντολή στο πλαίσιο πέντε διαφορετικών σεναρίων. Τα αποτελέσματα για τα διαφορετικά σενάρια παρουσιάζονται παρακάτω στην Εικόνα 0.0.5.



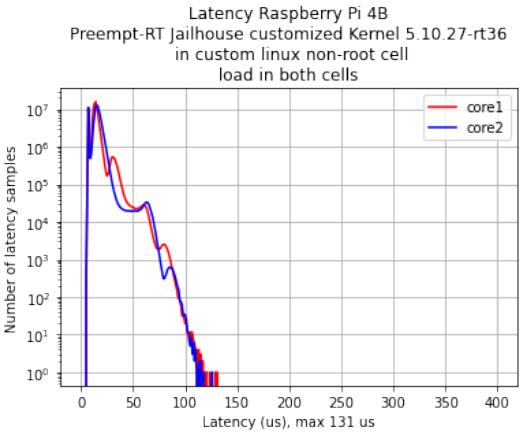
(a) Εκτέλεση στο root κελί - Χωρίς επιπλέον φορτίο



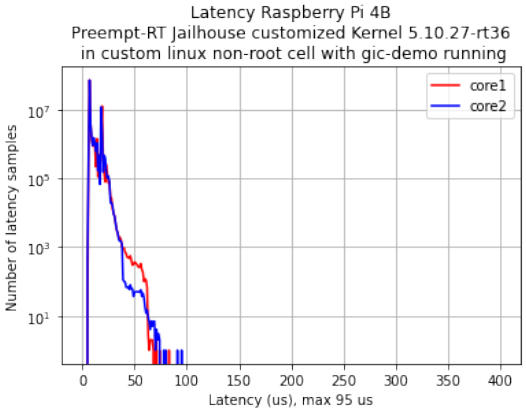
(c) Εκτέλεση στο non-root κελί - Βαρύ φορτίο στο non-root κελί



(b) Εκτέλεση στο non-root κελί - Χωρίς επιπλέον φορτίο



(d) Εκτέλεση στο non-root κελί - Βαρύ φορτίο και στα δύο κελιά



(e) Εκτέλεση στο non-root κελί - Τυπικό φορτίο ενός επιπλέον demo προς εκτέλεση

Figure 0.0.5: Σενάρια εκτέλεσης του cyclictest

Στην πρώτη περίπτωση βλέπουμε ένα σημαντικό αριθμό δειγμάτων καθυστέρησης που

ξεπερνούν τα 100us, με τη μέγιστη απόκριση καθυστέρησης να φτάνει τα 358us. Αυτή είναι η καλύτερη περίπτωση, καθώς δεν έχει προσομοιωθεί πρόσθετο φορτίο στο σύστημα. Κατά τη διάρκεια μιας τυπικής δοκιμής, το γενικό κόστος του Cyclictest είναι χαμηλό, επομένως δεν επιβάλλει σημαντική πίεση στο σύστημα. Σε όλες τις άλλες περιπτώσεις, ακόμη και με ελάχιστες παρεμβολές, είναι δυνατό να ανιχνευθεί υπέρβαση πάνω από 400us ή ακόμα και 1000us, αναμενόμενο για έναν πυρήνα μη πραγματικού χρόνου.

Στο δεύτερο σενάριο το ιστόγραμμά μας δείχνει ότι η καθυστέρηση απόκρισης είναι ντετερμινιστική, καθώς όλα τα δείγματα καθυστέρησης εμπίπτουν σε ένα προκαθορισμένο εύρος. Η μεγαλύτερη απόκριση καθυστέρησης, και κατά συνέπεια το ανώτερο όριο του εύρους, είναι 85us. Ωστόσο, αυτό απέχει πολύ από το χειρότερο σενάριο, επειδή το σύστημα δεν έχει υποστεί καμία πίεση λόγω επιπλέον φορτίου.

Το πείραμα επαναλαμβάνεται με προσομοίωση μεγάλου φορτίου στο Linux non-root κελί. Από το ιστόγραμμα που προκύπτει βλέπουμε ότι τα δείγματα των καθυστερήσεων υποδεικνύουν την ίδια συμπεριφορά πραγματικού χρόνου. Η μέγιστη καθυστέρηση απόκρισης είναι 139 us. Αυτή η εκτέλεση είναι 0,63 φορές πιο αργή από μια ισοδύναμη εκτέλεση χωρίς προσομοίωση πίεσης συστήματος, παρ' όλα αυτά αποτελεί ένα δυστοπικό αν όχι ρεαλιστικό σενάριο.

Στην τέταρτη περίπτωση επαναλαμβάνεται η εκτέλεση του cyclictest, αυτή τη φορά προσομοιάζοντας ταυτόχρονα βαρύ φορτίο και στα δύο κελιά. Παρατηρούμε ότι τα δείγματα των καθυστερήσεων διατηρούν την ίδια ντετερμινιστική συμπεριφορά πραγματικού χρόνου. Η μέγιστη καθυστέρηση απόκρισης είναι 131us και εμπίπτει στο προκαθορισμένο εύρος καθυστερήσεων. Για λόγους πληρότητας, τρέξαμε την εφαρμογή cyclictest και στο root κελί, με ένα πυρήνα μη πραγματικού χρόνου. Υπήρξαν αρκετές υπερχειλίσεις, συγκεκριμένα 0.04% για το πρώτο νήμα και 0.01% για το δεύτερο, ενώ η μέγιστη καθυστέρηση απόκρισης έφτασε τα 19757us ή αλλιως 0.019 δευτερόλεπτα.

Τέλος, θέλουμε να δούμε τι θα γινόταν σε μια μέση περίπτωση προσομοιάζοντας φορτίο μεσαίου μεγέθους. Αυτή τη φορά με την εκτέλεση του cyclictest στο non-root κελί, σηκώνουμε ένα ακόμα inmate κελί τρέχοντας το bare-metal gic-demo. Τα δείγματα των καθυστερήσεων διατηρούν την ίδια ντετερμινιστική συμπεριφορά πραγματικού χρόνου, ενώ η μέγιστη καθυστέρηση απόκρισης είναι 85us και εμπίπτει για άλλη μια φορά στο προκαθορισμένο εύρος καθυστερήσεων.

Συμπεράσματα & Μελλοντικές Κατευθύνσεις

Σε αυτή τη διπλωματική εργασία εξερευνήσαμε δύο τεχνικές εικονικοποίησης, εντυφώντας στον Jailhouse Hypervisor. Χτίσαμε και στήσαμε ένα πλήρως λειτουργικό Jailhouse οικοσύστημα με το οποίο αξιολογούμε την επίδραση του Jailhouse hypervisor πάνω σε αυτό και εν μέρει το συγκρίνουμε με αυτή των Docker κοντέινερς. Η κατάλληλη διαμόρφωση για το χτίσιμο και το στήσιμο του Jailhouse οικοσυστήματος ήταν μια αρκετά περίπλοκη και χρονοβόρα διαδικασία. Αφενός η φύση της διαδικασίας είναι τέτοια, ώστε ανάλογα με το ενσωματωμένο σύστημα που επιλέγεται να αλλάζουν αρκετά τα βήματα της και να προσαρμόζονται στην εκάστοτε πλατφόρμα. Αφετέρου για την συγκεκριμένη πλατ-

φόρμα που επιλέξαμε, δεν υπήρχαν συγκεντρωμένες οδηγίες και κατευθυντήριες γραμμές, οπότε έπρεπε να ψάξουμε αρκετά, ακόμα και να παραθέσουμε ερωτήσεις στην λίστα του Jailhouse Hypervisor.

Μελετώντας, λοιπόν, τον Jailhouse Hypervisor επάνω στο Raspberry Pi καταλήγουμε ότι τα Docker κοντέινερς προσθέτουν μικρές χρονικές δαπάνες στο σύστημα και μπορούν να συνδυαστούν επιτυχώς με τον Jailhouse εξασφαλίζοντας ένα απομονωμένο και ασφαλές περιβάλλον σε συνδυασμό με την ευελιξία που παρέχει η τεχνική της χρήσης κοντέινερς. Όσον αφορά την επίδραση σε συστήματα πραγματικού χρόνου, τα αποτελέσματα ήταν αρκετά ικανοποιητικά όσον αφορά την ντετερμινιστική συμπεριφορά που παρατηρήσαμε. Οι μέγιστες καθυστερήσεις θα μπορούσαν να ναι μικρότερες αλλά σε αυτό οφείλεται και το ίδιο το Raspberry Pi που δεν προτιμάται για την ανάπτυξη κρίσιμων συστημάτων πραγματικού χρόνου. Επιπλέον, επιβεβαιώσαμε τη δύναμη του Jailhouse, την απομόνωση, που δε μας δημιούργησε πρόβλημα σε κανένα σενάριο εκτέλεσης, αλλά και την αδυναμία του, τη μείωση της απόδοσης των εφαρμογών όταν υπάρχει επικοινωνία μεταξύ των κελιών ή αυξημένη κίνηση στο δίαυλο επικοινωνίας.

Μια μελλοντική κατεύθυνση της διπλωματικής αυτής θα ήταν ίσως μια εκτενέστερη και πληρέστερη αξιολόγηση της απόδοσης πραγματικού χρόνου μετρώντας για παράδειγμα την καθυστέρηση διακοπής με την χρήση εξωτερικών συσκευών και εργαλείων. Επίσης, ένα ζήτημα που επείγει να ληθεί είναι η περιορισμένη μνήμη που έχουμε στη διάθεσή μας, έτσι ώστε να μπορέσουμε να τρέξουμε π.χ. λογισμικό τεχνητής νοημοσύνης ή/και μηχανικής μάθησης που πολλές φορές απαιτεί αρκετούς διαθέσιμους πόρους για τη διαχείριση των δεδομένων. Επιπλέον, βλέποντας ότι ο Jailhouse έχει ήδη ξεκινήσει να γίνεται αγαπητός στη διαστημική κοινότητα θα ήταν ωραίο να αρχίσει να εξετάζεται ο συνδυασμός του Jailhouse με τα Docker containers και να δοκιμαστεί τελικά επάνω σε δορυφόρο.

Chapter 1

Introduction

We live in a data driven era where data collection and data processing play an essential role in people's everyday lives. Businesses, governments, industries, nonprofits, and scientific research produce and share an unprecedented amount of data. The space sector plays a key role in obtaining these data, whether quantitative or qualitative. We collect vital data from satellites that enables early detection of environmental and climatic changes.

Space is widely recognized as one of the most promising areas for future scientific and economic advancement. When it comes to boosting a country's competitiveness and growth while also providing the foundation for initiatives that attempt to better the lives of its population, the sector of satellite-based services and applications has gained widespread recognition as a valuable economic tool.

As opposed to the latest space sector market trends, traditional satellites are mainly focused on executing specific long-term missions in space. They are designed to execute only one particular task during their mission, and their lifetime is predetermined, meaning that after their mission they either burn up entering the earth's atmosphere or become space junk. Furthermore, the internal design of a traditional satellite suggests that each avionic function of the whole system (e.g. orbit control, attitude control, payload data processing etc) is implemented and contained in units so that it is quickly replaceable in case of partial failure. The main drawback to this perspective is that there is a large number of buses, networks and point-to-point connections being used to achieve successful communication and data transferring among all different processing modules, making the whole system a lot more complex and susceptible to human mistakes.

Consequently, development and testing time have very large duration, even larger than the mission itself sometimes, revealing the high costs of the satellite's deployment. That is the reason only few could manufacture and launch a satellite so far, thus the space market being limited. Finally, the point of view that dominates the space sector is to prioritize the deterministic behavior of the satellite. Thus, the technology used for the satellite's creation and launch is not the most updated one. Rather than that,

technology that was invented many years ago and tested repeatedly is preferred, making the whole concept not corresponding with today's world's real needs.

What is more, the amount of raw data generated by current instruments on board a satellite frequently exceeds the amount that can be transmitted to the ground. This necessitates the utilization of various data compression techniques in order to lower their volume. The entire process of acquiring raw data, transmitting and storing it on the satellite in order to compress and transmit it to the ground, where it will eventually be processed, is quite complex and time-consuming.

Over the past few years, the space business has expanded, bringing satellite as a service as a growing segment of the sector. Satellites have shifted from an upfront customer purchase approach to a service model, requiring just the creation of a satellite subscription plan as opposed to a satellite investment or purchase. Thus, the space business has a considerably broader target audience, as the cost of sending an individual project into space decreases by orders of magnitude.

Satellite as a Service's primary objective is to maximize the utilization of orbital resources while introducing novel ideas, such as the concept of data processing onboard a satellite. The most recent market trends suggest expanding the number of satellites adopting this concept, while introducing new technology to the space industry, such as Deep/Machine Learning and/or Artificial Intelligence in general. Missions such as providing remote sensing photography, radio-frequency signal gathering, communications, navigation, and other similar operations may now be carried out with the help of AI/ML systems that demonstrate great potential in terms of security and efficiency.

ML algorithms for onboard data processing present an innovative, efficient method of calculating the necessary findings, hence lowering the final data transfer to the ground, taking into account, for instance, the vast volume of data being produced for subsequent analysis on the ground.

It is evident from the above that SaaS is leading to flexible, software-defined technologies. Old, conventional, and purpose-built satellites should be changed or replaced. Each satellite will serve numerous applications and missions, -requiring a single hardware platform to support multiple software. Different SW must be deployed on the same platform, necessitating HW abstraction and a hardware-agnostic HW implementation. Virtualization is a key component of this transition.

1.1 Challenges

Virtualization is a broad concept with numerous specialized extensions and techniques. In order to determine which technology is most compatible with our needs, it is necessary to address certain challenges based on the general requirements and constraints of satellites and the execution of software with mixed criticality.

1. **Isolation - Recovery**
   From the security of individual electronic components to the safety of the

entire spaceship, the methodology of fault detection, isolation, and recovery (FDIR) is widely used in space engineering[17]. For instance, current space computer chips can execute calculations on a multiple, parallel basis, sometimes voting to select which of an inconsistent set of findings is the most likely to be reliable in an effort to recover from memory 'bit flips' caused by space radiation.

Similarly, the FDIR technique uses sensor cross-checks to detect defects in real time, isolating them as necessary before compensating using sensor or actuator reconfigurations or, in the case of an emergency, an autonomous collision avoidance maneuver.

In this context, we are looking for a virtualization method that correctly isolates virtual environments. No partitions should interact with one another so that, in the event of a failure, the failure does not spread to the satellite's vital components and satellite operation software. In addition, the system must recover as rapidly as feasible (in real-time, for real-time applications) and continue to function properly.

2. **Real-Time Performance** As previously indicated, Real-Time Performance must also be established, not only for the needs of a given mission, but also for certain satellite functions that must be done in real-time for the satellite to work as intended.

3. **Flexibility - Simplicity - Reusability** The SaaS concept, as mentioned above, demands the terms of flexibility and simplicity to be applied. A variety of different SW has to be able to run on board the satellite in combination with a fast, simple and efficient reconfiguration in orbit from mission to mission. As a natural consequence, the same hardware platform has to be reused hosting different SW with divergent needs and purposes.

4. **User friendliness** The user-friendliness of the virtualization approach to be chosen is also crucial. For executable code/projects to be uploaded and executed, the procedure must be straightforward to all potential users and easy to follow.

In this diploma thesis, we investigate the Jailhouse Hypervisor on an embedded platform in an effort to address the aforementioned challenges. Our goal is to examine and evaluate the impact of this hypervisor in the context of running mixed-criticality applications comparing this virtualization technique with containerization and, more specifically, Docker containers.

1.2 Contributions

The contribution of this thesis is twofold. First, we create a fully functional system consisting of a Linux-powered hardware platform, the Jailhouse Hypervisor, and a Linux non-root cell that has been customized for optimum operation. This system, which we refer to as the "Jailhouse ecosystem," will be used as a tool for our second contribution.

Specifically, we examine and assess the impact of Jailhouse Hypervisor on our Jailhouse ecosystem by generating and executing various execution scenarios.

For the first part of this thesis:

- We run the Jailhouse hypervisor in both x86-64 and aarch64 architecture environment offered by the QEMU emulator, including executing pre-existing examples, and gain insight into this topic.
- We install the Jailhouse hypervisor as a kernel module and deploy it on a Raspberry Pi 4B board.
- We check the Jailhouse hypervisors' functionality by running some basic commands regarding creation and distraction of cells, as well as running some basic demos.
- We investigate the proper configuration to be applied for our Linux non-root cell in depth. After creating our customized Linux non-root cell, we perform a series of setups to ensure that it is completely operational and adjusted to our needs.

For the second part of this thesis:

- We study the boot - recovery times of our Jailhouse ecosystem, an important aspect of system performance.
- We do performance benchmarking. We measure the execution time of an application to compare the impact of Jailhouse Hypervisor to that of Docker containers. We run the benchmarking code natively and within Docker containers both in Linux root and non-root cell and evaluate the different outcomes.
- We our Jailhouse ecosystem's check real time performance.

1.3 Thesis Structure

The thesis is divided into 7 chapters. The following is an overview of the contents of each chapter:

Chapter 2 contains a detailed survey of current representative cases of virtualization solutions and related techniques that are already being utilized or have the potential to be employed in the space sector, taking into account current industry trends and dimensions.

In Chapter 3 we discuss in detail the essential technologies underlying this study, including the concept of virtualization and its various sorts and the mechanism of hypervisor and its various types, paving the way for the Jailhouse Hypervisor chapter.

Chapter 4 is dedicated to the Jailhouse Hypervisor, the protagonist of this diploma thesis. We analyse the concept and operation of the Jailhouse Hypervisor, the primary virtualization technique to be studied, explored, and evaluated.

Our main research work is included in Chapters 5 and 6. Specifically, Chapter 5 is devoted to the investigation and development of our system, which consists of a real hardware target with custom Linux installed, the Jailhouse Hypervisor, and a fully operational custom Linux non-root cell. We detail the process of constructing this system, which will henceforth be known as the "Jailhouse ecosystem" and will be used to evaluate the Jailhouse Hypervisor extensively.

Chapter 6 is dedicated to the experiments conducted to measure the impact of the Jailhouse Hypervisor on application execution, as well as the evaluation of a number of metrics to determine the hypervisor's overall behavior. We focus on measuring and studying the following sections:

1. The boot and recovery time of our Jailhouse ecosystem
2. The impact of Jailhouse Hypervisor on applications' performance
3. The scheduling latency on the cells of the Jailhouse ecosystem

Finally, Chapter 7 summarizes the thesis's findings and presents possible future directions.

Chapter 2

Related Work

This chapter examines current representative cases of virtualization solutions and related techniques that are already being utilized or have the potential to be employed in the space sector, taking into account current industry trends and dimensions.

The examples are chosen from four categories that correspond to four trends in the current or future adoption of virtualization in industry domains[1]:

1. Kernel and microkernel separation-based solutions created primarily for industrial and embedded domains.
2. Solutions that attempt to leverage on the strengths of existing general-purpose hypervisors and adapt them to industry requirements.
3. Utilizing the most recent isolation features in hardware (e.g., ARM TrustZone) to achieve the isolation guarantees required by industry standards.
4. Compared to conventional virtualization methods, solutions that employ lightweight virtualization, such as containers or unikernels, in an effort to reduce the footprint and increase the flexibility

In addition, a new platform will be discussed that aims to enter the space community and alter the way that high performance safety-critical systems have been treated up until now.

2.1 Separation Kernel & Microkernel Approaches

2.1.1 XtratuM

Masmano et al. present the embedded hypervisor XtratuM [18]. It is a bare-metal partitioning hypervisor that supports paravirtualization. At compilation time, the data structures are predefined, so that every resource required is known beforehand. It has an interface for hypercalls, each of which has a known execution time. Utilizing a fixed cyclic scheduler, Xtratum facilitates temporal isolation across partitions. Each

partition also contains its own operating system and applications executing in user-mode without any shared memory region with other partitions, hence permitting spatial separation. These partitions operate continuously, communicate, and are governed by the hypervisor. In order to reduce temporal interferences, Xtratum only enables interruptions for partitions that are actively executing.

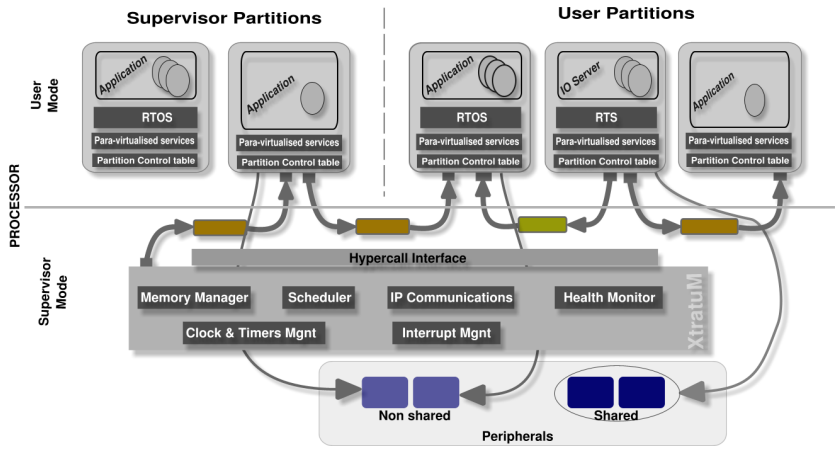


Figure 2.1.1: Complete Systems Architecture as described in [18]

The aircraft industry is utilizing XtratuM to construct software building blocks for future generic on-board software devoted to payloads management units. The Xtratum hypervisor is used in the aerospace sector due to the increased need for security as well as its capacity to meet the avionics' criteria about predictability[18][19].

Xtratum supports several processors like Intel x86 family, SPARCv8 family, ARMv7, ARMv8 and RISC-V[20]. In particular, Wessman et al. [21]present the De-RISC platform, the first RISC-V-based platform that includes the NOEL-V SoC from Cobham Gaisler and the XtratuM hypervisor from fentISS. De-RISC combines multiple technologies and solutions that overcome the obstacles to their effective adoption in mission-critical and safety-critical space systems.

2.1.2 PikeOS

SYSGO's PikeOS [22][23] is a commercial hypervisor used in the avionics industry. The real-time separation kernel approach of PikeOS enables the execution of separate guest environments as well as native tasks. PikeOS uses paravirtualization, hardware-assisted virtualization, and direct I/O access for running guest operating systems. The PikeOS architecture is built on the L4 microkernel and is compatible with Intel x86, ARM, PowerPC, SPARC v8/LEON, and MIPS processors. PikeOS natively supports platforms with multiple cores. This solution implements three different scheduling algorithms: priority-based, time-driven, and proportional share[1].

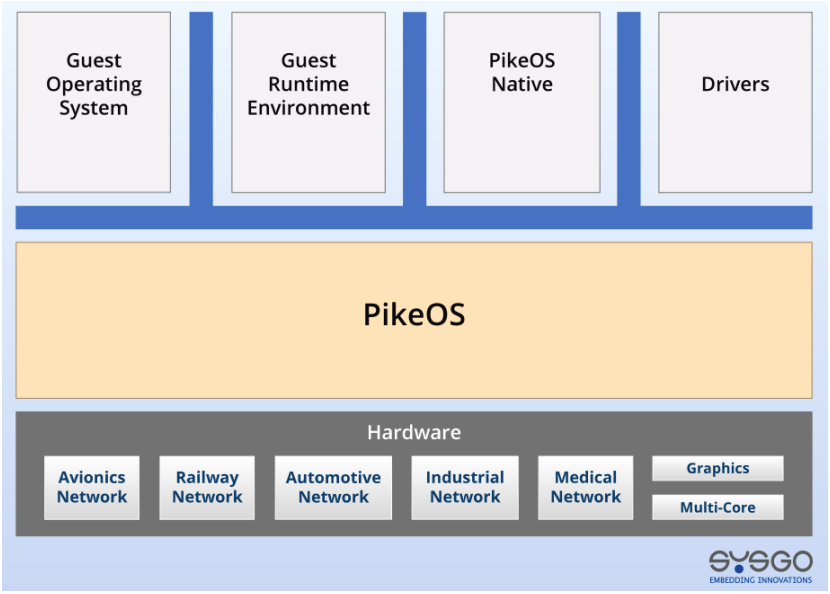


Figure 2.1.2: PikeOS hypervisor layers [22]

PikeOS combines time- and priority-driven scheduling with best-effort scheduling for non-critical tasks in order to payload applications. Specifically, it assigns a time-slice statically whenever a real-time partition has no tasks to execute and donates CPU time to non-real time partitions. PikeOS architecture complies to ARINC653 in that the PikeOS microkernel is the only privileged software and has complete control over virtual partitions. PikeOS supports several guest OSes like Android, RT-POSIX, ARINC653-based, Java and RTEMS.

2.1.3 Quest-V

Quest-V [24] is a separation kernel that aims for configurable partitioning of CPU, memory and I/O resources amongst guests with minimum monitor activity. In particular, each partition, or sandbox, encapsulates a subset of the machine's physical resources, which it manages independently of the hypervisor. Quest-V only requires a hypervisor to initialize the system, recover from errors, and establish communication channels between sandboxes. Quest-V uses paravirtualization techniques to boot Linux kernel as a guest and on hardware-assisted virtualization to achieve efficient resource (memory) partitioning and performance isolation for subsystem components.

Quest-V research [25]investigates how hardware resources can be managed to enable a power- and latency-aware system. The partitioning of virtual machines (VMs) onto separate machine resources enables power management per sandbox. Consequently, during periods of inactivity, a sandbox may place its hardware into a suspend state, thereby reducing its power consumption. Depending on their latency and energy requirements, sandboxes may be suspended to RAM or to disk. When required, the sandbox can then return to normal power consumption. Additionally, sandboxes can be migrated between hosts to equalize system resources and reduce energy consumption. This allows

for the placement of entire machines into low-power states upon the migration of all sandboxes from those machines. Quest-V differs from conventional hypervisors in that it permits virtual machines to suspend and resume hardware resources independently of other virtual machines running on the same physical platform. This enables the construction of systems that are cognizant of both power and latency.

2.1.4 Bao

Bao [26][27] is a lightweight, open-source, embedded hypervisor that aims to offer strong isolation and real-time guarantees. Bao provides an implementation of the partitioning hypervisor architecture that is minimal and built from scratch.

Designed primarily for mixed-criticality systems, Bao places a strong emphasis on fault containment isolation and real-time behavior. The static partitioning hypervisor architecture is implemented using only a minimal, thin layer of privileged software that leverages ISA virtualization support. In addition, resources are statically partitioned and assigned at the time of VM instantiation, and memory is assigned statically using two-stage translation. Regarding time, exclusive CPU assignment eliminates the need for a scheduler, which, in conjunction with the availability of per-CPU architectural timers managed directly by the guests, enables complete logical temporal isolation. IO is pass-through only, and virtual interrupts are mapped directly to physical ones.

Bao has no external dependencies, such as those on privileged VMs running large monolithic general-purpose operating systems (such as Linux), and as a result, its TCB is significantly smaller.

Currently, Bao is compatible with the Armv8 architecture. RISC-V experimental support is also available, but since it depends on the not-yet-approved hypervisor extensions, the hypervisor has been tested only on QEMU virtual environment.

2.2 General Purpose Hypervisors

Xen and KVM hypervisors were among the most popular server virtualization solutions in the past. Despite the fact that both Xen and KVM belong to the general-purpose hypervisors category, they can be used as customized and functional solutions for embedded systems and real-time cloud environments [28, 29, 30].

2.2.1 Xen

The Xen Project hypervisor [31] is a type-1 or baremetal open-source hypervisor responsible for managing CPU, memory, and interrupts. It is the first program to run following the exit of the bootloader. The hypervisor is not aware of the existence of I/O functions such as networking and storage.

The virtualized environments are divided into domains by the Xen hypervisor. It employs a specialized management interface, the Control Domain (or Domain 0 / Dom0),

to manage the hypervisor's runtime operation.

Dom0 can directly access the hardware, manages access to the system's I/O functions, and interacts with other Virtual Machines. The Xen Project hypervisor cannot function without Dom0, the first virtual machine launched by the system. In a typical configuration, Dom0 includes the following functions:

- System Services: such as XenStore/XenBus (XS) for managing settings, the Toolstack (TS) exposing a user interface to a Xen-based system, and XenServer (XS) for managing XenServer instances and Device Emulation (DE) in Xen-based systems that is based on QEMU.
- Native Device Drivers
- Virtual Hardware Device Drivers
- Toolstack: allows a user to manage the creation, destruction, and configuration of virtual machines.

The other created virtual machines, known as Guest Domains, each run their own operating system and applications. Several distinct virtualization modes are supported by the hypervisor. Guest VMs are completely isolated from the hardware, meaning they do not have access to hardware or I/O functionality. Consequently, they are also known as unprivileged domain (or DomU).

RT-Xen is one of the most prominent real-time applications of Xen, providing a hierarchical real-time scheduling framework. Xi et al. [32] conducted an empirical investigation of fixed-priority hierarchical scheduling in Xen, focusing on four real-time schedulers: Deferrable Server, Periodic Server, Polling Server, and Sporadic Server. Deferrable Server is better suited for soft real-time applications, whereas Periodic Server performs poorly in an overloaded environment.

Recently, Xen has been used as a component in Xilinx embedded systems. Xilinx chooses Xen for the following reasons:

- It is a reliable and robust solution
- Recent developments of Xen take full advantage of ARMv8 and underlying virtualization hardware, such as ARM System Memory Management Unit (SMMU)
- It has a free-of-charge license and an active user and developer community and technical support that confirms its maintainability

2.2.2 KVM

KVM-based solutions are mainly based on patching the host Linux kernel or improving KVM itself in order to comply with real-time constraints. The PREEMPT_RT [33] is a set of patches of the Linux kernel, which provide realtime guarantees (e.g., predictability, low latencies) still using a single-kernel approach. The PREEMPT RT patch provides several mechanisms like high-resolution timers, threaded interrupt

handlers, priority inheritance implementation, Preemptible Read-Copy-Update (RCU), real-time schedulers, and a memory allocator.

In order to be compliant with real-time constraints, KVM-based solutions primarily rely on modifying the host Linux kernel or enhancing KVM itself. PREEMPT_RT [33] is a collection of Linux kernel patches that provide realtime guarantees (such as predictability and low latency) while maintaining a single-kernel approach. The PREEMPT_RT patch offers a variety of mechanisms, including high-resolution timers, threaded interrupt handlers, priority inheritance implementation, Preemptible Read-Copy-Update (RCU), real-time schedulers, and a memory allocator.

Kiszka et al. [34] designed a task-level scheduler that employs paravirtualization techniques and collaborates with KVM via two new hypercalls in order to manage threads with varying priorities. KVM functions as a real-time hypervisor by assigning higher priorities to real-time threads within a virtual machine and lower priorities to threads operating at the host layer. Cucinotta et al. [35] extended the Linux cgroups interface to create a scheduling algorithm. The authors suggested using a variant of the CBS (Constant Bandwidth Server)/EDF scheduler for inter-VM scheduling (at the hypervisor level) and a fixed-priority scheduler within each guest virtual machine.

In a different study, Cucinotta et al. [36] investigate I/O issues. They intend to group VM threads, KVM threads, and kernel threads required for I/O virtualization (e.g., network or disk) in the same reservation. Zhang et al.[37] implemented numerous real-time tuning techniques. Using the PREEMPT_RT patch on a Linux host. They emphasized on the basis of a dual-guest scenario, they integrated an RTOS and GPOS on a single instance of KVM.

2.3 ARM TrustZone-assisted Virtualization

To increase the isolation of virtual domains, the research community investigated the possibility of utilizing hardware-assisted security solutions in the safety-critical domain. ARM architectures with TrustZone [38] are the most widespread solutions.

This technology offers temporal and spatial isolation between the two execution environments [39, 40, 41]. In particular, for virtualization purposes, these two worlds,non-secure and secure, are used to run multiple VMs managed by hypervisor, operating in monitor mode. Using ARM TrustZone with a dual-guest OS configuration, researchers ran a general-purpose operating system (GPOS) in the non-secure world and a real-time operating system (RTOS) with elevated privileges in the latter. This separates critical from non-critical tasks running on top of the RTOS.

Utilizing the ARM TrustZone technology, Pinto et al. [42] demonstrate an intriguing method for running a realtime operating system in parallel with Linux on a single CPU. Their strategy preserves real-time capabilities by employing fast interrupts (FIQs) exclusively for real-time-critical devices. These interrupts, unlike regular IRQs, arrive

directly in the secure environment, where the real-time OS and hypervisor execute. Normal interruptions occur in the non-secure environment, which is isolated from the secure world.

2.4 Lightweight Virtualization-Containerization

In industrial areas, lightweight solutions based on OS-level virtualization with containers and unikernels are starting to be explored [1]. A current trend in the realtime area is the adoption of OS-level or container-based virtualization. The objective is to replace virtual machines with containers in mixed-criticality systems to achieve isolation with a compact footprint [43]. In many situations, it is not necessary to host a whole operating system within a virtual machine (VM), especially when only certain OS features are required. The fundamental idea is to improve the OS process abstractions (called containers).The majority of Linux-based real-time container solutions in the literature take one of two approaches: 1) the use of co-kernels such as [44] or 2) the modification of the Linux scheduler [45].

2.5 SELENE Project

SELENE [2, 3] is a self-monitored Dependable platform for Safety-Critical Systems that proposes a new set of safety-critical computing platforms based on open source components such as the RISC-V instruction set architecture, GNU/Linux, and the Jailhouse hypervisor, which we will study in this Diploma thesis, to solve the problem of safety-critical systems with limited performance and/or lack of flexibility.

This sophisticated computing platform aims to:

1. build an abstract system and adjust it to the unique requirements and limits of different applications by changing its environmental settings as well as the system's conditions;
2. guarantee functional isolation properties, enabling the coexistence of mixed criticalities and performance requirements on the same hardware platform
3. achieve flexible diverse redundancy by leveraging the capabilities of the multicore
4. efficiently execute compute-intensive software applications with the help of specific accelerators

Chapter 3

Analysis of virtualization techniques

This chapter discusses in detail the essential technologies underlying this study, including the concept of virtualization and its various sorts and the mechanism of hypervisor and its various types, paving the way for the Jailhouse Hypervisor chapter.

3.1 Virtualization

Virtualization is a commonly used term that can be interpreted and defined in a variety of ways based on the context and the technologies employed. If we attempt to give a comprehensive but rather abstract definition of the word 'virtualization' we might say that it is a technology that allows us to take a single, physical hardware system and create multiple virtual environments, usually called Virtual Machines (VMs) by dividing it. Those environments can run simultaneously and independently from one another. One or more techniques such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service [4] are used, determining a specific type of virtualization.

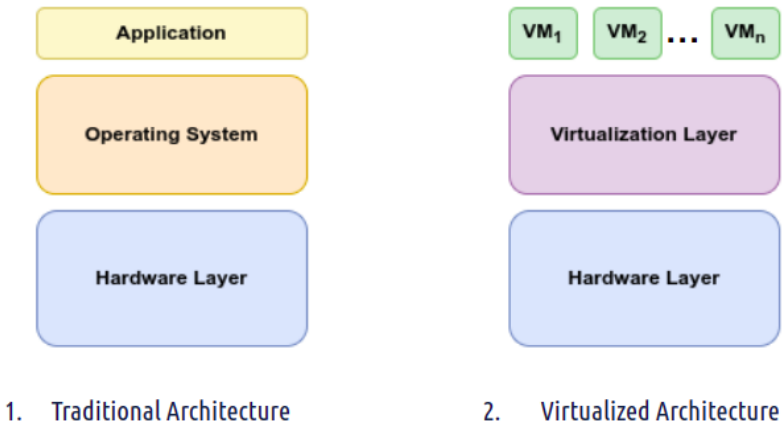


Figure 3.1.1: Traditional and Virtualized Architecture

3.1.1 Why does Virtualization matter?

Virtualization can be considered one of the most appealing architectural strategies for implementing mixed-criticality systems. i.e., to integrate software components with different levels of criticality on a shared hardware platform

Satellites consist of systems with mixed criticality. In other words, they often have at least one real-time component in which it is crucial to complete particular activities within a deterministic, guaranteed time frame. A non-real-time component may also be added, which is typically utilized for real-time information and data processing, system management or configuration.[5]

Real-time components can be impacted by non-real-time programs without virtualization, necessitating frequent separation and execution on a distinct physical CPU. Using virtualization, various components can be merged on a single platform while maintaining the system's real-time integrity.[5]

3.2 Virtualization Techniques

There are different approaches to implementing virtualization, suitable for different situations. The purpose of this section is to describe in general terms the virtualization techniques in common use today.

3.2.1 Trap and Emulate

Trap and emulate is a method in which the user applications and guest operating system of virtual machines both run in user mode, while the hypervisor runs in privileged mode. Non-privileged instructions are executed natively on the hardware. When executing a privileged instruction in virtual user mode, a trap to virtual kernel mode is triggered. This results in a trap for the VMM to handle. The hypervisor assumes control of the execution and emulates the behavior of the instruction to the guest operating system. The required corresponding operations are conducted in the underlying ISA, and the guest is then returned to user mode. Nevertheless, the Popek-Goldberg theorem states, "For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions" [46] which means that for virtualization to be feasible using the trap and emulate method, every sensitive instruction must also be privileged.

3.2.2 Full Virtualization - Binary Translation

The aforementioned issue can be resolved with the binary translation technique that enables full virtualization. In full virtualization, the primary hardware is made accessible to the guest operating system, which is oblivious of such abstraction and has no need for modification. Direct execution for non-sensitive instructions and binary translation for sensitive instructions or hardware traps allow for complete virtualization. The

translation of kernel code replaces non-virtualizable instructions with new sequences of instructions that have the desired impact on virtual hardware.

3.2.3 Paravirtualization

Paravirtualization is a technique that leverages communication between the hypervisor and the guest OS to improve performance efficiency. This method includes altering the kernel of the operating system to replace nonvirtualizable instructions with hypercalls that connect directly with the hypervisor of the virtualization layer. In addition to providing hypercall interfaces for memory management and interrupt handling, the hypervisor also provides interfaces for other crucial kernel activities, such as memory management and interrupt handling. Paravirtualization differs from full virtualization in that the unmodified operating system is unaware that it is virtualized and sensitive OS calls are intercepted using binary translation. Depending on the workload, the performance advantage of paravirtualization versus full virtualization can vary significantly. Since paravirtualization is incapable of supporting unmodified operating systems, its compatibility and portability are weak.

3.2.4 Static Partitioning

Setting up and configuring the aforementioned procedures can be difficult, particularly for embedded devices where simplicity is essential. In order to simulate independent systems, static partitioning isolates programs or tasks to specific pieces of the current hardware. The software partitions are connected to the hardware partitions, limiting the number of operating systems per physical core to one.

All approaches permit the separation of isolated tasks (such as numerous users sharing a system) and vital jobs from less-critical chores (such as separating a secure domain from a general-purpose domain). Nevertheless, the flexibility that full virtualization offers is sacrificed by static partitioning in exchange for some guarantees of determinism. Since different virtualization strategies react to hardware differently, each has inherent advantages for specific applications.

In a static partitioning approach, the platform's physical resources restrict the number of viable virtualized environments. Whether the mixed-criticality tasks are related to security, safety, or real-time operation, statically partitioned systems offer the same advantages for separating them. The scheduling of tasks is less impacted, however, because the physical resources are coupled more closely to the virtualized environments. This makes partitioned systems more appropriate for embedded systems with limited computing resources.

3.3 Types of Virtualization

Virtualization and its mechanism can be given a more comprehensive definition by identifying its various subcategories. The primary distinction can be based on the

technology or category of service to be virtualized [47]. This term is most commonly associated with server virtualization, which is natural given the widespread adoption of virtualization techniques on the server side. Nevertheless, there are numerous layers of technology that virtualize a portion of a computing environment, and as a result, there are various types of virtualization. Apart from server virtualization, terms like CPU virtualization, storage virtualization, network virtualization, I/O virtualization, and management/configuration

3.4 Concept of Hypervisor

A hypervisor (or sometimes referred to as Virtual Machine Monitor - VMM) is a software layer that abstracts hardware resources in order to run multiple Virtual Machines (VMs) or guests on the same physical machine.

Since the guest VMs are independent of the host hardware, hypervisors make it possible to utilize more of a system's available resources and increase IT mobility. This means that they can be easily transferred between servers. Because a hypervisor allows multiple virtual machines to run on a single physical server, it reduces energy, space, and maintenance requirements.

3.4.1 Types of Hypervisor

There are two primary hypervisor types; "Type 1" or "bare metal" and "Type 2" or "hosted". A type 1 hypervisor behaves as a lightweight operating system and runs directly on the hardware of the host, whereas a type 2 hypervisor runs as a software layer on an operating system, similar to other computer applications. Figure 3.4.1 depicts an abstract representation of the architecture's system "joining forces" with the different hypervisor types.

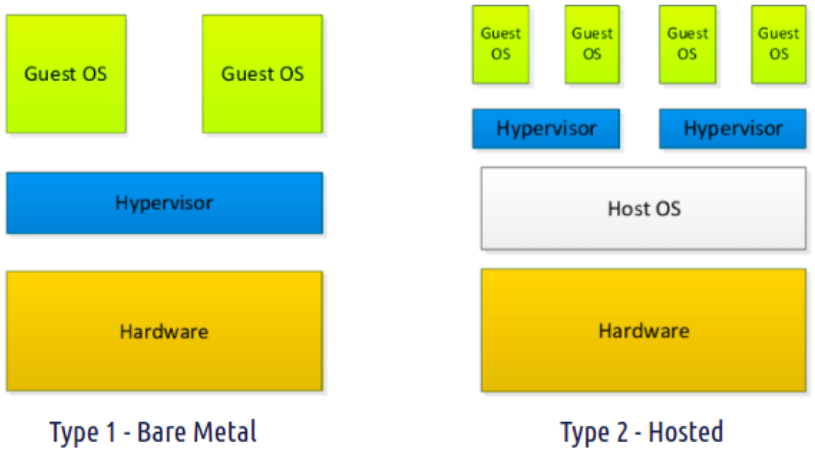


Figure 3.4.1: Types of Hypervisor taken from [48]

The type 1 or bare-metal hypervisor is the most frequently deployed type of hypervisor,

where virtualization software is installed directly on the hardware where the operating system is normally installed. Due to their isolation from the attack-vulnerable os, bare-metal hypervisors are extremely secure. Moreover, they perform better and are more efficient than hosted hypervisors. Due to these factors, the majority of enterprise organizations choose bare-metal hypervisors for data center computing requirements.

Hosted hypervisors run on top of the operating system (OS) of the host machine, as opposed to bare-metal hypervisors which run directly on the hardware. Despite the fact that hosted hypervisors operate within the OS, additional and different operating systems can be installed on top of the hypervisor. The latency of hosted hypervisors is larger than that of bare-metal hypervisors. This is due to the fact that communication between the hardware and hypervisor must pass through the additional OS layer. Hosted hypervisors are sometimes referred to as client hypervisors due to their frequent use with end users and software testing, where latency is less of an issue.

Chapter 4

The Jailhouse Hypervisor

This chapter is dedicated to the Jailhouse Hypervisor, the protagonist of this diploma thesis. We analyse the concept and operation of the Jailhouse Hypervisor, the primary virtualization technique to be studied, explored, and evaluated.

4.1 What is Jailhouse?

Jailhouse [6] [7] is a static partitioning hypervisor based on Linux, started by Jan Kiszka, a lead developer at Siemens, AG. In 2013, Siemens, AG. decided to open source it [8]. It offers a high level of isolation between its partitions, which can run either bare-metal applications or guest operating systems. The current version of Jailhouse is 0.12, and it has active developer and community support. It supports ARM, ARM64, and x86 architectures and has recently taken its first steps toward RISC-V support. [8] [2].

This scheme does not strictly adhere to the traditional classification of hypervisors [9] ; rather, it is a hybrid of Type-1 and Type-2 hypervisors [7] : Once activated, jailhouse hypervisor runs natively on hardware like a bare-metal hypervisor without an underlying system level, without any need of external support. Nevertheless, Linux is required for Jailhouse to be loaded and configured. In other words, Linux is used as a bootloader, but not as a host operating system. From one perspective, this could be interpreted as an attempt to successfully configure and port the hypervisor according to the selected platform's specific requirements, in a - relatively - abstract and simple way. It is difficult to support the variety of hardware currently available. Linux, on the other hand, is a robust operating system in terms of hardware support. This advantage is used by Jailhouse to hijack Linux. The unusual deferred activation procedure of the VMM has the substantial advantage of offloading the majority of hardware initialisation to Linux, allowing Jailhouse to focus solely on managing virtualisation extensions.

Jailhouse hypervisor prefers simplicity over feature-richness. Instead of using complex and time-consuming (para-)virtualisation schemes, like Xen or KVM, to emulate device drivers and share physical hardware resources, Jailhouse only provides isolation (by

exploiting virtualisation extensions) but intentionally neither provides a scheduler nor virtual CPUs. Only (few) resources that cannot, depending on the hardware support, yet be partitioned in that way are virtualised in software.

It converts symmetric multiprocessing (SMP) systems to asymmetric multiprocessing (AMP) systems by introducing "virtual barriers" between the system and I/O bus. The system is split into isolated partitions called "cells". Each cell runs one guest and has a set of assigned resources (CPUs, memory regions, PCI devices) that it fully controls. The hypervisor's job is to manage cells and maintain their isolation from each other, ensuring that cells will not interfere with each other in an unacceptable way. From a hardware perspective, the system bus is still shared, whereas software is jailed in cells and can only access a subset of physical hardware, the one assigned to them. This approach is most useful for virtualizing tasks that require full control over the CPU; examples include realtime control tasks and long-running high-performance computing tasks. .

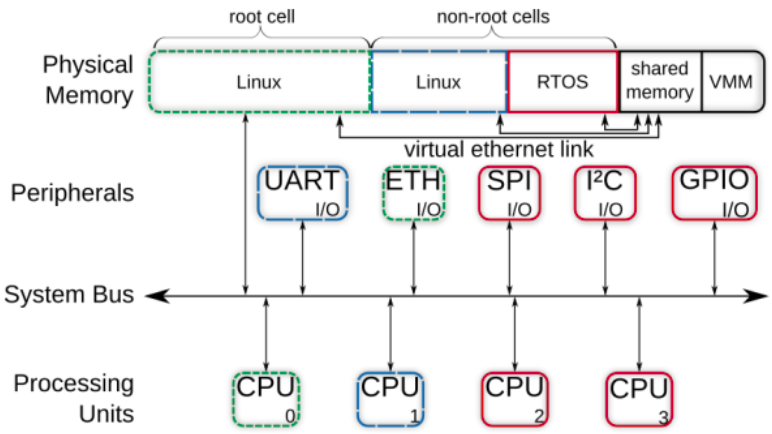


Figure 4.1.1: Concept of Partitioning in Jailhouse [7]

4.2 Phases of Operation

To activate the Jailhouse Hypervisor, it is necessary to build and load a Linux kernel module (jailhouse.ko) containing the necessary firmware. After the execution of the startup code and activation of the hypervisor, Linux continues to run as a virtual machine and guest of Jailhouse, "jailed" in the so-called root cell.

The direct assignment of hardware devices allows Linux to continue operating normally. Except for minimal, optional debug helpers, Jailhouse does not require any specific device drivers. Jailhouse assumes that guests do not need to share hardware resources. To create additional partitions (called non-root cells), Jailhouse reassigns Linux's hardware resources (such as CPU(s), memory, PCI, and MMIO devices) to the new partition. Therefore, Linux releases previously utilized hardware. This includes physical CPUs: the configuration of a partition consists of at least one physical CPU

and a certain amount of memory preloaded by the root cell with a secondary operating system or bare-metal application.

Linux takes certain CPUs offline and invokes the hypervisor to create a new cell by supplying a cell configuration describing the assigned resources. Other resources, such as PCI devices, memory-mapped devices, and I/O ports, can also be reassigned exclusively to the new guest. The hypervisor prevents subsequent cell access to these resources, thereby preventing accidental modifications. Non-root cells are capable of being dynamically created, destroyed (resources are returned to the root cell), and relaunched.

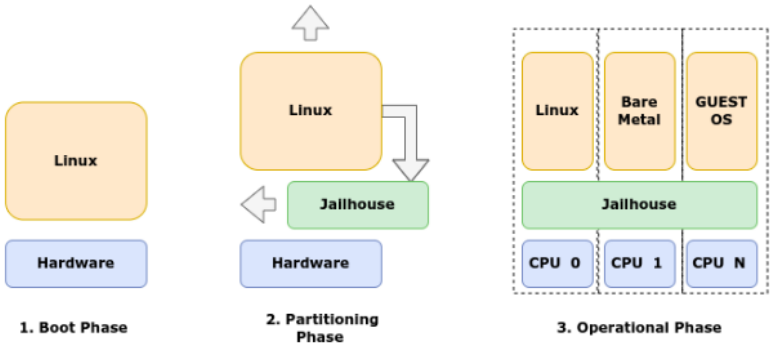


Figure 4.2.1: Phases of Operation

4.3 Enabling Jailhouse

To successfully enable Jailhouse we need to follow the steps decribed below:

- Build and install Linux on our target
- Compile(natively or cross-compilation) and Install Jailhouse on our target
- Provide the necessary prior reserved memory regions to the hypervisor to function properly. These memory reservations must be completed during the Linux boot.
- Load the jailhouse.ko module in our kernel
- Run jailhouse hypervisor

  Those steps will be described thoroughly in Chapter 5

For the third step, the memory region in which Jailhouse operates must be physically continuous. At this moment this can be done by prior reserving memory using the "memmap=" kernel command-line parameter or via device tree overlay. When Jailhouse is enabled, the loader linearly maps this memory into the virtual address space of the kernel. The offset from the base address of the memory region is stored in the page offset field of the header. This makes the conversion from virtual to physical host address (and vice versa) simple.

The fourth step initiates the subsequent operations [49, 50]. The jailhouse user-space program sends the JAILHOUSE_ENABLE request to /dev/jailhouse, which instructs the driver to invoke jailhouse cmd enable() (driver/main.c). In this function, CPU flags are examined by the driver to determine which virtualization technology this CPU employs, after that the configuration binary (signature in the header) is validated. Afterwards, it calls the request firmware() function, which searches the /lib/firmware folder. At the subsequent stage, the driver remaps the memory region reserved in step 3 to the kernel address space memory , allowing user-space access to the hypervisor. Driver copies this binary at the beginning of this memory region, followed by the subsequent cell configuration. The function jailhouse cell create() is then called.

The final step in enabling Jailhouse is initializing the CPU [49]. In brief, the hypervisor launches it by invoking the entry hypervisor() function for each CPU (leading to arch entry in hypervisor/x86/entry.S). Jailhouse must become an interface between cells (root cell with Linux at this early boot time) and CPU cores, thereby saving the system's state and configuring its environment when CPU0 initializes. It consists of configuring paging for the hypervisor and APIC or GIC, creating the Interrupt Description Table (IDT), creating the root cell and remapping Linux memory regions and devices, and configuring Virtual Machine Extensions (VME). It also configures UART communication to write debug information, allowing the proper information to be displayed on serial console. Next steps require the renew of the IDT and Global Descriptor Table (GDT), reset the CR3 register (page table pointer), and configure the Virtual Machine Control Structure (VMCS). Finally, the hypervisor sends a VMLAUNCH instruction, which returns control to Linux. However, Linux no longer runs "on bare metal" but rather in the "root cell" under Jailhouse.

4.4 Cell initialization and start

After the Jailhouse Hypervisor is enabled, different inmate cells can be initialized to run demos. To create a cell to host a demo we run the following command

```
jailhouse cell create <path/to/conf.cell>
```

The binary configuration of the to-be-created cell is read and stored in memory. Later, the JAILHOUSE_CELL_CREATE command is sent to the driver along with the loaded binary's address. The specified binary is copied from the user-space memory to the kernel space, where it is checked. An image for the guest is created and includes pointers to mapped memory for the guest's regions and PCI devices. The driver then leaves information about the new cell in sysfs and detaches Linux requested CPUs, PCI, and any hardware devices requested from the Linux (root) cell.

When the driver issues the JAILHOUSE_HC_CELL_CREATE hypercall, the next stage begins. Hypervisor, upon catching it, it first instructs all new cell processors to suspend except for the current one (which is executing this code).

The cell initialization process then begins. A procedure is invoked to save (already reallocated to the guest) memory-mapped device locations and handlers. Afterward,

after ensuring that no other entity owns any CPUs, all CPUs are claimed. The cell will then be added to the list of cells.

To execute some inmate in the new cell it is needed to move it to the cell's memory region. This is done the following command:

```
jailhouse cell load <name-of-cell> <inmate.bin>
```

Every inmate is treated as a raw binary. This binary's size must be less than or equal to the region of guest memory where it will be loaded. The mechanism for transferring the file into the cell's memory is similar to that of previous instances. The driver transmits JAILHOUSE_HC_CELL_SET_LOADABLE to the hypervisor and remaps loadable guest regions to the root cell address space. The message "Cell name of cell> can be loaded" should be displayed on the serial console at this point. The driver then stores binary at the specified address.

Finally, to start the loaded cell we execute the following command:

```
jailhouse cell start <name-of-cell>
```

It triggers the hypercall JAILHOUSE_HC_CELL_START, which causes the hypervisor's cell start() to unmap all loadable regions from the root cell to the guest. The state of the cell becomes JAILHOUSE_CELL_RUNNING and the cell starts.

Chapter 5

Development

This chapter is devoted to the investigation and development of our system, which consists of a real hardware target with custom Linux installed, the Jailhouse Hypervisor, and a fully operational custom Linux non-root cell. We detail the process of constructing this system, which will henceforth be known as the "Jailhouse ecosystem" and will be used to evaluate the Jailhouse Hypervisor extensively.

5.1 Jailhouse QEMU Demonstration

Prior to installing and running Jailhouse on a real hardware target, familiarizing oneself with the source code and its fundamental capabilities is advisable. Consequently, we chose to run Jailhouse on the QEMU emulator. Since QEMU already emulates any hypervisor-supported hardware device, it is relatively simple to run Jailhouse on "any machine and architecture" we desire.

QEMU Installation

QEMU is available from the repositories of the majority of distributions, but we will compile it from its source code [51].

```
$ wget https://download.qemu.org/qemu-version.tar.xz

$ tar xvJf qemu-version.tar.xz

$ cd qemu-version

$ ./configure --prefix=/path/to/qemu/build/dir --enable-kvm \
> --target-list=x86_64-softmmu

$ make
```

```
$ make install
```

We decide to connect using ssh on port 22223 from the host by executing **ssh -p 22223 root@localhost**. The VM listens on localhost's TCP port 22223 (host).

1. **Use of Jailhouse Images**

   We are going to use the Jailhouse Images [11] to generate a ready-to-use reference image for the Jailhouse Hypervisor. Our virtual target is QEMU and is only for examining the basic commands and functionality of Jailhouse.

   We clone the jailhouse images repository and follow the instructions in README. This repository is comprised of two primary components: build-images and the start-qemu script. To initialize, QEMU requires an image created by the first script. In this instance, the QEMU/KVM Intel-x86 and QEMU aarch64 virtualtarget emulators are chosen for configuring this type of machine. The second script initializes the emulator with the produced image and displays certain commands along with their parameters (found in the history) to run Jailhouse and work with apic-demo cell / gic-demo cell and a (minimal) non-root Linux cell.

2. **No use of Jailhouse Images** Following this method, we create a virtual machine from scratch, including all hardware requirements for the hypervisor. We used a Debian GNU/Linux operating system image. Shown below is the parameters we used for the VMs construction for x86 architecture

   ```
   exec $QEMU -machine q35,kernel_irqchip=split -m 1024 -enable-kvm \
   -smp 4 -device intel-iommu,intremap=on,x-buggy-eim=on \
   -cpu host,-kvm-pv-eoi,-kvm-pv-ipi,-kvm-asyncpf,-kvm-steal-time,-kvmclock \
   -drive file=./private.qcow2,id=disk,if=none \
   -device ide-hd,drive=disk -serial stdio -serial vc \
   -nic user,hostfwd=tcp:127.0.0.1:22223-:22 \
   -device intel-hda,addr=1b.0 -device hda-duplex
   ```

   After the VMs creation we install jailhouse according to the official repository's guide[6]. We should prior reserve contiguous piece of RAM for the hypervisor passing the reservation as parameter to the command line of the virtual machine's kernel.

5.2 Install and Run Jailhouse in real hardware target

After acquiring expertise with running jailhouse on a virtual target, we will attempt to install and run the hypervisor on actual hardware. It is a process significantly more complex but also more appropriate and accurate for testing Jailhouse's functioning and assess many metrics (scheduling latency, interrupt latency, temperature, etc.).

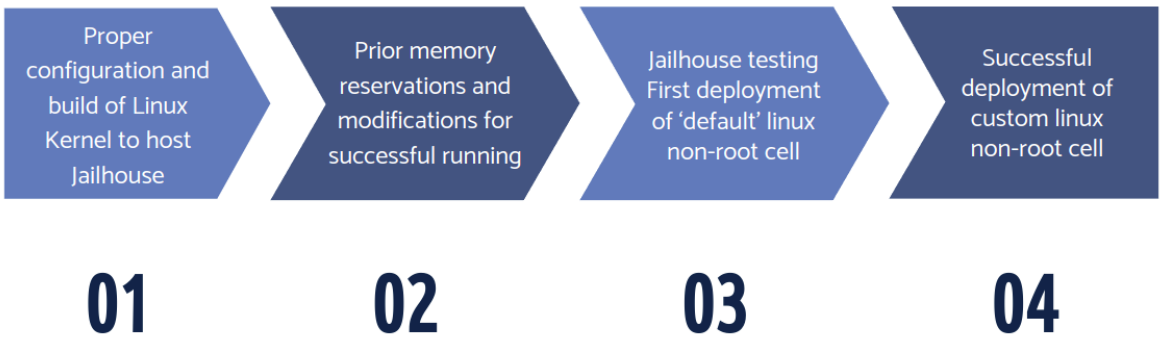


Figure 5.2.1: Steps for proper build and run of our Jailhouse "ecosystem"

5.2.1 Device Selection

After some discussion and comparisons between possible candidates, we concluded that Raspberry Pi 4 Model B is suitable for this study for the following reasons.

- Raspberry Pi 4B uses the 64-bit quad-core ARM Cortex-A72 processor @ 1.5 GHz. Its later model (@ 1.8GHz) has successfully qualified for launching to space has passed stringent total ionizing dose (TID) radiation tests, achieving 100krad resilience.
- Huge processing power in a compact board and withing an affordable price
- Many interfaces (HDMI, Ethernet, onboard Wi-Fi, many GPIOs)
- Large community support, suitable for beginners
- Supports Linux, Python (making it easy to build applications)

Raspberry Pi 4B Main Components

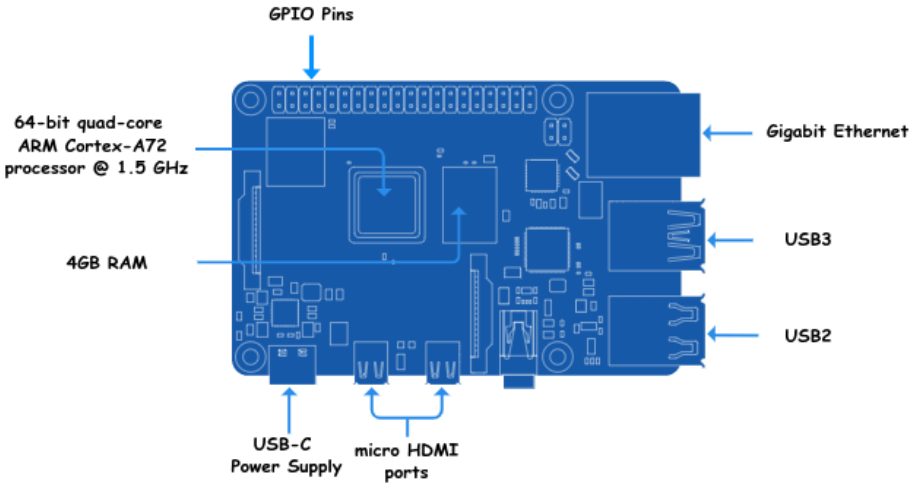


Figure 5.2.2: Raspberry Pi 4B Tech Specs

We will further state the rpi4b specifications

- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- 4GB LPDDR4-3200 SDRAM (depending on model)
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE Gigabit Ethernet
- 2 USB 3.0 ports; 2 USB 2.0 ports.
- Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards)
- 2 × micro-HDMI ports (up to 4kp60 supported)
- Micro-SD card slot for loading operating system and data storage
- 5V DC via USB-C connector (minimum 3A*)
- 5V DC via GPIO header (minimum 3A*)
- Power over Ethernet (PoE) enabled (requires separate PoE HAT)
- Operating temperature: 0 – 50 degrees C ambient

5.3 Jailhouse Installation on Raspberry Pi 4B

This section will describe the steps we took to install and run Jailhouse on a Raspberry Pi 4 board. Although some steps may appear inconsequential, they must be described in detail because even the smallest change can result in errors.

To successfully install and run Jailhouse on rpi4 board we should do the following things:

- Take the linux distribution of our choice
- Clone the Jailhouse repository
- Clone a Linux tree with jailhouse enabling support to build and boot our own custom kernel
- Build and run Jailhouse

Linux Distribution

To first boot our board we choose to port it with **Raspberry Pi OS Lite (64-bit)**, a Debian Bullseye distribution with no desktop environment. An easy way to do this is with the use of *rpi-imager*[52].

Kernel Versions: 5.10, 5.10-rt

5.3.1 Building The Kernel

Jailhouse need to be compiled with kernel objects. Thus we first need a copy of kernel source code and compile it.

Jailhouse requires Kernel Version >=4.7. Here, we'll use Kernel version 5.10.31 and/or 5.10.27-rt provided by Siemens/Jan Kiszka [10].

Cross-compilation on a x86 machine is definitely faster, nevertheless it is more susceptible to mistakes in comparison with native build.

Someone can choose the section that corresponds to their situation; either the native builds or cross-compilation. Although there are many common steps between the two, there are also significant differences.

Compilation on Raspberry Pi 4B Board

First, we install git and build dependencies.

```
$ sudo apt install git bc bison flex libssl-dev make
```

Next we get the sources, which takes quite some time.

```
$ git clone -b jailhouse-enabling/5.10 https://github.com/siemens/linux
                                        or
$ git clone -b jailhouse-enabling/5.10-rt https://github.com/siemens/linux
```

We need to adjust the kernel configuration so we take existing configuration [11] and put it in a file e.g. mini.config

We adjust the configuration file according to our needs. For this study we specifically added the following lines :

```
 # we need to enable support for thermal sensors on our board which is a Broadcom
 BCM2711 SoC
- CONFIG_BCM2711_THERMAL=y

# Depending if we want a linux kernel with or without the real time patch
- CONFIG_PREEMPT=y or CONFIG_PREEMPT\_RT=y

#  It is also very important to include (probably manually) the following
lines so that Docker works properly
 - CONFIG_NAMESPACES=y

 - CONFIG_NET_NS=y

 - CONFIG_PID_NS=y

 - CONFIG_IPC_NS=y
```

```
- CONFIG_UTS_NS=y
- CONFIG_CGROUPS=y
- CONFIG_CGROUP_CPUACCT=y
- CONFIG_CGROUP_DEVICE=y
- CONFIG_CGROUP_FREEZER=y
- CONFIG_CGROUP_SCHED=y
- CONFIG_CPUSETS=y
- CONFIG_MEMCG=y
- CONFIG_KEYS=y
- CONFIG_VETH=y
- CONFIG_BRIDGE=y
- CONFIG_BRIDGE_NETFILTER=y
- CONFIG_NF_NAT_IPV4=y
- CONFIG_IP_NF_FILTER=y
- CONFIG_IP_NF_TARGET_MASQUERADE=y
- CONFIG_NETFILTER_XT_MATCH_ADDRTYPE=y
- CONFIG_NETFILTER_XT_MATCH_CONNTRACK=y
- CONFIG_NETFILTER_XT_MATCH_IPVS=y
- CONFIG_IP_NF_NAT=y
- CONFIG_NF_NAT=y
- CONFIG_NF_NAT_NEEDED=y
- CONFIG_POSIX_MQUEUE=y
```

```
- CONFIG_OVERLAY_FS=y

$ cp mini.config .config

$ make olddefconfig
```

Next we build and install the kernel, modules, and Device Tree blobs. We use the flag **-j4** to split the work between all four cores, speeding up compilation significantly.

```
$ make -j4 Image modules dtbs

$ sudo make modules_install

$ sudo cp arch/arm64/boot/dts/broadcom/*.dtb /boot/

$ sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/

$ sudo cp arch/arm64/boot/Image /boot/Image_5.10
```

Cross Compilation for ARM64 on x86

First, we need a suitable Linux cross-compilation host. We use a x86 host with Ubuntu 18.04.Since Raspberry Pi OS is also a Debian distribution, many aspects, such as command lines, are similar.

To build the sources for cross-compilation, we make sure we have the dependencies needed on your machine

```
$ sudo apt install git bc bison flex libssl-dev make libc6-dev libncurses5-dev
```

We select the desired toolchain e.g. Linaro [53]

We adjust the .config file according to jailhouse configuration and our needs, just as in native build. Finally we execute:

```
$ make ARCH=arm64 olddefconfig
```

and to build the kernel, modules and Device Tree blobs:

```
$ make ARCH=arm64 CROSS_COMPILE=/path/to/gcc-linaro-version-x86_64_aarch64-linux-g
/bin/aarch64-linux-gnu- -j4 Image modules dtbs
```

To install the kernel, modules and dtbs we can either transfer the directory that contains kernel source to Raspberry Pi or mount the compiled kernel-source directory on target board.

```
#On Host,
$ sudo apt-get update && sudo apt-get install -y sshfs
```

```
#On RaspberryPi,
$ sudo apt-get update && apt-get install -y sshfs make gcc

$ mkdir ~/linux-src

$ sshfs <remote user>@<remote ip address>:<linux source path> ~/linux-src

$ cd ~/linux-src

$ make modules_install

$ cp -v arch/arm/boot/Image /boot/Image_5.10

$ sudo cp arch/arm/boot/broadcom/*.dtb /boot/

$ sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/

#now reboot
$ sudo reboot
```

We need to prior reserve memory for the jailhouse hypervisor and we do it via a device tree overlay The necessary code of jailhouse.dts:

Cross-Compilation of Arm Trusted Firmware:

On x86 Host:

```
$ git clone https://github.com/ARM-software/arm-trusted-firmware.git

$ cd arm-trusted-firmware

$ make CROSS_COMPILE=/path/to/gcc-linaro-version-x86_64_aarch64-linux-gnu/
/bin/aarch64-linux-gnu- PLAT=rpi4 DEBUG=1

$ scp ~/arm-trusted-firmware/build/rpi4/debug/bl31.bin pi@[rpi4-ip4]:~/
```

On target:

```
$ sudo mv bl31.bin /boot/

[. . .]

[all]
enable_uart=1
armstub=bl31.bin
enable_gic=1
kernel=Image_5.10.27
```

```
    dtoverlay=aliases
    dtoverlay=jailhouse
```

```
/dts-v1/;
/plugin/;
/ {
        compatible = "brcm,bcm2835";

        fragment@0 {
                target-path = "/";
                __overlay__ {
                        reserved-memory {
                                #address-cells = <2>;
                                #size-cells = <1>;
                                ranges;

                                jailhouse@10000000 {
                                        reg = <0 0x10000000 0x10000000>;
                                        no-map;
                                };
                        };
                };
        };

        fragment@1 {
                target-path = "/scb/pcie@7d500000";
                __overlay__ {
                        linux,pci-domain = <0x00000000>;
                };
        };


fragment@2 {
                target-path = "/";
                __overlay__ {
                        reserved-memory {
                                #address-cells = <2>;
                                #size-cells = <1>;
                                ranges;

                                jailhouselinux@40000000 {
                                        reg = <0 0x40000000 0x40000000>;
                                        no-map;
                                };
                        };
                };
        };
};
```

Listing 5.1: Jailhouse Device Tree Overlay

```
/*
 * Jailhouse, a Linux-based partitioning hypervisor
 *
 * Copyright (c) Siemens AG, 2021
 *
 * Authors:
 *  Jan Kiszka <jan.kiszka@siemens.com>
 *
 * SPDX-License-Identifier: MIT
 */

/dts-v1/;
/plugin/;
/ {
        compatible = "brcm,bcm2835";

        fragment@0 {
                target-path = "/";
                __overlay__ {
                        aliases {
                                /* Needed to enable UART1 for use by TF-A */
                                uart1 = "/soc/serial@7e215040";

                                /* Ensure stable /dev/mmcblk0 assignment */
                                mmc0 = "/emmc2bus/emmc2@7e340000";
                                mmc1 = "/soc/sdhci@7e300000";
                        };
                };
        };
};
```

Listing 5.2: Aliases Device Tree Overlay

5.4 Jailhouse GIC Demo Demonstration

GIC demo [6] (short for Generic Interrupt Controller) is an inmate typically used to demonstrate fundamental Jailhouse features and functions. It is also the ARM equivalent of APIC demo. It is a small program that configures an interrupt for the GIC timer to fire at 10Hz measuring the actual time between events. It represents the disparity between the anticipated and actual duration (called Jitter). The smaller the Jitter, the less impact the Jailhouse Hypervisor has on the performance of the system and applications.

We execute the GIC demo to familiarize ourselves with the fundamental jailhouse Linux commands. Consequently, we first enable the jailhouse hypervisor and then establish the inmate cell that will host gic-demo. Later, we load gic-demo into the rpi4 inmate cell and launch it.

```
jailhouse enable jailhouse/configs/arm64/rpi4.cell
jailhouse cell create jailhouse/configs/arm64/rpi4-inmate-demo.cell
```

```
jailhouse cell load rpi4-inmate-demo jailhouse/inmates/arch/arm64/gic-demo.bin
jailhouse cell start rpi4-inmate-demo
```

5.5 Linux Non-Root Cell Deployment

In this section we will describe the steps we followed to deploy our linux non-root cell.

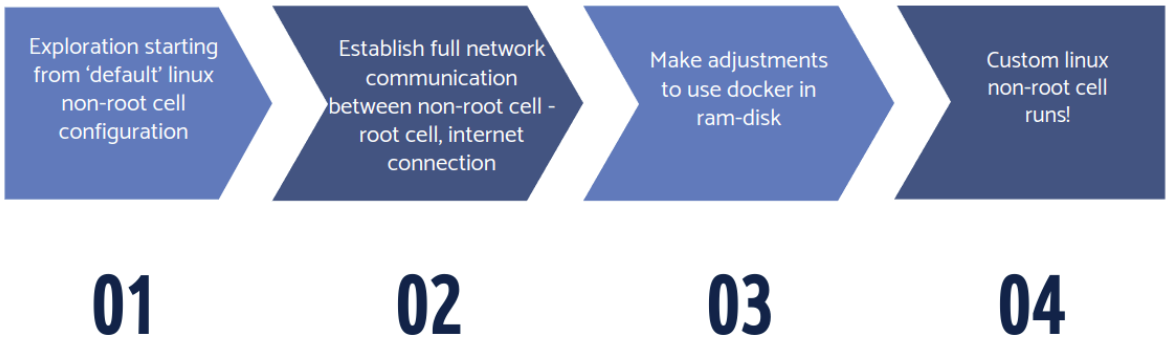


Figure 5.5.1: Steps for proper deployment of custom linux non-root cell

At first we tested the functionality of the non-root cell by loading the kernel and a minimum .cpio file, built inside jailhouse-images[11]. We use the existing configuration for arm64 architecture as a reference to build our custom kernel and image.

We built our custom Linux kernel and image with Buildroot[12]. It is a tool that simplifies and automates the process of constructing a complete and bootable Linux environment for an embedded system, leveraging cross-compilation to enable the construction of various target platforms on a single Linux-based development machine. Buildroot can build the necessary cross-compilation toolchain, create a root file system, compile a Linux kernel image, and generate a boot loader for an embedded system, or it can perform any independent combination of these tasks. For example, an independently installed cross-compilation toolchain can be utilized, whereas Buildroot just constructs the root file system[13].

We chose to work with Buildroot for a couple of reasons:

- It a simple structure that makes it easy to understand and extend. After all it is a set of Makefiles and scripts.
- It is designed with simplicity in mind. To prevent complexity and increased build times, the basic Buildroot tool has been kept as simple as possible. This makes it straightforward to learn and use.
- Buildroot generates a root filesystem image as opposed to a full-on distribution, and that is what we need with Jailhouse.

– Probably for the same reasons, Jailhouse Images use buildroot to create their minimum non-root linux cell.

5.5.1 Design and Build

We download the version of Buildroot we wish to work with. In our case it is the 2022-02-03. We use the configuration file for the rootfile system image generated by Jailhouse-Images as a reference and then we apply the necessary changes according to our preferences.

Kernel

We use the custom git repository we used to build our jailhouse-enabling kernel for Raspberry Pi, along with the same configuration. That way, we have the same kernel version so the evaluation of latencies, later described in Chapter 6 is valid. We build the real-time 5.10.27-rt36 kernel. We make sure that the kernel has the ivshmem-net driver, in order to establish a virtual network connection between linux root and non-root cell.

Root Filesystem Image

Buildroot does not come with a package manager. It is in the tool's philosophy that we build our image with all the packages we need prior deployment. In this context, for the root file system image we choose the necessary packages in order to support our scenarios' execution. We make the following choices:

– Instead of building a .cpio rootfs we build an initial RAM filesystem linked into the linux kernel. The reason behind this is that the final file is about two times smaller (125MB) than .cpio (250MB), which has to be extracted so the actual size is much bigger, and we do not have the privilege of extend memory use. Furthermore, as mentioned in Chapter 4, Jailhouse reserves continuous memory for its components (hardware, cells etc). In the */proc/iomem* file, the available RAM regions, among others, are stated. We observe that there is 1GB of continuous RAM available. This size of memory we reserve, as also described in 5.3.1.

– We select packages related to networking applications and packages such as iputils, ethtool, iptables etc, which help us establish an internet network connection.

– The package of Python language, along with external python modules are essential for our scenarios' execution.

– The whole docker platform (docker-cli, docker-engine, docker-compose, docker-proxy) is essential for our scenarios' execution.

– The rt-test suite contains cyclitest and hackbench, which we will need for our scenarios' execution.

5.5.2 Deployment

After our custom kernel and rootfs Image build, we transfer the files in Raspberry Pi to deploy the linux non-root cell.

We enable the Jailhouse Hypervisor and type the following command:

```
# jailhouse cell linux path/to/rpi4-linux-demo.cell path/to/Image_5.10-rt -d \
  path/to/inmate-rpi4.dtb  -c "console=ttyS0,115200 ip=10.1.1.3"
```

It is important to highlight the preconfigured PCI device in *rpi4-linux-demo.c*. This PCI device is of type JAILHOUSE_SHMEM_PROTO_VETH and because both cells have the ivshmem-net driver, we see that a new ethernet interface becomes available. A virtual ethernet connection has established and we can access the non-root cell via ssh.

5.5.3 Non-root cell internet access establishment

Although both cells can connect and communicate with each other via networking protocols, we essentially have an isolated network between the two cells. To connect the inmate to the internet, the root cell will have to become a network router. A simple solution could be to create a network bridge where we attach in the root-cell the real physical network interface and later the virtual one.

After doing that the inmate should be in the same network and can use DHCP to get a network configuration that will allow internet access just like the root-cell has.

Bridged clients will be on a separate subnet. We execute the script in Listing 5.3 in root-cell and afterwards we just disable and enable again the Jailhouse Hypervisor (basically reboot the system).

Later on, we go to the inmate cell and type

```
# route -n
```

to check the IP routing table.

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0        0.0.0.0         255.0.0.0       U     0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
```

As we can see we need a route to any other destination through our default gateway (we prior established its ip as 10.1.1.1). Thus, we execute the following command.

```
# route add default gw 10.1.1.1 eth0
```

Now, we can see that we can reach any other destination

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.1.1.1        0.0.0.0         UG    0      0        0 eth0
10.0.0.0        0.0.0.0         255.0.0.0       U     0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
```

In addition, we need to fix the */etc/resolv.conf* file. This file contains information that is read by the resolver routines the first time they are invoked by a process. It is basically used to configure the system's Domain Name System (DNS) resolver so that we can hit addresses with their human readable names and not by their IPs. Having said that we delete the existing file, create another one and type **nameserver 8.8.8.8** (Google's DNS Server).

```
#!/usr/bin/env bash

set -e

[ $EUID -ne 0 ] && echo "run as root" >&2 && exit 1

apt update && \
  DEBIAN_FRONTEND=noninteractive apt install -y \
    dnsmasq netfilter-persistent iptables-persistent

# Create and persist iptables rule.
iptables -t nat -A POSTROUTING -o wlan0 -j MASQUERADE
netfilter-persistent save

# Enable ipv4 forwarding.
sed -i'' s/#net.ipv4.ip_forward=1/net.ipv4.ip_forward=1/ /etc/sysctl.conf

# The Ethernet adapter will use a static IP of 10.1.1.1 on this new subnet.
cat <<'EOF' >/etc/network/interfaces.d/eth0
auto eth0
allow-hotplug eth0
iface eth0 inet static
  address 10.1.1.1
  netmask 255.255.255.0
  gateway 10.1.1.1
EOF

# Create a dnsmasq DHCP config at /etc/dnsmasq.d/bridge.conf. The Raspberry Pi
# will act as a DHCP server to the client connected over ethernet.
cat <<'EOF' >/etc/dnsmasq.d/bridge.conf
interface=eth0
bind-interfaces
server=8.8.8.8
domain-needed
bogus-priv
dhcp-range=10.1.1.2,10.1.1.254,12h
EOF

systemctl mask networking.service
```

Listing 5.3: Script to make Raspberry Pi a network bridge taken from [54]

5.5.4 Docker set-up in linux non-root cell

In this subsection, we will describe the Docker configuration required for proper operation. - As described in Chapter ref jailhouse chapter>, the Linux non-root cell boots and operates in RAM. By default, all files generated within a container are stored on a writable container layer, with data written to the filesystem of the host system. In our instance, the "host" actually runs in memory, so building and/or running a Docker container generates an error. Docker supports containers storing files in-memory on the host machine. Such files are not persisted. As we run Docker on Linux, we will use the **tmpfs mount** option to store files in the linux non-root cell's system memory. We

execute the following steps.

1. First, we stop the docker service and create a directory to use as mount point for the tmpfs.

```
# /etc/init.d/S60dockerd stop
# mkdir /mnt/ramndisk
```

2. We mount the tmpfs to our mount point

```
# mount -t tmpfs -o size=512m tmpfs /mnt/ramdisk
```

3. We configure the docker daemon to use the desired directory. We copy all the files from the docker default directory to our mount point.

```
# vim /etc/docker/daemon.json
# cp -r /var/lib/docker/* /mnt/ramdisk/
```

4. We export the DOCKER_RAMDISK variable to make Docker work as root is on a ramdisk.

```
# export DOCKER_RAMDISK=true
```

5. Finally, we start again the docker service.

```
# /etc/init.d/S60dockerd start
```

Now that Docker is properly configured and running, we have constructed our jailhouse ecosystem, which consists of a jailhouse hypervisor running on a hardware platform and a fully customized Linux non-root cell that is operational.

Chapter 6

Evaluation

This chapter is devoted to the experiments conducted to measure the impact of the Jailhouse Hypervisor on application execution, as well as the evaluation of a number of metrics to determine the hypervisor's overall behavior. We focus on measuring and studying the following sections:

1. The boot and recovery time of our Jailhouse ecosystem
2. The impact of Jailhouse Hypervisor on applications' performance
3. The scheduling latency on the cells of the Jailhouse ecosystem

6.1 Boot Time - Recovery

In this section, we will measure the boot time of our Jailhouse ecosystem. Boot time is an important aspect of system performance because users must wait for the device to boot before using it. In cases of failure-recovery, (re)boot time is also crucial. In our case, we have a system with a virtualized layer that can host both the satellite's critical software and the desired applications. It is crucial that

1. we are able to re-configure and essentially re-instantiate the system to host different (real-time) applications
2. in case of a software application's partial failure the system will reboot without affecting the applications' overall performance.

Therefore, having a quick boot time is critical.

We measure the boot time of our Jailhouse ecosystem's from the time we execute the command to enable the Jailhouse hypervisor until the time the command regarding the creation and loading of the Jailhouse linux non-root cell exits. We took 50 measurements boot time and the results are depicted in table 6.1 and figure 6.1.1. The distribution is symmetric.

Mean	Median	Standard Deviation
1.4518s	1.4515s	0.0189s

Table 6.1: Mean, Median, and std. Deviation of Jailhouse ecosystem's boot time

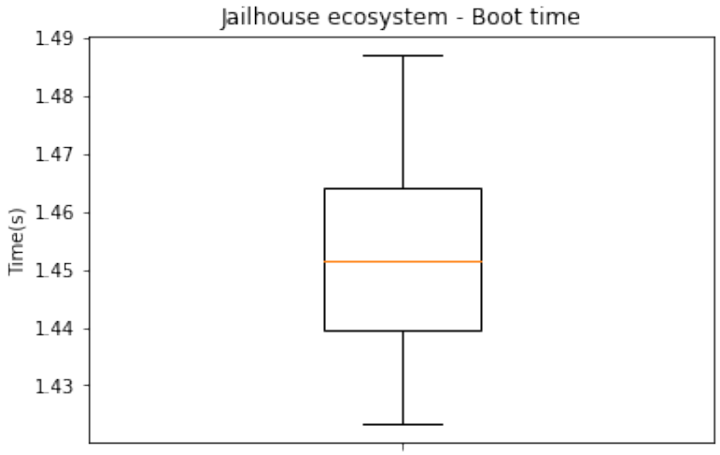


Figure 6.1.1: Box Plot of Jailhouse ecosystem's boot time

There might be cases where the linux non-root cell crashes and, therefore, has to be rebooted. We measure the reboot time of the Jailhouse's linux non-root cell from the time we execute the command to destroy the non-root cell until the time the command regarding the (re)creation and loading of the Jailhouse linux non-root cell exits. We took 50 measurements boot time and the results are depicted in table 6.2 and figure 6.1.2. The distribution is positively skewed and has a wider range than the distribution about Jailhouse ecosystem's boot time.

In our scenario, we intentionally caused a kernel panic, as shown in fig 6.1.5 , in order to freeze the linux non-root cell. Despite the fact that the linux image did not function, the cell appeared to be operational and continued to be in running state. Thus, we cause the non-root cell to reboot without having any interrupt by its state change from running to failed.

Mean	Median	Standard Deviation
1.5072s	1.4697s	0.1306s

Table 6.2: Mean, Median, and std. Deviation of linux non-root cell reboot time

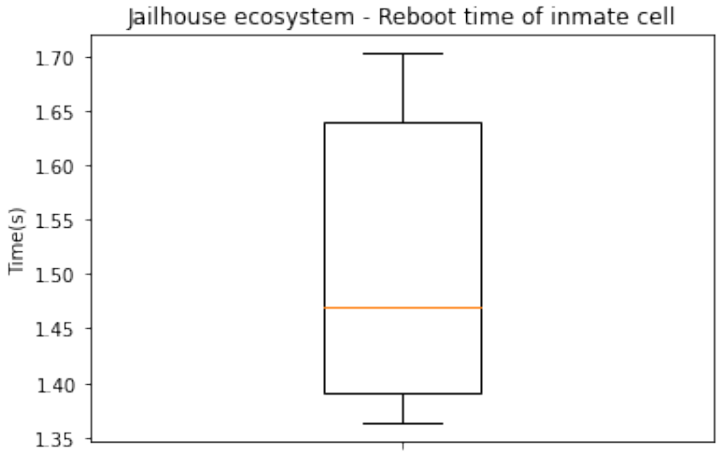


Figure 6.1.2: Box Plot of Jailhouse ecosystem's linux non-root cell reboot time

Finally, we measure the reboot time of our Jailhouse ecosystem's from the time we execute the command to disable the Jailhouse hypervisor until the time the command regarding the creation and loading of the Jailhouse linux non-root cell exits. We took 50 measurements boot time and the results are depicted in table 6.3 and figure 6.1.3. The distribution is symmetric and has a more narrow range than the two previously mentioned distributions.

Mean	**Median**	**Standard Deviation**
1.8419s	1.8232s	0.0753s

Table 6.3: Mean, Median, and std. Deviation of Jailhouse ecosystem's reboot time

Figure 6.1.3: Box Plot of Jailhouse ecosystem's reboot time

Our findings are presented in table 6.4 and figure 6.3.5. If we compare the medians of each box plot, it is evident that there is a difference between the third group (the

reboot time of the Jailhouse ecosystem) and the other two, as the median line of the third group's box plot sits outside the box of the comparison box plots. This is easily explicable. For the last group measurement we used one extra jailhouse command; **jailhouse disable**. This command adds about 0.4s extra latency in the overall (re)boot time compared to the other two groups.

Comparing the first two groups, the number of commands executed is identical. The distinction is on the command types employed. In the first scenario, **jailhouse enable** is executed, whereas in the second scenario, **jailhouse cell destroy** is used. This may account for the disparity between the interquartile ranges and score ranges, as the data reported for the second group are more dispersed and scattered.

Overall, the results indicate a predictable (re)boot time. However, the outlier in the box plot of the Jailhouse ecosystem's reboot time pushes us to seek a bigger supremum for the set of (re)boot times.

System-action	Mean	Median	Standard Deviation
JH ecosystem boot	1.4518s	1.4515s	0.0189s
JH linux non-root cell reboot	1.5072s	1.4697s	0.1306s
JH ecosystem reboot	1.8419s	1.8232s	0.0753s

Table 6.4: Mean, Median, and std. Deviation of Jailhouse ecosystem's (re)boot times

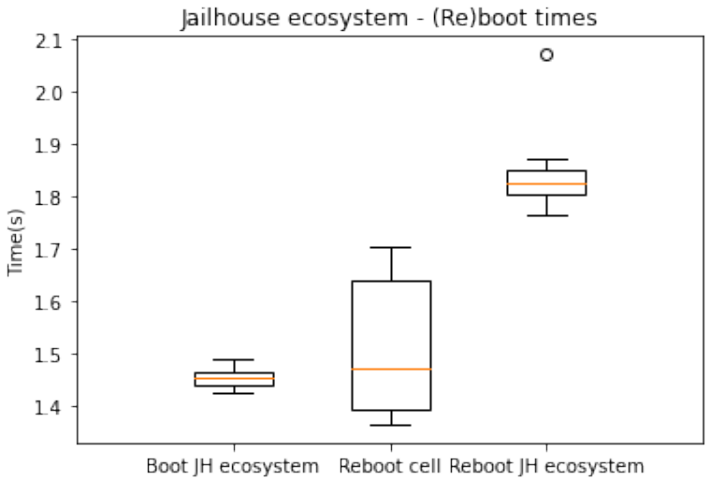


Figure 6.1.4: Box Plot of Jailhouse ecosystem's (re)boot times

```
Welcome to Buildroot
jailhouse login: root
# echo c > /proc/sysrq-trigger
[   55.188291] sysrq: Trigger a crash
[   55.188301] Kernel panic - not syncing:
[   55.188302] sysrq triggered crash
[   55.188305] CPU: 1 PID: 196 Comm: sh Not tainted 5.10.27-rt36 #3
[   55.188312] Hardware name: Jailhouse cell on Raspberry Pi 4 (DT)
[   55.188314] Call trace:
[   55.188316]  dump_backtrace+0x0/0x1b0
[   55.188330]  show_stack+0x18/0x70
[   55.188335]  dump_stack+0xd0/0x12c
[   55.188341]  panic+0xd0/0x364
[   55.188344]  sysrq_handle_crash+0x1c/0x20
[   55.188350]  __handle_sysrq+0x8c/0x1a0
[   55.188353]  write_sysrq_trigger+0x94/0x140
[   55.188357]  proc_reg_write+0xa8/0xf0
[   55.188363]  vfs_write+0xf0/0x2d0
[   55.188367]  ksys_write+0x6c/0x100
[   55.188371]  __arm64_sys_write+0x20/0x30
[   55.188374]  el0_svc_common.constprop.0+0x78/0x1a0
[   55.188380]  do_el0_svc+0x24/0x90
[   55.188384]  el0_svc+0x14/0x20
[   55.188389]  el0_sync_handler+0x1a4/0x1b0
[   55.188393]  el0_sync+0x174/0x180
[   55.292553] SMP: stopping secondary CPUs
[   55.296543] Kernel Offset: disabled
[   55.300079] CPU features: 0x0040022,21002000
[   55.304411] Memory Limit: none
[   55.307510] ---[ end Kernel panic - not syncing: sysrq triggered crash ]---
```

Figure 6.1.5: Intentional crash in linux non-root cell

6.2 Performance Benchmarking: N-Queens Problem

This section will concentrate on performance benchmarking. We will measure the execution time of an application to compare the impact of Jailhouse Hypervisor to that of Docker containers.

The N-queens problem is a generalization of the well-known 8-queens problem, according to which we have to find a way to put 8 queens in a classic 8x8 chessboard so that no queen can attack or be attacked by another. A queen can attack another if they are located on the same diagonal, row or column. In the case of the N-queens, N queens are placed on a NxN board. The number of solutions is known, for example there are 92 solutions for eight queens.

We found a python code on github [14]. It calculates all the solutions using the method of backtracking and has time complexity of $O(N^2)$. It also exploits parallelism, so we can execute the code for different numbers of threads.

6.2.1 Execution in root cell with Linux Kernel 5.10.27-rt36+

Execution in No Jailhouse environment

We execute the N-queens code in Raspberry Pi 4B in a Jailhouse-custom Linux Kernel 5.10.27-rt36+ and measure the execution time and temperature per iteration. The results for $N = 12$, 100 iterations and 1, 2, 4 threads are presented below in Fig.6.2.1.

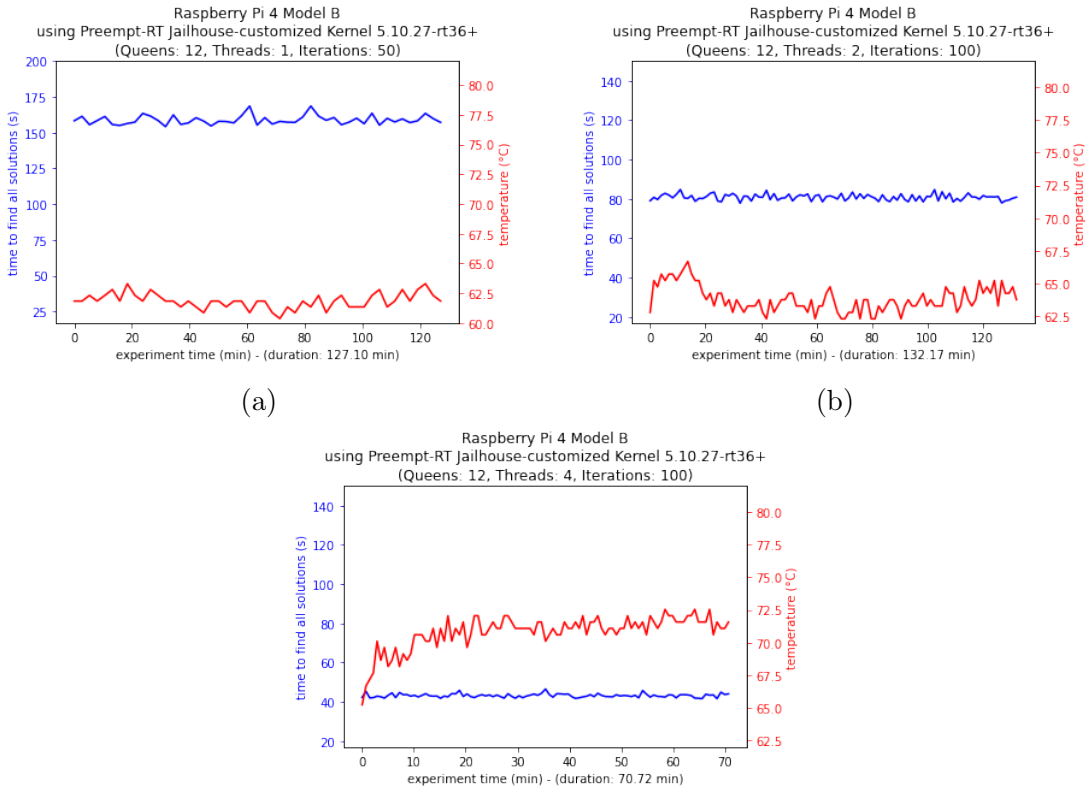


Figure 6.2.1: N-Queens Problem Results for 1, 2 and 4 threads. Kernel Version:5.10.27-rt36+

1. **Execution time**
   Execution Time seems problematic when we compare the results with similar execution scenarios. Specifically, the performance in PREEMPT RT 5.10.27-rt36+ Kernel is two times worse than PREEMPT RT Raspbian Kernel 4.19.59.-rt23-v7l+ in all three cases (1, 2, 4 threads) [55]. Nevertheless, we execute the same python code in the Linux non-root cell to compare the results.

2. **Temperature**
   Temperature rises as the number of threads increases, and mild temperature fluctuations can be observed in each experiment. For proper function and optimal performance, the Raspberry Pi Foundation recommends keeping the Raspberry Pi device's temperature below 85 degrees Celsius [**Linuxhint**]. In our case, the

maximum temperature our Raspberry Pi reaches is around 72.5 degrees Celsius, far below the maximum acceptable value.

Execution in Linux non-root cell

We execute the same python code in the Linux inmate cell we created and deployed. We should highlight although previously mentioned, that the kernel version is the same with the Linux root cell. The average execution times on the three environments are gathered in Table 6.5. The time execution for 1 and 2 threads presents a **73.9%** and **73.1&** increase respectively. This could not possibly caused by an overhead due Jailhouse's presence.

Kernel Version	Threads	Avg Time(s)
4.19.59.-rt23-v7l+	1	67.55
4.19.59.-rt23-v7l+	4	24.27
5.10.27-rt36+ without Jailhouse	1	159.23
5.10.27-rt36+ without Jailhouse	2	80.92
5.10.27-rt36+ without Jailhouse	4	43.23
5.10.27-rt36+ Jailhouse non-root	1	276.96
5.10.27-rt36+ Jailhouse non-root	2	140.57

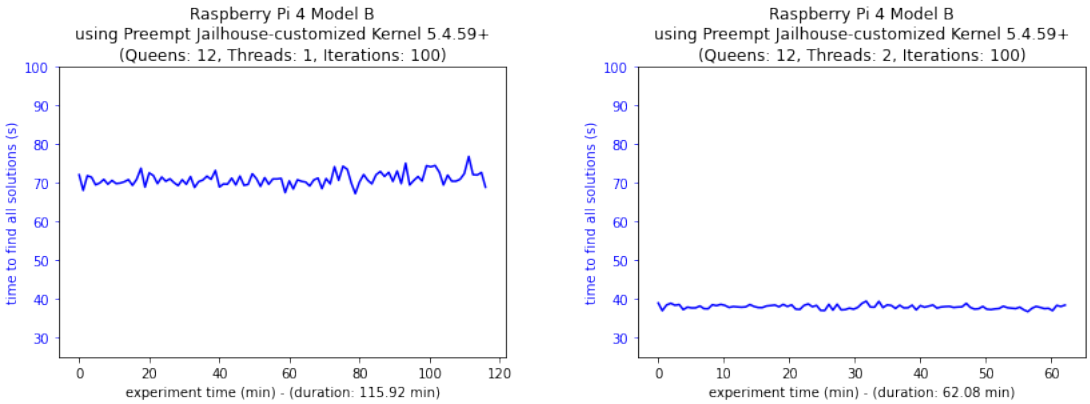
Table 6.5: N-queens benchmark: Average Execution Times in different environments

6.2.2 Execution in Linux Root-cell with Kernel 5.4.59+

We decide to choose another kernel version for our Linux root-cell. Specifically we build and port Jailhouse-custom Linux 5.4.59+ kernel. We repeat the previous experiments in our Jailhouse ecosystem. We run the N-queens problem code both in native environment and within a Docker container.

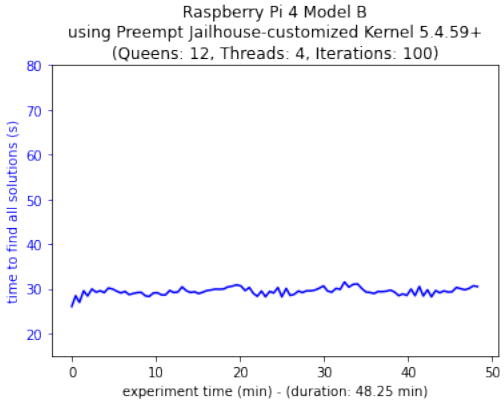
Native Execution

Again, we execute the N-queens code in Jailhouse-custom Linux Kernel 5.4.59+, focusing on execution time. The results for $N = 12$, 100 iterations and 1, 2, 4 threads are presented below in Figure 6.2.2. To have a complete evaluation, we measure the temperature as well, in the 4-threads execution. The results are depicted separetely in Figure 6.2.3.



(a)

(b)



(c)

Figure 6.2.2: N-Queens Problem Results for 1, 2 and 4 threads. Kernel Version:5.4.59+

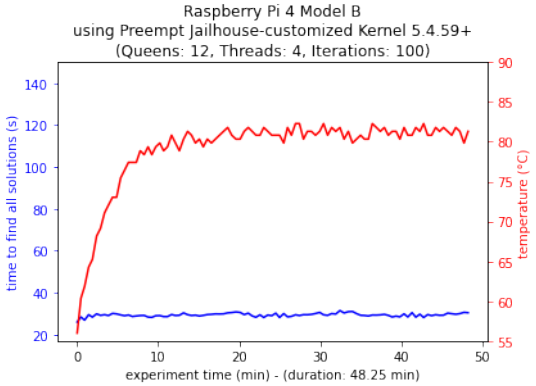


Figure 6.2.3: N-Queens Problem Result for 4 threads. Kernel Version:5.4.59+. Temperature along with execution time

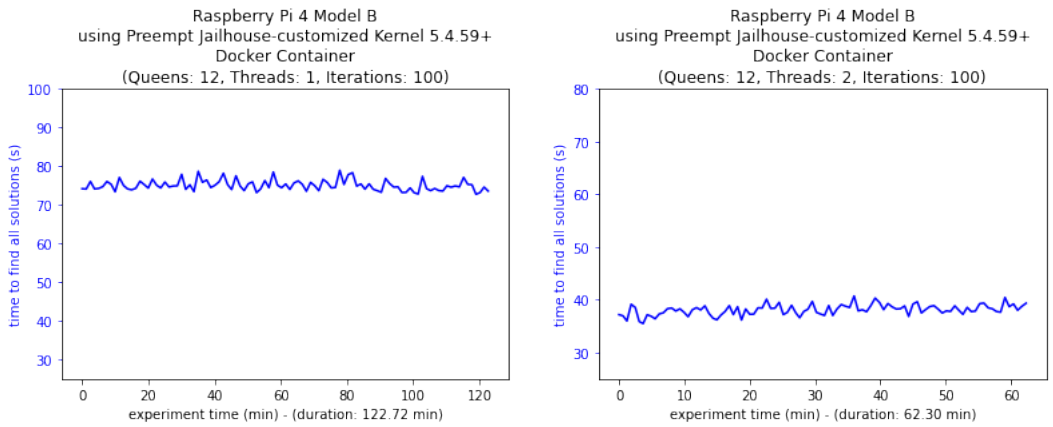
1. **Temperature**
   Temperature rises as the number of repetitions increases for the first 10 iterations.

After that, slight changes in temperature can be seen. Our Raspberry Pi reaches a maximum temperature of 82.29 degrees Celsius, which is not too far below the maximum permissible number. The Raspberry Pi's performance will undoubtedly begin to degrade as the temperature rises. In other words, a thermal throttling condition is present.

2. **Exexution Time**
   The execution time appears normal when compared to similar execution scenarios. In particular, the performance of N-queens code running on the PREEMPT 5.4.59+ Kernel is comparable to that of the PREEMPT RT Raspbian Kernel 4.19.59.-rt23.v7l+. As a result, we choose to conduct all of our experiments using the PREEMPT 5.4.59+ Kernel as the Linux root cell.

Execution in Docker container

We build and run a docker container using a python-alpine image. Our key concern is that the size of the final image is as minimal as possible so that it may be utilized in the Linux non-root cell, where memory is limited. The results for $N = 12$, 100 iterations and 1, 2 threads are presented below in Figure 6.2.4.



(a) (b)

Figure 6.2.4: N-Queens Problem Results for 1 and 2 threads. Inside Docker Container in Linux root cell. Kernel Version:5.4.59+

Our results are summarized in Table 6.6. For the 1-thread execution, the average time within a Docker container is 5.8% longer than the native environment, while for the 2-thread execution, the increase is 0.3%.

Environment	Threads	Avg Time
Natively	1	69.55s
Docker	1	73.63s
Natively	2	37.25s
Docker	2	37.38s

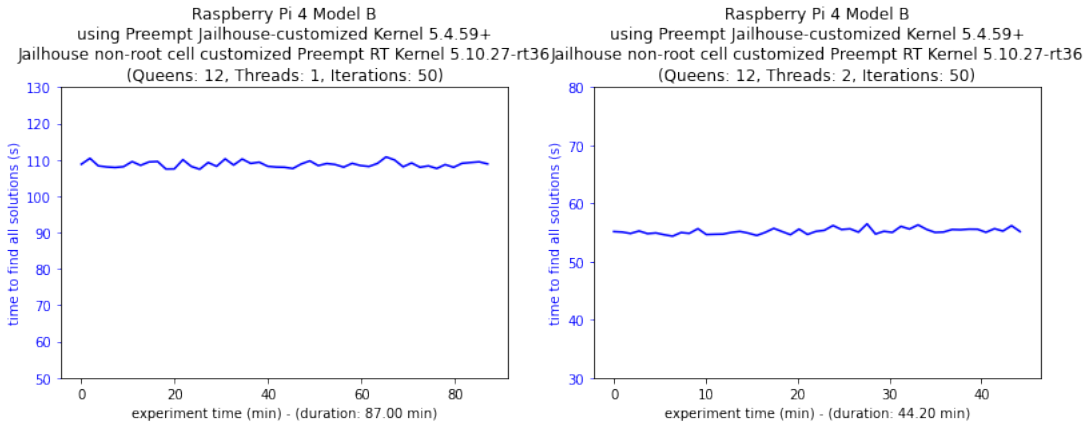
Table 6.6: Summarized results for N-queens benchmark: Jailhouse Linux Root-cell

6.2.3 Execution in Non-Root cell

We run the N-queen benchmark in the Linux inmate cell we created and deployed. We should highlight although previously mentioned, that the kernel version is 5.10.27-rt36.

Native Execution

We first execute the python code directly on the Linux non-root cell without any presence of additional layers. The results for $N = 12$, 50 iterations and 1, 2 threads are presented below in Figure 6.2.5.

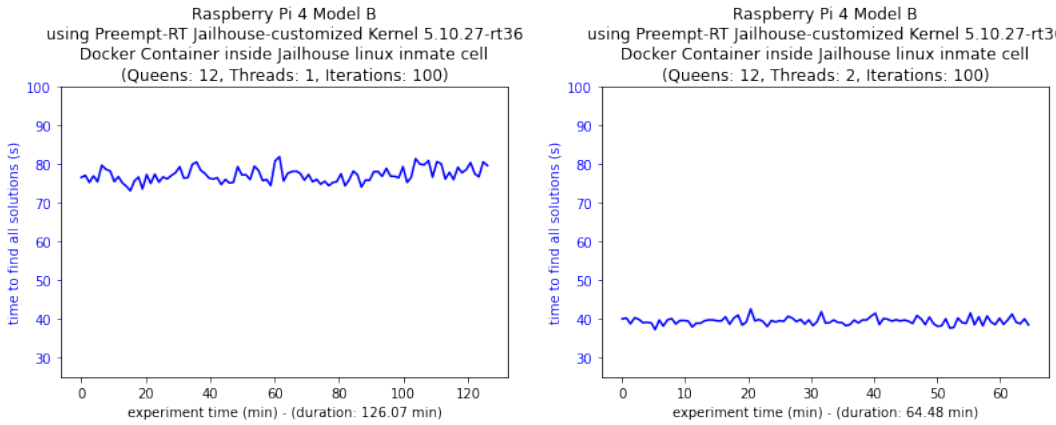


(a) (b)

Figure 6.2.5: N-Queens Problem Results for 1 and 2 threads. Inside Linux non-root cell. Kernel Version: root cell:5.4.59+, non-root cell:5.10.27-rt36

Execution in Docker Container

We build and run a docker container using the same python-alpine image for the execution in the Linux root cell. The results for $N = 12$, 100 iterations and 1, 2 threads are presented below in Figure 6.2.6.



(a) (b)

Figure 6.2.6: N-Queens Problem Results for 1 and 2 threads. Inside Docker Container in Linux non-root cell. Kernel Version: root cell:5.4.59+, non-root cell:5.10.27-rt36

The summary of our observations is shown in Table 6.7. Native execution in a non-root Linux cell for 1 and 2 threads takes an average of 69.55 and 37.25 seconds, respectively. Comparing these findings to Table 6.6 reveals a 50.1% and 42.3% increase in execution time, accordingly. Similarly, the execution time of a docker container increases by 2.7% and 3.5% for one and two threads, respectively.

Environment	Threads	Avg Time
Natively	1	104.40s
Docker	1	75.64s
Natively	2	53.04s
Docker	2	38.08s

Table 6.7: Summarized results for N-queens benchmark: Jailhouse Linux Non-Root cell

Regarding native execution directly on a non-root Linux cell, the influence of the Jailhouse Hypervisor is insufficient to justify the time increase. In addition, the iterations' durations measured are significantly longer than those for the execution within a docker container. This indicates that the issue is related to the distribution of operating systems in the root and non-root cells. We should mention that the kernels and images we use have been customized by the jailhouse hypervisor's contributors. In addition, the 5.10 kernel is configured and customized for arm architecture targets in general, whereas the 5.4 kernel is customized particularly for raspberry pi, which likely explains the enormous discrepancy in execution times that we discovered before.

If we overlook the unexpected performance degradation in native execution and focus on results from execution in docker containers, the minor increase in execution time demonstrates that the jailhouse hypervisor has a negligible effect on the performance

of software applications. In fact, the extremely small fluctuations depicted in Figure 6.2.6 demonstrate the predictable behavior of our ecosystem.

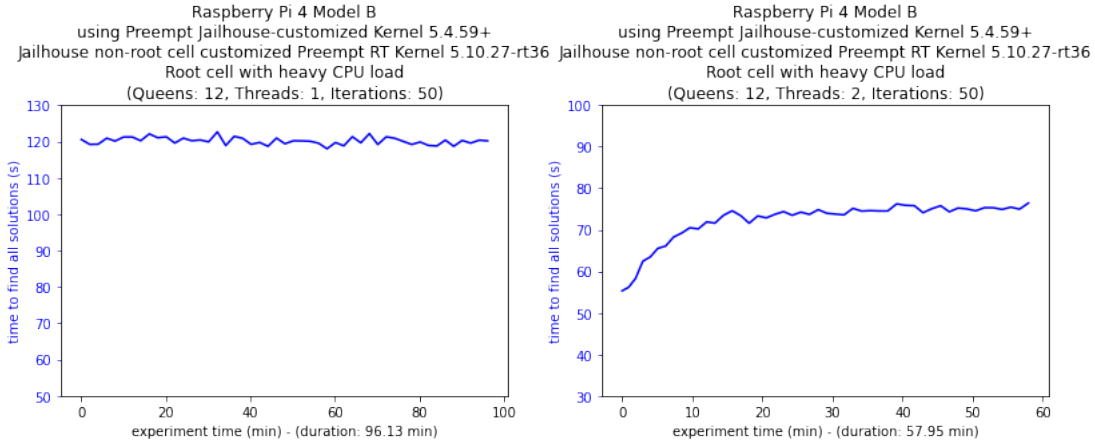
6.2.4 Execution in Non-Root cell - Additional CPU Load

As of now, we have executed the N-queens code in both root and non-root cell, evaluating the impact of the jailhouse hypervisor on application performance. Apart from the execution of the N-queens code, the system did not have any "significant" tasks to complete, hence this scenario could not be considered realistic. In fact, the system is intended to execute multiple programs concurrently, typically in separate cells. These applications may or may not be constrained by real-time limitations and consist of user software, or mission-critical avionics software.

Therefore, we will repeat the experiment by executing the N-queens code in a Linux non-root cell and simulating significant CPU load in a Linux root cell. The process of the load's creation is further examined in Section 6.3.

Native Execution

We execute the python code again directly on the Linux non-root cell without any presence of additional layers. The results for $N = 12$, 50 iterations and 1, 2 threads are presented below in Figure 6.2.7

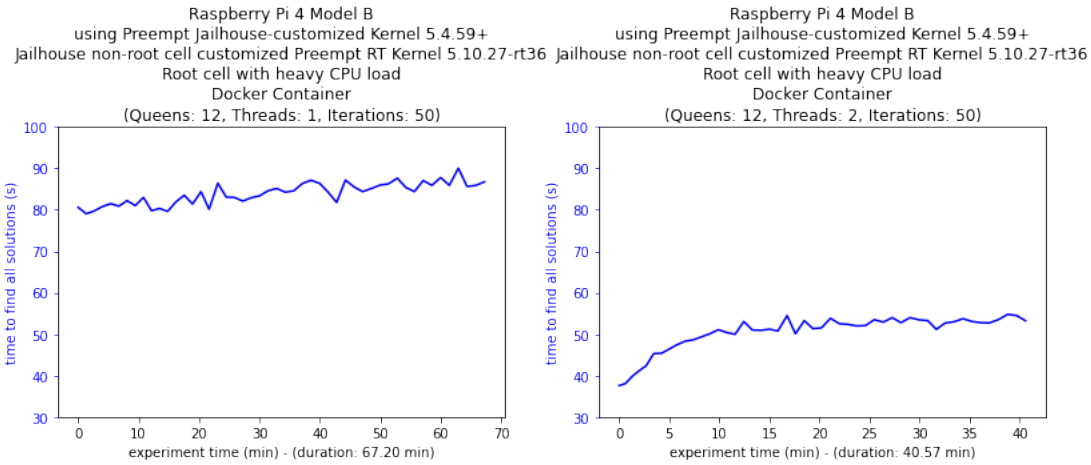


(a) (b)

Figure 6.2.7: N-Queens Problem Results for 1 and 2 threads. Inside Linux non-root cell. Heavy cpu load in root cell. Kernel Version: root cell:5.4.59+, non-root cell:5.10.27-rt36

Execution in Docker Container

We build and run the same docker container as used before, for the execution in the Linux root cell. The results for $N = 12$, 50 iterations and 1, 2 threads are presented below in Figure 6.2.8.



(a) (b)

Figure 6.2.8: N-Queens Problem Results for 1 and 2 threads. Inside Docker Container in Linux non-root cell. Heavy cpu load in root cell. Kernel Version: root cell:5.4.59+, non-root cell:5.10.27-rt36

The summary of our observations is shown in Table 6.8. Native execution in a non-root Linux cell for 1 and 2 threads takes an average of 115.40 and 69.54 seconds, respectively. Comparing these findings to Table 6.7 reveals a 10.5% and 33.1% increase in execution time, accordingly. Similarly, the execution time of a docker container increases by 6.6% and 25.8% for one and two threads, respectively.

Environment	**Threads**	**Avg Time**
Natively	1	115.40s
Docker	1	80.64s
Natively	2	69.54s
Docker	2	48.68s

Table 6.8: Summarized results for N-queens benchmark: Jailhouse Linux Non-Root cell with heavy CPU Load

We note that the percentage increase in average execution time from one thread to two threads is significant in both execution environments. This result is to be expected, as the benchmark we are running demands numerous memory accesses (reads and writes). Given our limited cache resources, it is certain that we will experience enough cache misses that require a further search in RAM. However, our Linux non-root cell has been statically assigned its resources and cannot request more. With limited resources and RAM allocated for the Linux non-root cell, it is inevitable that several page faults will occur, causing the program to look for the page in storage. Linux's **perf** application confirms the program's high number of pagefaults.

The bus is shared by the cells in our Jailhouse ecosystem, as previously stated. All of these cache misses and page faults in combination with the heavy CPU Load in the

Linux root-cell, which heavily utilizes the bus, result in increased traffic (which rises with the number of threads) and hence significant delays.

Our investigation confirms the disadvantage of the Jailhouse Hypervisor in the case of higher bus traffic and, consequently, the negative influence on the performance of applications.

6.3 Scheduling Latency on Real Time: Cyclictest

Our next step on Jailhouse's ecosystem evaluation is to check its real time performance. Real-time tasks are typically triggered by external events or the periodic expiration of a timer. Real-time jobs with the highest priority should be scheduled immediately after activation, however in reality, there is a delay between the moment the activating event takes place and the moment the task begins executing[15]. This delay, known as scheduling latency, impacts the reaction times of all jobs and imposes a lower bound on the system's ability to meet deadlines. Therefore, scheduling latency must be considered when deciding if a system can give the appropriate temporal assurances. For an empirical evaluation of scheduling delay, we employ cyclictest, a tracing program that treats the kernel as a black box and reports scheduling latency directly.

As stated in its documentation, cyclictest[16] accurately and repeatedly measures the difference between a thread's intended wake-up time and the time at which it actually wakes up in order to provide statistics about the system's latencies. It can measure latencies in real-time systems caused by the hardware, the firmware, and the operating system.

Our test suite is based on the following command

```
sudo cyclictest -l100000000 -m -S -p90 -i200 -h400 -q  output.txt
```

Specifically, we use the following options:

- l100000000: 100M iterations (about 6.5 hours);
- m: lock current and future memory allocations to prevent Cyclictest pages from being paged out of memory
- S: Standard SMP testing: options -a -t -n [56] and same priority of all threads (Raspberry Pi has 4 Cores then 4 Threads)
- p90: priority of highest prio thread set to 90 (for the 4 threads, then: 90 89 88 87)
- i200: interval for the first thread (in us).
- h400: dump histogram for max latency (up to 400us).
- q: print a summary only on exit.

6.3.1 Execution with no system Load

Cyclictest - Non RT Kernel No system Load

Despite the fact that cyclictest is a real-time latency benchmark, we will attempt to execute it on the Linux root cell kernel, i.e. PREEMPT 5.4.59+, which does not contain Real Time Linux Patch. We anticipate that the results will not be satisfactory, yet we conduct the study for comparison and thoroughness.

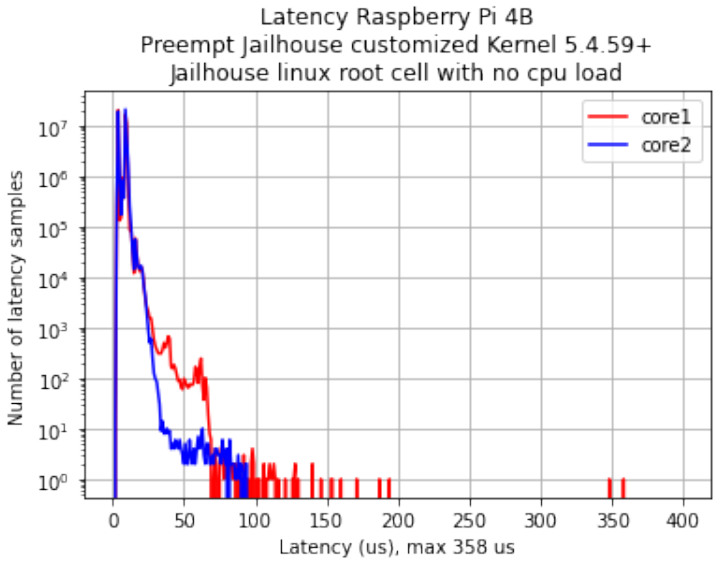


Figure 6.3.1: Cyclictest execution in Linux root cell without any additional load

The generated histogram is depicted in Figure 6.3.1. We see a significant number of latency samples that exceed 100us, with the maximum latency response reaching 358us. This is the best-case situation, as no additional system stress has been simulated. During a standard test, Cyclictest's overhead is low, therefore it does not impose a significant stress on the system. In all other circumstances, even with minimal interference, it is possible to detect an overrun above 400us or even 1000us.

Cyclictest - RT Kernel - No system Load

This time, the procedure is repeated for Linux non-root cells with a real-time kernel, version 5.10.27-rt36. We anticipate that the results will reveal the system's real-time behavior.

Figure 6.3.2 shows the resulting histogram. Our histogram illustrates that response latency is deterministic, as all latency samples fall inside a predetermined range. The greatest delay response, as well as the range's upper bound, is 85us. However, this is hardly the worst-case scenario because the system has not been subjected to any stress-load.

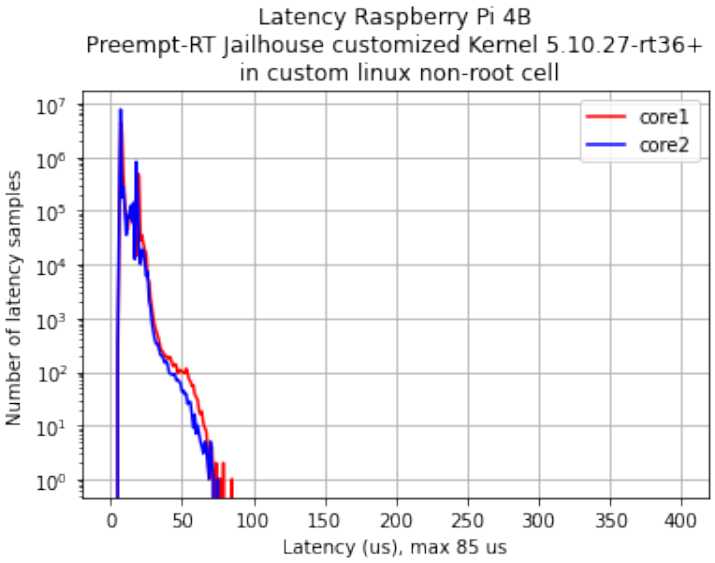


Figure 6.3.2: Cyclictest execution in Linux non-root cell without any additional load

6.3.2 Execution with heavy CPU Load

To obtain the most precise latency measurements, Cyclictest should be run while the assessed system is executing a load that is as comparable as possible to the real time application for which the system is intended. The easiest technique to estimate the latencies that an real time application would encounter on a particular system is to run the actual RT application alongside Cyclictest and any other non-RT applications that would usually be running concurrently. This approach has some drawbacks. For instance, if programs produce system latencies infrequently, it could take Cyclictest a very long period to detect them [57].

A second option is to simulate the load produced by the final application. In lieu of accurately replicating the load that an application would apply to a system, as it might be very complex and difficult, it is common practice to conduct a simulation that stresses the system more than the application is expected to stress it.

We will simulate CPU load by using

1. Hackbench

2. Continuously read and write large amount of random data in a file

Hackbench [58] which is included in the rt-tests package, is a benchmark and stress test for the kernel scheduler. Hackbench stresses the memory subsystem by repeatedly creating and destroying threads, as well as interprocess communication. This stress test can be combined with Cyclictest, which evaluates scheduling latencies, to simulate system load.

Load in Non-Root Cell

The experiment was repeated by simulating the aforementioned heavy load on a Linux non-root cell. Figure 6.3.3 displays the resulting histogram. We see that the latency samples indicate the same real-time behavior. The maximum response latency is 139 us. This execution is 0.63 times slower than an equivalent run without simulated system stress. Nevertheless, this still remains a dystopian if not a realistic scenario

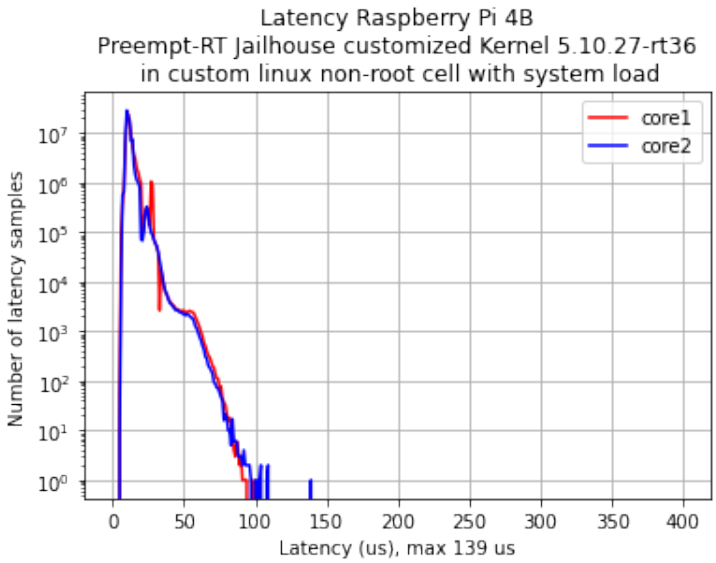


Figure 6.3.3: Cyclictest execution in Linux non-root cell - heavy CPU load in non-root cell

Load in both cells

The cyclictest execution is repeated, this time simulating a heavy load on both cells simultaneously. The results are depicted in Figure 6.3.4. We observe that the sample latencies retain the same real-time behavior that is deterministic. The maximum response latency is 131us, which falls within the range of acceptable latency. To be thorough, we also ran the cyclictest application on the root cell's non-realtime kernel. There were multiple overflows, 0.04 percent for the first thread and 0.01 percent for the second, and the maximum response delay reached 19757us, or 0.01 seconds.

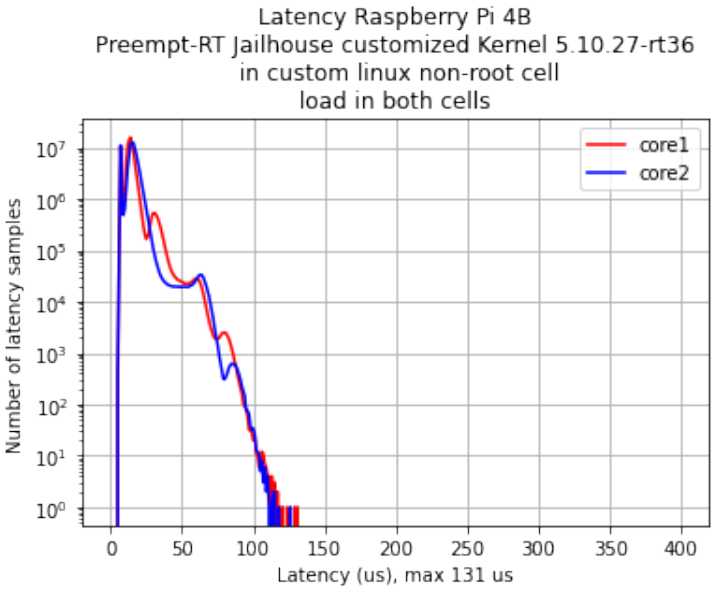


Figure 6.3.4: Cyclictest execution in Linux non-root cell - heavy CPU load in both cells

Average system load

Finally, we want to simulate a medium-sized load to determine what would happen in a typical scenario. This time, by executing cyclictest on the non-root cell we create another inmate cell and load it with the bare-metal gic-demo. The sample latencies maintain the same real-time deterministic behavior, whereas the maximum response delay is 85us, which falls back within the predefined latency range.

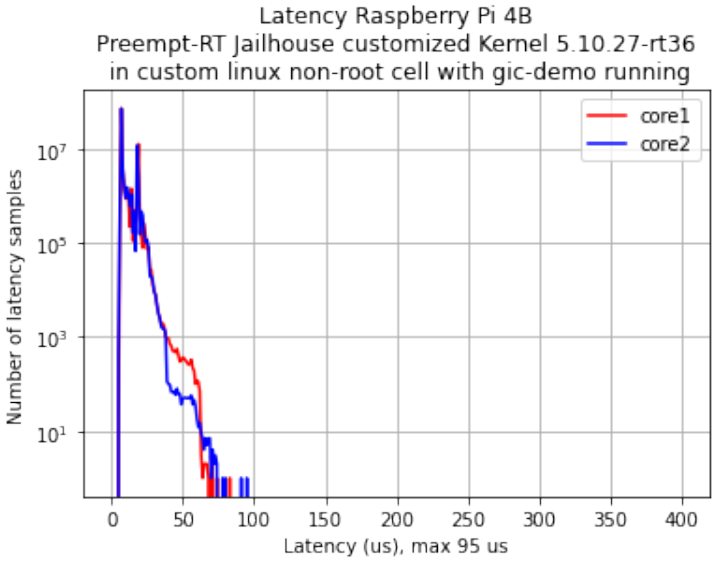


Figure 6.3.5: Cyclictest execution in Linux non-root cell - average system load

Chapter 7

Conclusions & Future Work

In this Diploma Thesis, we explored two virtualization techniques, diving into the Jailhouse Hypervisor. We constructed and configured a fully functional Jailhouse ecosystem to evaluate the effect of the Jailhouse hypervisor on it and partially compare it to that of Docker containers. The process of correctly configuring and establishing the Jailhouse ecosystem was complex and time-consuming. On the one hand, the nature of the process is such that, based on the selected integrated system, a number of its steps change and adapt to the respective platform. On the other hand, there are no centralized instructions and guidelines for the platform we chose, so we had to conduct extensive research, including posting questions on the Jailhouse Hypervisor list.

The study of the Jailhouse Hypervisor on the Raspberry Pi led us to the conclusion that Docker containers add a small amount of overhead to the system and can be combined with Jailhouse to provide an isolated and secure environment with the flexibility offered by the containerization technique. Regarding the effect on real-time performance, the results were quite satisfactory in terms of observed deterministic behavior. Peak latencies could be reduced, but this is due to the Raspberry Pi, which is not favored for the development of mission-critical real-time systems. In addition, we confirmed Jailhouse's strength, isolation, which did not pose a problem in any execution scenario, as well as its weakness, the decrease in application performance when there is communication between cells or an increase in system bus traffic.

A potential future direction for this thesis could be a more extensive and comprehensive evaluation of real-time performance, for instance by measuring interrupt latency using external devices and tools. A further issue that must be resolved is the limited memory available to us, so that we can run software such as artificial intelligence and/or machine learning, which frequently require a great deal of available resources to manage the data. In addition, given that Jailhouse is already gaining popularity in the space community, it would be beneficial to begin investigating the possibility of combining Jailhouse with Docker containers and testing it onboard a satellite.

Bibliography

[1] Marcello Cinque et al. "Virtualizing mixed-criticality systems: A survey on industrial trends and issues". In: *Future Generation Computer Systems* 129 (Apr. 2022), pp. 315–330. DOI: 10.1016/j.future.2021.12.002.

[2] *Selene Project*. URL: https://www.selene-project.eu/open-platform/.

[3] Carles Hernàndez et al. "SELENE: Self-Monitored Dependable Platform for High-Performance Safety-Critical Systems". In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 2020, pp. 370–377. DOI: 10.1109/DSD51259.2020.00066.

[4] Rogier Dittner and David Rule. "An Introduction to Virtualization". In: Dec. 2007, pp. 1–35. ISBN: 9781597492171. DOI: 10.1016/B978-1-59749-217-1.00001-0.

[5] Ulf Kulau et al. *The Impact of Open Standards on Next Generation Data Handling Systems*. URL: https://indico.esa.int/event/335/attachments/3807/5348/MOSAIC.pdf. White Paper. 2020.

[6] *JAILHOUSE*. URL: https://github.com/siemens/jailhouse.

[7] Ralf Ramsauer et al. "Look Mum, no VM Exits! (Almost)". In: *ArXiv* abs/1705.06932 (2017).

[8] *Jailhouse Mailing list*. URL: https://groups.google.com/g/jailhouse-dev.

[9] Robert P. Goldberg. "Architectural Principles for Virtual Computer Systems". In: 1973.

[10] *Linux Kernel*. URL: https://github.com/siemens/linux.

[11] *Jailhouse Images*. URL: https://github.com/siemens/jailhouse-images.

[12] *Buildroot*. URL: https://buildroot.org/.

[13] Alexander Sirotkin. “Roll Your Own Embedded Linux System with Buildroot”. In: *Linux Journal* (2011).

[14] *rPI-Tests*. URL: https://github.com/lemariva/rPI-Tests.

[15] Felipe Cerqueira and Björn B. Brandenburg. “A Comparison of Scheduling Latency in Linux, PREEMPT-RT, and LITMUS RT”. In: 2013.

[16] *Cyclictest*. URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest

[17] *Fault detection, isolation and recovery*. URL: https://www.esa.int/Space_Safety/Hera/Fault_detection_isolation_and_recovery.

[18] M. Masmano et al. “Xtratum: a hypervisor for safety critical embedded systems”. In: *In: Proceedings of the 11th Real-Time Linux Workshop*. 2009.

[19] A. Crespo, I. Ripoll, and M. Masmano. “Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach”. In: *2010 European Dependable Computing Conference*. 2010, pp. 67–72. DOI: 10.1109/EDCC.2010.18.

[20] *XtratuM Hypervisor*. URL: https://fentiss.com/products/hypervisor/.

[21] Nils-Johan Wessman et al. “De-RISC: the First RISC-V Space-Grade Platform for Safety-Critical Systems”. In: *2021 IEEE Space Computing Conference (SCC)*. 2021, pp. 17–26. DOI: 10.1109/SCC49971.2021.00010.

[22] Robert Kaiser and Stephan Wagner. “Evolution of the PikeOS Microkernel”. In: Feb. 2007.

[23] *PikeOS Product Overview*. URL: https://www.sysgo.com/fileadmin/user_upload/data/flyers_brochures/SYSGO_PikeOS_Product_Overview.pdf.

[24] Ye Li, Richard West, and Eric Missimer. “A Virtualized Separation Kernel for Mixed Criticality Systems”. In: *SIGPLAN Not.* 49.7 (Mar. 2014), pp. 201–212. ISSN: 0362-1340. DOI: 10.1145/2674025.2576206. URL:

[25] Craig Einstein, Prof. Richard West, and Bandan Das. “A partitioning hypervisor for latency-sensitive workloads”. In: *Red Hat Research Quarterly* 1 (3 2019), p. 34.

[26] José Martins et al. “Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems”. In: *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Ed. by Marko Bertogna and Federico Terraneo. Vol. 77. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 3:1–3:14. ISBN: 978-3-95977-136-8. DOI: 10.4230/OASIcs.NG-RES.2020.3.

[27] *Bao - a lightweight static partitioning hypervisor*. URL: https://github.com/bao-project/baohypervisor.

[28] Luca Abeni and Dario Faggioli. "Using Xen and KVM as real-time hypervisors". In: *J. Syst. Archit.* 106 (2020), p. 101709.

[29] Sisu Xi et al. "RT-Open Stack: CPU Resource Management for Real-Time Cloud Computing". In: *2015 IEEE 8th International Conference on Cloud Computing*. 2015, pp. 179–186. DOI: 10.1109/CLOUD.2015.33.

[30] Paolo Bonzini. "Realtime KVM". In: *LWN.net* (2015).

[31] *The Xen Project wiki*. URL: https://wiki.xenproject.org/wiki/Main_Page.

[32] Sisu Xi et al. "RT-Xen: Towards Real-Time Hypervisor Scheduling in Xen". In: *Proceedings of the Ninth ACM International Conference on Embedded Software*. EMSOFT '11. Taipei, Taiwan: Association for Computing Machinery, 2011, pp. 39–48. ISBN: 9781450307147. DOI: 10.1145/2038642.2038651.

[33] P. McKenney. "A realtime preemption overview". In: *LWN.net* (2055).

[34] Jan Kiszka. "Towards Linux as a Real-Time Hypervisor". In: (Jan. 2009).

[35] Tommaso Cucinotta, Gaetano Anastasi, and Luca Abeni. "Respecting Temporal Constraints in Virtualised Services". In: *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 02*. COMPSAC '09. USA: IEEE Computer Society, 2009, pp. 73–78. ISBN: 9780769537269. DOI: 10.1109/COMPSAC.2009.118. URL:

[36] Tommaso Cucinotta et al. "Providing Performance Guarantees to Virtual Machines Using Real-Time Scheduling". In: *Euro-Par 2010 Parallel Processing Workshops*. Ed. by Mario R. Guarracino et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 657–664. ISBN: 978-3-642-21878-1.

[37] Jun Zhang et al. "Performance analysis towards a KVM-Based embedded real-time virtualization architecture". In: *5th International Conference on Computer Sciences and Convergence Information Technology* (2010), pp. 421–426.

[38] *TRUSTZONE FOR CORTEX-M*. URL: https://www.arm.com/technologies/trustzone-for-cortex-m/.

[39] Oliver Schwarz, Christian Gehrmann, and Viktor Do. "Affordable Separation on Embedded Platforms". In: *Trust and Trustworthy Computing*. Ed. by Thorsten Holz and Sotiris Ioannidis. Cham: Springer International Publishing, 2014, pp. 37–54. ISBN: 978-3-319-08593-7.

[40] Sandro Pinto et al. “LTZVisor: TrustZone is the Key”. In: *ECRTS*. 2017.

[41] Soo-Cheol Oh et al. “Acceleration of dual OS virtualization in embedded systems”. In: *2012 7th International Conference on Computing and Convergence Technology (ICCCT)* (2012), pp. 1098–1101.

[42] Sandro Pinto et al. “Towards a Lightweight Embedded Virtualization Architecture Exploiting ARM TrustZone”. In: Sept. 2014. DOI: 10.13140/RG.2.1.3588.1129.

[43] Václav Struhár et al. “Real-Time Containers: A Survey”. In: *Fog-IoT*. 2020.

[44] Marcello Cinque et al. “RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets”. In: *ECRTS*. 2019.

[45] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. “Container-based real-time scheduling in the Linux kernel”. In: *ACM SIGBED Review* 16 (Nov. 2019), pp. 33–38. DOI: 10.1145/3373400.3373405.

[46] Gerald J. Popek and Robert P. Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures”. In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073.

[47] A. Murphy. *VIRTUALIZATION DEFINED - EIGHT DIFFERENT WAYS*. URL: https://www.f5.com/pdf/white-papers/virtualization-defined-wp.pdf. White Paper.

[48] Khoa Pham. “Embedded Virtualization of a Hybrid ARM - FPGA Computing Platform”. PhD thesis. Mar. 2014. DOI: 10.13140/2.1.4945.2328.

[49] Valentine Sinitsyn. “Understanding the Jailhouse hypervisor, part 1”. In: *LWN.net* (2014).

[50] Maxim Baryshnikov. “Jailhouse Hypervisor”. Czech Technical University in Prague, 2016.

[51] *QEMU*. URL: https://www.qemu.org/.

[52] *rpi-imager*. URL: https://github.com/raspberrypi/rpi-imager.

[53] *Linaro toolchain*. URL: https://www.linaro.org/downloads/.

[54] Will Haley. *Raspberry Pi 4 Model B WiFi Ethernet Bridge*. URL: https://willhaley.com/blog/raspberry-pi-wifi-ethernet-bridge/?fbclid=IwAR0Q38Z6Yc1vhVb99xois4xeh3qfDj-ValbDlAth0ATXiAPiT-3jUxmsLDY. 2018.

[55] Mauro Riva. “Raspberry Pi 4B: Real-Time System using Preempt-RT (kernel 4.19.y)”. In: (2019).

[56] *Cyclictest Man Page*. URL: https://manpages.debian.org/jessie/rt-tests/cyclictest.8.

[57] *Cyclictest-Test Design*. URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/test-design.

[58] *Hackbench*. URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/hackben