KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING
Department of Computer Science

Enabling Behavior-Based Energy Consumption and Memory Footprint Optimizations in Native Contexts

Christos Lamprakos

Dissertation presented in partial fulfillment
of the requirements for the degree of
Doctor of Engineering Science (PhD):
Electrical Engineering

July 2025

Supervisors:
Prof. dr. ir. Dimitrios Soudris
(NTUA)
Prof. dr. ir. Francky Catthoor
Prof. dr. ir. Johan De Boeck

ENABLING BEHAVIOR-BASED ENERGY CONSUMPTION AND MEMORY FOOTPRINT OPTIMIZATIONS IN NATIVE CONTEXTS

Christos LAMPRAKOS

Supervisors:
Prof. dr. ir. Dimitrios Soudris
(NTUA)
Prof. dr. ir. Francky Catthoor
Prof. dr. ir. Johan De Boeck

Members of the
Examination Committee:
Prof. dr. ir. Hendrik Van Brussel, chair
Prof. dr. ir. Ioanna Roussaki
(NTUA)
Prof. dr. ir. Sotirios Xydis
(NTUA)
Prof. dr. ir. George Theodoridis
(University of Patras)
Dr. ir. Manu Perumkunnil
(IMEC)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

July 2025

Uitgegeven in eigen beheer, Christos Lamprakos, Lamprou Katsoni 67-69, 11471 Athens (Greece)

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

Καθολικό Πανεπιστήμιο του Λέουβεν
Σχολή Ηλεκτρολόγων Μηχανικών
DOCT Επιστήμης Μηχανικης

Συμπεριφοροκεντρική Βελτιστοποίηση Ενεργειακής Κατανάλωσης και Αποτυπώματος Μνήμης

Διδακτορική Διατριβή
του
Χρήστου Λαμπράκου

Επιβλέποντες: Δημήτριος Σούντρης (ΕΜΠ)
Francky Catthoor (ΕΜΠ)
Johan De Boeck (KUL)

Αθήνα, Ιούλιος 2025

National Technical University of Athens
School of Electrical & Computer Engineering
Division of Communication, Electronic and Information Engineering

KU LEUVEN

KU Leuven
Electrical Engineering
Doctor of Engineering Science

Enabling Behavior-Based Energy Consumption and Memory Footprint Optimizations in Native Contexts

Ph.D. Thesis
of
Christos Lamprakos

Supervisory Committee: Dimitrios Soudris (Professor NTUA)
Francky Catthoor (Visiting Professor NTUA)
Johan De Boeck (Professor KUL)

Approved by the advisory committee on July 10th, 2025.

.....	.....	.....
Dimitrios Soudris	Francky Catthoor	Johan De Boeck
Professor NTUA	Visiting Professor NTUA	Professor KUL
.....	.....	.....
Dionisios Pnevmatikatos	Konstantinos Siozios	Manu Perumkunnil
Professor NTUA	Associate Professor AUTH	Senior Researcher IMEC
.....	.....	.....
Sotirios Xydis	Ioanna Roussaki	George Theodoridis
Assistant Professor NTUA	Professor NTUA	Associate Professor UPAT

Athens, July 2025

.....
Christos Lamprakos
Doctor of Philosophy in Engineering NTUA
Doctor of Engineering Science KUL

Acknowledgements

The reason why I'm leaving academia is the constant need it imposes to prove myself in ways irrelevant to the work done. Thankfully I did not have to do that when it came to starting my graduate studies. My advisors, **Prof. Dimitrios Soudris** and **Prof. Francky Catthoor**, granted me with their ultimate trust from the very outset despite barely knowing me. In a world collapsing under absurdity, they gave me a straightforward and reasonable professional haven. I shall remain forever grateful to them, and all the wisdom that they so generously shared with me during the past six years. Speaking of teachers to whom I owe everything, here is a list that I have never stopped going through in my head and heart: my English tutor **Kostas Galanis**, for all the joyful hours he invested in shaping my speech, perception and self-esteem. My teacher during the final couple of years in elementary school, **Tasos Stamoulis**, for encouraging me to acknowledge and honor my strengths. My mentors from the Poreia Tutor School while preparing for the national university entrance exams: **Thodoris Antonopoulos** (greek language), **Maria Vagia** (mathematics), **Tasos Melas** (physics), **Nikos Anastasopoulos** (economics). My life has drawn me far from all of you, and I am sorry that I was not decent enough to keep contact. Please remember that your time with me was not for nothing. As for my friends and family, you know who you are and how much I love you. Individuals are an illusion, relationships are all that matters, I wish that our happy little tree of life never withers.

Funding

The research work presented in this text was financially supported by:

- the European Union's Horizon 2020 research and innovation programme under grant agreement No 780572 SDK4ED (`www.sdk4ed.eu`).

- the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101021274 PRAETORIAN (`https://cordis.europa.eu/project/id/101021274`).
- the European Union's Horizon Europe research and innovation programme under Grant Agreement No. 101096110 PRIVATEER (`https://www.privateer-project.eu/`).
- the European Union's Horizon research and innovation programme under grant agreement No 101070374 CONVOLVE (`https://convolve.eu/`).
- the Hellenic Foundation for Research and Innovation (HFRI) under the 3rd Call for HFRI PhD Fellowships (fellowship number: 61/512200).

Popularized Abstract

Computers control a large part of modern life despite being a (very) small part of history. And while exotic new applications such as ChatGPT attract most of the interest from both consumers *and* practitioners, the first principles underlying our world's digital infrastructure remain timeless. This thesis studies the following such principles through the lens of energy consumption and memory footprint: (i) *a central aspect of program behavior is its dynamic requests for memory*, (ii) *approximately optimal solutions to memory allocation can be computed offline* and (iii) *software is the result of iterative decision-making over source code transformations.*

Along the way, we make a series of original contributions. We show the complex impact that specific dynamic memory allocation implementations have on the extremely popular Python programming language; we describe a principled methodology for capturing program-allocator interaction and quantifying memory fragmentation; we contribute a static memory planning implementation outperforming the SOTA in a wide range of heavyweight, challenging benchmarks; and we demonstrate a flexible, agnostic framework for improving software.

Join us in a thrilling intellectual adventure spanning several levels of abstraction, featuring arcane algorithms and data structures, and introducing an entire new interpretation to the term ***deep*** *learning*! At the end of this text, we promise to have enlightened you with a satisfying conclusion to the Holywood-scale story starting with:

```
Once upon a time, three allocators walked into a bar...
```

Gepopulariseerde Samenvatting

Computers beheersen een groot deel van het moderne leven, ondanks dat ze slechts een (zeer) klein deel van de geschiedenis uitmaken. En hoewel exotische nieuwe toepassingen zoals ChatGPT de meeste interesse wekken bij zowel consumenten als professionals, blijven de basisprincipes die ten grondslag liggen aan de digitale infrastructuur van onze wereld tijdloos. Dit proefschrift bestudeert de volgende principes vanuit het perspectief van energieverbruik en geheugenvoetafdruk: (i) *een centraal aspect van programmagedrag is de dynamische geheugenaanvraag*, (ii) *bijna-optimale optimale oplossingen voor geheugentoewijzing kunnen offline worden berekend* en (iii) *software is het resultaat van iteratieve besluitvorming over broncodetransformaties.*

Onderweg leveren we een reeks originele bijdragen. We laten de complexe impact zien die specifieke implementaties van dynamische geheugentoewijzing hebben op de extreem populaire programmeertaal Python; we beschrijven een principiële methodologie voor het vastleggen van de interactie tussen programma's en geheugentoewijzers en het kwantificeren van geheugenfragmentatie; we een statische geheugenplanningsimplementatie die de bestaande literatuur overtreft in een breed scala aan zware, uitdagende benchmarks; en we demonstreren een flexibel, agnostisch raamwerk voor softwareverbetering.

Ga met ons mee op een spannend intellectueel avontuur dat verschillende abstractieniveaus beslaat, met mysterieuze algoritmen en datastructuren, en een geheel nieuwe interpretatie van de term ***deep*** *learning* introduceert! Aan het einde van deze tekst beloven we u te hebben geïnformeerd met een bevredigende conclusie van het verhaal op Holywood-schaal, beginnend met:

```
Er waren eens drie toewijzers die een bar binnenliepen...
```

Abstract

The fact that modern society depends on computers in order to function may obscure the fact that computer science, and consequently computer engineering, is a young discipline. Each decade since the 1990s has captivated the public's attention with yet another digital miracle, starting with the Internet, then social media, and now artificial intelligence. The ever growing impact of each aforementioned advancement on the economy has in return directed the respective research spotlights: the Web spawned parallel processing, Facebook spawned cloud computing, and though the dust has yet to settle, one cannot deny the push that LLMs have given to hardware accelerators and novel memory-centric technology such as CXL and compute-in-memory.

On the one hand, a scientific discipline being driven by its applications is a most natural phenomenon. It is after all the degree to which it benefits the public from where a breakthrough draws its value. On the other hand, we cannot help but wonder: *could this progress be happening too fast?* How can we be certain that all fruits have been reaped before moving on to the next paradigm? Intuition suggests we cannot. But each next paradigm is built on top of the previous one. This thesis is founded upon the belief that **meaningful work remains to be done in the fundamentals of computer systems**.

Let us begin with a definition of what we mean with the term "computer system". At the bottom of the abstraction hierarchy lies the "hardware", which we view as a compute unit interacting with a memory module. At the top we find a user application consisting of (i) source code that a developer wrote and (ii) third-party libraries imported as dependencies, written by other developers. Collectively, the application's source code is turned into executable binary instructions via the help of a compiler. In between the executable and the system's hardware lies the OS, which provides the illusion that this is a dedicated machine, hides the hardware's *actual* complexity behind interfaces, and takes care of running everything safely and efficiently.

For a long time now, the above definition has been a comfortable fabrication: compute units are many and heterogeneous, memory could encompass address spaces in *other* machines and/or be heterogeneous itself, and so on. But a second necessary remark is that, nevertheless, the simplified computer system remains the model used (if any) by most developers. A reasonable means to resolve this tension is to **incorporate in our view the specific effects that an application's execution brings upon aspects of the system that we care about**; in other words, its *behavior*. Modern system architects should let laymen developers retain their simplified model, while working underground to deliver infrastructure that adapts to its tasks in real time. This idea may sound synonymous to that one of abstraction, which is as old as computing itself—and rightfully so. Our emphasis regards what aspects of the system should the aforementioned abstractions adapt toward.

Energy consumption and memory footprint are often overlooked in the race to minimize execution time, which is the chief concern of all users. To give a concrete example, consider the ubiquitous dynamic memory allocation interface comprising GNU's `malloc` family of functions. By first principles, memory fragmentation is the main enemy of any allocator. But little work on defining fragmentation has been conducted, and allocator designers prefer to focus on problems impacting latency or security. Another example stems from the deep learning domain, where workloads are facing a so-called "memory wall"—apart from excessive energy requirements. This is mostly about memory bandwidth but partly also about the massive storage capacity needed, especially for exploding demand for machine learning-powered services.

Thus, we are brought to the research objectives of this dissertation, all of whom can be viewed as operations at a relatively high abstraction level where application programmers do not want to "see" the complex synergy between their code, the OS and the physical hardware:

- **Objective A**: *Conduct a principled, informed study of workload-allocator interaction and memory footprint.*
- **Objective B**: *Deliver a scalable SOTA implementation for static memory planning.*
- **Objective C**: *Provide application developers with assistance to better evaluate the impact of source-to-source transformations.*

Keywords: static memory planning, dynamic storage allocation, memory management, energy accounting, software engineering

Beknopte samenvatting

Het feit dat de moderne samenleving afhankelijk is van computers om te functioneren, kan het feit verhullen dat computerwetenschappen, en dus ook computertechniek, een jonge discipline is. Elk decennium sinds de jaren negentig heeft de aandacht van het publiek getrokken met weer een nieuw digitaal wonder, beginnend met het internet, vervolgens sociale media en nu kunstmatige intelligentie. De steeds grotere impact van elke bovengenoemde vooruitgang op de economie heeft op zijn beurt de respectievelijke onderzoeksspots in de schijnwerpers gezet: het web bracht parallelle verwerking voort, Facebook bracht cloud computing voort, en hoewel het stof nog moet neerdalen, kan men niet ontkennen dat LLM's hardwareversnellers en nieuwe geheugengerichte technologieën zoals CXL en compute-in-memory hebben gestimuleerd.

Aan de ene kant is het een heel natuurlijk fenomeen dat een wetenschappelijke discipline wordt gedreven door haar toepassingen. Het is immers de mate waarin het publiek er baat bij heeft, waaraan een doorbraak zijn waarde ontleent. Aan de andere kant kunnen we ons niet onthouden afvragen: gaat deze vooruitgang misschien te snel? Hoe kunnen we er zeker van zijn dat alle vruchten geplukt zijn voordat we naar het volgende paradigma gaan? Intuïtie suggereert dat dit niet mogelijk is. Maar elk volgend paradigma bouwt voort op het vorige. Deze these is gebaseerd op de overtuiging dat **er nog zinvol werk verricht moet worden in de basisprincipes van computersystemen**.

Laten we beginnen met een definitie van wat we bedoelen met de term "computersysteem". Onderaan de abstractiehiërarchie bevindt zich de "hardware", die we zien als een rekeneenheid die interageert met een geheugenmodule. Bovenaan vinden we een gebruikersapplicatie die bestaat uit (i) broncode die een ontwikkelaar heeft geschreven en (ii) bibliotheken van derden die als afhankelijkheden zijn geïmporteerd, geschreven door andere ontwikkelaars. Gezamenlijk wordt de broncode van de applicatie omgezet in uitvoerbare binaire instructies met behulp van een compiler. Tussen het uitvoerbare bestand en de hardware van het systeem bevindt zich het besturingssysteem, dat de illusie wekt dat het om een speciale machine gaat, de werkelijke complexiteit van de hardware achter interfaces verbergt en ervoor zorgt dat alles veilig en efficiënt

verloopt.

De bovenstaande definitie is al lange tijd een comfortabele constructie: rekeneenheden zijn talrijk en heterogeen, geheugen kan adresruimten in *andere* machines omvatten en/of zelf heterogeen zijn, enzovoort. Maar een tweede noodzakelijke opmerking is dat het vereenvoudigde computersysteem desalniettemin het model blijft dat de meeste ontwikkelaars (indien van toepassing) gebruiken. Een redelijke manier om deze spanning op te lossen, is om **naar onze mening de specifieke effecten te integreren die de uitvoering van een applicatie heeft op aspecten van het systeem die ons belangrijk lijken**; met andere woorden, het *gedrag ervan*. Moderne systeemarchitecten zouden onbekende ontwikkelaars hun vereenvoudigde model moeten laten behouden, terwijl ze ondergronds werken om infrastructuur te leveren die zich in realtime aanpast aan de taken. Dit idee klinkt misschien synoniem aan dat van abstractie, dat net zo oud is als het computergebruik zelf – en terecht. Onze nadruk ligt op welke aspecten van het systeem de bovengenoemde abstracties zich moeten aanpassen.

Energieverbruik en geheugengebruik worden vaak over het hoofd gezien in de race om de uitvoeringstijd te minimaliseren, wat de belangrijkste zorg is van alle gebruikers. Om een concreet voorbeeld te geven, neem de alomtegenwoordige interface voor dynamische geheugentoewijzing, die bestaat uit GNU's `malloc`-functiefamilie. Volgens de basisprincipes is geheugenfragmentatie de grootste vijand van elke geheugentoewijzer. Maar er is weinig onderzoek gedaan naar de definitie van fragmentatie, en ontwerpers van geheugentoewijzers richten zich liever op problemen die van invloed zijn op latentie of beveiliging. Een ander voorbeeld komt uit het deep learning-domein, waar workloads te maken hebben met een zogenaamde "geheugenmuur" – afgezien van de overmatige energiebehoefte. Dit gaat vooral over geheugenbandbreedte, maar deels ook over de enorme opslagcapaciteit die nodig is, met name voor de exploderende vraag naar diensten die draaien op machine learning.

Zo komen we bij de onderzoeksdoelstellingen van dit proefschrift, die allemaal gezien kunnen worden als bewerkingen op een relatief hoog abstractieniveau, waarbij applicatieprogrammeurs de complexe synergie tussen hun code, het besturingssysteem en de fysieke hardware niet willen 'zien':

- **Doelstelling A:** *Een principieel, geïnformeerd onderzoek uitvoeren naar de interactie tussen werklast en toewijzer en de geheugenvoetafdruk.*
- **Doelstelling B:** *Een schaalbare meer optimale implementatie leveren voor statische geheugenplanning.*
- **Doelstelling C:** *Applicatieontwikkelaars ondersteunen bij het beter evalueren van de impact van bron-naar-bron-transformaties.*

Extended Abstract in Greek

Το γεγονός ότι η σύγχρονη κοινωνία εξαρτάται από τους υπολογιστές για να λειτουργήσει μπορεί να συσκοτίζει το γεγονός ότι η επιστήμη των υπολογιστών, και κατ΄ επέκταση η μηχανική υπολογιστών, είναι ένας νέος κλάδος. Κάθε δεκαετία από τη δεκαετία του 1990 έχει αιχμαλωτίσει την προσοχή του κοινού με ένα ακόμη ψηφιακό θαύμα, ξεκινώντας από το Διαδίκτυο, στη συνέχεια τα μέσα κοινωνικής δικτύωσης, και τώρα την τεχνητή νοημοσύνη. Ο συνεχώς αυξανόμενος αντίκτυπος κάθε προαναφερθείσας προόδου στην οικονομία έχει με τη σειρά του κατευθύνει τα αντίστοιχα ερευνητικά φώτα: ο Παγκόσμιος Ιστός γέννησε την παράλληλη επεξεργασία, το Φαςεβοοκ γέννησε το υπολογιστικό νέφος, και παρόλο που η σκόνη δεν έχει καταλαγιάσει ακόμα, δεν μπορεί κανείς να αρνηθεί την ώθηση που τα LLM έχουν δώσει στους επιταχυντές υλικού και στη νέα τεχνολογία με επίκεντρο τη μνήμη, όπως το CXL και ο υπολογισμός-εντός-μνήμης.

Από τη μία πλευρά, το να καθοδηγείται ένας επιστημονικός κλάδος από τις εφαρμογές του είναι ένα απολύτως φυσικό φαινόμενο. Σε τελική ανάλυση, η αξία μιας ανακάλυψης πηγάζει από τον βαθμό στον οποίο ωφελεί το κοινό. Από την άλλη πλευρά, δεν μπορούμε παρά να αναρωτηθούμε: *μήπως αυτή η πρόοδος συμβαίνει υπερβολικά γρήγορα·* Πώς μπορούμε να είμαστε βέβαιοι ότι όλοι οι καρποί έχουν συλλεχθεί πριν προχωρήσουμε στο επόμενο παράδειγμα· Η διαίσθηση υποδηλώνει ότι δεν μπορούμε. Αλλά κάθε επόμενο παράδειγμα χτίζεται πάνω στο προηγούμενο. Αυτή η διατριβή βασίζεται στην πεποίθηση ότι **απομένει σημαντικό έργο να γίνει στα θεμελιώδη των υπολογιστικών συστημάτων**.

Ας ξεκινήσουμε με έναν ορισμό του τι εννοούμε με τον όρο 'υπολογιστικό σύστημα'. Στο κάτω μέρος της ιεραρχίας αφαίρεσης βρίσκεται το 'υλικό' (hardware), το οποίο θεωρούμε ως μια υπολογιστική μονάδα που αλληλεπιδρά με μια μονάδα μνήμης. Στην κορυφή βρίσκουμε μια εφαρμογή χρήστη που αποτελείται από (ι) πηγαίο κώδικα που έγραψε ένας προγραμματιστής και (ιι) βιβλιοθήκες τρίτων που εισάγονται ως εξαρτήσεις, γραμμένες από άλλους προγραμματιστές.

Συλλογικά, ο πηγαίος κώδικας της εφαρμογής μετατρέπεται σε εκτελέσιμες δυαδικές εντολές με τη βοήθεια ενός μεταγλωττιστή. Ανάμεσα στο εκτελέσιμο αρχείο και το υλικό του συστήματος βρίσκεται το λειτουργικό σύστημα, το οποίο παρέχει την ψευδαίσθηση ότι πρόκειται για μια αποκλειστική μηχανή, κρύβει την *πραγματική* πολυπλοκότητα του υλικού πίσω από διεπαφές και φροντίζει για την ασφαλή και αποδοτική εκτέλεση των πάντων.

Για μεγάλο χρονικό διάστημα, ο παραπάνω ορισμός ήταν μια βολική επινόηση: οι υπολογιστικές μονάδες είναι πολλές και ετερογενείς, η μνήμη θα μπορούσε να περιλαμβάνει χώρους διευθύνσεων σε *άλλες* μηχανές ή/και να είναι η ίδια ετερογενής, και ούτω καθεξής. Αλλά μια δεύτερη απαραίτητη παρατήρηση είναι ότι, παρ΄ όλα αυτά, το απλοποιημένο υπολογιστικό σύστημα παραμένει το μοντέλο που χρησιμοποιείται (αν χρησιμοποιείται κάποιο) από τους περισσότερους προγραμματιστές. Ένας λογικός τρόπος για να επιλυθεί αυτή η ένταση είναι να **ενσωματώσουμε στην οπτική μας τις συγκεκριμένες επιδράσεις που έχει η εκτέλεση μιας εφαρμογής σε πτυχές του συστήματος που μας ενδιαφέρουν**· με άλλα λόγια, τη *συμπεριφορά* της. Οι σύγχρονοι αρχιτέκτονες συστημάτων θα πρέπει να αφήνουν τους απλούς προγραμματιστές να διατηρούν το απλοποιημένο μοντέλο τους, ενώ εργάζονται υπογείως για να παρέχουν υποδομή που προσαρμόζεται στα καθήκοντά της σε πραγματικό χρόνο. Αυτή η ιδέα μπορεί να ακούγεται συνώνυμη με εκείνη της αφαίρεσης, η οποία είναι τόσο παλιά όσο και η ίδια η πληροφορική—και δικαίως. Η έμφασή μας αφορά προς ποιες πτυχές του συστήματος θα πρέπει να προσαρμόζονται οι προαναφερθείσες αφαιρέσεις.

Η κατανάλωση ενέργειας και το αποτύπωμα μνήμης συχνά παραβλέπονται στον αγώνα για την ελαχιστοποίηση του χρόνου εκτέλεσης, που αποτελεί το κύριο μέλημα όλων των χρηστών. Για να δώσουμε ένα συγκεκριμένο παράδειγμα, ας εξετάσουμε την πανταχού παρούσα διεπαφή δυναμικής εκχώρησης μνήμης που περιλαμβάνει την οικογένεια συναρτήσεων `malloc` της GNU. Εξ ορισμού, ο κατακερματισμός της μνήμης είναι ο κύριος εχθρός οποιουδήποτε εκχωρητή. Αλλά λίγη δουλειά έχει γίνει στον ορισμό του κατακερματισμού, και οι σχεδιαστές εκχωρητών προτιμούν να εστιάζουν σε προβλήματα που επηρεάζουν τον λανθάνοντα χρόνο ή την ασφάλεια. Ένα άλλο παράδειγμα προέρχεται από τον τομέα της βαθιάς μάθησης, όπου οι φόρτοι εργασίας αντιμετωπίζουν το λεγόμενο 'τείχος της μνήμης'—πέρα από τις υπερβολικές ενεργειακές απαιτήσεις. Αυτό αφορά κυρίως το εύρος ζώνης της μνήμης αλλά εν μέρει και την τεράστια χωρητικότητα αποθήκευσης που απαιτείται, ειδικά για την εκρηκτική ζήτηση για υπηρεσίες που βασίζονται στη μηχανική μάθηση.

Έτσι, φτάνουμε στους ερευνητικούς στόχους αυτής της διατριβής, οι οποίοι όλοι μπορούν να θεωρηθούν ως λειτουργίες σε ένα σχετικά υψηλό επίπεδο αφαίρεσης όπου οι προγραμματιστές εφαρμογών δεν θέλουν να 'βλέπουν' την πολύπλοκη συνέργεια μεταξύ του κώδικά τους, του λειτουργική συστήματος και του φυσικού

υλικού:

- **Στόχος Α**: *Διεξαγωγή μιας αρχειοθετημένης, τεκμηριωμένης μελέτης της αλληλεπίδρασης φόρτου εργασίας-εκχωρητή και του αποτυπώματος μνήμης.*
- **Στόχος Β**: *Παράδοση μιας κλιμακούμενης υλοποίησης SOTA για στατικό σχεδιασμό μνήμης.*
- **Στόχος Γ**: *Παροχή βοήθειας στους προγραμματιστές εφαρμογών για την καλύτερη αξιολόγηση του αντικτύπου των μετασχηματισμών από πηγαίο-σε-πηγαίο κώδικα.*

Λέξεις κλειδιά: στατικός σχεδιασμός μνήμης, δυναμική κατανομή μνήμης, διαχείριση μνήμης, καταμέτρηση ενέργειας, μηχανική λογισμικού

List of Abbreviations

2DBP Two-Dimensional rectangular Bin Packing. xxiv, xxvii, 31–35, 37, 40, 42, 92

AI Artificial Intelligence. 45

AMD Advanced Micro Devices Inc.. 97

ARM Advaned RISC Machine Inc.. 97

BA The boxing algorithm by Buchsbaum et al.. xxvii, 50–55, 57, 60–62

CIA Change Impact Analysis. 89, 90

CPU Central Processing Unit. 87, 96

CSV Comma-Separated Values. 37, 38

CXL Compute Express Link. xi, xv, 1

DDTR Dynamic Data Type Refinement. 93

DM Decision Maker. 83, 85–88

DRAM Dynamic Random Access Memory. 19, 24, 26, 27, 29, 41, 98

DSA Dynamic Storage Allocation. xxix, 5, 8, 15–17, 19, 20, 22, 23, 25, 29, 31, 33, 42, 45–49, 51, 53, 54, 62, 92–94

ECDF Empirical Cumulative Distribution Function. xxvii, 22–24

EOF End Of File. 105

FBWM Fuzzy Best-Worst Method. 87

FU Functional Unit. 52, 54, 62

GNU GNU's Not Unix. xii, xvi, 4, 20, 36–38, 45

GPT Generative Pre-trained Transformer. vii, ix, 2

GPU Graphics Processing Unit. 87

HPT Hierarchical Performance Testing. 16, 17, 19, 27

HTML HyperText Markup Language. 21

I/O Input-Output. 6, 103

IEEE Institute of Electrical and Electronics Engineers. 97

IGC Interval Graph Coloring. 49, 54, 61

IR Intermediate Representation. 99, 100, 104

JSON JavaScript Object Notation. 21

LLM Large Language Model. xi, xv, 1, 9, 93

LLVM Low-Level Virtual Machine. xxv, 96, 99, 100, 104, 105, 109

LTO Link-Time Optimization. xxiv, 20, 23, 24, 26, 29

MCDM Multiple-Criteria Decision Making. xxviii, 9, 10, 82, 83, 85–87, 89, 94

MMU Memory Management Unit. 19, 26, 29

N/A Not Applicable. 28

NFR Non-Functional Requirement. 9, 83, 86–88, 94

NP Non-deterministic Polynomial time. 8, 42

OS Operating System. xi, xii, 3–5, 38, 45, 49, 92, 93

PARSEC Princeton Application Repository for Shared-Memory Computers. 16

PC Personal Computer. xxix, 21, 25, 28

PGO Profile-Guided Optimization. xxiv, 20, 23, 24, 26, 29

PT Processor Tracing. xxv, xxviii, 96–99, 104, 105

RAPL Running Average Power Limit. xxv, xxviii, 15, 19, 26, 27, 96–98, 100, 102–105, 109, 110

RISC Reduced Instruction Set Computer. 110

RSS Resident Set Size. 7, 8, 31, 32, 37, 40–42, 92, 93

SoC System-on-Chip. 21–23, 28

SOTA State Of The Art. vii, xii, xvii, 6, 8–11, 46, 48, 51, 96

SSA Static Single Assignment. 100

XML eXtensible Markup Language. 21

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The fact that modern society depends on computers in order to function may obscure the fact that computer science, and consequently computer engineering, is a young discipline. To the extent that one can trust Wikipedia[1], we know for example that humans have been studying numbers for more than 5,000 years; geometry 4,000 (though it took more than the first thousand before flourishing); anatomy 3,000 and so on. However, the paper which introduced the notion of the computer in the sense of a flexible information-processing machine, i.e., Alan Turing's seminal work on Hilbert's Entscheidungsproblem [118], needs another eleven years to become a century old. And as a reminder of the non-trivial amount of effort separating the world of *notions* from the *physical* world, almost a decade had to pass before the genius of John von Neumann turned Turing's concepts into a real machine [120].

But reality did not give our discipline time to mature before the popularity of its applications exploded. Each decade since the 1990s has captivated the public's attention with yet another digital miracle, starting with the Internet, then social media, and now artificial intelligence. The ever growing impact of each aforementioned advancement on the economy has in return directed the respective research spotlights: the Web spawned parallel processing, Facebook spawned cloud computing, and though the dust has yet to settle, one cannot deny the push that LLMs have given to hardware accelerators and novel memory-centric technology such as CXL and compute-in-memory. Note that providing and accurate history of progress in computing is out of this dissertation's scope. The claims in this paragraph stem from our judgement, to the extent that it has been informed from spending the past few years thinking about computers.

[1] https://en.wikipedia.org/wiki/Timeline_of_scientific_discoveries

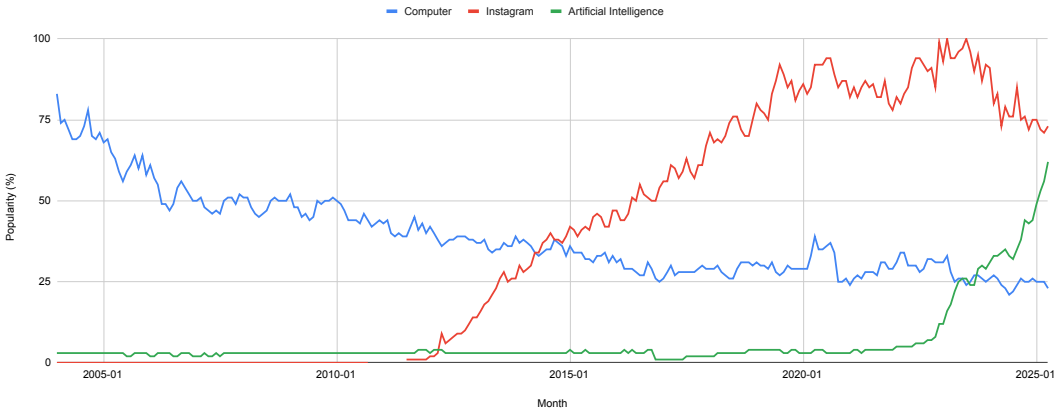


Figure 1.1: Popularity comparison of the terms “Computer”, “Instagram” and “Artificial Intelligence”, as searched on Google worldwide. Chart extracted via Google Trends.

One cannot deny, however, that there *have* been trends and attractors of the community’s collective interest.

On the one hand, a scientific discipline being driven by its applications is a most natural phenomenon. It is after all the degree to which it benefits the public from where a breakthrough draws its value. On the other hand, we cannot help but wonder: *could this progress be happening too fast?* How can we be certain that all fruits have been reaped before moving on to the next paradigm? Intuition suggests we cannot. But each next paradigm is built on top of the previous one. So who is to blame for the fact that training ChatGPT-3 requires as much energy as 130 American homes consume in a year [16]? If it were something inherent in deep learning, DeepSeek would not have been able to make headlines all around the world with a model both better *and 10x more efficient* than its competition[2].

According to Figure 1.1, ten years have passed since computers themselves were as interesting to the world as Instagram. At the time of writing, AI is as popular as the machines it runs on were fifteen years ago. In fact one can observe a slow yet steady decline in the amount of attention that we pay to what has become the world’s infrastructure. This thesis is founded upon the belief that **meaningful work remains to be done in the fundamentals of**

[2]https://www.tomshardware.com/tech-industry/artificial-intelligence/deepseeks-ai-breakthrough-bypasses-industry-standard-cuda-uses-assembly-like-ptx-programming-instead

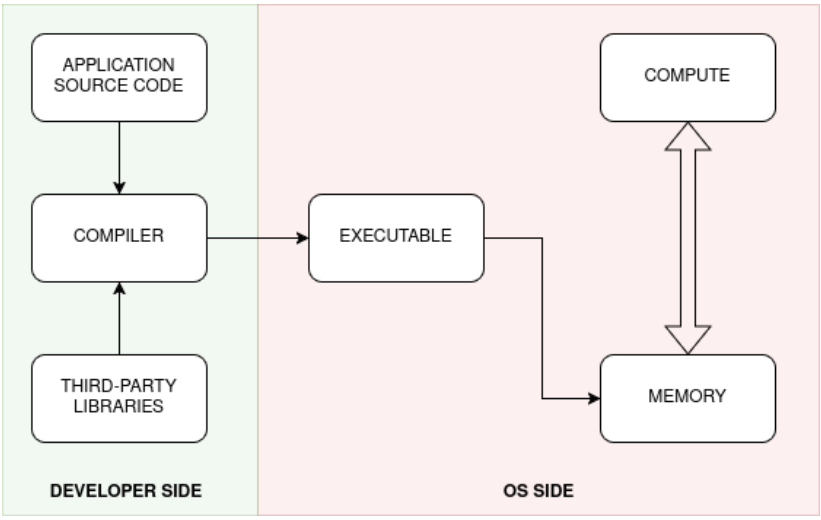


Figure 1.2: A (simplified) model of a computer system.

computer systems.

1.1 Scientific Approach

Let us begin with a definition of what we mean with the term "computer system". Figure 1.2 serves as an aid. At the bottom of the abstraction hierarchy (right side of the figure) lies the "hardware", which we view as a compute unit interacting with a memory module. At the top (left side) we find a user application consisting of (i) source code that a developer wrote and (ii) third-party libraries imported as dependencies, written by other developers. Collectively, the application's source code is turned into executable binary instructions via the help of a compiler. In between the executable and the system's hardware lies the OS, which provides the illusion that this is a dedicated machine, hides the hardware's *actual* complexity behind interfaces, and takes care of running everything safely and efficiently.

The first remark to be made at this point is that, for a long time now, the above definition has been a comfortable fabrication [30]: compute units are many and heterogeneous, memory could encompass address spaces in *other* machines and/or be heterogeneous itself, and so on. But a second necessary remark is that, nevertheless, the simplified computer system of Figure 1.2 remains the model used (if any) by most developers today [12]. A reasonable means to

resolve this tension is to **incorporate in our view the specific effects that an application's execution brings upon aspects of the system that we care about**; in other words, its *behavior*. Modern system architects should let laymen developers retain their simplified model, while working underground to deliver infrastructure that adapts to its tasks in real time. This idea may sound synonymous to that one of abstraction, which is as old as computing itself—and rightfully so. Our emphasis regards what aspects of the system should the aforementioned abstractions adapt toward.

Energy consumption and memory footprint are often overlooked in the race to minimize execution time, which is the chief concern of all users [104, 10]. To give a concrete example, consider the ubiquitous dynamic memory allocation interface comprising GNU's `malloc` family of functions. The largest part of the research presented in this thesis was sparked by a seminal survey on `malloc` published thirty years ago [121]. The main point of that survey was that, by first principles, memory fragmentation is the main enemy of any allocator. Moreover, it is a consequence of the particular interaction between the program and the allocator. Last but not least, a quantitative definition of it is elusive. But little work on defining fragmentation has been conducted ever since, and allocator designers prefer to focus on problems impacting latency such as thread contention [25, 89], producer-consumer relationships [79], and language runtimes [76]. And yet, as both this thesis and other works have shown, the interaction between a specific application and a memory allocator significantly impacts the resulting energy consumption and process memory footprint. Another example stems from the deep learning domain, where workloads are facing a so-called "memory wall" [39]—apart from the excessive energy requirements mentioned already. This is mostly about memory bandwidth but partly also about the massive storage capacity needed, especially for exploding demand for machine learning-powered services.

At this point we have laid the ground to elaborate on the terms comprising the title of this thesis. By doing so, we hope to achieve a clear description of our scientific approach. Let us start parsing in reverse: the title suggests that our work takes place in **native contexts**. We are interested in no further virtualization than that imposed by the OS on a single machine. Higher-level abstractions such as virtual machines and distributed computation will be viewed, if present, as traditional OS processes interacting with the host's hardware. It was this narrow scope that we were referring to earlier as "fundamentals"; in other words, operations and interfaces that are ubiquitous in computing. By positioning our research in this way, we hope to provide insights and tools with as wide an applicability as possible.

In this context, we want to **enable** optimizations. We have witnessed more than once during the course of our studies the presentation of research that

takes *imaginary* capabilities as a given. We have done so ourselves as the second Chapter will show. But there is a world of difference between building what one imagined and building on top of it. We are interested in the first half. Our claim of "enabling" optimizations is a subtle way of acknowledging our failure to complete building what we imagined, but we nevertheless deliver a couple of concrete first steps.

The optimizations at hand will be **behavior-based**. We allude here to the notion that there is no free lunch in today's systems, and probably there never was. Workloads have characteristics that make them distinct from one another. When expressed on different hardware platforms, their impact on performance is also different. It is our firm belief that such characteristics have to be exploited en route to optimization. The main characteristic we will be dealing with is *dynamic memory allocation*.

The rest of the title, i.e., specific ties to energy consumption and memory footprint, will be unpacked in the background sections of each Chapter.

1.2 Research Objectives

Thus, we are brought to the research objectives of this dissertation, all of whom can be viewed as operations at a relatively high abstraction level where application programmers do not want to "see" the complex synergy between their code, the OS and the physical hardware.

Objective A

Conduct a principled, informed study of workload-allocator interaction and memory footprint.

Dynamic memory allocation is a both ubiquitous and fundamental operation. It is telling of our epistemic status around it that in performance-critical situations such as real-time embedded systems, dynamic memory allocation is avoided as if it posed existential risks. The mystery at the heart of the problem is memory fragmentation, the amount of memory beyond a program's needs that was wasted. Very little work around taming or even defining fragmentation exists. We ascribe this situation to an up-to-now lack of tools for describing the source of fragmentation, which is the interaction between a workload's requests for memory and the corresponding allocator's placement policy. We show that the bin-packing variant of DSA is a perfect fit for the task, and propose a novel fragmentation measure on top of it.

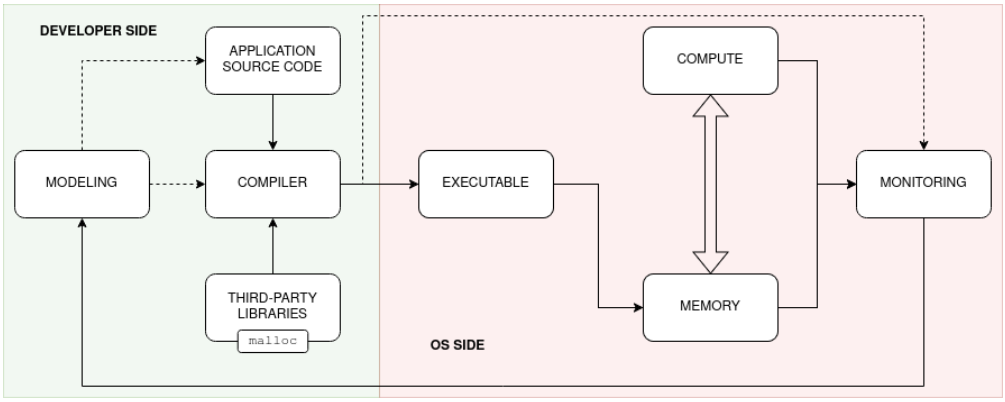


Figure 1.3: Computer system model extended with components and relationships that this thesis treats. Bold arrows stand for I/O relationships. Dotted arrows for assistance. Objective A focuses on `malloc`, a fundamental operation closely tied to program behavior and affecting both energy consumption and memory footprint. Objective B offers a tool for investigating the theoretical limits of `malloc` by providing a scalable, SOTA off-line oracle for it. Objective C adopts a bird's eye view of the application development lifecycle and proposes an agnostic decision-making framework for source-level code transformations leveraging lower-level information.

Objective B

Deliver a scalable SOTA implementation for static memory planning.

Future allocators must be informed by "perfect" solutions to the allocation patterns of the workloads they face. These solutions must by definition have access to the entire allocation history of a process. The task of producing such solutions is *static memory planning*. Existing implementations are either not scalable or excessively wasteful. We come up with an implementation that exhibits none of those hindrances. We achieve this goal by (i) building on top of the best known algorithm to date, (ii) discovering and reinforcing its weak spots, and (iii) leveraging modern systems programming techniques to deliver a safe, parallel implementation.

Our implementation's source code and the benchmarks used for evaluation are publicly available on GitHub.

Objective C

Provide application developers with assistance to better evaluate the impact of source-to-source transformations.

Developers writing software for a specific computer system do not have the ability to change the system itself. They thus embark on optimizing applications through source-to-source transformations. Due to the underlying complexity of both the system and the application, these transformations are bound to affect energy consumption and memory footprint in non-trivial ways. But additional goals may be put on the optimization task, such as technical debt elimination or security enhancements. To make their final decision, the user must operate on information linking each candidate transformation to changes in each aspect of interest. We provide a structured way of handling such information.

1.3 State of the Art

We will describe the state of the art per objective, and identify the gaps addressed in each case. Be advised that parts of the following sections have been copied from the author's list of publications.

1.3.1 Objective A

For this objective, we aim to open the black box standing between workload-allocator interaction and memory fragmentation. This link was first put to paper by Wilson et. al in their seminal 1995 survey on dynamic memory allocation [121]. A couple of years later, some of the authors returned with the most comprehensive study of fragmentation to date [57]. While acknowledging that many possible definitions exist, they focused on two and concluded that fragmentation is practically "solved". Their experiments, however, were trace-based simulations operating entirely on virtual memory and did not take *physical memory* into consideration.

In most recent times, practitioners have converged to a coarse view of fragmentation which is based on RSS, that is, the amount of physical memory consumed by a process (which is *different* from the amount of memory it allocated due to demand paging). Using two different allocators on the same workload, they conclude that the one yielding less fragmentation is the one resulting in smaller peak RSS [106, 81, 50]. We find this stance rather hazardous, since it (i) obscures the fact that allocators operate on *virtual* memory, (ii) and

mixes together the impact of an allocator's policy and its *implementation*, i.e., the data structures it internally uses for book-keeping.

Yet, as novel work occasionally shows, fragmentation remains elusive [87]: lacking a principled methodology to quantify it, we are adopting indirect approaches to merely inspect its effects. Chapter 2 motivates our focus on dynamic memory allocation by showing the impact of different allocators on the CPython runtime's energy consumption and memory footprint. Chapter 3 proposes a novel, structured view of workload-allocator interaction based on DSA, and a fragmentation measure which demonstrably correlates with RSS without sacrificing placement policy observability.

- **Gap in the SOTA:** An agreed-upon representation of memory fragmentation as incurred by dynamic memory allocators.
- **Contribution:** A demonstration of the suitability of DSA to capture fragmentation, and open source tools to study it further.

1.3.2 Objective B

The mathematical formulation of dynamic memory allocation is DSA. It is an NP-complete combinatorial optimization task that is a variation of bin packing. Our goal in this objective is to implement a golden standard for DSA.

A summary of theoretical work on DSA up to two decades ago can be found in the Introduction section of Buchsbaum et al. [13] The same paper presents the SOTA algorithm for the general case. Kierstead and Saoub have introduced generalized DSA, which allows some spatial overlap between rectangles [62]. Related but different problems are the Storage Allocation Problem [91], 2D Geometric Knapsack [45] and Unsplittable Flow on a Path [42].

As far as implementations go, Maas et al have proposed hybridizing heuristics and meta-optimization with TelaMalloc [83]. Moffitt improved upon TelaMalloc with an isomorphism between DSA and lattice theory [90]. Lamprou et al. introduced a heuristics-based solution with additional constraints [73].

Such recent efforts have been motivated by the memory-saving benefits of applying DSA to deep learning compilers for both training and inference. As a result, evaluation sections are always limited in that context. A rigorous evaluation with a focus on general-purpose DSA has not been conducted. We undertake this task in Chapter 4, along with contributing a golden standard implementation.

- **Gap in the SOTA:** A scalable, efficient and effective implementation for static memory planning. A rigorous evaluation of existing solutions. Necessary conceptual and technical insights for future work.
- **Contribution:** All of the above.

1.3.3 Objective C

The task at hand is evaluating a set of candidate code changes under a set of oftentimes conflicting criteria of software quality.

The term "software quality" refers to a program's characteristics outside its functional specification. Several standards have been proposed, the most popular of which is ISO/IEC 25010[3]. Prior art does not exhibit any consensus on the meaning of the term. In their 1996 survey on software quality, Osterweil et al. limit its notion to the consistency between a program's intended and actual behavior [98]. Cavano and McCall recognized as early as 1978 the inherent difficulty in both defining and measuring software quality, which should in their opinion be application-specific [15]. Since we shall be taking software quality criteria as a given, the key insight from this paragraph is that the process we are after must be extremely flexible.

In particular, we are interested in the impact that source code changes, i.e., source-to-source transformations, have on aspects orthogonal to the program's function. Such aspects have been named NFRs in the literature [41, 86, 5, 54]. We treat the problem of deciding upon a pool of heterogeneous code changes, i.e., not all of them improving the same NFR. Similar works exist. For instance, Ouni et al. perform an automated search in refactoring space, adopting a genetic algorithm to evaluate possible code changes [99]. Such search-based formulations have evolved to incorporate LLMs in the decision making process [33]. We refrain from automated methods since it is a known fact that teams of developers prefer manual actions when refactoring [93].

Our work in Chapter 5 thus provides decision support for the manual application of NFR-targeting code changes. To the best of our knowledge, the closest intellectual relative is Zhao and Hayes' work on rank-based decision support [124]. The key difference of our contribution is that it does not subscribe to any particular type of NFR, and formulates the problem as an instance of MCDM [55]. For instance, there is a whole line of work focusing on the economic impact of refactoring, mostly technical debt [14, 28, 29, 26, 32].

[3]https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

- **Gap in the SOTA:** Generic formulation of source-to-source transformations targeting *arbitrary* criteria.
- **Contribution:** View refactoring as MCDM.

1.4 A Note on Memory Fragmentation

A main theme across a large portion of this text is memory fragmentation. As already noted, multiple quantitative definitions of it exist. A commonly agreed upon *qualitative* definition is "memory that is available in aggregate, but not contiguous". In order to avoid developing further confusion around the phenomenon, we give here a brief descriptions of the two perspectives from which we approach fragmentation in this dissertation. Chapter 3 introduces an *aggregate*, *page-local* definition aligned with the demand paging mechanism of Linux. In that context, we were interested in summarizing workload-allocator interaction *across time* into a single number, which would hopefully correlate with the real memory footprint. Our definition is good to the extent that the expected monotonic relationship does indeed appear, but better alternatives could always exist. Chapter 4 adopts a much more conservative approach, measuring the amount of memory wasted as the difference between maximum memory allocated and maximum memory usage. This definition is useful for contexts such as the real, contiguous, physical address space of hardware accelerators with no virtualization. For the reasons explained in that Chapter, we are confident that this definition is the most appropriate for setups where memory is not virtual, i.e., contiguous memory pages correspond to contiguous memory.

1.5 Remaining Text Organization

The main body of this dissertation is organized as follows. Chapter 2 describes the case study initializing Objective A, and Chapter 3 demonstrates the core of our work there. Chapter 4 is an extensive account of how we achieved Objective B. Chapter 5 showcases our work on Objective C. A global discussion on the above and proposed future work are the matter of Chapter 6.

We depict the relationships between objectives, chapters and stages during our research trajectory on Figure 1.4. In the beginning, our intention was to (i) study the memory footprint and energy consumption of some fundamental operation and (ii) investigate ways of assisting the *integration* of systems-derived insights to the application development lifecycle. This spawned Objectives A and C, on

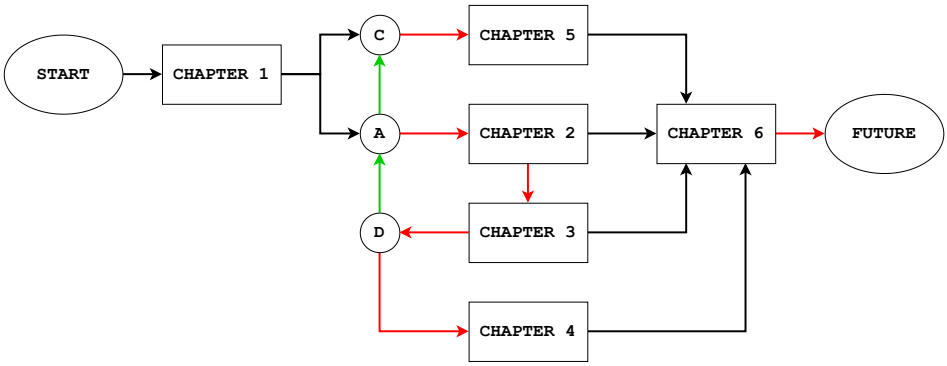


Figure 1.4: An illustration of the organization of this text, annotated with extra information. Ellipses are initial and "terminal" states. Circles are research objectives. Rectangles are thesis chapters. Black arrows represent **logical implication**. Red arrows indicate the **research trajectory** followed. Last but not least, green arrows indicate **synergies** between objectives. For instance Objective A can, by assisting the construction of high-level models around different `malloc` implementations, inform Objective C.

which we worked in parallel. The motivational study of `malloc` and CPython (Chapter 2) led us to the bin packing representation of workload-allocator interaction (Chapter 3). We were then led to investigate the `malloc`-related bin packing SOTA, and our work there is presented in Chapter 4.

Chapter 2

The Impact of Dynamic Storage Allocation on CPython Execution Time, Memory Footprint and Energy Consumption: An Empirical Study

This Chapter is a verbatim copy of the author's publication cited below:

Lamprakos, C. P., Papadopoulos, L., Catthoor, F., and Soudris, D. The impact of dynamic storage allocation on cpython execution time, memory footprint and energy consumption: An empirical study. In *Embedded Computer Systems: Architectures, Modeling, and Simulation* (Cham, 2022), A. Orailoglu, M. Reichenbach, and M. Jung, Eds., Springer International Publishing, pp. 219–234

CPython is the reference implementation of the Python programming language. Tools like machine learning frameworks, web development interfaces and scientific computing libraries have been built on top of it. Meanwhile, single-

board computers are now able to run GNU/Linux distributions. As a result CPython's influence today is not limited to commodity servers, but also includes edge and mobile devices. We should thus be concerned with the performance of CPython applications. In this spirit, we investigate the impact of dynamic storage allocation on the execution time, memory footprint and energy consumption of CPython programs. Our findings show that (i) CPython's default configuration is optimized for memory footprint, (ii) replacing this configuration can improve performance by more than $1.6x$ and (iii) application-specific characteristics define which allocator setup performs best at each case. Additionally, we contribute an open-source means for benchmarking the energy consumption of CPython applications. By employing a rigorous and reliable statistical analysis technique, we provide strong indicators that most of our conclusions are platform-independent.

2.1 Introduction

Today Python is one of the most popular high-level programming languages[1]. Its reference implementation is CPython[2]. Python source code is usually compiled to bytecode and executed on the CPython virtual machine, which is implemented in C.

CPython is often criticized for its performance, which has previously been compared to that of other programming languages [103]. The results do indeed validate the criticism, but Python's extreme popularity cannot be ignored. Several Python libraries have dominated the programming landscape. SciPy [119] has democratized scientific computing, scikit-learn [102] has done the same for introductory machine learning projects, Pytorch [8] has almost monopolized deep learning pipelines in both academia and the industry.

CPython is not deployed just on servers or home computers, but is becoming all the more present on embedded systems[3][4]. This is largely owed to the availability of cheap single-board computers (SBCs) like the Raspberry Pi[5] and the BeagleBone[6], which are capable of running GNU/Linux.

We thus consider improving the execution time, memory footprint and energy consumption of CPython applications a worthwhile endeavor. We believe low-hanging fruit should be reaped before having to examine CPython's internals.

[1] https://insights.stackoverflow.com/survey/2020/
[2] https://github.com/python/cpython
[3] https://wiki.python.org/moin/EmbeddedPython
[4] https://www.zerynth.com/blog/the-rise-of-python-for-embedded-systems/
[5] https://www.raspberrypi.com/
[6] https://beagleboard.org/bone

Consequently, we focus on configurations exposed by the language runtime that cause observable, non-random effects on performance[7].

As regards the particular configuration under study, we picked DSA[8] for the following reasons: firstly, CPython does provide the option of configuring it. Secondly, DSA is a cornerstone operation used by most real-world programs and as a result demands attention. Finally, it is not well-understood [121], so treating it as a black box and benchmarking several versions of it may yield useful results.

2.1.1 Contributions

We conducted a reliable, statistically rigorous empirical study of DSA's impact on the execution time, memory footprint and energy consumption of CPython programs. Our first contribution is providing quantitative answers to the research questions: (**I**) to what extent can a CPython application's performance with respect to execution time, memory footprint and energy consumption be improved by modifying the runtime's DSA configuration? (**II**) how much do optimizations in the runtime itself affect the expected improvement? (**III**) are performance improvements sensitive to application-specific characteristics? By addressing these questions, we pave the way towards predicting the optimal DSA configuration for a given application without resorting to brute-force methods. Note, however, that for the purposes of this Chapter, we are only interested in acquiring a coarse impression of whether DSA *can* have substantial impact. Transitioning from such an epistemic status to a methodology for picking the optimal allocator per case is non-trivial and, as our results show, extremely case-sensitive. Future practitioners' best hopes lie, in our opinion, in solutions that adapt in a fully dynamic way to the characteristics of the access patterns in the currently present application phase.

Our second contribution is an open-source modification of the `pyperformance` and `pyperf` packages, which enables the benchmarking of Python programs with respect to energy consumption. It can be used on all Linux-running machines featuring Intel's RAPL power capping interface [22, 47]. All of our results and accompanying code are publicly available on GitHub[9].

[7] From this point onward, we will use the term "performance" to refer collectively to the set of execution time, memory footprint and energy consumption. See Section **??** for details.

[8] In this Chapter, DSA stands for "traditional" dynamic memory allocation in operating systems.

[9] https://github.com/cappadokes/cpythondsa

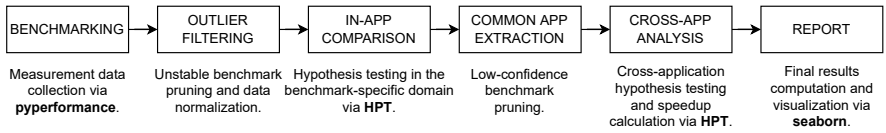


Figure 2.1: The method followed to produce this Chapter's results.

2.2 Method

Figure 2.1 summarizes the general flow of our work. We used `pyperformance`[10] for data collection and the HPT method [19] for statistical performance analysis.

`pyperformance` is the official benchmark suite of the Python community. It contains a variety of real-world applications, which is a necessity when studying DSA (synthetic benchmarks have long been known to be unreliable in characterizing allocator performance [121]). `pyperformance` also offers utilities for the reproducibility and stability of results, like commands for system tuning and compiling isolated versions of CPython, test calibration and warmup, as well as automatic identification of unstable benchmarks. We used `pyperformance` to collect raw benchmark data, and filter unstable applications out (first and second boxes in Figure 2.1).

HPT is a statistical analysis method for reliable performance comparison of systems. It is integrated in the PARSEC benchmark suite [11]. HPT employs hypothesis testing in order to answer questions of the form "X has better/worse performance than Y with a confidence of Z". It also computes quantitative, single-number speedup[11] comparisons (again, including the respective confidence)[12]. We used HPT to process the benchmark data recorded by `pyperformance` and derive fair, informed answers to the research questions stated in Section 2.1.1. This constitutes boxes 3-5 in Figure 2.1.

Our method rests upon the fact that HPT allows *cross-application* deductions stemming from application-specific data[13]. The flow is repeated for each metric of interest (execution time, memory footprint, energy consumption):

[10] https://github.com/python/pyperformance

[11] The term "speedup" normally hints toward improvement in execution time. For this Chapter, we extend the term's semantics so as to include memory footprint and energy consumption as well. Thus a speedup of $1.2x$ should be interpreted as achieving 1.2 times less execution time, memory footprint, or energy consumption with respect to some baseline.

[12] According to the authors of HPT, merely relying on the geometric mean for summarizing computer performance is problematic [19].

[13] The curious reader is encouraged to consult [19] for a complete treatment of why this is feasible.

- **Benchmarking:** each application is calibrated and a warmup value is computed and discarded. 60 values per application are then measured. A suite consisting of B benchmarks thus produces $60 \cdot B$ values. All this work is conducted by `pyperformance`. This step is repeated for all the DSA configurations that we want to compare. A dataset corresponding to N configurations contains $60 \cdot B \cdot N$ values.

- **Outlier filtering:** `pyperformance` prints warnings for potentially unreliable benchmarks, if they include outlier values different than the mean by a multiple of the standard deviation. We discard such cases and normalize the remaining ones as speedups over a reference configuration. We propagate forward the greatest subset of benchmarks common across all configurations.

- **In-app comparison:** the qualitative aspect of HPT is employed for this step. Each candidate configuration C is compared to the reference configuration R over each benchmark T. The comparison tests the hypothesis "C has better performance than R on benchmark T". The result is summarized as a confidence value. If this confidence is lower than a predefined threshold (e.g. 95%) the comparison is discarded.

- **Common app extraction:** not all (C, R) pairs end up having identical sets of benchmarks with high-confidence comparisons. To mitigate this, we again identify the greatest subset of common benchmarks. We end up with a dataset that, for all configurations, (i) is stable and (ii) guarantees statistically significant comparisons.

- **Cross-app analysis:** in-app comparison is now conducted for the set of stable benchmarks. The result is used as input to a quantitative computation: each candidate configuration C is assigned a cross-application speedup S over the reference configuration R, along with a confidence value as usual. S has three possible types of values:
  - a **floating point** number denoting actual speedup, which states that C performs better than R across all benchmarks with high confidence. The speedup is a lower bound on the expected impact to an arbitrary application's performance[14].
  - an "**MI**" placeholder denoting Marginal Improvement. The candidate configuration C might improve an application's performance, but to a negligible degree. Cases may exist where C performs worse than the reference. No horizontal conclusion should be drawn.

[14] No speedup value stands on its own, but must be co-interpreted with the respective confidence value.

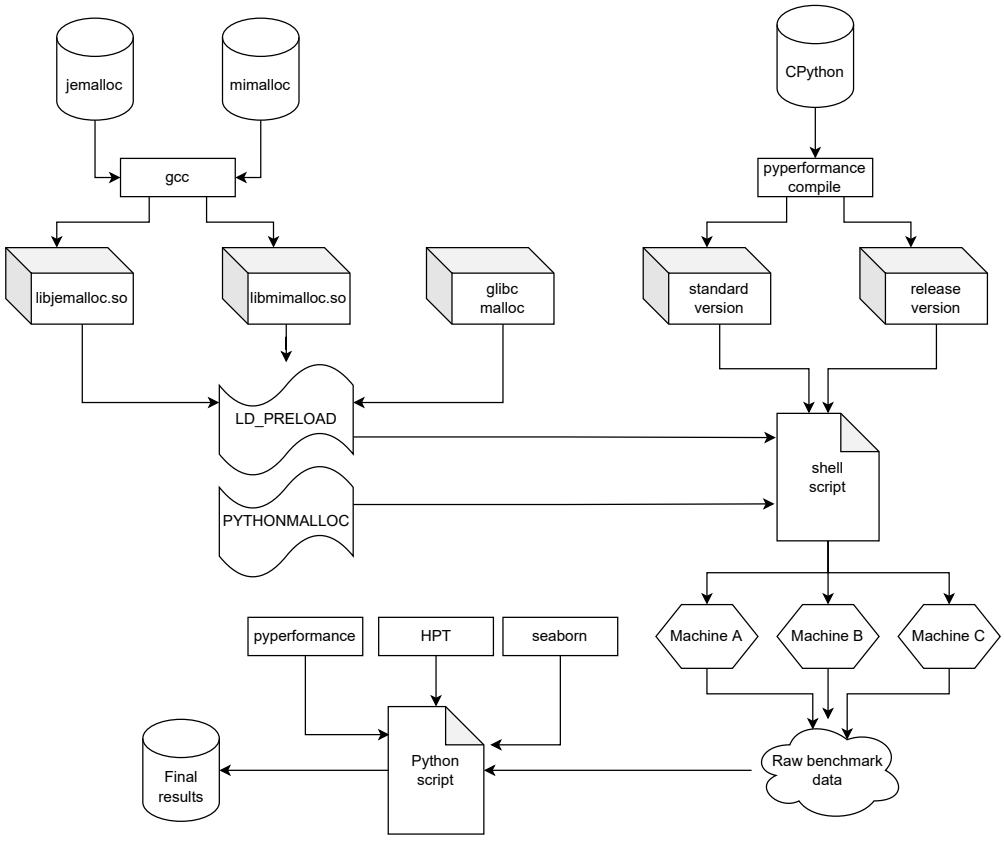


Figure 2.2: Experimental setup summary. Note the different symbols for: trusted data (cylinders), software operations (rectangles), scripts, environment variables (flags), binary files (3D rectangles), raw data (cloud), machines (hexagons).

– a "**PD**" placeholder denoting Performance Degradation. This implies that C will cause worse performance for an arbitrary application compared to the reference configuration.

2.3 Experimental Setup

Figure 2.2 encapsulates our tools, experiment workflow and results generation process. In the next few paragraphs, we describe it in further detail. A

Table 2.1: Hardware and software used for the study.

Component	Machine A	Machine B	Machine C
Type	Server	Embedded SoC	PC
Processor	8x Intel Core i7-6700	4x ARMv8 Processor rev 1	12x Intel Core i7-8750H
Clock frequency	3.4 GHz	1.5 GHz	2.2 GHz
System memory	32 GiB	4 GiB	16 GiB
Operating system	Ubuntu 20.04.4 LTS	Ubuntu 16.04.7 LTS	Ubuntu 20.04.3 LTS
Linux kernel version	5.13.0-35-generic	4.4.38-tegra	5.13.0-37-generic
GNU glibc version	2.31	2.23	2.31
Energy interface	Intel RAPL [47, 22]	N/A	Intel RAPL
CPython	3.10.2		
pyperformance	1.0.4 (modified)		
pyperf	2.3.1 (modified)		
mimalloc [76]	2.0.5		
jemalloc [25]	5.2.1		

comprehensive list of materials can be found at Table 2.1. With regard to the benchmarked applications, we mention $\sim \frac{1}{3}$ of them[15] in Table 2.2. Our main method, as described analytically in Section A.2, is implemented by a Python script which processes the raw benchmarking data offline.

We are interested in cross-application conclusions on the impact of DSA on CPython's performance. We define **conclusions** as answers to our research questions (Section 2.1.1). We define **performance** as the set of execution time, memory footprint and energy consumption (T, M, E). We used the default version of the `pyperformance run` command to measure execution time, and the `-track-memory` option for memory footprint. We modified `pyperformance` so as to report energy consumption readings in Linux machines which feature Intel's RAPL [47] power-capping interface. E is the sum of core and DRAM energy consumption, E_C and E_M. According to RAPL's documentation, the core part includes caches and the MMU. Note that $E_C + E_M < E_T$ if E_T is the total energy consumption of the platform. We define **impact** as the speedup S achieved by configuration C against a reference measurement R. Formally, $S = \frac{R}{C}$. This is also the metric used by our main statistical analysis tool, the HPT method [19].

[15] For the full catalogue, please consult https://pyperformance.readthedocs.io/benchmarks.html.

2.3.1 PGO, LTO sensitivity

Profile-guided and link-time optimizations (PGO, LTO) are available when compiling the CPython runtime. We want to investigate how these affect the performance impact caused by DSA–a sensitivity check between two extremes: a "standard" version which is the one built by default, and a "release" one that uses the `-enable-optimizations` and `-with-lto` options to enable PGO and LTO.

2.3.2 Configuration points

CPython DSA may be configured with two degrees of freedom:

- enabling/disabling the use of CPython's internal allocator `pymalloc`. When enabled, it is invoked for request sizes up to 512 bytes. Controlled via the `PYTHONMALLOC` environment variable[16]. Enabled by default.
- selecting the `malloc` implementation which CPython invokes when requesting memory from the operating system. Controlled via the `LD_PRELOAD`[17] trick. The default one is the system's allocator, which is normally `glibc` in GNU/Linux-running machines.

Thus N allocators produce $2 \cdot N$ candidate configuration points. In the case of this Chapter, $N = 3$ [`malloc` (`glibc`), `mimalloc`, `jemalloc`] and as a result we have 6 configuration points available. The aforementioned allocators were selected as popular representatives of the state of the art in DSA. Our process can be extended to other allocator libraries with minimal effort.

We refer to the `pymalloc`-enabled, `malloc`-linked configuration as reference, since it is the default setting.

2.3.3 Benchmarking script

Batches of data are collected via a shell script which repeatedly executes `pyperformance run` commands[18], each time for a different configuration point. When available, `pyperf system tune` is used prior to data collection for ensuring more stable measurements. Raw benchmarking measurements are

[16] https://docs.python.org/3/c-api/memory.html
[17] https://man7.org/linux/man-pages/man8/ld.so.8.html
[18] https://pyperformance.readthedocs.io/usage.html

Table 2.2: Some of the benchmarked applications.

Name	Description
chameleon	HTML/XML template engine.
django_template	High-level Python web framework.
dulwich	Implementation of the Git file formats and protocols.
fannkuch	From the Computer Language Benchmarks Game.
float	Artificial, floating point-heavy benchmark originally used by Factor.
genshi	Library for parsing, generating, and processing HTML, XML or other textual content..
html5lib	Library for parsing HTML.
json	API to convert in-memory Python objects to a serialized representation known as JavaScript Object Notation (JSON) and vice-versa.
pathlib	Tests the performance of operations of the pathlib module of the standard library.
pickle	Uses the cPickle module to pickle a variety of datasets.
regex_compile	Stresses the performance of Python's regex compiler, rather than the regex execution speed.
richards	The classic Python Richards benchmark. Based on a Java version.
spectral_norm	MathWorld: "Hundred-Dollar, Hundred-Digit Challenge Problems", Challenge #3.
telco	Benchmark for measuring the performance of decimal calculations. From the Computer Language Benchmarks Game.
tornado_http	Web framework and asynchronous networking library, originally developed at FriendFeed.

saved in compressed JSON format (the resulting files correspond to the arrow leaving the `BENCHMARKING` box in Figure 2.1).

2.3.4 Platform independence

Our experiments are conducted on three platforms: a server-class workstation, an embedded SoC and a laptop PC. In each case, we follow the method described in Section A.2 to compute the cross-application speedups of all configuration points on all performance metrics.

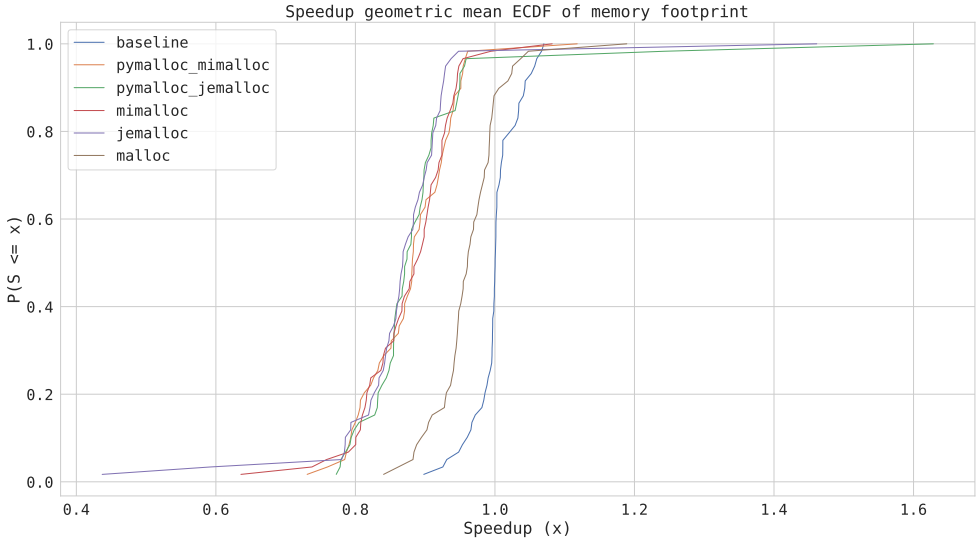


Figure 2.3: ECDF of speedup over memory footprint (embedded SoC, release version). The baseline configuration outperforms all alternatives in almost all cases. This figure partly supports our first finding according to which CPython's default DSA is optimized for memory footprint. See Table 2.5 for details.

If (i) similar speedups are computed for the same configuration on different platforms and (ii) this holds true for both the standard and release versions (Section 2.3.1), we may consider our experiments platform-independent. This test relies on mere common sense and must not be taken for a formal method; its validity lies in the multiplicity of the tested machines and the reliability of our analysis method (Section A.2).

2.4 Results

This section presents our study's findings. Before we proceed, however, we shall do our best to accustom the reader's intuition with our results' format. Let us thus map the research questions stated in Section 2.1.1 to suiting structures:

I. *To what extent can a CPython application's performance with respect to execution time, memory footprint and energy consumption be improved by*

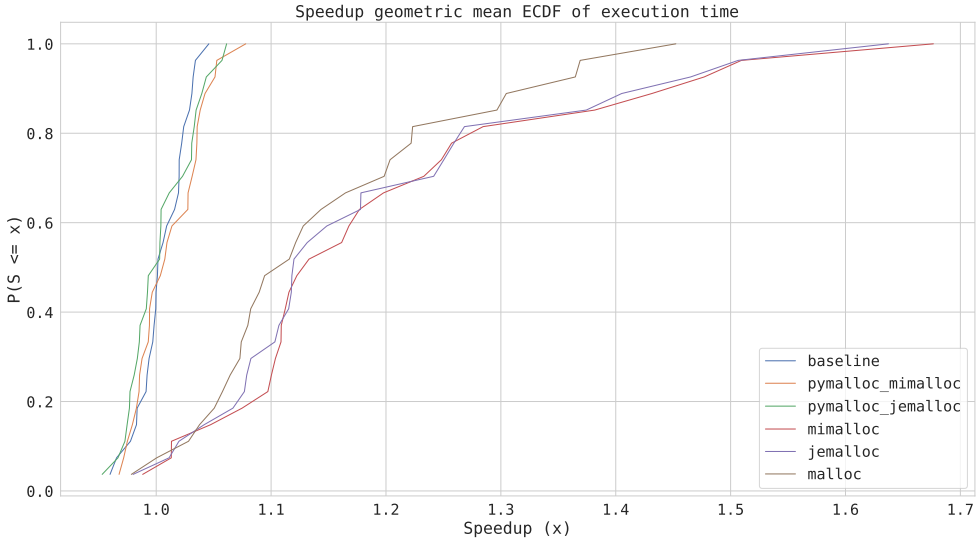


Figure 2.4: ECDF of speedup over execution time (embedded SoC, standard version). All DSA configurations that make exclusive use of some external allocator outperform the ones employing CPython's `pymalloc`. This partly supports our finding that in builds without PGO and LTO, discarding `pymalloc` almost always improves performance. See Table 2.5 for details.

modifying the runtime's DSA configuration? We need to reason about an arbitrary application (not necessarily one included in the benchmark suite). As explained in Section A.2, the HPT method computes cross-application speedups and the respective confidence values. We thus create a table of $(speedup, confidence)$ tuples with configurations as rows and performance metrics as columns, like Table 2.3.

II. *How much do optimizations in the runtime itself affect the expected improvement?* We create tables as the one described above for both the standard and release CPython builds (Tables 2.3, 2.4, 2.5).

III. *Are performance improvements sensitive to application-specific characteristics?* We summarize a configuration's performance in a single benchmark as the geometric mean of all measured speedups. We attain visual answers to this

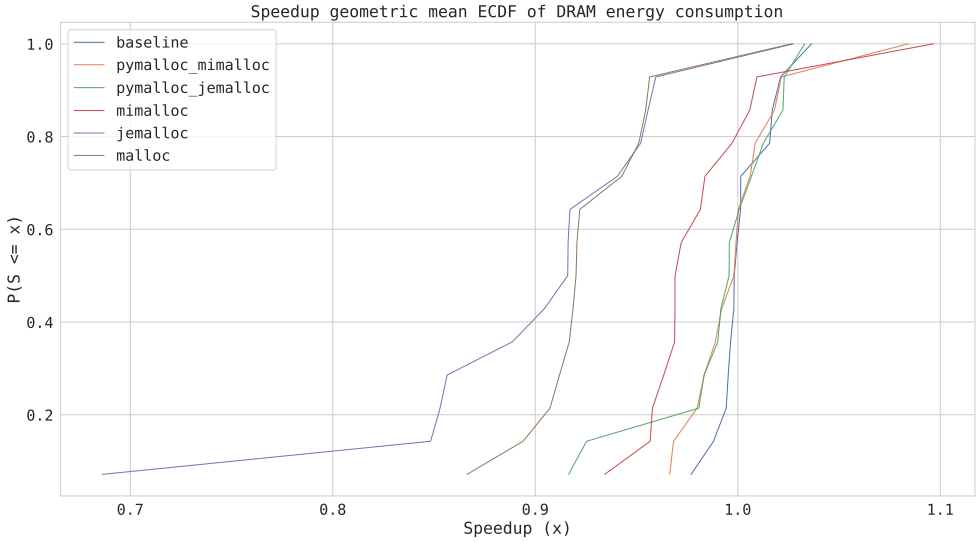


Figure 2.5: ECDF of speedup over DRAM energy consumption (server-class workstation, release version). An horizontal solution is impossible to find. Impact against the baseline is marginal at best. This partly supports our finding that the energy consumption of a PGO-LTO optimized CPython runtime is very hard to improve upon, and is sensitive to application-specific characteristics. See Table 2.3 for details.

question by printing the ECDF of geometric mean speedups (Figures 2.3-2.5)[19]; if variability exists in the best-performing allocator setup per benchmark, the answer is positive.

2.5 Discussion

We now proceed with the findings extracted from the collected data. We shall refer to the non-optimized CPython build as "standard" and to the PGO-LTO

[19]A previous footnote mentions the inadequacy of performance summarization via the geometric mean. It refers, however, to cross-application results. In the present case, we are interested in application-specific speedups, which the geometric mean is normally used to summarize. Note that the harmonic mean could prove closer to the ground truth in special circumstances [53].

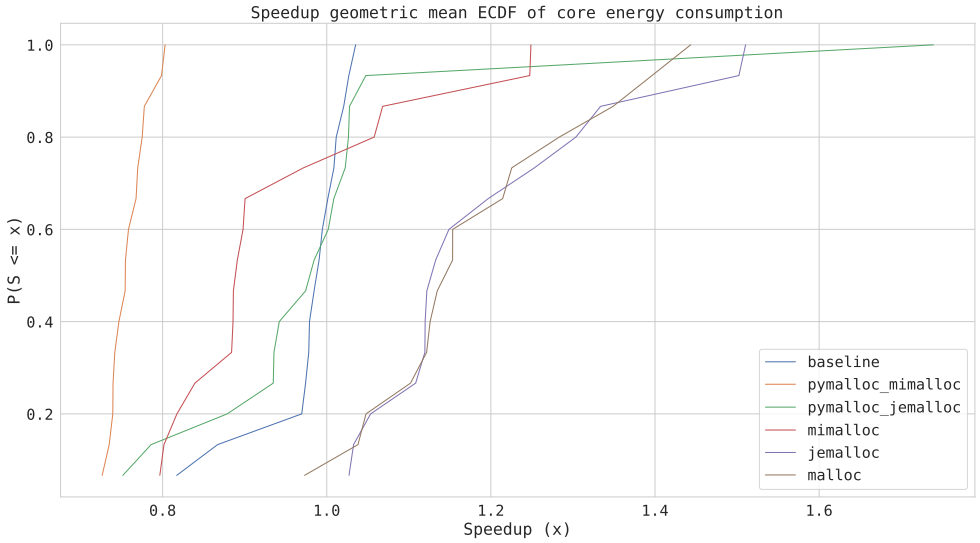


Figure 2.6: ECDF of speedup over core energy consumption (laptop PC, standard version). The configurations featuring exclusive use of `jemalloc` and `malloc` yield significant improvements across most cases. This figure partially supports our finding that cross-application performance enhancement can be achieved in standard builds by discarding CPython's `pymalloc`. It also shows that core energy consumption can at times be extremely counterintuitive, as happens in the case of `pymalloc_jemalloc`. See Table 2.4 for details.

one as "release". Although we display ECDF graphs for all three machines and both types of CPython, not enough space exists for including everything. The complete code and data can be found at our accompanying repository[20]. For now, we limit ourselves to representative cases that partially support our findings and reveal the broadest possible area of our experiments' space.

Degree of influence: modifying CPython's DSA has considerable impact when using the standard build (see "Standard" sections of Tables 2.3, 2.4 and 2.5 as well as Figures 2.4 and 2.6). Particular gains can surpass $1.6x$. To the contrary, performance is very hard to improve in the case of a release build–yet if constraints are tight, marginal improvements could be gained for specific applications (see for example Figure 2.5).

[20]https://github.com/cappadokes/cpythondsa

External allocator optimality: in the standard case performance across all metrics of interest will be improved by disposing of `pymalloc` and using an external allocator for all request sizes (Tables 2.3-2.4, Figures 2.4 and 2.6). This is possibly owed to the fact that a standard runtime introduces substantial overheads compared to native allocator code. An unresolved question here is why memory footprint is also better without `pymalloc`. We suspect but have not investigated a potential impact of PGO and LTO themselves on the runtime's final memory layout.

Footprint optimality: if a release CPython is employed the default configuration (`pymalloc`+`glibc`) will achieve the best memory footprint with very high certainty (e.g. Figure 2.3). If no other metrics are of interest, all alternatives should be avoided.

Platform independence: execution time and memory footprint impacts for all configurations in both standard and release builds are very similar across all three tested machines, as shown in Tables 2.3, 2.5 and 2.4. We consider this a very strong indicator that conclusions involving these two metrics are *platform-independent*.

Energy complexity: a weaker statement on platform indpendence can be made about DRAM energy consumption, since it is supported by two out of three machine datasets (Tables 2.3 and 2.4); the only outlier here is `pymalloc_jemalloc` in the release case. Core energy consumption on the other hand seems to be tied to each platform's microarchitecture (Tables 2.3 and 2.4, Figure 2.6). Decisions involving it should always be driven by extensive profiling and careful study. Both DRAM and core energy are very difficult to improve in the release case[21].

2.6 Limitations

The allocators used for our study are not the only ones available–though they are popular enough to represent the state of the art. Options like Hoard [9], `supermalloc` [64] and others should be evaluated too for completeness.

We use Intel's RAPL tool [22] for making energy measurements. RAPL does not report true energy consumption, but is rather a hardware model that has been shown to possess adequate accuracy [24]. There do exist methodologies for

[21] Recall that "core" energy includes cache memories and the MMU apart from the actual processor cores. Access to direct measurements from these subsystems would be very interesting for the scope of this Chapter, but the RAPL implementations we used did not expose such an option.

Table 2.3: Cross-application speedup and confidence values from the server experiment. MI stands for Marginal Improvement, PD for Performance Degradation. Note that MI results have the lowest confidence, which signifies to sensitivity application-specific characteristics. Also note that the reported speedups are lower bounds on the expected performance improvements. It is natural for cases like Figure 2.6 to contain benchmark-specific speedups that are higher than their cross-application counterparts.

Configuration	Execution time	Memory footprint	Core energy	DRAM energy
Standard				
pymalloc_mimalloc	MI (58%)	PD (100%)	PD (100%)	MI (27%)
pymalloc_jemalloc	MI (48%)	PD (100%)	PD (100%)	MI (15%)
mimalloc	**1.125** (93.96%)	**1.035** (92.98%)	**1.135** (93.2%)	**1.115** (90.31%)
jemalloc	**1.105** (92.73%)	**1.075** (93.41%)	**1.09** (93.32%)	**1.1** (94.97%)
malloc	**1.115** (91.29%)	**1.11** (89.79%)	**1.12** (89.27%)	**1.1** (94.8%)
Release				
pymalloc_mimalloc	MI (35%)	**PD** (100%)	MI (45%)	MI (21%)
pymalloc_jemalloc	MI (21%)	**PD** (100%)	PD (99.58%)	PD (95.44%)
mimalloc	PD (99.99%)	**PD** (100%)	MI (11%)	PD (98.84%)
jemalloc	PD (100%)	**PD** (100%)	PD (99.95%)	PD (99.99%)
malloc	PD (100%)	**PD** (100%)	PD (99.99%)	PD (99.99%)

ensuring minimal error when using RAPL to measure the energy consumption of short code paths [47]; we did not implement them due to lack of time. As a result, the subsets of common stable benchmarks across configurations for core and DRAM energy were smaller than the ones for execution time and memory footprint. Future work should focus on producing more stable measurements across all metrics.

We showed in our findings that, particularly for release builds of CPython, performance improvements in some respects are possible yet marginal and sensitive to application-specific characteristics. Our work devotes no effort on actually defining these characteristics; an idea we did not manage to realize is to integrate analytical heap profiling to `pyperformance` and `pyperf`. Even if we had done this, however, more complex methods than HPT should be employed to analyze and categorize the collected profiles since heap behavior cannot be summarized with a single number. Static analysis methods are another idea. Future research must consider both of these routes.

Table 2.4: Cross-application speedup and confidence values from the PC experiment. All comments on Table 2.3 apply here as well.

Configuration	Execution time	Memory footprint	Core energy	DRAM energy
Standard				
pymalloc_mimalloc	MI (82%)	PD (100%)	PD (99.95%)	MI (8%)
pymalloc_jemalloc	MI (7%)	PD (100%)	MI (18%)	MI (10%)
mimalloc	**1.125** (93.97%)	**1.03** (94.81%)	MI (10%)	**1.12** (93.55%)
jemalloc	**1.115** (99.3%)	**1.03** (89.65%)	**1.11** (94.21%)	**1.11** (93.55%)
malloc	**1.115** (91.39%)	**1.11** (90.96%)	**1.11** (94.21%)	**1.11** (93.55%)
Release				
pymalloc_mimalloc	MI (66%)	**PD** (100%)	MI (17%)	MI (30%)
pymalloc_jemalloc	MI (89%)	**PD** (100%)	**1.005** (83.71%)	MI (56%)
mimalloc	PD (99.99%)	**PD** (100%)	MI (7%)	PD (99.76%)
jemalloc	PD (100%)	**PD** (100%)	MI (99.76%)	PD (99.98%)
malloc	PD (100%)	**PD** (100%)	PD (100%)	PD (99.99%)

Table 2.5: Cross-application speedup and confidence values from the embedded SoC. Energy measurements not available for this platform. The N/A entries denote corrupt data which we could not refine.

Configuration	Execution time	Memory footprint
Standard		
pymalloc_mimalloc	MI (84%)	N/A
pymalloc_jemalloc	MI (19%)	N/A
mimalloc	**1.12** (94.66%)	N/A
jemalloc	**1.11** (94.66%)	N/A
malloc	**1.095** (93.13%)	N/A
Release		
pymalloc_mimalloc	**1.01** (76.09%)	**PD** (100%)
pymalloc_jemalloc	MI (93%)	**PD** (100%)
mimalloc	PD (99.87%)	**PD** (100%)
jemalloc	PD (99.99%)	**PD** (100%)
malloc	PD (100%)	**PD** (100%)

2.7 Conclusions

This work explored the impact of configuring CPython's dynamic storage allocation mechanism on execution time, memory footprint and energy consumption. It is motivated by Python's wide adoption in systems spanning from the edge to the server domain.

We used and extended the official benchmark suite of the language, `pyperformance`, to collect our measurements. We analyzed the data with HPT [19], a statistically rigorous method for making cross-application deductions.

According to our findings, the performance of standard CPython can be improved in all three performance aspects by exclusively using an external allocator (e.g. `jemalloc`) for all request sizes. Gains can surpass $1.6x$. Moreover, a runtime built with PGO and LTO provides optimal memory footprint out of the box. Energy consumption in the PGO-LTO case can only be improved marginally, and the suiting DSA configuration is sensitive to application-specific characteristics.

Our experiments took place in three different platforms. We show strong evidence that DSA's impact on execution time and memory footprint is platform-independent. Slightly weaker evidence for the independence of DRAM energy consumption is also provided. As regards core energy consumption (processor, caches, MMU) no similar statement can be made. To the best of our knowledge, this is the first rigorous study of the relationship between CPython and DSA.

Chapter 3

Beyond RSS: Towards Intelligent Dynamic Memory Management

This Chapter is a verbatim copy of the author's publication cited below:

Lamprakos, C. P., Xydis, S., Kourzanov, P., Perumkunnil, M., Catthoor, F., and Soudris, D. Beyond rss: Towards intelligent dynamic memory management (work in progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (New York, NY, USA, 2023), MPLR 2023, Association for Computing Machinery, p. 158–164

The main goal of dynamic memory allocators is to minimize memory fragmentation. Fragmentation stems from the interaction between workload behavior and allocator policy. There are, however, no works systematically capturing said interaction. We view this gap as responsible for the absence of a standardized, quantitative fragmentation metric, the lack of workload dynamic memory behavior characterization techniques, and the absence of a standardized benchmark suite targeting dynamic memory allocation. Such shortcomings are profoundly asymmetric to the operation's ubiquity.

This Chapter presents a trace-based simulation methodology for constructing

representations of workload-allocator interaction. We use 2DBP as our foundation. 2DBP algorithms minimize their products' makespan, but virtual memory systems employing demand paging deem such a criterion inappropriate. We see an allocator's placement decisions as a solution to a 2DBP instance, optimizing some unknown criterion particular to that allocator's policy. Our end product is a data structure by design concerned with events residing entirely in virtual memory; no information on memory accesses, indexing costs or any other factor is kept.

We bootstrap our contribution's utility by exploring its relationship to maximum RSS. Our baseline is the assumption that less fragmentation amounts to smaller peak RSS. We thus define a fragmentation metric in the 2DBP substrate and compute it for both single- and multi-threaded workloads linked to 7 modern allocators. We also measure peak RSS for the resulting pairs. Our metric exhibits a monotonic relationship with memory footprint 94% of the time, as inferred via two-tailed statistical hypothesis testing with at least 99% confidence.

3.1 Introduction

In their 1995 survey, Wilson et al. contributed a comprehensive taxonomy and a grounded critique of dynamic storage[1] allocation (DSA) [121], noting the inherent difficulty in defining fragmentation, the inadequacy of basing designs on synthetic workloads, and the lack of novelty in new allocator policies. To this day, we have not converged to a single, measurable definition of fragmentation [84], neither do we possess a method for workload characterization–despite the fact that program behavior partly controls fragmentation.

Most noticeably, there is no standardized memory allocation benchmark suite. Motivation sections often adopt synthetic test cases [79] even though we know such practices to be inadequate. Applications used for evaluation are selected on intuitive grounds of being "dynamic enough". Certain classes, such as database and web browsing workloads, are preferred over others with no proper justification. Worse, "internal" workloads are at times used [81], obstructing transparency and reproducibility. We claim that hidden costs, such as scarce physical memory contiguity [123], are imposed to systems from the aforementioned gaps, and amplified by the ubiquitous nature of DSA.

This Chapter introduces a systematic methodology for representing workload-allocator interaction as instances of two-dimensional rectangular bin packing (2DBP) [21, 13]. To conclude whether any information of practical value is

[1]We use "storage" instead of "memory" as a tribute to Paul R. Wilson et al. [121] The matter at hand is non-moving virtual memory allocation.

captured, we explore our product's relationship to maximum RSS. We define fragmentation as the ratio between gaps and used memory in the 2DBP space, and measure it for 34 real workloads linked to 7 modern allocators. 94% of the time, 2DBP-based fragmentation and maximum RSS exhibit a monotonic relationship–as found by conducting statistical hypothesis tests with a significance value of at least 99%. Our contributions can thus be summarized as:

- a novel perspective emphasizing the need for a principled study of workload-allocator interaction
- a methodology for constructing 2DBP representations of arbitrary workloads and non-moving allocators
- a first empirical study of 2DBP's informational content
- a novel definition of memory fragmentation
- a discussion on our results' implications for DSA, motivating future research

Section A.1 elaborates on our representation and 2DBP-based fragmentation. Section A.2 describes the mechanisms implemented to actualize our methodology. We present our results in Section 3.4 and discuss their implications in Section 4.7. Related work is presented in Section 3.5, and Section A.4 closes the main text with an overview of our conclusions.

3.2 Background

Allocators receive a series of requests from the programs they are linked to. Two main request types exist: *allocation* of n bytes and *deallocation* of a previously allocated object. The requests' creator may range from application developers, as happens in C, to garbage-collected language runtimes (CPython), to compiler-injected directives (Rust).

Real allocation requests come in several variations. A program may need specifically aligned objects, or objects initialized as a zero-valued array. It may even ask for an object to be resized. Upon successful allocation, a pointer to the newly acquired memory is returned. Deallocation requests are straightforward. The program informs the allocator via a previously obtained memory pointer that it does not need the corresponding object any more.

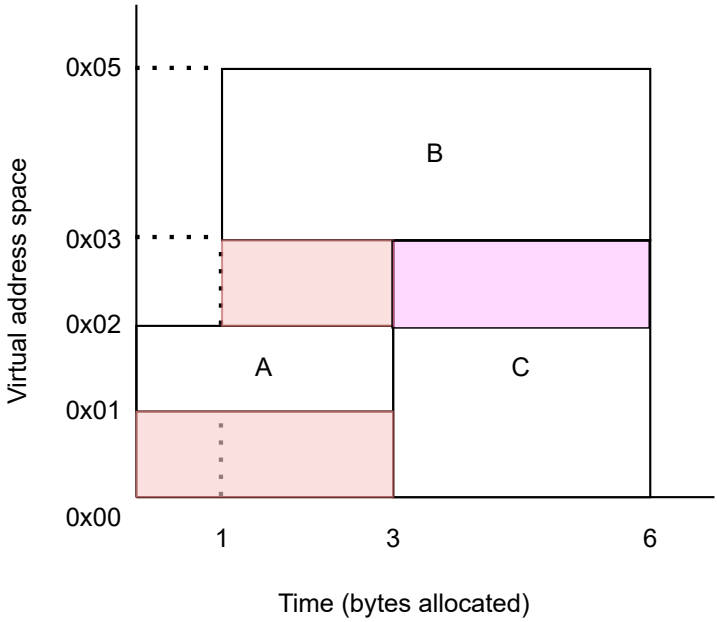


Figure 3.1: A simple 2DBP example. External fragmentation is marked in red; internal in magenta (assume that the allocator decided to put object `C` in the 3-byte size class, despite the program requesting only 2 bytes).

An allocator's decisions on object placement and free memory management form its *policy*. On the program's side, the distribution of allocation sizes requested as well as the particular sequence of requests jointly form its *behavior*. The goal of a good policy is to minimize *fragmentation*[2], which means to waste minimal amounts of extra memory beyond what the program requested. Two types of fragmentation exist: *internal* fragmentation treats wasted memory within objects (i.e., returning more bytes than requested); *external* fragmentation focuses between objects (e.g., putting objects that die together in non-consecutive places). Both types are functions of the *interaction* between allocator policy and program behavior [121]. Several definitions have been proposed over the years [57, 84, 9].

A 2DBP instance comprises a series of *unplaced* objects in the form of $(start, end, height)$ tuples. An acceptable *solution* to 2DBP is a placement

[2]We remain aware of the complex memory/performance tradeoffs faced by allocator designers. We focus on memory explicitly because (i) viewing DSA from first principles automatically makes memory a first-class citizen and (ii) most research over the past decades targets performance already [78].

with no overlapping objects. For the purposes of our Chapter there is no need to distinguish between placed and unplaced objects, so with the term "2DBP" we refer both to the requests *and* the allocator's responses to each request (all objects are already placed by the time we depict them). The concepts involved are best described by example. Let us consider the below requests sequence:

1. `A = malloc(1)`
2. `B = malloc(2)`
3. `free(A)`
4. `C = malloc(2)`
5. `free(B)`
6. `free(C)`

Figure 3.1 combines these requests with an imaginary allocator's responses, placing object A at virtual address 0x01, object B at 0x03 and object C at 0x00. Each object is formed by pairing two requests, one for allocation and one for deallocation, involving the same memory pointer (stored in A/B/C variables in this example). The figure's horizontal axis measures time in allocated bytes. Time progresses forward after each allocation request, and remains unaltered after each deallocation request.

Normally 2DBP algorithms optimize a placement's make-span, meaning the total address range used (in Figure 3.1 the makespan equals to 5). We have already emphasized that in the scope of this Chapter, the allocators are the ones producing the placements; we are merely recording their decisions *as if* they were solving a 2DBP problem. We cannot know the precise criterion that each allocator optimizes, but it is probably not makespan; disjoint virtual pages may be mapped to contiguous physical ones and vice versa.

There is thus no point in restricting the range of virtual addresses used. There is quite a point, however, in restricting overall memory usage–or to minimize physical memory fragmentation. So the question is, in the context of the representation we are constructing, what could fragmentation look like? Our proposed answer is indicated by the three shaded rectangles in Figure 3.1. Recall that one description of fragmentation is "memory wastage"; the shaded areas are like gaps in a Tetris game. They represent segments which the allocator left unused, thus reserving higher addresses in order to handle all requests.

One might judge our formulation as too strict, since a non-moving allocator could not break object B in two and slide the left part down to cover the top

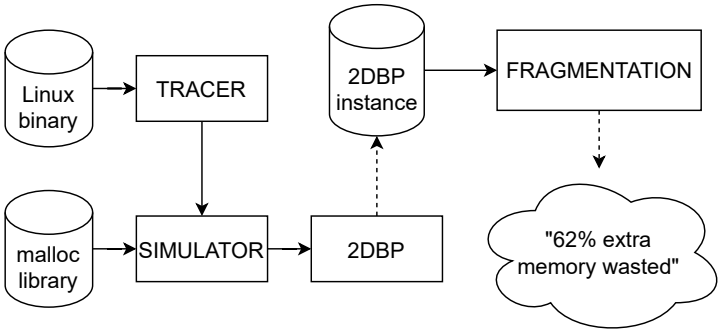


Figure 3.2: An overview of our method to produce 2DBP representations and to compute their fragmentation. Bold arrows are inputs and dotted arrows are outputs.

fragmented area. Two points must be raised here: fragmentation is partly defined by the program's behavior, and it thus makes sense for portions of it to be inevitable. Moreover, what matters most is 2DBP itself. Computations performed on it, fragmentation included, are secondary. This statement does not mean to devalue fragmentation as a phenomenon–such a stance would go against our own motivation. It just stresses the importance of *first* establishing a useful substrate. In our Chapter, fragmentation plays the crucial role of bootstrapping 2DBP in the sense of a 2DBP-derived signal correlating with the real world. But again, nothing else must be considered more primary than the representation itself.

3.3 Proposed Method

Our goal is to represent arbitrary pairs of Linux binaries and `malloc` implementations as 2DBP instances. An overview of our method is shown at Figure 3.2. Inspired by [121] and [57] we aimed for trace-based simulation.

3.3.1 2DBP construction

We log all of a program's calls to allocation functions. The resulting trace, along with the `malloc` implementation of interest, feeds our simulation module. The 2DBP component produces the final representation. Our architecture is

Table 3.1: Rules for unpacking request traces to `malloc` and `free` operations. `s` stands for "size", `p` for "pointer", `n` for "number", `a` for "alignment".

Original Operation	Transform
`malloc(s)`	`malloc(s)`
`free(p)`	`free(p)`
`calloc(s,n)`	`malloc(n*s)`
`realloc(p,s)`	`free(p); malloc(s)`
`posix_memalign(p,a,s)`	`malloc(s)`
`aligned_alloc(a,s)`	`malloc(s)`
`valloc(s)`	`malloc(s)`
`memalign(a,s)`	`malloc(s)`
`pvalloc(s)`	`malloc(s)`

modular to enable optimizations in each stage, since it must eventually handle realistic workload sizes.

Requests tracing

A reasonable question is why did we not leverage existing solutions such as `mtrace`[3], `heap- track`[4], or tracing capabilities built in `malloc` implementations. Our decision was driven by the below points:

- `mtrace` demands that the program be modified so as to initialize the tool, while access to the application source code may not be feasible in practice.
- `heaptrack` and similar alternatives are extra dependencies which the user may want to avoid.
- existing tracers impose larger overheads to store additional data, e.g., stack traces and call site addresses

Our tracer is required to be *complete*, catching allocations and deallocations all across the program's call stack. It must also be *non-intrusive*, that is to imply zero actions regarding code instrumentation and compilation. It finally needs to be *correct*: logged calls should belong to the traced program only, and not be polluted by dynamic memory operations of the tracer itself. To satisfy these requirements we target typical Linux processes forking no children. We also make use of several Linux and GNU utilities reported in the following

[3]https://linux.die.net/man/3/mtrace
[4]https://github.com/KDE/heaptrack

Table 3.2: Trace file structure. The `els_num` field is used for tracing `calloc`, which returns a number of elements, each element of a certain size. To facilitate the study of multi-threaded programs, we record the caller thread's ID in the `call_tid` field.

CSV Field	Request 1	Request 2	Request 3
req_type	`malloc`	`free`	`calloc`
in_address	`(nil)`	`0x55A`	`(nil)`
out_address	`0x55A`	`(nil)`	`0x63B`
el_size	`12`	`(nil)`	`128`
els_num	`1`	`(nil)`	`1000`
call_tid	`26`	`36`	`31`

paragraphs. Our mechanism is general enough to operate on any program in this context, from command line tools to application virtual machines.

The tracer is a shared library employing `dlsym`[5] to interpose calls to `malloc`, `free`, `calloc`, `realloc`, `posix_memalign`, `memalign`, `aligned_alloc`, `pvalloc` and `valloc`. These were selected according to GNU's guidelines on replacing `malloc`[6]. Beyond interposing the allocation interface, our tracer spawns a new process which writes the actual logs to a CSV file. The structure of the stored tracing data is shown at Table 3.2.

Placement simulation

2DBP perceives only two kinds of requests, namely allocation of n bytes and deallocation of occupied memory. But a real trace file may include operations with more complex semantics, such as `calloc`. We thus unpack all calls to combinations of the two elementary operations, `malloc` and `free`. The counterargument to address is the proposed unpacking's effect on original program behavior. A short yet concise answer is that if along our course we distorted program behavior more than we should, no connection with RSS would have been uncovered. The unpacking scheme is described in Table 3.1.

Policy simulation does not reproduce the original program's RSS waveform, since no memory access information is stored during the tracing stage. 2DBP lives entirely in virtual, not physical, memory. This works to our advantage, since it enables us to examine the extent to which events in virtual memory affect real-world performance.

[5] https://man7.org/linux/man-pages/man3/dlsym.3.html
[6] https://www.gnu.org/software/libc/manual/html_node/Replacing-malloc.html.

To record block sizes we use the values returned by `malloc-_usable_size`[7]. If the simulated allocator includes metadata in its block layout (like the GNU implementation does), this is also taken into account. Our decision to record *both* the amount of memory requested by the program *and* the final size of the block returned by the allocator incorporates a significant aspect of allocator policy, that is, the size classes that it uses. It also allows us to measure internal fragmentation. Memory *mappings* are consulted via the process-specific `/proc/[PID]/maps`[8] file. A good discussion of why modern allocators spawn memory mappings under the hood may be found on StackOverflow[9]. Thus our simulator must keep track of object traffic within said mappings if we want it to capture the complete picture. Hence another aspect of allocator design is captured. However, apart from the workload and the allocator, the OS *itself* is a major factor in the eventual memory footprint, and there are aspects of its interaction with the other two that are crucial and yet not captured. The most self-evident such aspect is memory access information. Thanks to the demand paging mechanism, we know that allocated memory is not mapped until the first time that the program attempts to access it. By omitting such information, we overestimate fragmentation because we consider memory to be mapped earlier than what it actually is.

Time is updated whenever a `malloc` request has been scanned. The final placement data is also structured as CSV.

3.3.2 Fragmentation

We define fragmentation as *the area of unused memory within occupied virtual pages, divided by the area of allocated memory.* We compute it across all M mappings spawned by a workload-allocator pair via Equation 3.1:

$$F_T = \frac{\sum_{i=1}^{M} F_{mi}}{\sum_{i=1}^{M} L_{mi}} \tag{3.1}$$

Recall that by design our representation captures virtual memory across time; that is, one can by traversing it track virtual pages getting occupied, emptied, or loaded with more allocated objects. The term F_{mi} is derived by summing the spatiotemporal areas of unused memory *belonging to occupied virtual pages*

[7] https://man7.org/linux/man-pages/man3/malloc_usable_size.3.html
[8] https://man7.org/linux/man-pages/man5/proc.5.html
[9] https://stackoverflow.com/questions/64029219/why-does-malloc-call-mmap-and-brk-interchangeably

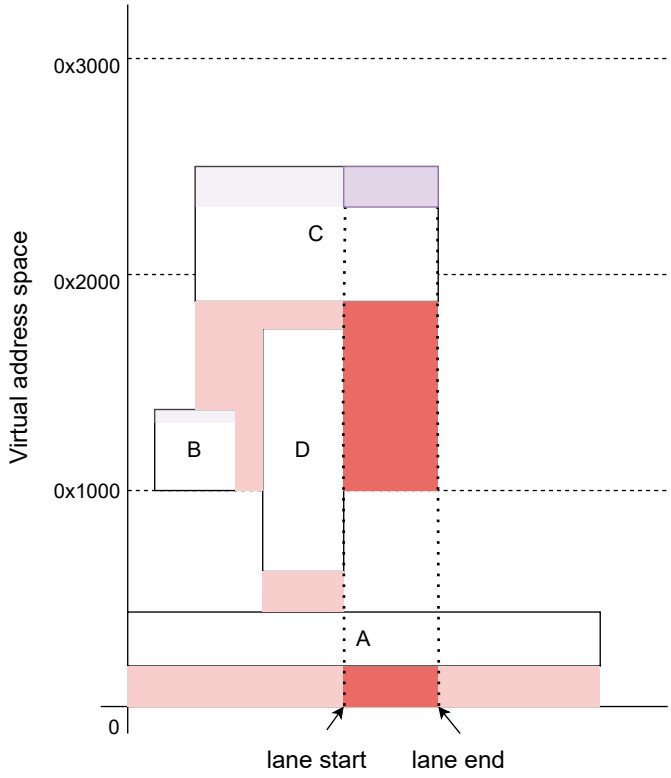


Figure 3.3: Gap identification algorithm. Axes are identical to those of Figure 3.1. Horizontal dashed lines are page boundaries. White rectangles are objects, i.e., allocated memory. Gaps contributing to external fragmentation are marked with red, internal with purple.

within each mapping. L_{mi} stands for total allocated memory–again, across time.

We illustrate our algorithm in Figure 3.3: gaps between and inside objects are shown as lightly and darkly shaded areas. Our plot is drawn *in medias res*–lightly shaded areas were and will be accounted for in previous and future iterations, while darkly shaded ones are captured by the present iteration. To this we focus. It involves a vertical slice that we call a *lane*. Lanes are delimited by object beginnings and endings. Within them nothing new happens; thus they can be traversed *vertically* for new gaps to be found.

Virtual page boundaries are drawn as horizontal dashed lines. We do not

allow gaps to cross those boundaries, since there is no guarantee of maintained contiguity between virtual and physical memory. Gaps must always have a same-page object as their ceiling. This puts more pressure on the allocator's placement decisions and discounts the effect of limitations it cannot overcome.

3.4 Evaluation

We have proposed a methodology that captures workload-allocator interaction. To evaluate our claim, a connection between our representation and a valuable physical memory-based measure must be made. We select maximum RSS as our target and assume that the cost of high fragmentation is most evident at the moment of highest memory usage [121], i.e., at peak RSS. If 2DBP actually captures workload-allocator interaction, then computing fragmentation on it yields a good approximation of real[10] fragmentation. Consequently, 2DBP-based fragmentation correlates with peak RSS *if and only if 2DBP as a whole is a valid representation.*

Allow us to further clarify our reasoning before proceeding. On the practitioner's side, the meaning of focusing on peak RSS is self-evident. Now one could ask why we went for an aggregate measure in the 2DBP plane: why not follow the same rationale and focus on the moment of *maximum* fragmentation? Earlier research has indeed suggested that we could do so [57]. A leap of intuition is required here, and we hope that the text helps the Reader do it. We want to compress as much information about workload-allocator interaction in our metric as possible. Decisions are made all across a program's lifetime, not only at its peak allocation/usage point in time. By aggregating fragmentation, our hope was to capture the effect of all those decisions (with all of the implied noise and probability of error). Whether such an expectation is reasonable remains to be backed or falsified by the experiments themselves. Nevertheless, by this decision our method remains aligned with our aforementioned desire for dynamically adaptive systems.

The correlation we are looking for is monotonically increasing; we expect higher fragmentation to cause higher peak RSS. We thus conduct two-tailed statistical hypothesis testing [7], the null hypothesis being that *2DBP-based fragmentation and peak RSS do not correlate monotonically.* Before proceeding to the results, let us elaborate a little more on our experiments' procedure.

First we traced all workloads with the mechanism described in Section 3.3.1. The simulator of Section 3.3.1 was then fed with trace-allocator pairs to collect

[10] Recall that the hardness and ambiguity of measuring real fragmentation was this Chapter's starting point.

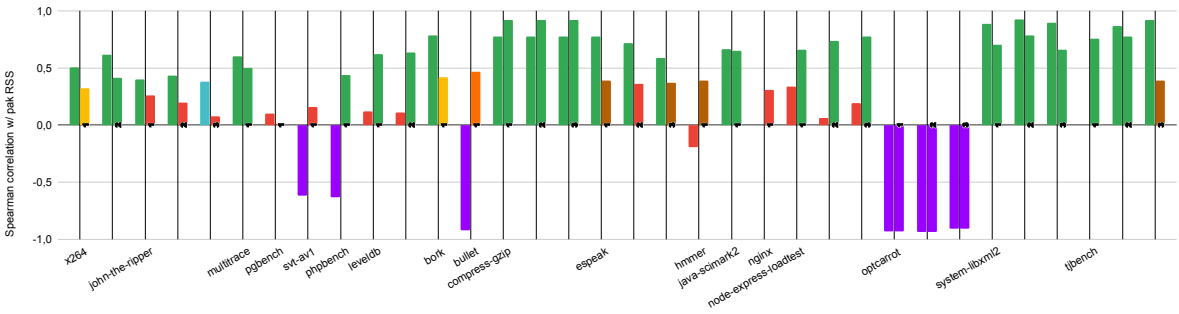


Figure 3.4: Hypothesis testing results. We studied 34 workloads in total. We associate 2 bars to each workload, i.e., Spearman correlation of external (left) and internal (right) fragmentation with peak RSS. Bar heights signify correlation strength, while colors signify confidence: 99.95% (green), 99.9% (cyan), 99.75% (orange), 99.5% (yellow), 99% (brown). Red bars validate the null hypothesis of no existing correlation. Purple bars are counterintuitive cases of *negative* monotonicity. 32 out of 34 workloads exhibit correlation between peak RSS and at least one type of fragmentation, with at least 99% probability that said correlation was not a matter of chance.

placement data, on top of which we measured fragmentation. In parallel, we executed each workload-allocator pair 10 times and measured peak RSS; each bar in Figure 5.2 stems from 70 data points. That way, we both take non-determinism into account, and reinforce the validity of the hypothesis testing procedure. Unlabeled bar pairs correspond to running the last-labeled application with different inputs/configurations, e.g., `x264` was run with 2 inputs, `multitrace` with 1, `system- libxml2` with 3 and so on. Last but not least, we computed workload-specific Spearman correlation coefficients for peak RSS and fragmentation and compared them to corresponding significance values of at least 99% confidence [6].

All experiments were run on a commodity x86_64 Ubuntu 20.04 machine with 16 GiB DRAM. All workloads are real applications from OpenBenchmarking.org[11] and include both single-threaded and multi-threaded programs. The allocators used were the GNU malloc implementation[12], jemalloc [25], mimalloc [76], tcmalloc [13], snmalloc [79], rpmalloc[14] and the Hoard allocator [9].

[11] https://openbenchmarking.org/
[12] https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html
[13] https://github.com/google/tcmalloc
[14] https://github.com/mjansson/rpmalloc

As can be seen on Figure 5.2, most of the time there is at least one type of fragmentation per workload which correlates with memory footprint across the allocators tested. This Chapter being a work-in-progress submission, we cannot elaborate further on the presented results; nevertheless, we consider them interesting enough to attract future research interest. In the following section we list some ideas on what said research could be concerned with.

3.5 Related Work and Comparison

Wilson et al. have written the seminal treatment on DSA and the central role of fragmentation [121]. Johnstone and Wilson conduct the first study of RSS-based fragmentation definitions [57]. Berger et al. show that modern allocators perform acceptably well with respect to RSS-based fragmentation [10]. Maas et al. propose a novel fragmentation definition incorporating chances of immediate memory reuse [84]. Powers et al. and Maas et al. contribute notably unorthodox ways to deal with fragmentation [106, 81].

On the theoretical side Robson has computed worst case fragmentation bounds for the best fit and first fit placement policies [109]. Optimal placement is reported as NP-complete by Garey and Johnson [**?**]. Chrobak and Ślusarek formulate it as a 2DBP instance [21]. Buchsbaum et al. develop the state-of-the-art ϵ-optimal algorithm for solving the general case with minimal makespan [13]. Given our focus on 2DBP, we do not mention other formulations such as graph coloring [60].

Tracing workload dynamic memory behavior correctly and efficiently has been tackled in the context of garbage collection research [49]. The closest real-world example of capturing workload-allocator interaction as bin packing comes from Maas et al [83]. 2DBP is there viewed as a useful tool for the specific case of ML compilers, where all dynamic memory requests are known in advance. With this Chapter we hope to convince the reader that the general case of DSA has much to benefit from 2DBP as well. The logical conclusion of using 2DBP has been explored by Lamprakos et al. [69]

3.6 Conclusion

This Chapter forms a connection between theoretical dynamic memory allocation and its real-world counterpart. It is motivated by a profound asymmetry between dynamic memory allocation's omnipresence and the scarcity of principled methods for understanding workload-allocator interaction. It describes a

mechanism for extracting representations of workload-allocator pairs in the form of two-dimensional bin packing, and then proposes a novel fragmentation definition built on top. Despite operating on entirely virtual, simulation-generated data, our measure correlates with the memory footprint of a variety of workloads. Our study serves as a first piece of empirical evidence towards adopting bin packing-based methods for dynamic memory allocation.

Chapter 4

Futureproof Static Memory Planning

This Chapter is a verbatim copy of the author's **submission** to ACM Transactions on Programming Languages and Systems, currently under review.

The NP-complete combinatorial optimization task of assigning offsets to a set of buffers with known sizes and lifetimes so as to minimize total memory usage is called dynamic storage allocation (DSA). Existing DSA implementations bypass the theoretical state-of-the-art algorithms in favor of either fast but wasteful heuristics, or memory-efficient approaches that do not scale beyond one thousand buffers. The "AI memory wall", combined with deep neural networks' static architecture, has reignited interest in DSA. We present `idealloc`, a low-fragmentation, high-performance DSA implementation designed for *million*-buffer instances. Evaluated on a novel suite of particularly hard benchmarks from several domains, `idealloc` ranks first against four production implementations in terms of a joint effectiveness/robustness criterion.

4.1 Introduction

Deep learning is causing significant shifts in professional and civilian life. Several technical challenges, however, remain open. For instance, there is a profound asymmetry between progress in compute capability and memory capacity/bandwidth [40]. This so-called "AI memory wall" has sparked substantial research and engineering efforts targeting the memory effectiveness of deep learning. The particular line of work in scope for this Chapter deals with assigning offsets to a set of buffers with known sizes and lifetimes in order to pack them in as small an address space as possible [90, 83, 73, 116, 2, 51, 77, 105, 125]. In deep learning such problems appear thanks to (i) neural networks' static architecture and (ii) hardware accelerators' physical memory contiguity.

Nevertheless, beyond providing motivation for what shall be presented, deep learning is not of the essence here. The problem is old and well-studied [34, 61, 60, 37, 38, 13]. It is known as *dynamic storage allocation* (DSA), a variation of two-dimensional bin packing. DSA has been proven NP-complete.

4.1.1 Against a Common Misunderstanding

Despite its name, DSA is a *static* problem, in the sense of having available all the information that it needs from the outset. "Dynamic storage allocation" has also been used for the dynamic variant (what `malloc` implementations deal with) [122, 107, 108], causing considerable confusion. We shall be using "DSA", "memory planning", "static offset assignment" and "static memory allocation" interchangeably in this text. In a similar vein, we will be referring to DSA implementations, i.e., programs solving DSA instances, as "allocators". Dynamic non-moving virtual memory allocators such as GNU's `malloc` are out of scope—we use "OS allocators" in the few times that we must mention them.

4.1.2 Motivation and Related Work

We are concerned with *real-world implementations* of DSA, their *effectiveness*, *efficiency* and *robustness* in the face of arbitrarily large inputs. Our founding assumption is that sooner or later, in deep learning or elsewhere, DSA instances comprising *millions* of buffers will emerge. For instance, large language models are already pushing compiler engineers to come up with ever more aggressive optimizations, yielding complex and massive memory allocation patterns in return [51, 43]. Another example is the Linux user applications domain, where

`malloc` traces are used for off-line analysis and/or optimization [72, 111, 70, 82, 97, 94].

Our main observation after surveying the SOTA was that allocators are bypassing the algorithms published in the DSA literature in favor of schemes that are simpler to implement. Alternatives can be sorted in two broad categories: heuristics [105, 73, 114, 77] and isomorphisms [90, 83, 112], e.g., integer linear programming, machine learning regression, simulated annealing, and hill-climb optimization. We ask what costs accompany circumventing the decades-old literature around an NP-complete problem for which one seeks a practical, general solution. The only way to find out *if* such costs exist would be to build an allocator informed by that literature, and then evaluate it rigorously against the SOTA. Hence `idealloc`, the allocator at this Chapter's center, was born. In terms of the heuristics/isomorphisms dichotomy, it is a *stochastic bootstrapped heuristic*.

A second observation was that apart from the micro-benchmarks published by the authors of `minimalloc`, a SOTA allocator [90], no DSA benchmark suites exist. We thus formed a novel set of benchmarks ranging from hundred- to half-a-million buffers and used it, along with the aforementioned micro-benchmarks, for evaluation. From a strict effectiveness-only perspective `idealloc` rarely beats all of its competition, comprising `minimalloc` and three other production allocators. But from a robustness and efficiency perspective that same competition (with one exception) rarely manages to even produce a solution in reasonable time. Under a joint ranking criterion incorporating both perspectives `idealloc` achieves top score.

4.1.3 Contributions

Along the course of designing, developing and testing `idealloc`, we gathered a multifaceted set of insights. On the algorithmic front, we identified and fixed several blind spots of the original theorems, published by Buchsbaum et al. in 2003 [13] [1]. We also devised a second set of algorithms, related not to the DSA core itself, but to forming a scalable infrastructure around it. On the benchmarks front, we collected a novel suite of challenging, large-scale inputs from domains such as Linux databases, parallel training of deep learning models, and distributed inference.

All in all, our contributions are:

[1] We have exchanged emails with the algorithm's original authors, who have validated that (i) transition from theory to practice always involves trickiness and (ii) there are no other known implementations of their work.

1. `idealloc`, a DSA implementation designed to handle inputs of arbitrary size and complexity
2. crucial theoretical extensions to the algorithms on which `idealloc` is based
3. various insights and techniques of general applicability to future DSA design tasks
4. the first rigorous evaluation of the DSA DSA

Section 4.2 provides background knowledge on DSA. Section 4.3 describes the core algorithm powering `idealloc`. The design of our allocator is exposed in detail in Section 4.5, and the experiments conducted for evaluating it are reported in Section 4.6. Section 4.7 discusses limitations and ideas for future work, and Section A.4 concludes our exposition.

4.2 Dynamic Storage Allocation

Rectangle packing [63] is the combinatorial optimization problem of placing rectangles of various widths and heights into arrangements where (i) no two rectangles overlap and (ii) the arrangement's enclosing rectangle has minimum height. Rectangles may move in two degrees of freedom (vertically or horizontally). This problem is NP-complete.

DSA is a constrained variation of rectangle packing. It owes its name to the interpretation of one dimension as available address space, and the other as time. Each rectangle encodes a pair of requests for the allocation and deallocation of some specific amount of memory at specific points in time. Allocators have no power over the timing of incoming requests, so the only degree of freedom they have is the spatial one. DSA is NP-complete

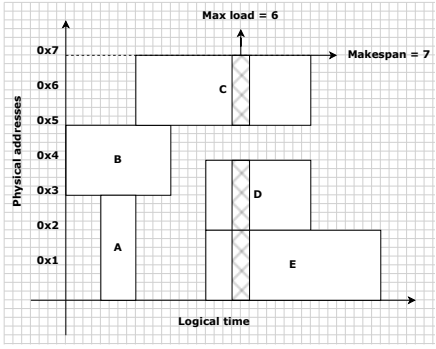


Figure 4.1: A more detailed illustration of the dynamic storage allocation (DSA) problem. This instance comprises five buffers and a (suboptimal) solution, i.e., offset assignment to each of the buffers, is depicted.

in the general case of non-uniform request sizes. A toy illustration comprising three rectangles is shown at Figure 4.1. If the horizontal axis represents time, the allocator may move rectangles vertically.

A DSA **input** comprises buffers defined as (h, t_s, t_e) tuples, where h stands for buffer size. All of the data involved are **discrete**, more precisely non-negative integers. A buffer is **live** in the open interval (t_s, t_e). We refer to (t_s, t_e) as the buffer's **lifetime**. We refer to t_s and t_e as **allocation time** and **de-allocation time** respectively. A buffer's **lifespan**, i.e., the size of its lifetime, i.e., the total number of time units at which the buffer is live, is computed as below:

$$l = t_e - t_s - 1 \tag{4.1}$$

Two buffers **overlap** if their respective lifetimes overlap. An input's **load** at moment t is the size sum of all buffers live at t. We refer to the maximum load measured across all t as **max load** (L). In Figure 4.1, the max load is the length of the cross-hatched stripe. The small gap between the two pieces does not contribute to it because it does not belong to any buffer, it's just unused space. By **placement** we mean annotating an input's buffers with valid offsets. A placement's **makespan** or **max memory usage** (M) is the address space size needed to fit all buffers. **Fragmentation** (F) is the difference between an input's max load and the actual makespan of some placement $(7 - 6 = 1$ byte in Figure 4.1):

$$F = M - L \tag{4.2}$$

The NP-completeness of DSA has led researchers toward approximation algorithms. The quality of each algorithm is expressed as upper bounds for fragmentation. For instance, a 6-approximation algorithm guarantees that it will never produce a makespan six times bigger than the max load. The current SOTA in DSA is a $(2 + \epsilon)$-approximation algorithm by Buchsbaum et al [13]. ϵ is described as a "sufficiently small" real number and is input-dependent.

4.2.1 Elementary Cases

There are certain instances of the problem which can be solved optimally, i.e., with zero fragmentation. One can recognize such instances in linear time. It suffices to traverse the input once, and check if (i) any overlapping buffers, or (ii) more than one buffer sizes exist. If no buffers overlap, they can all be placed at offset zero.

If all buffers share the same size, the problem is reduced to meeting room scheduling and can be solved with greedy interval graph coloring (IGC). Since we shall make use of IGC later, we remind it to the reader via Figure 4.2.

```
1  Function IntervalGraphColoring(B)
       input  : B = { b | b = (h, t_s, t_e) }
       output : O = { o | o ∈ ℕ :
                OffsetsValid(B, O) }
2      O ← HashMap.new();
       // Buffer-row mapping.
3      l ← HashMap.new();
4      free ← PriorityQueue.new();
5      next_row ← 0;
6      evts ← GetEvents(B);
       // Each .pop() spawns an "e".
       // Each "e" holds a "buff".
7      while evts.pop() do
8          if IsAlloca(e) then
9              if free.empty() then
10                 offset ← next_row;
11                 next_row += 1;
12             else
13                 offset ← free.pop();
14             end
15             l.insert((buff, offset));
16             O.insert((buff, offset));
17         else
18             freed ← l.remove(buff);
19             free.push(freed_row);
20         end
21     end
22     return O;
23 end
```

Figure 4.2: Interval Graph Coloring.

4.2.2 Heuristics

In Section 4.1 we claimed that existing DSA implementations can be categorized as either heuristics or isomorphisms. While "isomorphisms" is a deliberately vague term, by "heuristics" we mean a specific family of solutions.

In this Chapter, we define a heuristic as a two-phase operation comprising (i) a *sorting* step and (ii) a *fitting* step. In the first step, buffers are ordered according to some arbitrarily complex criterion, e.g., decreasing size, increasing allocation time, etc. Then, during the fitting step, the sorted buffers are traversed and assigned an offset in a best- or first-fit fashion. These fits differ from what the corresponding terms mean in the OS allocators context, since DSA also cares about lifetimes. By rejecting gaps lower in the address space for better-sized gaps higher up, DSA best-fit risks being unable to fill the lower gaps later because of conflicts in the temporal domain. A counterintuitive fact stemming from this is that *first-fit often incurs less fragmentation than best-fit.* Figure 4.3 describes first-fit in detail.

4.3 The Boxing Algorithm by Buchsbaum et al.

The best known DSA "algorithm" is a 2-approximation technique published more than two decades ago [13]. We put quotes around the term since, as will be shown in this section, we are dealing in fact with a complex system of interacting algorithms. From now on we will be referring to that original paper as "BA".

We have studied BA once more in the past [70]. Our previous implementation, despite being an indispensable research milestone, carried serious weaknesses. First of all, we never published its source code. Moreover, it suffered from severe instability, e.g., yielding out-of-memory errors for two thousand buffers, but converging as it should for twenty thousand. Most importantly, *it was incorrect*: BA has latent invariants which we had not discovered back then. Violating those invariants may lead to convergence, but the converged-upon output will be far from ideal. In consequence, we were getting nonsensical results where on-line algorithms

```
1  Function FirstFit(B)
       input  : B = { b | b = (h, t_s, t_e) }
       output : O = { o | o ∈ ℕ :
                OffsetsValid(B, O) }
2      O ← HashMap.new();
       // Each .pop() spawns a "buff".
3      while B.pop() do
           // For traversing the
              address space.
4          run ← 0;
           // Scan placed, conflicting
              buffers
           // in ascending offset
              order.
5          for conf in GetConflicts(O,
            buff) do
               // conf.offset - run ≥
                  buff.size
6              if Fits(buff, run, conf) then
7                  break;
8              else
                   // conf.offset +
                      conf.size
9                  run ←
                    GetNextAddr(conf);
10             end
11         end
12         O.insert((buff, run));
13     end
14     return O;
15 end
```

Figure 4.3: First-fit placement.

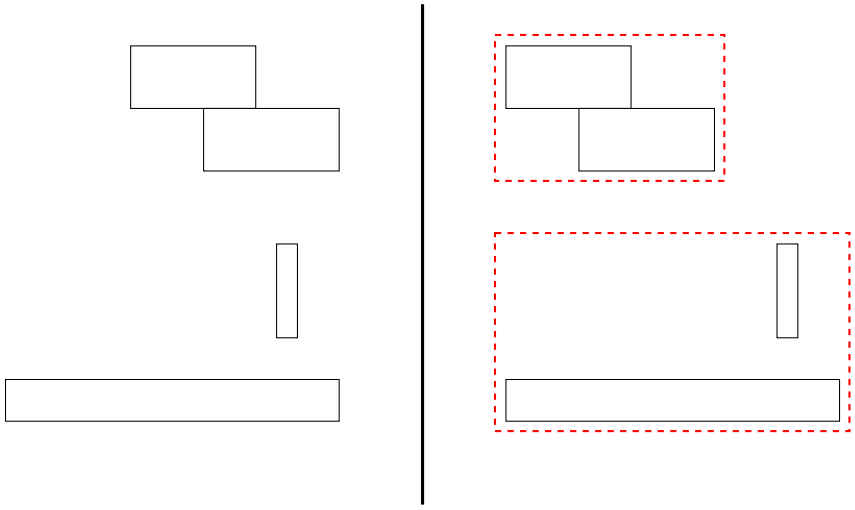


Figure 4.4: An illustration of BA's main idea, that is, boxing buffers into Matryoshkas. The buffers on the left have 4! = 24 possible orderings. By boxing them into two distinct groups the number of possible orderings has been reduced by a factor of 3. In their paper, Buchsbaum et al. do not care about this reduction in complexity; they use the boxes to reason about worst-case fragmentation.

were incurring less fragmentation than our off-line, supposedly SOTA allocator [2].

This Chapter aims to establish an open-source reference implementation that is correct, robust and fast. The present section handles the part about correctness. We shall do a guided tour of BA, which is `idealloc`'s beating heart. We will clarify which parts of it we kept, which ones we modified and how, and what novel additions we had to make in order to bring it to life.

4.3.1 Overview

The most important thing to understand about BA is that it is *incomplete*. In the heuristics terminology introduced in Section 4.2.2, BA is a partial sorting step. It accepts a set of buffers as input, and yields a set of Matryoshka doll-like

[2]This was an "intellectual abstract" paper, with the focus being on the ideas instead of the experiments. The main idea was to view OS allocators as black-box DSA agents and see how they fare against a "standard" DSA solution.

boxes as output. These boxes contain other boxes, and so on until some level of depth where subsets of the original input's buffers reside (see Figure 4.4). To keep consistent with BA's terminology, we will be referring to both the input's buffers and the output's boxes as **jobs**. The key characteristic of the outermost jobs is that they all share the same size, and as Section 4.2.1 notes, they can be optimally placed with IGC. How offsets given to the top-level Matryoshkas should bleed through each boxing layer, eventually to reach the original buffers at the bottom, is not treated by BA's authors. We shall return to this question in Section 4.4. For now, let us focus on the process followed to convert BA into source code.

Like any mathematics paper, BA comprises lemmas, theorems and corollaries. We will be referring to these constructs collectively as *functional units* (FUs). FUs are numbered in the order that they appear in the paper: Lemma 1 is followed by Theorem 2, then comes Lemma 3 and so on.

Each FU comprises a *statement*, and a *proof* testifying to the correctness of the statement. The rather convenient characteristic of BA is that all of its proofs are made by construction. Every step of every proof either *calculates* something (e.g., "compute the min/max ratio of input job sizes") or *invokes* some other FU. Thus, to implement BA it suffices to view each FU as a program function, and each proof as the corresponding function body. To give a concrete example, consider Corollary 17, which we initially took to be BA's "entry point":

COROLLARY 17. *There exists a polynomial-time algorithm that takes an arbitrary set X of jobs as input and produces a feasible solution to DYNAMIC STORAGE ALLOCATION on X with makespan at most* $(1+O((h_{max}/L)^{1/7}))L$.

Proof. Apply Theorem 16 to X with $\epsilon = (h_{max}/L)^{1/7}$. □

Recall from Section 4.2 that L stands for the input's max load. Thus if Corollary 17 were a function, its input would be a set of jobs, and its output would be a set of valid offsets with which to annotate the input. Moreover, its body would comprise (i) a computation of ϵ and (ii) an invocation of Theorem 16.

Though Corollary 17 proved inappropriate as an entry point, it was useful in the sense of fixing our attention to Theorem 16. To that FU we now turn [3]. As

[3]The numbering of FUs in the original BA publication carries an implicit indication of strength, i.e., width of applicability and/or degree of approximation. Lemma 1 operates on unit-size jobs that are all live at the same time. Theorem 2 treats unit-size jobs with arbitrary lifespans, thus removing the simultaneous liveness constraint and widening its applicability. Theorem 16 deals with arbitrary input sets and guarantees solutions with makespan at most $(1+c\epsilon)L + O(h_{max}/\epsilon^6)$ for some constant c and some real ϵ. The strongest algorithm in the paper is featured in Theorem 19, which nevertheless cannot be implemented as a computer program (see the Appendix for an elaboration).

regards its proof, we omit mathematical arguments in between computational steps. We make omissions explicit via the symbol "[...]".

THEOREM 16. *Let $\epsilon \in (0, 1]$. There exist a constant c and a polynomial-time algorithm that takes ϵ and an arbitrary set X of jobs as input and produces a feasible solution to DYNAMIC STORAGE ALLOCATION on X with makespan at most $(1 + c\epsilon)L + O(h_{max}/\epsilon^6)$.*

Proof. [...] We are going to apply Corollary 15 repeatedly, boxing the smallest jobs so as to increase the minimum job height h_{min} until it gets close enough to the maximum job height h_{max} that we can finish with a last application of Corollary 15.

[...] Let r denote the ration h_{max}/h_{min}. Assume first that $(log_2 r)^2 \geq 1/\epsilon$, and set $\mu = \epsilon/(log_2 r)^2$ and $H = \lceil \mu^5 h_{max}/(log_2 r)^2 \rceil$. Consider the partition $X = X_s \cup X_l$, where X_s denotes the jobs of height at most μH and $X_l = X \setminus X_s$. Now apply Corollary 15 to X_s with box-height parameter H and error parameter μ. This yields a set B_s of boxes of height H into which the jobs of X_s fit such that [...].

Now consider B_s as a set of jobs and the revised problem on $X' = B_s \cup X_l$. [...] Iterate the above boxing of small jobs, each time using new error parameter $\mu' = \epsilon/(log_2 r')^2$ until it yields a problem X^* with minimum job height h^*_{min} for which the ratio $r^* = h_{max}/h^*_{min}$ is such that $(log_2 r^*)^2 < 1/\epsilon$. [...]

Now apply Corollary 15 to all of X^* with box-height parameter $H = h_{max}/\epsilon$ [...] and error parameter ϵ; this is the "last application" of Corollary 15 to which we alluded earlier. [...] □

It must now be obvious that Theorem 16 is the crux of BA, i.e., its "main" function. It accepts an arbitrary set of jobs and a real number, and produces the corresponding DSA solution. The following remarks apply:

- the execution of Theorem 16 is governed by ϵ, h_{min} and h_{max}. Everything else is a function of these three quantities.
- the actual output of Theorem 16 is not a complete DSA solution. As we can see, the proof is built around repeated applications of Corollary 15, and terminates with such an application. According to the proof's own phrasing, however, Corollary 15 produces *boxes*; not offsets.
- the loop that is executed while $(log_2 r)^2 \geq 1/\epsilon$ demands that $h_{min} \leq \mu H$, else X_s turns out empty. Then B_s is empty as well, the ratio r remains unchanged, and the loop never ends.

The first remark is self-explanatory. As regards the second remark, its validity does not harm the purpose of BA's authors. Their argument rather exploits the fact that Corollary 15 produces same-height boxes. Recall from Section 4.2.1 that IGC applied on identical sizes yields zero fragmentation, i.e., the solution's makespan equals the input's max load. In the parts of the proof that we have omitted for brevity, the authors bound the max load of Corollary 15's output, thus bounding the makespan of the boxes' contents as a result. From the perspective of a programmer who wants to actually solve DSA, implementing Theorem 16 is insufficient. Hence the first paragraph of the present subsection.

The final remark is in fact the opening of the rabbit hole which led us to discovering BA's latent invariants.

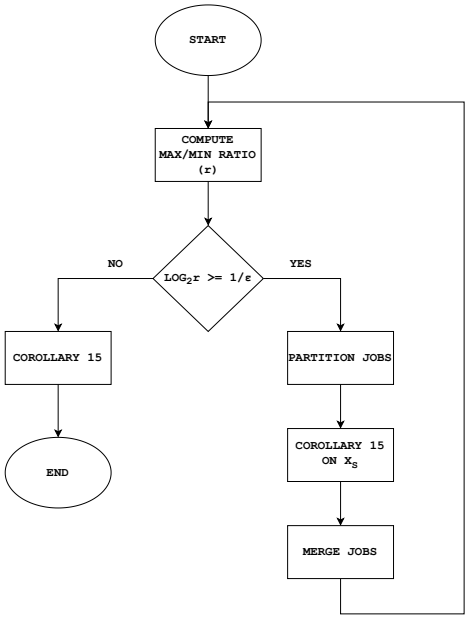


Figure 4.5: T16 flow diagram.

Interlude: Programming as Archaeology

Allow us to clarify our stance before proceeding. From the outset of our efforts to this day, we have put our ultimate trust on BA's superiority. We view its FUs as priceless ancient artifacts buried in the sands of abstract thought, and our work as that of an archaeologist who must unearth those artifacts in the most intact form possible. This act of excavation, this transition from theory to practice, from the abstract to the executable, unavoidably entails points of necessary intervention. Our unshakeable trust on BA dictates (i) minimizing the number and degree of said interventions, as well as (ii) being certain about their soundness. It is these two implications that the following Section serves. A formal treatment of our findings is beyond both our powers and intentions.

4.3.2 Latent Invariants

By this Section's title we are referring to the following non-trivial conclusions:

1. the real-valued ϵ of Theorem 16 has an input-dependent range of "legal" values which can be greater than 1.
2. the real-valued μ of Theorem 16 has a universal upper bound equal to $\frac{\sqrt{5}-1}{2}$.
3. it is necessary that every input satisfies the inequality $h_{max} \geq \lceil 2216.53 \cdot h_{min} \rceil$.

We shall show that all three invariants can be derived from BA's original text without any additional moves. Let us start with some definitions from the proof of Theorem 16, particularly that branch of execution where $(log_2 r)^2 \geq 1/\epsilon$:

$$r = \frac{h_{max}}{h_{min}} \tag{4.3}$$

$$\mu = \frac{\epsilon}{(log_2 r)^2} \tag{4.4}$$

$$H = \lceil \mu^5 h_{max}/(log_2 r)^2 \rceil \tag{4.5}$$

As we have already remarked, in order for that branch to avoid looping forever, it should hold that $h_{min} \leq \mu H$. Let us unwrap this expression:

$$h_{min} \leq \mu H \xRightarrow{(4.5)}$$

$$h_{min} \leq \mu \lceil \mu^5 h_{max}/(log_2 r)^2 \rceil \xRightarrow{\mu > 0}$$

$$\frac{h_{min}}{\mu} \leq \lceil \mu^5 h_{max}/(log_2 r)^2 \rceil \Rightarrow$$

$$\frac{h_{min}}{\mu} - 1 < \mu^5 h_{max}/(log_2 r)^2$$

To the last equation, we can without loss of generality tighten its left hand:

$$\frac{h_{min}}{\mu} - 1 < \mu^5 h_{max}/(log_2 r)^2 \Rightarrow$$

$$\frac{h_{min}}{\mu} \leq \mu^5 h_{max}/(log_2 r)^2 \xrightarrow[log_2 r > 0]{(4.3)}$$

$$\frac{(log_2 r)^2}{r} \leq \mu^6 \xrightarrow{(4.4)}$$

$$\frac{(log_2 r)^2}{r} \leq \frac{\epsilon^6}{(log_2 r)^{12}} \Rightarrow$$

$$\epsilon \geq \sqrt[6]{\frac{(log_2 r)^{14}}{r}} \tag{4.6}$$

We have arrived at a condition for ϵ which, given the fact that Theorem 16 operates on *arbitrary* sets of jobs, i.e., for any r, does by no means guarantee that $\epsilon \in (0, 1]$. Let us move forward. Recall that we are undergoing this investigation in order to arrive at conditions which guarantee that the algorithm described in the proof of Theorem 16 runs "as it should". Also recall that we are for now focusing on the top branch of said proof, namely that one where $(log_2 r)^2 \geq 1/\epsilon$. There, Corollary 15 is called on X_s with box-height parameter H and error parameter μ. Here's Corollary 15:

COROLLARY 15. *Let H be a positive integer box-height parameter and $\epsilon > 0$ be a sufficiently small error parameter. Given a set Z of jobs, each of height between h_{min} and ϵH, there exist a set B of boxes, each of height H, and a boxing of Z into B such that for all x-coordinates t,*

$$L_B(t) \leq (1 + 9\epsilon) L_Z(t) + O(\frac{H(log_2(H/h_{min}))^2}{\epsilon^4})$$

Proof. We construct such a boxing. First, round the job heights: each height h is rounded up to $\lfloor (1+\epsilon)^i \rfloor$, where i is defined by $(1+\epsilon)^{i-1} < h \leq (1+\epsilon)^i$. Let Y denote the resulting set of rounded jobs.

Now, partition the jobs according to their heights. For each rounded height h, let Y_h denote the set of jobs of height h. Divide the heights of all jobs in Y_h by h; apply Theorem 2 with box-height parameter $\lfloor H/h \rfloor$; and then multiply all box heights by h to get a set B_h of boxes of height at most H. The output is a set $B = \bigcup_h B_h$ of boxes, which we can assume are all of height H. [...] □

A non-obvious yet key detail is that we must not call Theorem 2 with a box-height parameter equal to zero (since zero-height boxes do not make sense). We

see from the proof that that box-height parameter is determined by the size classes to which the input jobs have been rounded up. We know that there exists a i_{max} for which the largest jobs in Z are rounded to $h_m = \lfloor (1+\epsilon)^{i_{max}} \rfloor$. It suffices to ensure $\lfloor H/h_m \rfloor \geq 1$:

$$\begin{aligned}
\lfloor H/h_m \rfloor \geq 1 \Rightarrow \\
H/h_m \geq 1 \Rightarrow \\
h_m \leq H \Rightarrow \\
\lfloor (1+\epsilon)^{i_{max}} \rfloor \leq H \Rightarrow \\
(1+\epsilon)^{i_{max}} < H+1 \xRightarrow{\epsilon = \mu} \\
(1+\mu)^{i_{max}} < H+1 \qquad (4.7)
\end{aligned}$$

Due to the fact that Corollary 15 is called on X_s ($Z = X_s$), we know that for all sizes h in Z:

$$h_{min} \leq h \leq \lfloor \mu H \rfloor \qquad (4.8)$$

We can thus expand Inequality 4.7 with another branch on its left side, since $\lfloor \mu H \rfloor \leq h_m$:

$$\begin{aligned}
\lfloor \mu H \rfloor \leq (1+\mu)^{i_{max}} < H+1 \Rightarrow \\
\lfloor \mu H \rfloor < H+1 \Rightarrow \\
\mu H < H+1 \Rightarrow \\
H(1-\mu) > -1
\end{aligned}$$

The above is always true as long as $1-\mu \geq 0 \Rightarrow \mu \leq 1$. A rather sensible requirement given the fact that, overall, Corollary 15 boxes jobs of height up to μH into H-sized boxes.

Before examining Theorem 2, let us backtrack to consider the second execution path of Theorem 16, that where $(log_2 r)^2 < 1/\epsilon$. BA's authors suggest to invoke Corollary 15 one last time, with box-height parameter $H = h_{max}/\epsilon$ and error

parameter ϵ. Our analysis, however, forces us to reject this course of action. We have already shown that (i) ϵ may end up greater than 1 and (ii) Corollary 15 demands an error parameter that is *at most* 1. An alternative is necessary.

The repeated applications of Corollary 15 during the top branch of Theorem 16 increase the minimum job height to h^*_{min}. We thus know that $r^* = h_{max}/h^*_{min}$ is smaller than all the previous values of r. As a result, $\mu^* = \epsilon/(log_2 r^*)^2$ is the *maximum* value for μ. What if we used μ^* in the place of ϵ for the last invocation of Corollary 15? Similarly with before, we would have:

$$(1+\mu^*)^{i_{max}-1} < h_{max} \leq (1+\mu^*)^{i_{max}} \tag{4.9}$$

Demanding that the largest size class does not yield a zero box-height parameter for Theorem 2 leads us to:

$$\begin{aligned}
&\lfloor (1+\mu^*)^{i_{max}} \rfloor \leq H \Rightarrow \\
&(1+\mu^*)^{i_{max}} < H+1 \xrightarrow{H=h_{max}/\mu^*} \\
&(1+\mu^*)^{i_{max}} < \frac{h_{max}}{\mu^*} + 1
\end{aligned}$$

To simplify our algebra, we can once again without loss of generality prune the last inequality to $(1+\mu^*)^{i_{max}} \leq \frac{h_{max}}{\mu^*}$. Dividing all members of Inequality (4.9) with μ^* and keeping the left side, we have $\frac{(1+\mu^*)^{i_{max}-1}}{\mu^*} < \frac{h_{max}}{\mu^*}$. We must now decide about the relation between $(1+\mu^*)^{i_{max}}$ and $\frac{(1+\mu^*)^{i_{max}-1}}{\mu^*}$. Nothing obstructs us from declaring the below:

$$\begin{aligned}
&(1+\mu^*)^{i_{max}} \leq \frac{(1+\mu^*)^{i_{max}-1}}{\mu^*} \Rightarrow \\
&(1+\mu^*)\mu^* \leq 1 \Rightarrow \\
&\mu^{*2} + \mu^* - 1 \leq 0
\end{aligned}$$

The corresponding equation has roots $\mu^*_{1,2} = \frac{-1\pm\sqrt{5}}{2}$. Since μ^* is by definition positive, the only way for the inequality to be less or equal than zero is:

$$\mu^* \leq \frac{\sqrt{5}-1}{2} \simeq 0.618033... \tag{4.10}$$

This is a very convenient result. First of all, it abides to our requirement with respect to the error parameter given to Corollary 15. In other words, we *can* use μ^* instead of ϵ for the last invocation of Corollary 15, as long as Inequality 4.10 holds. Secondly, it is independent from the input. The only problem is, μ^* is a quantity "from the future": BA has to execute properly and reach the low branch of Theorem 16 before r^*—and thus μ^*—becomes available. In contrast, we want to control BA's execution via configuring quantities that are available from the outset, like ϵ and r. Thankfully, μ^* is a function of ϵ. Having decided to use μ^* for the last Corollary 15 invocation, and knowing the necessary condition for this to work (Inequality 4.10), we can impose it to ϵ in the here and now:

$$\mu^* \leq \frac{\sqrt{5}-1}{2} \xrightarrow{\mu^* = \frac{\epsilon}{(log_2 r^*)^2}}$$

$$\epsilon \leq \frac{\sqrt{5}-1}{2}(log_2 r^*)^2 \xrightarrow[(4.6)]{r^* < r}$$

$$\sqrt[6]{\frac{(log_2 r)^{14}}{r}} \leq \epsilon \leq \frac{\sqrt{5}-1}{2}(log_2 r)^2 \tag{4.11}$$

There is, however, no reason to believe that Inequality 4.11 will be valid for *all* possible inputs. In order to be certain we must make one last demand:

$$\sqrt[6]{\frac{(log_2 r)^{14}}{r}} < \frac{\sqrt{5}-1}{2}(log_2 r)^2 \Rightarrow$$

$$\frac{(log_2 r)^{14}}{r} < (\frac{\sqrt{5}-1}{2})^6 \cdot (log_2 r)^{12} \Rightarrow$$

$$\frac{(log_2 r)^2}{r} < (\frac{\sqrt{5}-1}{2})^6 \tag{4.12}$$

According to WolframAlpha, an approximate solution for Inequality 4.12 is $r > 2216.53$. This concludes our design. Inequalities 4.11, 4.10 and 4.12 correspond to each of the three invariants listed in the beginning of this Section. Incorporating them to our source code has allowed `idealloc` to treat a wide variety of inputs without any unexpected behavior.

4.3.3 Critical Point Injection

The latent invariants of the preceding Section do the "heavy lifting" of ensuring that BA works as it should. Our tour, however, is not over. There is one last intervention that we needed to make. It is time to visit Theorem 2:

THEOREM 2. *Given a set Z of jobs, each of height 1, an integer box-height parameter H, and a sufficiently small positive ϵ, there exist a set B of boxes, each of height H, and a boxing of Z into B such that for all x-coordinates t,*

$$L_B(t) \leq (1+4\epsilon)L_Z(t) + O(\frac{Hlog_2H}{\epsilon^2}log_2\frac{1}{\epsilon})$$

Proof. We are going to apply Lemma 1 many times, boxing the unresolved jobs into additional boxes as we go along. Our general goal is to keep the wasted load (free space) in those additional boxes small at any x-coordinate.

We use the following recursive method. Given are

- A set X of jobs and an open *bounding interval* I, such that $\forall j \in X,\ I_j \subseteq I$.
- A nonempty finite set of *critical x-coordinates* $T = \{infI = t_o < t_1 < \ ... \ < t_q < t_{q+1} = supI\} \subseteq I \cup \{infI,\ supI\}$.
- A set F of *free spaces*. Each free space is an open sub-interval of I of height 1 *having endpoints in* T. Any free space $f \in F$ is called *spanning* if $f = I$ and *non-spanning* otherwise.

Initially, $X = Z$, $I = (0,\ 1)$, $T = \{0,\ t,\ 1\}$ for some arbitrary t at which some job from Z is live, and $F = \emptyset$. Recall that $I_j = (x_j,\ y_j)$ denotes the interval of job j. With the help of T, define partition

$$X = (R_1 \cup R_2 \cup \ ... \ \cup R_q) \cup (X_0 \cup X_1 \cup \ ... \ \cup X_q)$$

as follows. First, define $X_i = \{j \in X : I_j \subseteq (t_i,\ t_{i+1})\}$ for $0 \leq j \leq q$.

Then define the R_i's recursively. Define $X' = X \setminus (X_0 \cup X_1 \cup \ ... \ \cup X_q)$. Note that $q \geq 1$. Define $R_{\lceil q/2 \rceil} = \{j \in X' :\ t_{\lceil q/2 \rceil} \in I_j\}$. Define P to be the set of remaining jobs j of X' with $y_j < t_{\lceil q/2 \rceil}$, and define Q to be the set of remaining jobs j of X' with $t_{\lceil q/2 \rceil} < x_j$. If $P \neq \emptyset$, recursively partition P using $\{t_1,\ t_2,\ \ ... \ , t_{\lceil q/2 \rceil - 1}\}$. Afterward, if $Q \neq \emptyset$, recursively partition Q using $\{t_{\lceil q/2 \rceil + 1},\ t_{\lceil q/2 \rceil + 2},\ ...,\ t_q\}$.

Now to each X_i associate a set F_i of intervals (free spaces), initially empty. As sections of free spaces in F are used to box jobs in the R_i's, the unused fragments will be deposited into the appropriate F_i's for use deeper in the recursion (to box jobs in the X_i's).

To box the jobs in the R_i's, first apply Lemma 1 to each R_i, $1 \leq i \leq q$, in any order; note that all jobs in R_i are live at t_i. For each i, this boxes all the jobs of R_i except for at most $2H\lceil 1/\epsilon^2 \rceil$ unresolved jobs. Now consider the set U of all the unresolved jobs from all the R_i's. Derive an optimal packing of U using interval graph coloring (Recall that all jobs are of height one). This packing has makespan L_U.

Let $s(F)$ denote the subset of spanning free spaces of F. If $|s(F)| < L_U$, create $\lceil (L_U - |s(F)|)/H \rceil$ boxes of height H and horizontal extent I. This yields $H\lceil (L_U - |s(F)|)/H \rceil$ new spanning free spaces; add them to F. Now there are at least as many spanning free spaces in F as rows of the packing of U.

For each $1 \leq j \leq L_U$, remove one spanning free space from F, and use it to place all the jobs in row $j$of the packing. This creates *gaps*, or unused portions, in the original free space, each of the form $[\alpha,\ \beta]$ where for some i, j: $t_i < \alpha < t_{i+1}$ and $t_j < \beta < t_{j+1}$; recall that $t_0 = inf I$ and $t_{q+1} = sup I$. For each such $[\alpha,\ \beta]$, if $i \neq j$ then split $[\alpha,\ \beta]$ into $(\alpha,\ t_{i+1})$, $(t_{i+1},\ t_{i+2})$, ..., $(t_{j-1},\ t_j)$, $(t_j,\ \beta)$; and add $(\alpha,\ t_{i+1})$ to F_i, $(t_{i+1},\ t_{i+2})$ to F_{i+1}, ..., $(t_{j-1},\ t_j)$ to F_{j-1}, and $(t_j,\ \beta)$ to F_j. Otherwise $(i = j)$, simply deposit $(\alpha,\ \beta)$ into F_i. This *fragments* the gaps.

Now all the jobs in all the R_i's are boxed. Consider the unused free spaces in F, if any. Each is of the form $(t_i,\ t_j)$ for some $i \neq j$. Split each such $(t_i,\ t_j)$ into $(t_i,\ t_{i+1})$, $(t_{i+1},\ t_{i+2})$, ..., $(t_{j-1},\ t_j)$. Add $(t_i,\ t_{i+1})$ to F_i, $(t_{i+1},\ t_{i+2})$ to F_{i+1}, ..., and $(t_{j-1},\ t_j)$ to F_{j-1}. This *passes down* the remaining unused free spaces to the sub-problems.

In parallel for each $\ell = 0,\ 1,\ 2,\ ...,\ q$, if $X_\ell \neq \emptyset$, recursively apply the construction with new $X \leftarrow X_\ell$, new free space set $F \leftarrow F_\ell$, new bounding interval $I \leftarrow (t_\ell, t_{\ell+1})$ and new criticall x-coordinate set $T \leftarrow$ {endpoints of elements of F_ℓ} $\cup$ $\{t_\ell,\ t_{\ell+1}\}$. [...] □

By now our initial point that BA is not simply an "algorithm" must be obvious. We discourage the reader from devoting excess effort to grasping every last word of Theorem 2 (as we shall show in Section 4.5, some parts of it are redundant). For the time being, it suffices to pay attention to the fact that in order for the boxing procedure to advance, there must exist at least one critical x-coordinate in T at which at least one job in Z is live. In other words, there must exist at least one R_i. At each recursion level, it is only jobs in R_i's that are being boxed, some via Lemma 1, and others via IGC. This need is made explicit at

Table 4.1: Findings and remedies applied to BA's FUs.

BA FU	Finding	Remedy
Corollary 17	Incompliant with latent invariants of Theorem 16 and Corollary 15.	Input preprocessing, ϵ-calibration (Section 4.5.6).
Theorem 16	1. Last Corollary 15 invocation uses ϵ (unsafe). 2. Yields boxes instead of offsets.	1. Use μ^* instead (Section 4.3.2). 2. Unbox and place (Section 4.4).
Theorem 2	R can be empty.	Critical point injection (Section 4.3.3).
Corollary 15	As is.	N/A
Lemma 1		

the start of the proof, where attention is drawn to "some arbitrary t at which some job from Z is live". In our experience, however, it is possible deeper in the recursion for critical point sets T to appear carrying no such t. In those cases, we append one more (appropriate) time point to T.

The only remaining FU in BA's chain is Lemma 1. To keep the main body of our Chapter as short as possible, and due to the fact that Lemma 1 works "out of the box", we have moved its definition to the Appendix.

To summarize, the entire Section 4.3 demonstrates our approach as regards `idealloc`'s core component, namely the boxing algorithm by Buchsbaum et al. [13]. We have gone through the algorithm's parts and limitations, and have either presented, or hinted toward, ways to overcome said limitations. The main takeaways are listed in Table 4.1.

4.4 Unboxing and Final Placement

We have already mentioned that BA does not produce offsets, as would normally be the case if one wanted to solve DSA. Instead Theorem 16 returns a set of equal-height, Matryoshka doll-like boxes. The problem addressed by the present Section can be stated as: *how can the outer Matryoshkas' IGC-derived offsets be diffused all throughout the boxing's hierarchy until the original buffers are found and accordingly placed?*

The process is sketched in Figure 4.6. We will be using "buffers" to refer to original buffers and "boxes" for the Matryoshkas. Like boxing, this is a recursive procedure. Two questions are driving decisions at each level of recursion:

- **Line 3:** do input elements share the same size?
- **Line 5:** are input elements non-overlapping?

Apart from buffers/boxes, a watermark is also given as input—initialized at zero before the first ever call. It signifies the starting offset from which placement should commence. The watermark is updated and inherited by deeper recursion levels. Hence we ensure that the contents of each box end up placed within their container's boundaries.

Let us now visit all possible answers to the above questions. If jobs share the same size, we exploit the fact that DSA for uniform sizes is optimally solved with IGC. The role of `PlaceSameSizes` is to traverse all IGC-produced rows, place the contents of each at the current watermark, and bump the watermark at the row's tip. If the jobs don't overlap in time, the decision is trivial. We unbox each input element and recursively call the procedure with the same watermark (lines 7, 8).

Finally, if none of the above conditions hold, we partition the jobs by size and place each subset

```
1  Function UnboxAll(J, w)
       input  : J = {j|j = (h, t_s, t_e)}, w
       output : O = {o|o ∈ ℕ :
                OffsetsAreValid(J, O)}
2      O ← Init();
3      if SameSize(J) then
4          return PlaceSameSizes(J,w);
5      else if not Overlap(J) then
6          for job in J do
7              placed ← UnboxAll(Unbox(J),
                w);
8              O ← MergeOffsets(O, placed);
9          end
10     else
11         for jobs in PartitionBySize(J) do
12             placed ← PlaceSameSizes(jobs,
                w);
13             w ← MaxAddr(placed);
14             O ← MergeOffsets(O, placed);
15         end
16     end
17     return O;
18 end
19 Function PlaceSameSizes(J,w)
       input  : J = {j|j = (h, t_s, t_e)}, w
       output : O = {o|o ∈ ℕ :
                OffsetsAreValid(J, O)}
20     O ← Init();
21     for row in IGC(J) do
22         placed ← UnboxAll(row,w);
23         w ← MaxAddr(placed);
24         O ← MergeOffsets(O, placed);
25     end
26     return O;
27 end
```

Figure 4.6: Unboxing pseudocode.

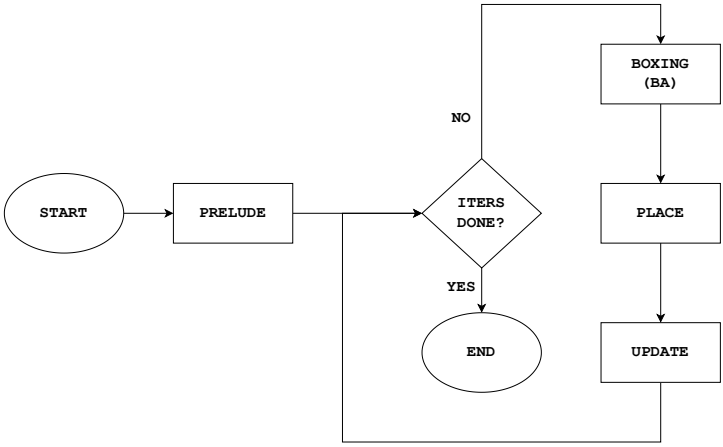


Figure 4.7: `idealloc` flow diagram.

independently (lines 12-14). But we are not done yet! Due to the repeated round-ups of box sizes in Corollary 15, as well as the recursive nature of Theorem 16, the offsets produced by the above procedure are *sparse*. So we view all work up to this point, i.e., BA-derived boxing and the offsets produced by unboxing, as an intricate sorting step according to the terminology of Section 4.2.2. To finalize the output, we "squeeze" the buffers via first-fit placement, traversing them in increasing offset.

4.5 Design and Implementation

Figure 4.7 gives an overview of `idealloc`. Its design is owed (i) to our goal of robust and high performance, and (ii) to the inherent *stochasticity* of BA, due to the critical points of Theorem 2 (see Section 4.3.3). Though we have more to say on this later, keep in mind that even the simplest of operations, namely sorting by size, is stochastic: how should one break ties between equal sizes? Enforced determinism, i.e., using some unique ID for such occasions, may help with data visualization but harms best-case fragmentation. One does not tame randomness by putting it under the rug.

4.5.1 Interface

`idealloc` accepts the following parameters:

- **original input:** a collection of jobs to place.
- **worst-case fragmentation:** an upper bound for the quality of the output. If that amount or less fragmentation is achieved at any point, the execution terminates early.
- **start address:** base location S to which all offsets refer. The address of a buffer with offset O is $S + O$.
- **iterations:** an upper bound for the total number of times the box-and-place kernel is allowed to run. If exhausted and worst-case fragmentation is not yet beaten, the next-best result is returned.

Note that, as regards fragmentation, in our opinion the only optimal value is zero. But we have included the respective parameter in response to allocators like `minimalloc` [90] and the one featured in Apache's TVM compiler, who include a "maximum makespan" parameter to their interfaces. We find it erroneous to decouple worst-case storage from the input, since it is the input itself, and specifically its max load, which bounds makespan (from below, not from above). Certain maximum makespans may not be achievable for certain inputs.

4.5.2 Input Representation

The fundamental data structure of `idealloc` is the `Job`. Its fields are:

- **allocated size:** self-explanatory.
- **(start, end):** the respective allocation and deallocation times. In line with DSA theory, we adopt *exclusive lifetime semantics* in `idealloc`. This means that a job is **not** live at neither its start, nor its end. Numerous bugs have crunched our nighttime due to ours not being strict enough about lifetime semantics.
- **alignment:** if any, the final address of the buffer is guaranteed to be a multiple of this value.
- **requested size:** owed to the beginnings of `idealloc` being in studying `malloc` traces, kept because someone else may decide to do so in the future. By knowing the difference between requested and allocated size one can measure *internal* fragmentation, out of scope for this paper.

- **contents:** a job may be a box spawned by BA, holding other jobs inside. Both such boxes *and* the original buffers of the input are represented with the same struct.
- **id:** self-explanatory.

Some further remarks on how we handle the input. First of all, there is a list of security checks that must be conducted *before* `idealloc` is invoked. Zero-valued sizes are not allowed. Start- equal or greater than end-times are not allowed. Zero-valued alignment (different than *no* alignment) is not allowed. Non-empty contents are not allowed. Last but not least, we do not allow allocated sizes to be smaller than requested sizes.

4.5.3 Event Traversal

A common situation in `idealloc` is that of computations operating on subsets of buffers. In our experience, avoiding quadratic complexity in such cases is crucial to the allocator's execution time and scalability. Take the max load L of Section 4.2 as an example. Recall that L amounts to the maximum amount of memory that is concurrently live at any time. A naive quadratic solution is to traverse all allocation and deallocation times of all buffers, and for each one traverse the buffers themselves, and aggregate the sizes of those that are live. Luckily there is a better approach.

Imagine a priority queue consisting of *events*: each event carries (i) a timestamp, (ii) a type, i.e., whether it marks the allocation or deallocation of a job, and (iii) a reference to the job itself. Earlier events have precedence over later ones, and deallocations have precedence over allocations. The max load L of N buffers can be computed by consuming this priority queue once, thus by processing $2N$ events. We make heavy use of event traversal across `idealloc` and consider it a fundamental operation. Its underlying principle is that *no change of any kind occurs between consecutive events.*

4.5.4 Working with Different Lifetime Semantics

Fellow allocators and/or benchmarks ascribe different interpretations to buffers' intervals. For instance, XLA's best-fit heap simulator views jobs as live at the endpoints as well as the in-between. `minimalloc` is start-inclusive end-exclusive. `idealloc` adopts exclusive semantics for its internal operation.

Suppose the very real scenario of needing to conduct the experiments accompanying this paper. Given the aforementioned variety of semantics in

the SOTA, one needs to be certain that they are comparing apples to apples. In other words, allocators with different semantics must agree, regarding the buffers described by a specific input benchmark, on which pairs of buffers do or do not overlap. A necessary but not sufficient condition when pursuing such an agreement is that the reported max load of the *same* dataset expressed in exclusive semantics be equal to the one reported when using any other semantics. We make active use of this check in our measurement scripts.

Assume we are in possession of a benchmarks suite employing start-inclusive, death-exclusive semantics. We will be referring to this interpretation as `InEx` from now on, and will be using `In` and `Ex` for start-inclusive-end-inclusive and start-exclusive-end-exclusive semantics respectively. Assume, further, that we want to evaluate on this suite three allocators: the first uses `InEx`, the second `In`, and the last one `Ex`. Last but not least, assume that the task of reading a DSA solution, validating its feasibility, and reporting statistics of interest such as its max load and makespan, is carried out by an analyzer program also using `Ex` semantics. This description largely resembles our real experiments setup.

The missing component is an *adapter*, its input being (i) a DSA solution file, (ii) the semantics of that file and (iii) the semantics to which the file's contents must be transformed. By making use of this adapter, we can for example start from an `InEx` dataset, feed it to the `In`-allocator, and then pass its output to the `Ex`-analyzer. Regardless from the point of departure, the analyzer must always report the same max load and the same number of conflicts (i.e., distinct pairs of overlapping buffers) for the same benchmark. The `idealloc` source code includes such an adapter. Its operating principles are:

- `In` $\longleftrightarrow$ `InEx`: add or subtract one from the buffer's de-allocation time, depending on the direction of the arrow
- `InEx` $\longleftrightarrow$ `Ex`: the two types are *equivalent*. The condition for conflict with a buffer allocated at a and de-allocated at b is in both cases $\neg(x \leq a \ \vee \ y \geq b)$, where $x,\ y$ stand for the respective endpoints of some other buffer

4.5.5 Bootstrapping and Early Stopping

Due to its stochastic nature, the quality of solutions that `idealloc` may yield at each iteration exhibits great variety. In order to waste as little time as possible on sub-optimal solutions, we use a simple bootstrapping scheme: we keep a record of the smallest makespan achieved up to now. During final placement's first-fit, we check whether the resulting offset drives the buffer at hand to exceed our record. In that case, we stop, discard the present boxing, and start anew.

We initialize our bootstrapping value with what we consider to be the best heuristic available: sort by size, break ties by lifespan, and do first-fit. A fitting name for it would be "big-rocks-first". The bootstrapping value is updated whenever `idealloc` yields a smaller makespan.

4.5.6 Prelude Analysis

Certain tasks need take place only once across `idealloc`'s flow. Before doing anything else, we bundle the following tasks into a single event traversal: (i) check for elementary cases (Section 4.2.1), (ii) compute max load, minimum and maximum height, and (iii) construct the interference graph (Section 4.5.7).

If any of the elementary cases holds, execution proceeds accordingly and an optimal solution is found in minimum time. Else, `idealloc` must prepare to iterate on its box-and-place core (Sections 4.3, 4.4). More specifically:

- if the max-to-min height ratio r does not comply with Inequality 4.12, a "dummy" job of height equal to $\lceil 2216.53 \cdot h_{min} \rceil$ and lifetime spanning all of the input is added to the buffers to be boxed
- bootstrapping takes place as described in Section 4.5.5
- the real number ϵ governing the boxing algorithm is configured as described below

Recall that according to Inequality 4.11, it is only within a specific range that ϵ may move. A simple iterative process is followed to pick the final value: we initialize ϵ to be equal to the left arm of Ineq. 4.11. We run the boxing algorithm *up to the point where r^* is computed* (see Section 4.3.1). Next, we increase ϵ by 1% of the remaining range and repeat. We keep that value which yields the smallest r^*.

4.5.7 Fast and Correct Final Placement

Two extra operations to what was described in Section 4.4 are necessary: if a "dummy" job was inserted during prelude analysis, we *ignore* it during unboxing, i.e., we do not assign it any offset and proceed as if it did not exist. Secondly, we ensure that offsets calculated in the final first-fit pass are compliant with each job's potential alignment requirements. Recall that we know both the start address of the range as well as each buffer's alignment (Sections 4.5.1, 4.5.2).

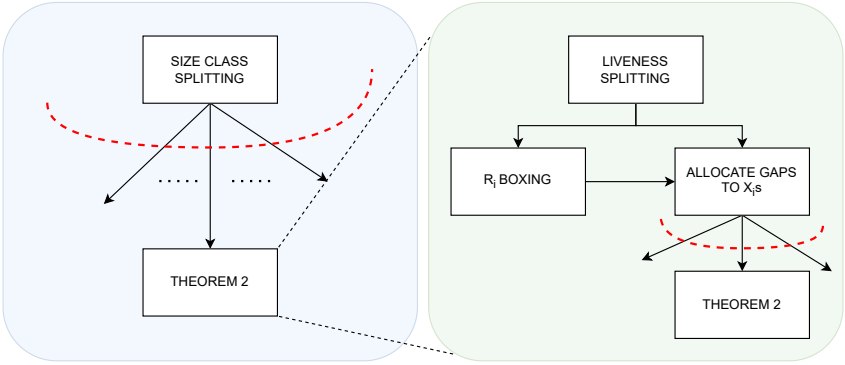


Figure 4.8: Illustration of parallelism opportunities as thick red dashed curves. The light blue box (left) is Corollary 15. Theorem 2 is invoked on each size class independently. The light green box (right) is a simplified unpacking of Theorem 2. Recursive calls to self are issued for each X_i once all R_is are boxed and gaps shared. Each call is independent from the rest.

One further optimization we introduce is an *interference graph*, i.e., a hash map with job IDs as keys, and vectors of concurrently live buffers as values. We use this graph during the first-fit stage, to avoid an otherwise quadratic-complexity overlap check (to be precise, worst-case complexity is still quadratic but in practice rarely does every buffer overlap with everyone else).

4.5.8 Theorem 2 Simplification

The one thing to keep in mind as regards Theorem 2 is that it is expected to box all jobs it is given by Corollary 15 into boxes of size H. For reasons tied to their mathematical arguments, Buchsbaum et al. must pretend that first, Corollary 15 scales jobs down to unit height and *then* passes them to Theorem 2 with height parameter $\lfloor H/h \rfloor$, before scaling the returned boxes up back to H. `idealloc` is concrete evidence that the process can both be simplified and remain correct.

The *actual* interface used by Theorem 2 comprises: (i) the set of buffers to be boxed, (ii) the quantity $\lfloor H/h \rfloor$, (iii) box size H, (iv) the usual error parameter ϵ, (v) the definition's bounding interval, and (vi) the definition's vector of critical coordinates. There are no "free spaces" needed. Boxing happens in two places only: Lemma 1 (see Appendix) and after grouping its unresolved jobs to rows

via IGC. As long as `idealloc` asserts when boxing that the load of the jobs to be boxed does not exceed the expected box height H, execution may proceed.

Also note that, when either initializing the critical coordinates vector or injecting points to it as per Section 4.3.3, it suffices to consider only those points that appear during event traversal.

4.5.9 Parallel Boxing

There are two opportunities for coarse-grain parallelism in the boxing flow. Both are shown on Figure 4.8. The first opportunity appears in Corollary 15: the buffers of each size class can be boxed by Theorem 2 independently. The second opportunity appears in Theorem 2, where the recursive calls for each X_i can also be made in parallel. In both cases, no dependencies between parallel tasks exist. We exploit them accordingly to minimize execution time.

4.5.10 Doors to Randomness

Apart from the critical coordinate selection in the context of Theorem 2, there are numerous other spots in our source code that behave non-deterministically *in a baked-in manner*. For instance, there are places where jobs have to be sorted according to some arbitrary criterion, e.g., in reverse de-allocation time. In each such case, again to minimize execution time, we utilize *unstable* sorting, which may re-order equal elements. Another example is the priority queue we are using for event traversal, which does not guarantee that the insertion order of equal elements is preserved.

It is the systemic interaction of all these random effects that gives `idealloc` its stochasticity.

4.6 Evaluation

We ask the following research questions:

1. Superiority against toy heuristics

We have characterized `idealloc` as a “stochastic bootstrapped heuristic”. Does it outperform the simplest of heuristics in terms of fragmentation?

Table 4.2: Experimental setup used for evaluating `idealloc`.

ALLOCATORS			
Compiler	**Algorithm**	**Commit**	**Build Remarks**
XLA	Some complex best-fit heuristic.	`896c02`	`-O3` flag worsened performance.
MindSpore	`SOMAS` [73]	`4308a56`	CMake build type "Release" improved performance, so we kept it.
TVM	`hillclimb`	cfe1711	Same as `SOMAS`, Triton.
N/A	`minimalloc` [90]	`987b3c1`	None.
N/A	`idealloc` (this paper)	N/A	Cargo `-release` flag and LTO enabled.

BENCHMARK SUITES				
Name	**Type**	**# of Benchmarks**	**(Smallest, Largest) # of Buffers**	**Retrieved Via**
minimalloc	TPU Inference	11	(154, 454)	minimalloc GitHub repo ("challenging" suite).
MindSpore	NPU Training	2	(1042, 18692)	Emails with the authors of SOMAS [73].
In-house	ASPLOS Contest Track, LevelDB tracing	4	(816, 567573)	Custom code.

2. Degree of randomness

Given the high degree of stochasticity elaborated in Section 4.5.10, how probable is that event where applying first-fit to a completely random permutation of the input yields less fragmentation than `idealloc`?

3. Competence against the SOTA

Allocators must (i) produce solutions (ii) of low fragmentation (iii) in reasonable time. We encode this requirement in the following per-benchmark grading scheme: if for *any* reason (e.g., segmentation fault, floating point exception) an

allocator crashes, it loses as many points as the allocators that did not. The same if it has not terminated after 15 minutes. In the rest of cases, the allocator earns as many points as the allocators that it outperformed. How many points does `idealloc` earn under this grading scheme?

4. Core latency

The interface of `idealloc` exposes the total number of iterations over its box-unbox-place core as a user option (Section 4.5.1). How cheap is each such iteration?

5. Futureproofness

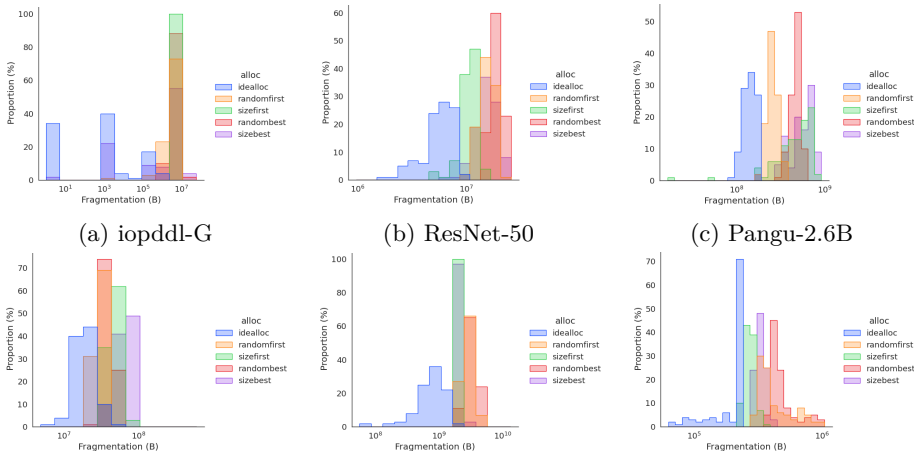
From the outset we have emphasized our interest on DSA instances of arbitrary size and complexity. We want `idealloc` to fare well against the hardest of possible inputs. If we define hardness as the bootstrap heuristic's fragmentation (we will be calling that heuristic "SLFF" from this point onwards), how much better than SLFF is `idealloc` as hardness grows?

The materials used for our experiments are listed in Table A.1. Note that it was particularly difficult to find non-trivial benchmarks in the sense of SLFF yielding non-zero fragmentation.

Our measurements took place on a commodity workstation with eight Intel i7-6700 cores clocked at 3.4 GHz, 128 KiB L1 data and instruction caches, 1 MiB L2 and 8 MiB L3. The machine had 32 GiB DRAM and was running Ubuntu 22.04 inside a privileged-mode Docker container. We instrumented all allocators to report allocation time in microseconds excluding I/O. Max memory usage was computed by processing each run's output files and measuring makespan [4]. We executed each benchmark-allocator pair 10 times to ensure statistical integrity. We assigned a maximum allowable time window of 15 minutes per individual run. All measurement scripts were run with a niceness value of -20 and minimal background noise.

In addition to the SOTA allocators, we fed each benchmark to `idealloc` and configured it to run for 100 iterations—except for the LevelDB benchmark, due to whose size we used 10 iterations. In all cases, we repeated our measurements 100 times to let `idealloc`'s stochasticity express itself as much as possible.

[4]We assume that the target device has no virtual memory and its addresses are physically contiguous. Thus measuring max memory usage offline is accurate. Fellow publications, e.g., minimalloc [90], follow the same practice.



(a) iopddl-G (b) ResNet-50 (c) Pangu-2.6B

(d) iopddl-S (e) iopddl-Y (f) LevelDB

Figure 4.9: Fragmentation histograms against heuristics.

4.6.1 Questions 1 and 2

In Figure 4.9 we are comparing `idealloc`'s fragmentation with four heuristics: the first heuristic (`sizefirst`) sorts the buffers by decreasing size and then applies first-fit. It is stochastic since, as mentioned, size ties had better be solved at random. The second heuristic (`randomfirst`) again applies first-fit, but this time on a random permutation of the input buffers. `sizebest` and `randombest` are the corresponding best-fit flavors. `idealloc`'s superiority in all cases is evident.

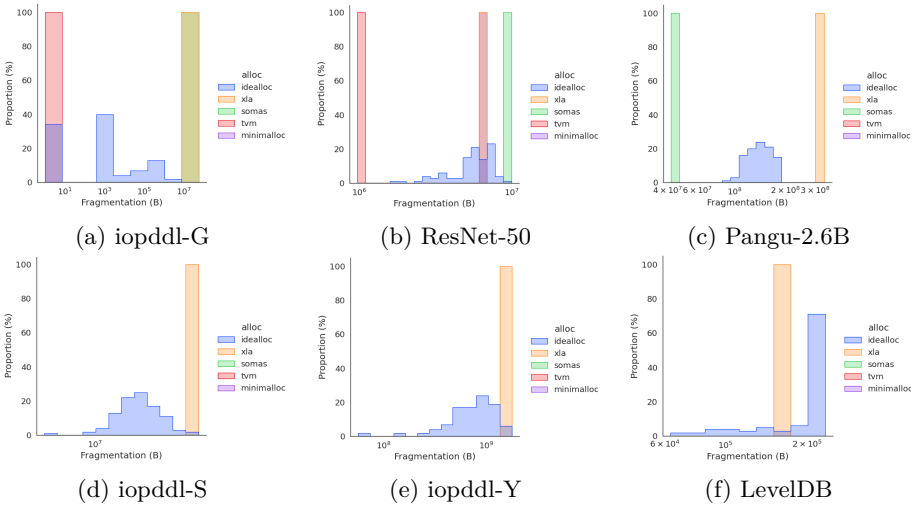
As a side note, there is no clear indication w.r.t. the superiority of some heuristic over the others. Which one is best, and how they compare to each other varies wildly across benchmarks. Thus using the same heuristic horizontally is guaranteed to waste memory.

4.6.2 Question 3

From the opponent allocators, XLA uses `In` semantics, and `minimalloc`, SOMAS and TVM use `InEx`. `idealloc`, on the other hand, uses `Ex`. To ensure fairness we conducted the analysis described in Section 4.5.4 and decided to assume that all of our benchmarks use `InEx` semantics. We then took our measurements and plotted fragmentation histograms like the ones shown in Figure 4.10. The

Table 4.3: Fragmentation measurements and corresponding points.

Benchmark (#bufs.)	Allocator	Fragmentation	Points
MINIMALLOC POINTS	XLA	N/A	1
	TVM		**39**
	SOMAS		11
	minimalloc		36
	idealloc		18
iopddl-G (816)	XLA	54.9 MiB	1
	TVM	**0 MiB**	**4**
	SOMAS	8 MiB	2
	minimalloc	FAILED	-4
	idealloc	81 KiB	3
ResNet-50 (1042)	XLA	6.4 MiB	1
	TVM	**946 KiB**	**4**
	SOMAS	9.5 MiB	0
	minimalloc	6.1 MiB	2
	idealloc	5.5 MiB	3
Pangu-2.6B (18692)	XLA	322.8 MiB	2
	TVM	FAILED	-3
	SOMAS	**40 MiB**	**4**
	minimalloc	FAILED	-3
	idealloc	135.2 MiB	3
iopddl-S (28526)	XLA	42.5 MiB	3
	TVM	FAILED	-2
	SOMAS	FAILED	-2
	minimalloc	FAILED	-2
	idealloc	**18.9 MiB**	**4**
iopddl-Y (62185)	XLA	1.6 GiB	3
	TVM	FAILED	-2
	SOMAS	FAILED	-2
	minimalloc	FAILED	-2
	idealloc	**771.7 MiB**	**4**
LevelDB (567573)	**XLA**	**160 KiB**	**4**
	TVM	FAILED	-2
	SOMAS	FAILED	-2
	minimalloc	FAILED	-2
	idealloc	198 KiB	3
REST POINTS	XLA	N/A	14
	TVM		-1
	SOMAS		0
	minimalloc		-11
	idealloc		**20**
TOTAL POINTS	XLA	N/A	15
	TVM		**38**
	SOMAS		11
	minimalloc		25
	idealloc		**38**



(a) iopddl-G (b) ResNet-50 (c) Pangu-2.6B

(d) iopddl-S (e) iopddl-Y (f) LevelDB

Figure 4.10: Fragmentation histograms against the SOTA.

respective rankings are listed in Table 4.3. The same table includes a summary of the rankings formed for the `minimalloc` micro-benchmarks.

4.6.3 Question 4

We plot allocation time as a function of the buffer count in Figure 4.11. Particularly w.r.t. `idealloc` we have plotted *single-iteration* latency, which includes one prelude analysis (Section 4.5.6) and a single box-unbox-place pass (Sections 4.3, 4.4). Regardless from the size of the input, `idealloc`'s core latency is faster than any alternative.

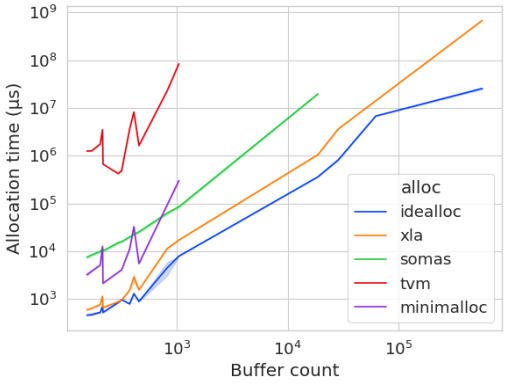


Figure 4.11: `idealloc`'s single-iteration latency versus its competition, as a function of total buffer count. Note the interference graph's impact at the far end of the curve.

4.6.4 Question 5

We see in Figure 4.12 that `idealloc` outperforms SLFF in

a steady fashion as the input hardness increases. The ideal but impossible scenario would be for the drawn line to coincide with $y = x$, i.e., for boxing to always eliminate fragmentation completely. It nevertheless stays close enough.

4.7 Discussion

We began our exposition by declaring our interest in real allocators and how they behave under pressure. We have now presented evidence that (i) there is a gap in the SOTA as regards effective *and* scalable solutions, and (ii) `idealloc` fills that gap. That said, we are aware of the subtleties involved in the process toward making such strong statements. The first half of this Section examines said subtleties from close distance. We then discuss meaningful future activities to either improve or utilize our allocator.

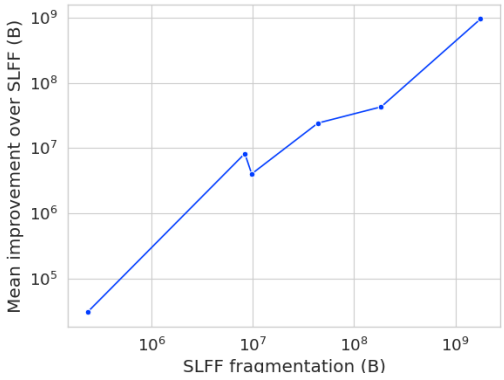


Figure 4.12: `idealloc`'s mean improvement over its bootstrap heuristic as a function of the bootstrap heuristic's own fragmentation.

4.7.1 Results and Their Interpretation

An important point to agree on is whether the selected allocators listed in Table A.1 reflect what we mean by "DSA SOTA". Our initial measurements also included three greedy algorithms from LiteRT (formerly TensorFlow Lite) [105] and one from OpenAI's Triton [116]. Furthermore, XLA features a second allocator based on heap simulation[5], mimicking an on-line OS allocator. TVM has heuristics similar to `sizefirst` besides the hillclimb algorithm[6]. IREE's one and only algorithm is a sort-by-allocation-time best-fit heuristic[7]. We included all these as well, but their performance was poor and we decided to

[5] `https://github.com/openxla/xla/blob/main/xla/service/heap_simulator/heap_simulator.h`

[6] `https://github.com/apache/tvm/blob/cfe1711934f82e56f147f2f5f9f928b5a9b92b3e/src/tir/usmp/algo/greedy.cc`

[7] `https://github.com/iree-org/iree/blob/15ca58e19ec76fab94c4aba8f75091c532282d51/compiler/src/iree/compiler/Dialect/Stream/Transforms/LayoutSlices.cpp`

keep our tables and figures from getting too crowded. The only "popular" deep learning compiler we did not inspect was Meta's Glow, an omission owed to lack of time, not of meticulousness. ILP formulations of DSA are known to be inferior due to poor scalability [83, 90]. We thus believe to have cast a wide and informed enough gaze.

Let us now visit some more specific issues:

Grading System Fairness

Since the crux of our argument is the rankings of Table 4.3, asking if our grading system is fair is a fair question. We used a *tournament* comprising many *races* as a model. The results of each race, i.e., benchmark, are translated to points for each allocator. Whoever has collected the most points after the last race is the tournament's winner. This model is fair to the extent that the points translation scheme is.

Our scheme *rewards* allocators with as many points as the allocators they beat. The number includes both those that yielded worse fragmentation and those that failed. The only objection we can think of is that *differences* in fragmentation are not accounted for. However, the same holds in an actual racing tournament: individual times don't matter.

Moreover, our scheme *punishes* failing allocators with as many points as the allocators that did not fail. Why did we not use a fixed punishment, i.e., losing one point at each failure? Imagine a tournament of N contestants. Focus on contestants A and B. In the first race of the tournament, A finishes first and B is the only contestant that did not finish at all. In the second race, B is the only finisher. Under a fixed-punishment scheme, A and B would end up with $N - 2$ points. Under our scheme, the respective points would be $N - 2$ and *zero*. Which one is fairest?

It depends on the type of tournament winner we are searching for. Since almost everyone finished it, the first race of our example was rather easy (think about iopddl-G in Table 4.3). The converse holds for the second race (think LevelDB). So do we want to incentivize "laziness" in easy races for the sake of potential triumph in hard ones? To the authors of this paper, a positive answer sounds like gambling.

On Normalized Fragmentation

Despite including normalized values for fragmentation in Table 4.3, i.e., absolute fragmentation divided by the benchmark's max load, we do not encourage their use. In the age of "memory walls" [40] memory savings are valuable regardless from necessary memory investment. The reason is simple: most of the time, memory is shared. Savings that look insignificant in proportion to max load can still be used to host data that is foreign to the problem at hand. Only when considering things *in isolation* do absolute quantities lose their weight.

If the above was not convincing enough, consider that by relying on normalized fragmentation, wasting 1 KiB under a max load of 10 KiB looks identical to wasting 1 GiB under a max load of 10 GiB. Both cases have 10% normalized fragmentation, but the second case is clearly more damaging.

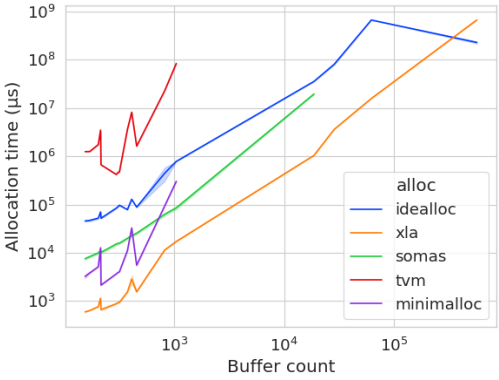


Figure 4.13: `idealloc`'s total latency versus its competition.

Core vs. Total Latency

As noted by Figure 4.11's caption, the plotted blue line stands for `idealloc`'s *single-iteration* latency. For LevelDB, however, we configured `idealloc` to repeat 10 iterations, and for the rest of the benchmarks 100. The following remarks apply:

- our intention was to highlight the fact that each `idealloc` iteration takes minimum time compared to the SOTA
- even when scaled to its real latency (Figure 4.13), `idealloc` (i) is up to two orders of magnitude faster than TVM, and (ii) ends up faster than XLA in LevelDB's context
- if total allocation time is the user's main concern, off-the-shelf heuristics are the way to go. Otherwise trading off latency for lowering fragmentation stands to reason

Hardness Definition

While forming our research questions for Section 4.6, we did not explain our decision to define hardness as SLFF's fragmentation. We hope to give a convincing answer here.

The context was that of "futureproofness": that arbitrarily hard DSA instances will emerge was, as stated in Section 4.1, our founding assumption. Our prime interest is to ensure that `idealloc` will be able to deal with them. The hardness we have in mind concerns the *topology* of an instance, that is, the complexity of the landscape formed by the co-existence of a given set of buffer conflicts and the corresponding buffer sizes. For example, an instance where all buffers overlap is not at all hard/complex/non-trivial: even bump allocation would yield zero fragmentation!

We posit that *a reasonable way to gauge the hardness of an instance is to measure the fragmentation incurred by a simple yet decent heuristic.* Consider the problem of packing one's suitcase before a long trip: does it not make sense to start with the biggest of items, and work our way down? If we place all of our items in this fashion, our baggage was not hard to treat. If on the other hand the "big-rocks-first" strategy fails, our baggage is as hard as the total size of items that we were forced to leave out. Choosing SLFF as our hardness measure is the DSA equivalent of what we described.

4.7.2 Proposed Future Work

Sampling Many ϵ-values

`idealloc`'s boxing core is governed by the error parameter ϵ (Section 4.3.1), which must be confined into an input-specific range of values (Section 4.3.2). Our current approach is to conduct a preprocessing step where we iterate on the aforementioned range and finally set ϵ to that value which minimizes the resulting boxing's max-to-min height ratio (Section 4.5.6). There is no concrete reason behind this strategy, only the intuition that deeper boxing recursions lead to lower fragmentation. A promising alternative would be to sample ϵ at random, thus eliminating significant overhead from our prelude analysis and adding more variety to the placements explored.

Statistical Inference

Given `idealloc`'s stochastic nature (Section 4.5.10), it would be good to have an estimate of how many iterations are needed to become certain that most of the solution space has been explored. This implies a statistical inference component observing each iteration's makespan and using it to refine an on-line distribution. However, such observability would incur performance overhead: to monitor all makespans we would have to remove early stopping (Section 4.5.5). Moreover, extra time would be needed for ste statistical inference core itself.

Randomness Taming

It is tempting to think of some meta-optimization over (i) the selection of ϵ (Sections 4.3.1, 4.5.5) and (ii) Theorem 2 critical points (Section 4.3.3). This would help us avoid "useless" iterations. The main problem with setting up such a mechanism is that our current implementation has non-deterministic elements that are outside our control (Section 4.5.10). On top of that, meta-optimization would need to keep and act on some global state, which would need to be synchronized between threads. In turn this would make everything slower.

4.7.3 A Note on Time and Space

For the entirety of this text we have been interpreting the horizontal dimension as "time" and the vertical one as "space". We owe this to the fact that our research has its roots in computer systems' memory allocation. Nevertheless, other interpretations could enable using `idealloc` (or any similar piece of related work) in completely different contexts. For instance, one could view the horizontal axis as a spectrum of frequencies, and the vertical one as time. Each "buffer" could thus encode a radio host's request to broadcast over a specific band of frequencies for a specific amount of time. Solving DSA in that context would ensure that (i) all hosts receive a slot for their show and (ii) the overall spectrum is "reserved" for as little time as possible.

Room for nuance exists even within the standard time/space interpretation. Whether the vertical axis stands for physical or virtual addresses is left to the hands of the end user. Whether time is wall clock time or, e.g., the total number of bytes allocated by a program, or the indices of a topologically sorted computation graph's nodes, again this decision belongs to the user. DSA itself is indifferent to these decisions. In order for its output to be useful, however, the following invariants must hold: (i) both dimensions must be *contiguous*, i.e., elements that overlap in one dimension cannot do so in the other and (ii)

elements are *fixed* in one dimension, and are allowed to "slide" only along the other.

4.8 Conclusion

Static memory planning is an NP-complete problem with applications of great potential. Existing solutions are either scalable or memory-efficient. We have presented `idealloc`, an implementation designed with low fragmentation, high performance and scalability in mind. Along the way we have reported numerous insights that may prove useful to practitioners and theorists in the future.

We have open-sourced `idealloc` and the benchmarks used [8].

[8] https://github.com/cappadokes/idealloc

Chapter 5

Translating Quality-Driven Code Change Selection to an Instance of Multiple Criteria Decision Making

This Chapter is a verbatim copy of the author's publication cited below:

Lamprakos, C. P., Marantos, C., Siavvas, M., Papadopoulos, L., Tsintzira, A.-A., Ampatzoglou, A., Chatzigeorgiou, A., Kehagias, D., and Soudris, D. Translating quality-driven code change selection to an instance of multiple-criteria decision making. *Information and Software Technology 145* (2022), 106851

The definition and assessment of software quality have not converged to a single specification. Each team may formulate its own notion of quality and tools and methodologies for measuring it. Software quality can be improved via code changes, most often as part of a software maintenance loop. This Chapter contributes towards providing decision support for code change selection given a) a set of preferences on a software product's qualities and b) a pool of heterogeneous code changes to select from. We formulate the problem as an instance of MCDM, for which we provide both an abstract flavor and a prototype implementation. Our prototype targets energy efficiency, technical

debt and dependability. This prototype achieved inconsistent results, in the sense of not always recommending changes reflecting the decision maker's preferences. Encouraged from some positive cases and cognizant of our prototype's shortcomings, we propose directions for future research. This Chapter should thus be viewed as an imperfect first step towards quality-driven, code change-centered decision support and, simultaneously, as a curious yet pragmatic enough gaze on the road ahead.

Our main contributions are: (i) quality-driven code change selection is defined as a MCDM problem and an abstract, extensible methodology built around the MCDM core is presented, (b) a prototype focusing on energy efficiency, technical debt and dependability, is described and evaluated, (c) related future work is proposed through a qualitative critique of the prototype's shortcomings.

5.1 Main Approach

The abstract form of our methodology is depicted in Figure 5.1. A DM wants to select one or more code changes that will be applied to a software project in order to improve one or more quality attributes, as those are defined in the project's NFRs. The proposed methodology requires the following:

- a set of NFRs[1]
- a set of code change[2] recommender processes, each targeting a separate non-functional requirement[3]
- a uniform set of impact models for quantifying the degree to which a code change could affect a quality attribute
- a set of functions that map candidate code changes to expected impacts on all quality attributes
- an MCDM algorithm

[1]From this point, we will use the term "non-functional requirements" for both the quality attributes' definition **and** the means for their assessment (metrics, thresholds, combination of metrics, etc.

[2]We define a code change as a **single**, contiguous edit in the source code. Thus, removing two unused variables from two non-neighboring spots of a program equals to two different code changes.

[3]Keeping a generic point of view, we do not impose any limitations on what these processes could look like. Maybe they are part of a static analysis tool, or an expert's opinion. The only invariant should be that each process generates a set of code change suggestions.

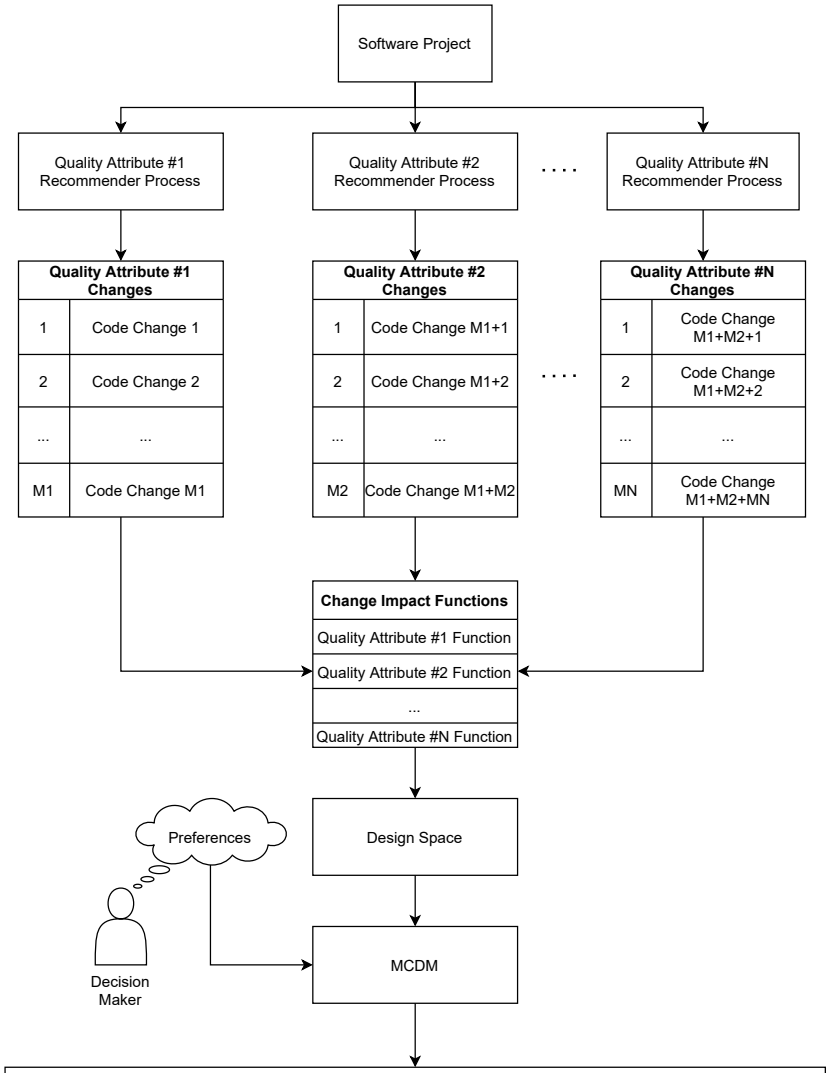


Figure 5.1: Functional diagram of the proposed method.

Each distinct quality attribute-centered recommender process generates a list of code change suggestions. Each candidate code change is given as input to all of the change impact functions, and thus estimates of how this change would affect each quality attribute are computed[4] Thus we get a design space on which the trade-offs between each candidate change are imprinted. A more concrete example can be seen in Table 5.1, created in the context of the prototype that we present in Section 5.2.

This design space and the DM's preferences are the inputs to the MCDM algorithm comprising our methodology's decision-making core.[5] The final output is a ranking of the code changes, based on each change's fitness to the DM's preferences, captured by the "Decision Value" field.

5.1.1 Decision-Making Core

We define two discrete spaces X and Ψ, comprising quality attributes and code changes respectively. We also define the set of real numbers $\mathbb{R}$ as the range of the change impact models. Each model is expressed by a function $f_{n \in X} : \Psi \to \mathbb{R}$. We denote each set of code change recommendations with $\Psi_i,\ \ i \in (1, 2, ..., N)$. There could exist empty sets or sets with overlapping elements but in any case, we are concerned with the union of all suggestion sets, which will be a subset of Ψ: $\Psi_p \coloneqq \bigcup_{i=1}^{N} \Psi_i$.

One can now visualize a table of $|\Psi_p|$ rows and $N + 1$ columns like Table 5.1[6]. This can be viewed as the *Design Space* block shown in Figure 5.1.

Table 5.1: A sample from the design space generated by this Chapter's prototype. Impacts on qualities were based on empirical measurements.

Code Change	**Energy**	**Technical Debt**	**Dependability**
Fix input/output issues	No Impact	Improve	Improve
Eliminate dead code segments	No Impact	Hinder	Improve
...	...	...	...

[4] An alternative wording is the notion of "trade-off analysis".

[5] MCDM is related to whatever scenario requires decision support given a list of available options, and multiple criteria based on which the options are evaluated.

[6] This table was developed for our prototype instance. Things left abstract in Section 5.1 are here specified. For example, we treat refactoring operations instead of generic code changes. We pick fuzzy sets as the range of the change impact functions. Further details are provided in Section 5.2.

The value of each suggested change (that is, its fitness to the DM's preferences) can be expressed as:

$$\begin{aligned} V_j &\coloneqq w_1 \cdot f_1(j) + w_2 \cdot f_2(j) + ... + w_N \cdot f_N(j) \\ &\coloneqq \sum_{i=1}^{N} w_i \cdot f_i(j) \\ j &\in \Psi_p, \quad i \in X \end{aligned} \tag{5.1}$$

For Eq. 5.1 to be evaluated, the key component missing is the N-dimensional weight vector $\overrightarrow{w}$. This corresponds to a quantitative representation of the DM's preferences[7]. To compute $\overrightarrow{w}$, most MCDM algorithms require from the user to provide a form of her preferences with respect to the criteria under discussion. In our case, these criteria are the chosen quality attributes.

5.2 Prototype

This section describes an elementary implementation (prototype) of the abstract methodology presented above. Our prototype was developed in the context of the Horizon H2020 project SDK4ED—see Section **??**. The specific software qualities chosen are energy efficiency (E), technical debt (TD) and dependability (D). It is not possible to elaborate on the details of our prototype's setup (metrics and derivation of each quality, change recommender processes, etc.). We do however provide a compact list of design decisions below[8]:

- **NFRs:** Energy consumption (milliJoules), principal technical debt (U.S. dollars), dependability (custom index). These were selected in the context of SDK4ED, the pilot use cases of which were either embedded, IoT or safety-critical applications. Technical debt aside (which could be considered universally important), the needs for high energy efficiency and optimal dependability are evident for such applications.
- ***Code change format:*** Refactoring. Architectural changes do not belong in this Chapter's scope[9].

[7]Strange as it may sound, a central problem addressed by MCDM is precisely this quantification of preferences, which according to the literature is non-trivial if consistency and reliability are sought after.

[8]For further information, please feel welcome to consult the project's publicly available deliverable 6.4 at `https://sdk4ed.eu/documents/`.

[9]That said, we see no particular reason why the abstract version of our methodology could not be applied to architecture-oriented changes.

- ***Change recommender process:*** Automatic identification of refactoring opportunities via static (technical debt, dependability) and dynamic (energy efficiency) analysis tools.
- ***Impact models:*** Lexicographical categories mapped to fuzzy sets. As already mentioned, a sample of the design space may be seen in Table 5.1.
- ***Derivation of impact functions:*** Static empirical analysis via iterative application of refactoring operations on several open-source code segments, and subsequent quality assessment on the SDK4ED platform[10].

For the MCDM component, we implemented the FBWM algorithm by Guo and Zhao [44]. Like other MCDM algorithms, its inputs are an encoding of the DM's preferences and the impact-annotated design space of available options (in our case, refactoring opportunities). *fbwm* is flexible enough to support a wide spectrum of preference scenarios, ranging from single objective optimization (improve only one non-functional requirement) to joint, three-way optimization (improve all NFRs).

For the impact models' range, we defined three triangular membership functions in $[-1, 1]$ with equal overlaps of 0.5 units (*Hinder, No Impact, Improve*). For more details on FBWM (preference encoding, mathematical description), the reader may consult [44].

5.3 Results

We devised 3 'preference scenarios', each with a different ordering of importance on the qualities. The software project used for evaluation was Rodinia [17], a widely-used benchmark suite for heterogeneous computing. It comprises both CPU and GPU implementations of a wide array of computational kernels, from backpropagation to video editing. The data depicted below are **average** improvements measured after applying, for each benchmark, the proposed top-ranked refactoring[11].

[10] See Footnote 8.

[11] The SDK4ED platform provides analysis infrastructure for direct measurement of the stated software qualities. We performed measurements both before and after applying the MCDM-proposed code change.

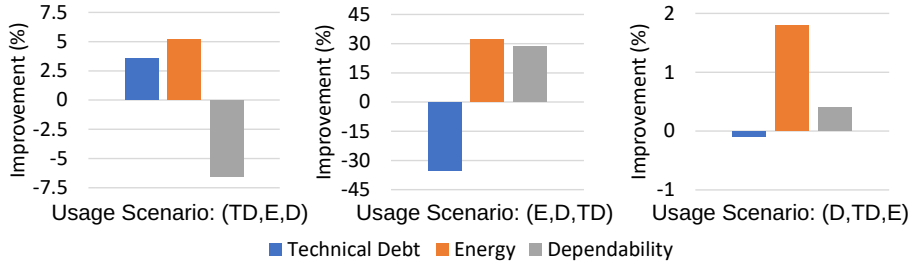


Figure 5.2: Experimental results: Average improvement (i.e. across all Rodinia benchmarks) of qualities after implementing the prototype's top-ranked suggestions.

5.3.1 Threats to Validity

- **Construct Validity:** To counteract the ambiguity of this manuscript's claims, we explicitly describe the proposed architecture components and we demonstrate results obtained through a first proof of concept. We take care to map all the proof of concept's components to the ones of the general architecture.
- **Internal Validity:** to ensure a cause-and-effect relationship in our findings, we isolate open-source benchmark applications and perform precisely what changes the analysis toolboxes suggest. We do this by a) choosing (mostly) refactoring operations as a means of code maintenance, b) adhering to well-known definitions for each refactoring and c) for Energy Toolbox's Acceleration (which is not a refactoring in the traditional sense) we stick to implementations contained in the benchmarks themselves.
- **External Validity:** the results and ideas in this Chapter ought to be generalizable. We thus define our experimental part as nothing more than a proof of concept. We list potential limitations, and provide a future work section.

Figure 5.2 displays the results retrieved from applying the top-ranked refactoring operations in three usage scenarios. *Usage Scenario* is a tuple showing the hierarchy of quality attribute importance as provided by the DM. From left to right, the elements denote a **best-worst-remaining** sequence. A result is positive if it respects the hierarchy posed by the DM, in the sense of improving the best NFR most and the remaining NFR less. As regards the worst NFR (in which the DM is the least interested), it is irrelevant whether it gets improved

or not by the made decision. A result is negative if it does not respect the imposed preference hierarchy in any way.

According to Figure 5.2, our results are inconsistent. We provide brief comments for each scenario in a left-to-right fashion.

- **(TD,E,D)**: negative. The best criterion (TD) is indeed improved, but as regards the other two qualities, the hierarchy is reversed (worst gets improved, remaining gets hindered). The respective percentages also do not follow the hierarchy (TD should see the biggest improvement).
- **(E,D,TD)**: mostly negative, in a similar vein with the above. A hopeful detail is that percentages are more fitting here, since the best criterion (E) does indeed see the biggest improvement.
- **(D,TD,E)**: mostly positive. Best criterion (D) gets improved, worst criterion (TD) gets hindered. Of course, E should have seen smaller improvement than D. But having **both** of the top qualities improved, just in inverse proportions, is in our opinion the least painful of all possible headaches.

5.4 Discussion and Future Work

Refactoring is known as a controversial means of quality improvement. We selected this code change paradigm due to our funding project's specifications. Future research could focus on alternative methods.

Moreover, a decision process is only as good as its inputs. We consider the main culprit behind our results' inconsistency the fact that our prototype uses a static, empirical model for the expected impact of refactoring on software quality. In this way it ignores the interdependencies in a specific project, which as a result distorts the design space (values in Table 5.1).

Existing works in the field of CIA can be used for locating which parts of code would be affected by a candidate refactoring, making a first step towards shedding the coarseness of our fuzzy impact model. The next step would then be to develop fine-grained local impact models. Such an approach would add a far greater amount of information in the design space, and the user could receive heterogeneous sequences of refactoring. This direction is orthogonal to the choice of the particular MCDM algorithm.

Last but not least, no comparison of our proposal with relevant works in the literature is presented. Acknowledging the issues discussed above does not

leave any room for sparring with the state-of-the-art, even if similar tools exist (inconsistent results appear for the simplest of baselines).

5.5 Conclusion

This work concerns quality-driven decision making in the source code level during the maintenance and evolution stages of an application's development lifecycle. We presented an abstract methodology for integrating arbitrary software quality definitions in a workflow producing a ranked list of quality-enhancing code changes. The central component of our proposal is the multiple-criteria decision making theory and family of algorithms.

We also presented a prototype of our methodology targeting the overall security, technical debt and energy efficiency of the Rodinia benchmarks. Even though our experimental results proved to be inconsistent, we believe our main idea to be worthy of further exploration—particularly its coupling with existing techniques of CIA.

Chapter 6

Conclusion

6.1 Summary of Presented Work

The central thesis of this text claims that meaningful work remains on the fundamentals of computer systems. Chapter 2 set the stage by demonstrating the impact of *dynamic memory allocation*, a fundamental operation, on the energy consumption and memory footprint of the CPython runtime. Chapter 3 proposed a mathematical representation of workload-allocator interaction and demonstrated its utility by defining a novel fragmentation measure on top of it, shown to correlate with actual memory footprint. Chapter 4 made a deep dive in the theoretical limits of `malloc` by delivering a high-performance, low-fragmentation solution for *static memory planning*, outperforming corresponding implementations from companies such as Google and OpenAI. Last but not least, Chapter 5 zoomed out from any particular optimization domain, adopting the perspective of application developers—it is them, after all, who are expected to apply the results of any deeper insights in practice and en mass.

Numerous other examples exist in the literature of researchers andor practitioners studying and advancing elementary operations: Colton, Krapivin and Kuszmaul recently proved that the insertion of elements in hash tables can be made much faster [27]. Haeupler et al. showed Dijkstra's algorithm to be universally optimal roughly a year ago [46]. If these sound too theoretical, the reader can instead contemplate how the Rust programming language, which enabled most of the work described here, is taking the world by storm[1].

[1] https://rustfoundation.org/members/

6.2 Future Work

Let us conclude with how the research described in this thesis could be utilized and extended in later efforts.

6.2.1 Dynamic Memory Allocation

Chapter 3 forms a connection between theoretical dynamic memory allocation and its real-world counterpart. It is motivated by a profound asymmetry between dynamic memory allocation's omnipresence and the scarcity of principled methods for understanding workload-allocator interaction. It describes a mechanism for extracting representations of workload-allocator pairs in the form of two-dimensional bin packing, and then proposes a novel fragmentation definition built on top. Despite operating on entirely virtual, simulation-generated data, our measure correlates with the memory footprint of a variety of workloads. Our study serves as a first piece of empirical evidence towards adopting bin packing-based methods for dynamic memory allocation.

Before proceeding, let us emphasize that the study described in Chapter 3 is incomplete and amenable to significant future extensions. For the moment we cannot answer precisely why some workloads exhibit negative correlation with 2DBP fragmentation. Noise is being added from (i) our transformations of complex calls to elementary `malloc/free` pairs, and (ii) noise from the OS. Extracting useful features from 2DBP representations is an important piece of future research. Another direction would be to incorporate, as mentioned, more fine-grain information such as a program's memory access trace.

Assume, now, that we know DSA to capture workload-allocator interaction. How does one capitalize on this knowledge? A first application would be identifying workloads that are provably sensitive to allocator policy–that is, workloads where significant savings in physical memory are expected if better placements are found. Such workloads would be perfect candidates for a benchmark suite evaluating placement policies. Next, assume a sensitive workload that is to be executed on a memory-constrained machine. It is critical to ensure that when deployed, the workload's peak RSS (or some other metric) is the minimum possible. A sandbox could be set up where different policies are iteratively tried on the workload's request trace, until the best one is found. The whole process would run offline, and not even access to the executable itself would be needed. Its request trace and a modifiable allocator would be the only required elements.

The generation of (approximately) optimal placements with respect to some more relevant criterion than the classical makespan could also be studied. Lower

bounds would then be assigned to sensitive workloads' achievable fragmentation. If the distance between said bounds and the top performing allocator were small, exploring custom policies for a particular workload would not be worth the effort. In the opposite case sandbox approaches like the one mentioned above could be explored.

Most importantly, DSA could yield more complex products: it could assist in performing feature extraction of workload-allocator pairs, for use in relevant machine learning tasks. We wonder what such tasks would look like; can, for instance, an allocator's policy be "learned"? Can similarity measures for allocators or workloads be established? We find great value in exploring such questions.

Last but not least, the same qualitative reasoning must be applied to quantitatively different contexts. The use of huge pages both on the OS and the allocator's side is common practice at the time of writing. To present a coherent view across as many allocators as possible, we decided to refrain from huge page-based experiments in the context of this dissertation (some allocators do not still support them). But exploring that space in the future would be more than useful. Our expectation is that fragmentation would correlate with peak RSS even more then, since page boundaries would be further apart.

6.2.2 Static Memory Planning

As regards Chapter 4, the most immediate domain of applying `idealloc` apart from the OS allocators research path mentioned above is *deep learning compilers*, where requests for memory are known in advance due to the fixed nature of neural networks. A particularly interesting fact is that DSA can be applied not only in "classical" deep learning workloads, but up to the immensely popular tasks of LLM inference and training as well [126, 43, 52].

As recent research has shown, there is also space for static methods in the context of dynamic, i.e., real-time, optimization. For instance, the DDTR framework [58] contains a static memory optimization step. We argue that the main consideration for such applications are the latency constraints imposed by each individual application enclosing this dynamic refinement step. We have already shown that `idealloc` features extremely low and configurable latency.

On the front of extending and optimizing `idealloc`, we find the task of making it even faster and memory-efficient *in itself* both interesting and valuable. The most evident weakness in our current implementation is that the interference graph quickly reaches gigabytes of required size. More compact data structures with the same function should be explored. Moreover, due to the boxing

algorithm's semantics, additional constraints could be introduced. For example, buffers that for any reason must be placed in contiguous memory (SOMAS introduces such constraints in the context of parallel training [73]) could be boxed together as a pre-processing step and viewed as an "original" job.

Another interesting piece of future work would be to enlarge the domain of *shapes* that `idealloc` handles. For instance, De Greef et al. have shown that array storage in embedded applications often exhibits trapezoid overall patterns [23]. Could the boxing semantics of `idealloc` be made so as to accommodate such shapes? Of course, this would imply the need for new *theory* as well, since DSA is by definition limited to rectangles.

Of course, the hardest but most rewarding `idealloc`-related future research path is that of *taming randomness*. Though we hold the belief that stochasticity works to our allocator's advantage, we find the idea of adding some on-line statistical inference module for judging whether it makes sense to keep iterating pretty interesting. Apart from that, one could try to decouple implementation-generated randomness (e.g., Rust's unstable sorting algorithm) to algorithm-generated randomness, that is, the critical point selection of Theorem 2. The reader should be warned, though, that our current intuition is that such decoupling is impossible. If we're wrong, researchers could proceed, once the decoupling is made, to add some meta-optimization element controlling the algorithm-generated randomness. The main difficulty we see in that enterprise is the inability to maintain a coherent state space given the parallelism involved.

6.2.3 Source-level Model Construction

Chapter 5 formulates the task of improving software quality as an instance of MCDM. It presupposes the existence of high-level models predicting the impact of candidate refactorings on a predefined set of NFRs. Actualizing this assumption is the most valuable piece of future work. For example, as regards energy consumption, a high-level model could be built via utilizing low-level information, e.g., context-aware consumption of basic blocks. In Appendix A we demonstrate the feasibility of collecting such information.

Nevertheless, we must acknowledge that such expectations may not be feasible to realize. The process of maintaining and refining large-scale software projects involves iteration and manual actions instead of automation and reliance on off-line oracles.

Appendix A

Reliable Basic Block Energy Accounting

Our goal is to conduct energy accounting of basic blocks using commodity hardware and software. With everyone nowadays carrying a small computer in their pocket, as well as trends such as edge computing and the Internet of Things, many researchers have turned to the energy efficiency of software [104, 36].

The term "energy accounting" refers to measuring the energy consumption of an entity of interest. Neugebauer and McAuley, for example, proposed process-level energy accounting in 2001 [95]. The further down the abstraction ladder one climbs, the bigger problems they face due to the increased frequency of events. The seminal work on the instruction level, conducted by Tiwari et al. in 1996 [117], utilizes physical measurements of electrical current to that end.

We find basic blocks of machine instructions to be a reasonable compromise residing in a low enough level to be exploitable by the compiler, while at the same time being coarse enough in granularity to allow energy consumption measurement. We are not the first to reach this idea. Ten years ago, Mukhanov et al. presented a tool for basic block-level energy profiling [92]. Jayaseelan et al. posited that a basic block's energy consumption can be inferred without resorting to measurements first, and based their derivation to instruction and architecture-specific models [56]. Then they assigned worst-case energy consumption bounds on basic blocks. A similar approach is followed by Pallister et al. [100] Chen et al. operate on the basic block level but are interested in instruction throughput instead of energy accounting [20, 88] (they also deal with steady-state blocks while we are interested in monitoring them in the wild, i.e., along with their

execution's context).

Here, we show that it is possible to achieve reliable basic block energy accounting *with commodity equipment*, just by utilizing existing technologies in the modern software stack.

- **Gap in the SOTA:** Seamless energy consumption measurement at the basic block level without any special equipment or software beyond what modern computers offer as "batteries included".
- **Contribution:** Open source methodology leveraging (i) integrated power models like RAPL, (ii) modular compiler infrastructure like LLVM and (iii) low-overhead processor tracing like PT.

This Chapter is a verbatim copy of the author's publication cited below:

Lamprakos, C. P., Bouras, D. S., Catthoor, F., and Soudris, D. Reliable basic block energy accounting. In *Embedded Computer Systems: Architectures, Modeling, and Simulation* (Cham, 2023), C. Silvano, C. Pilato, and M. Reichenbach, Eds., Springer Nature Switzerland, pp. 193–208

Modeling the energy consumption of low-level code will enable (i) a better understanding of its relationship to execution time and (ii) compiler/runtime optimizations tailored for energy efficiency. But such models need reliable ground truth data to be trained on. We thus attack extracting machine-specific datasets for the energy consumption of basic blocks–a problem with surprisingly few solutions available. Given the impact of execution context on energy, we are interested in recording *sequences* of basic blocks coupled to corresponding energy measurements. Our design is lightweight and portable; no manual hardware/software instrumentation is required. Its main components are an energy estimation interface with sufficiently high refresh rate, access to an application's complete execution trace, and LLVM pass-based instrumentation. We extract half a million basic block-energy mappings overall, and achieve a mean whole-program error of $\sim$3% on two different machines. This Chapter demonstrates that commodity resources suffice to perform a very crucial task on the road to energy-optimal computing.

Recent years have witnessed a proliferation of low-power embedded devices [110] with power ranging from few milliwatts (battery-powered) to microwatts (batteryless), and a plethora of techniques have been produced that yield significant results [1]. Furthermore, "green" commercial CPUs have become more and more available on the market [48], especially for mobile phones due to

their battery needs [85]. However, improvements in battery density and energy harvesting have failed to mimic Moore's law. Battery density has the slowest improvement in mobile computing and it does not scale exponentially [101]. Battery capacity has increased very slowly, with a factor of 2 to 4 over the last 30 years, while computational demands have drastically risen over the same time.

There is also a concern that energy efficiency improvements will not be sustained, as the "low hanging fruit" have already been reaped, and that the continued increase in compute demand might not be offset in the coming years. Thus, energy remains a formidable bottleneck. The ability of energy efficient hardware to satisfy the increasing computational needs of the market while keeping energy and power stable has turned into an uphill battle. Thus more and more research has turned towards energy efficient software [104, 36].

This Chapter's premise is that *existing commodity tools can be leveraged in order to study the energy consumption of programs*, without needing any special instrumentation. The first step for such a study is the extraction of reliable data in as low an abstraction layer as possible. We measure the energy consumption of basic blocks; such fine-grain accounting is known to be of great value in profile-guided optimization [92]. We build on the commercially available infrastructure provided by Intel's RAPL [22], the accuracy of which has been extensively validated [59, 47]. Contrary to the state-of-the-art, our method is simpler, more universal, and of equivalent effectiveness. It can be applied to any platform exposing a RAPL-like interface and offering processor tracing functionality (there already exists work replicating RAPL for AMD architectures [113], and processor tracing has long been standardized via Nexus IEEE 5001 [115], with two well-known instances being Intel's PT and ARM's CoreSight [4]).

Overall, our contributions can be summarized as:

- a lightweight, portable methodology for reliable basic block energy accounting
- an open-source implementation[1] of our method.
- an empirical evaluation on real programs for two `x86_64` machines

The rest of this Chapter is organized as follows. Section A.1 exposes the necessary background, while Section A.2 describes our measurement methodology. Section A.3 presents our evaluation procedure and discusses the corresponding results. Last but not least, Section A.4 draws our final conclusions.

[1]The source code is available at https://github.com/jimbou/energy_profiling

A.1 Background

This section provides a concise exposition of all concepts related to the work presented.

A.1.1 Basic blocks

A basic block is a straight line of machine instructions that are executed in sequence. It includes no branches, jumps, function calls, and in general any commands that disrupt normal control flow. Basic blocks usually contain few instructions. Figure A.1 offers a handy example derived from the programs used for evaluating this Chapter.

Basic blocks are a fundamental abstraction in compiler design [3]. They offer opportunities for optimization via, for instance, basic block reordering [96]. As such, attempts of modeling basic block properties such as instruction throughput [88] have been made.

A.1.2 RAPL

Intel's RAPL tool was conceived toward power capping DRAM power [22]. As part of its functionality, it employs hardware counter-based modeling to estimate a system's energy consumption in real time. The model's outputs are integrated with Linux and exposed to users via a simple, hierarchical file interface[2]. The files contain accumulated values of the energy consumed since startup. A respective interface for power estimates is also available.

RAPL measurements come in four different granularities: (i) package, (ii) core, (iii) DRAM and (iv) uncore. The package granularity is an aggregate of the rest. The accuracy of the exposed data has been rigorously validated in prior work [59, 47, 24].

A.1.3 PT

Intel's PT is an architectural extension that collects information about software execution such as control flow, execution modes and timings, and formats it into highly compressed binary packets. Trace data is recorded and must then be decoded, which amounts to walking the object code and matching the trace

[2]https://www.kernel.org/doc/html/next/power/powercap/powercap.html

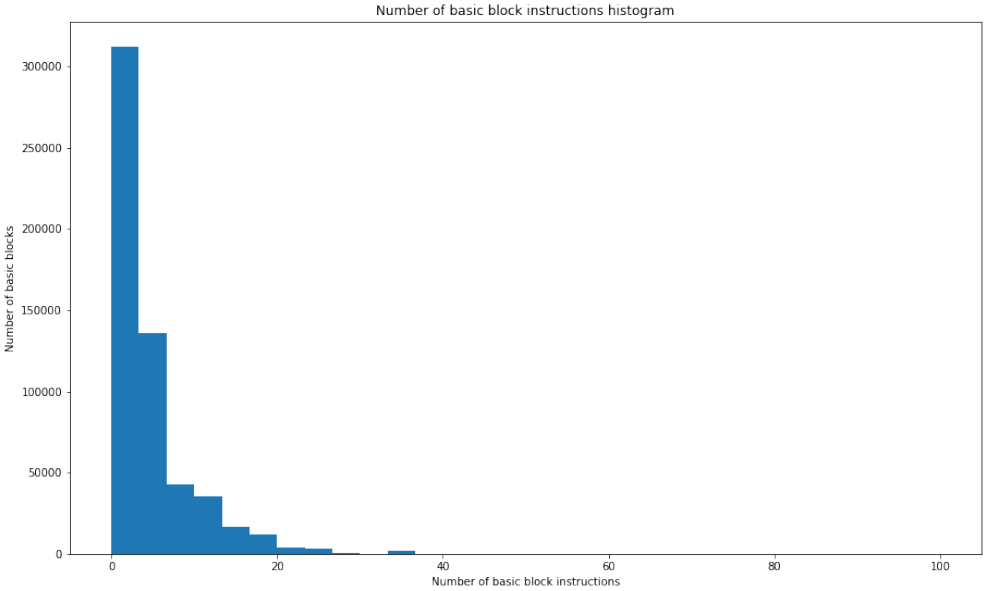


Figure A.1: Histogram for the number of instructions per basic block. To draw it, we traced all the basic blocks that got executed during our evaluation phase. Note that most of the time, a basic block is expected to contain 10 instructions or less.

data packets. PT has already been used as a basic building block of many research works [35, 80, 18, 127].

The main distinguishing feature of Intel PT is that software does not need to be recompiled, so it works with debug or release builds. A limitation is that it produces huge amounts of data (hundreds of megabytes per second per core) which takes a long time to decode–two to three orders of magnitude longer than what it took to collect. The performance impact of tracing itself varies depending on the use case and architecture.

A.1.4 Clang-enabled LLVM passes

LLVM is a compiler toolchain that can be used to develop a front-ends and back-ends for various programming languages and instruction set architectures [75]. It is designed around an agnostic IR that serves as a portable, high-level assembly language to be optimized via a variety of transformations over multiple passes.

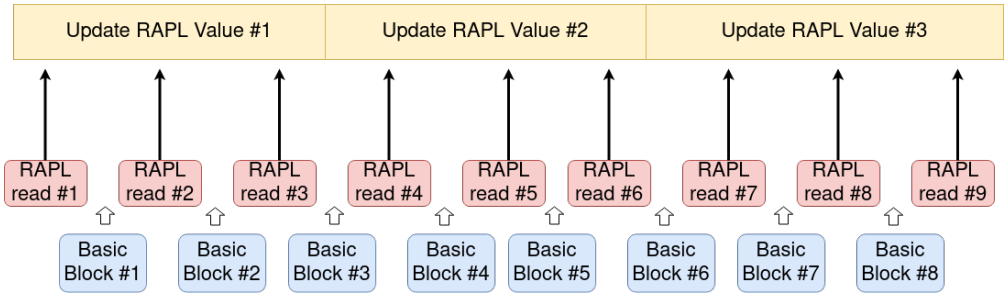


Figure A.2: Consecutive RAPL reads between consecutive basic blocks measure same energy due to RAPL's low refresh rate.

Each LLVM IR instruction is in SSA form to simplify dependency analysis between program variables.

Clang is a compiler for C, C++ and other C-derived languages and frameworks, which operates in tandem with LLVM [74]. In this Chapter, Clang acts a gateway to the LLVM Pass Framework. Passes perform the transformations and optimizations, build the analysis products to be used by said transformations and are, above all else, a structuring technique for compiler code. They can be used to mutate the IR code, e.g. print a number upon entry to a basic block, or compute properties, i.e. count the total number of function calls.

A.2 Method

The purpose of this work is to map energy consumption to basic blocks of executed code. To achieve this, the main idea is to perform an energy read before and after a basic block is executed. In theory, subtracting the two measured values provides us with the consumed energy:

$$E_{BB} = E_{after} - E_{before} \tag{A.1}$$

We implement our idea via writing an LLVM pass that instruments each basic block's entry point. The injected functionality amounts to opening the file to which RAPL writes, reading its value, and noting that value down to another file (created during the application's runtime to hold all energy readings). Every read value is accompanied by an identifier denoting the basic block that was thereafter executed. We will be referring to this read-write process as "RAPL reads" from now on.

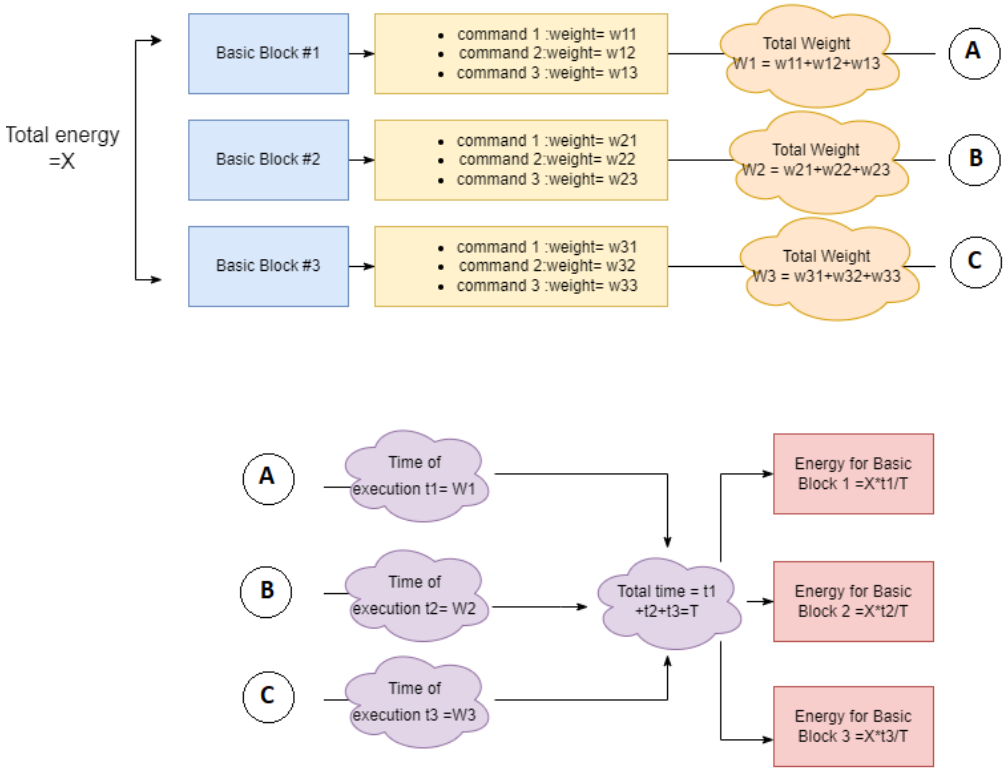


Figure A.3: Splitting energy between multiple basic blocks based on each block's total throughput. With the term "weight" the figures refer to the amount of cycles needed to execute an assembly instruction. The weights are, of course, instruction specific.

A.2.1 Obstacles and workarounds

Although at first glance the outlined strategy seems like a viable and simple solution, it presents a number of issues that demand a different approach. We now elaborate on these issues, and respective measures taken for mitigation.

RAPL read granularity

Our basic block instrumentation scheme is based on the idea that the RAPL energy registers are updated with higher frequency than that of basic block execution. If this does not hold, it is possible to assign zero energy consumption to basic blocks that execute faster than RAPL's refresh rate.

As noted earlier, RAPL has a refresh rate of ∼1 kHz, and basic blocks are most often sets of less than 10 instructions executed on processors with a clock frequency in the GHz order of magnitude. This leads to situations like the one illustrated in Figure A.2.

To deal with the coarse refresh rate of RAPL's registers, we used the workaround sketched in Figure A.3. Since RAPL updates occur slower than the retirement of basic blocks, it is obvious that multiple blocks will have been retired until the next time that the RAPL register is updated. To allocate this newly measured energy, denoted as X to the intermediate basic blocks, we make two assumptions: **(i)** the energy consumed by a basic block is analogous to its execution time and **(ii)** we can trust the technical report in [31] to derive individual clock cycle data per instruction.

Each of the above assumptions raises critical points to be addressed. With respect to the validity of [31], we pose our empirical evidence as a counterargument. If the instruction-specific clock cycle data we used were erroneous, we would not have managed to measure such a small error in our experiments. Note that the authors of [88] also use this resource. About our treating energy and execution time as linearly dependent, we view it as a heuristic rule that allows us to overcome a particular obstacle–not as an absolute, universal fact. In a similar fashion, Tiwari et al. [117] do assign execution time base costs on individual instructions, but at the same time emphasize that the actual relationship between energy consumption and latency is not trivial to formulate.

Thus a basic block comprising M instructions, each needing w_{iz} cycles to execute, has a total latency of $t_i = \sum_{z=1}^{M} w_{iz}$. N basic blocks execute in $T = \sum_{i=1}^{N} t_i$ processor cycles. Figure A.3 dictates that from the initial energy measurement X, each intermediate basic block gets the following quantity allocated to it:

$$E_{BB,i} = \frac{X \cdot t_i}{T} \tag{A.2}$$

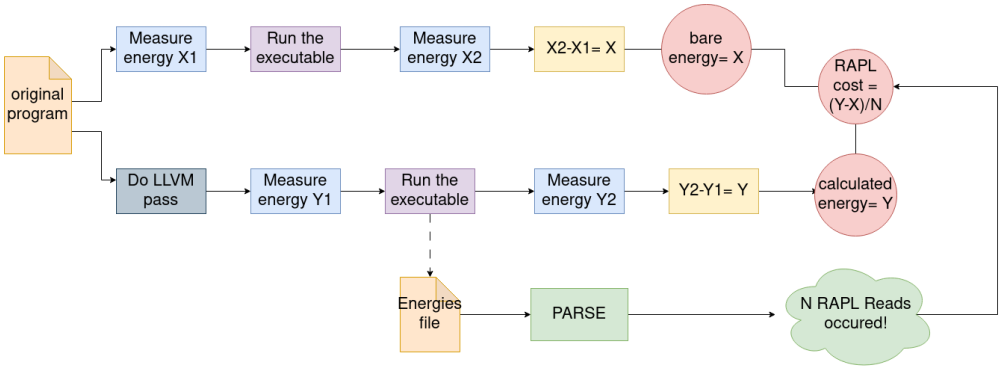


Figure A.4: Measure the cost of a RAPL read via comparing a "bare", i.e. uninstrumented program's energy consumption with that of an instrumented one.

RAPL read imposed energy overhead

A second problem noticed early on was the unaccounted cost of the RAPL read function itself, which could easily overshadow the basic block's own cost. RAPL reads amount to C functions doing file I/O to parse and record the model's measurements and as such are quite more complex than the basic blocks in between which they are instrumented.

We quantified and validated the involved energy overhead of RAPL read operations in two ways:

- do a RAPL read, execute a program that does a RAPL read N times, then do a final RAPL read. Subtract the difference between the last and the first reads with N to get a result.
- apply the method illustrated by Figure A.4. We execute 2 versions of a program: a bare one, having undergone no transformations, and an instrumented one on which our basic block-targeting pass has been applied. By parsing the RAPL readings file created by the instrumented flavor, we can deduce how many such reading were done in total. Dividing the energy difference between the 2 versions with the number of readings done provides a good approximation of a RAPL read's cost.

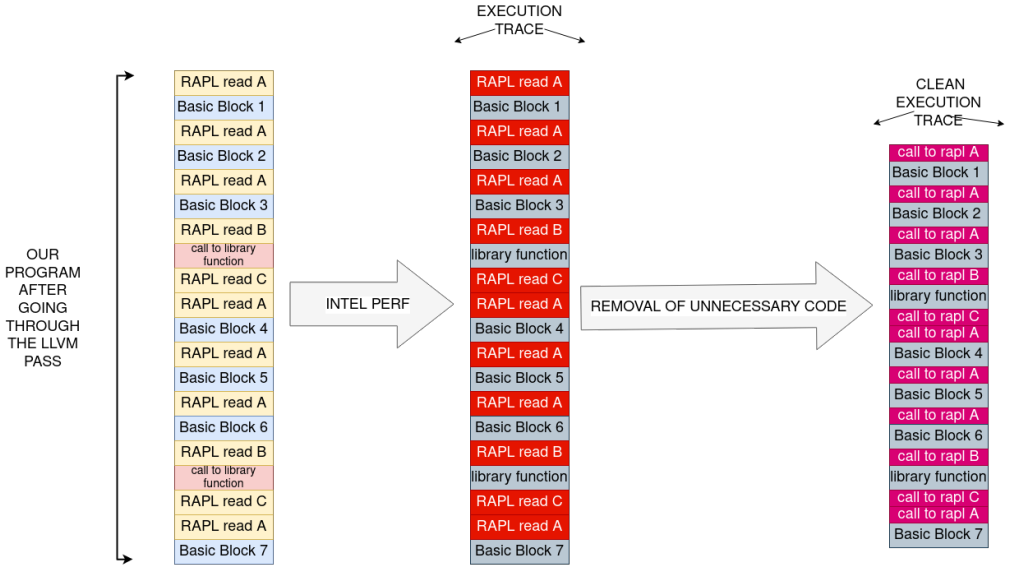


Figure A.5: Mitigation of external library code via Intel PT and two additional RAPL read tags.

Shared library code

The most important issue is that an LLVM pass operates only on the basic blocks of the application *itself*, and cannot reason about library functions linked at a later stage of the compilation process. Given the fact that the bulk of commands executed by applications are very often owed to external function invocations, we cannot ignore this situation.

A first mitigation we tried was to statically link compiled applications and lift the resulting binaries back to LLVM IR. We used `revng`[3], `llvm-mctoll`[4] and `mcsema`[5] but none of them proved to have plug-and-play compatibility with our method. On the occasions that we managed to lift binaries back to LLVM IR and apply the RAPL read pass, execution halted with segmentation fault.

To this end we devised the solution shown at Figure A.5. As a first step, we defined three possible tags, i.e. names, for our RAPL read function:

[3]https://github.com/revng/revng
[4]https://github.com/microsoft/llvm-mctoll
[5]https://github.com/lifting-bits/mcsema

- `RAPL_read_A` denotes readings taking place before normal basic blocks local to the application code that is being compiled
- `RAPL_read_B` corresponds to readings happening right before calling some external function
- `RAPL_read_C` represents, accordingly, readings made right after returning from some external function

Then we compile the targeted application while also applying our LLVM pass defining the three different RAPL read flavors. Note that the function body at each time stays the same–we only introduce additional names to differentiate between local and external cases.

Upon executing the instrumented binary via `perf-intel-pt`[6], we collect the total trace of its execution thanks to Intel PT. As a final step, we remove code corresponding to the RAPL reads themselves from the trace, and as a result receive a structure like the one at the far right of Figure A.5. Our final task is to parse this trace and assign energy costs to basic blocks according to the following procedure, the parts of which have been the subject of the present section:

1. compute or retrieve from storage the estimated energy overhead of an individual RAPL read.
2. begin parsing the trace.
3. in parallel, begin parsing the energy measurements file created by the RAPL read functionality.
4. if the trace traversal has reached a `RAPL_read_A` call, what follows are legitimate basic blocks. If it is a `RAPL_read_B`, what follows is external function code ending at `RAPL_read_C`. Split this segment in basic blocks by identifying branches, jumps etc.
5. stack parsed basic blocks for as long as the difference between consecutive energy measurements is zero. When it becomes non-zero, subtract the computed RAPL read energy overhead and allocate the rest between parsed blocks as illustrated in Figure A.3.
6. keep parsing until reaching EOF.

The end product is a sequence of basic blocks, many of them duplicates, each mapped to a particular energy cost. Repeating this process for many programs yields our final dataset.

[6] https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html

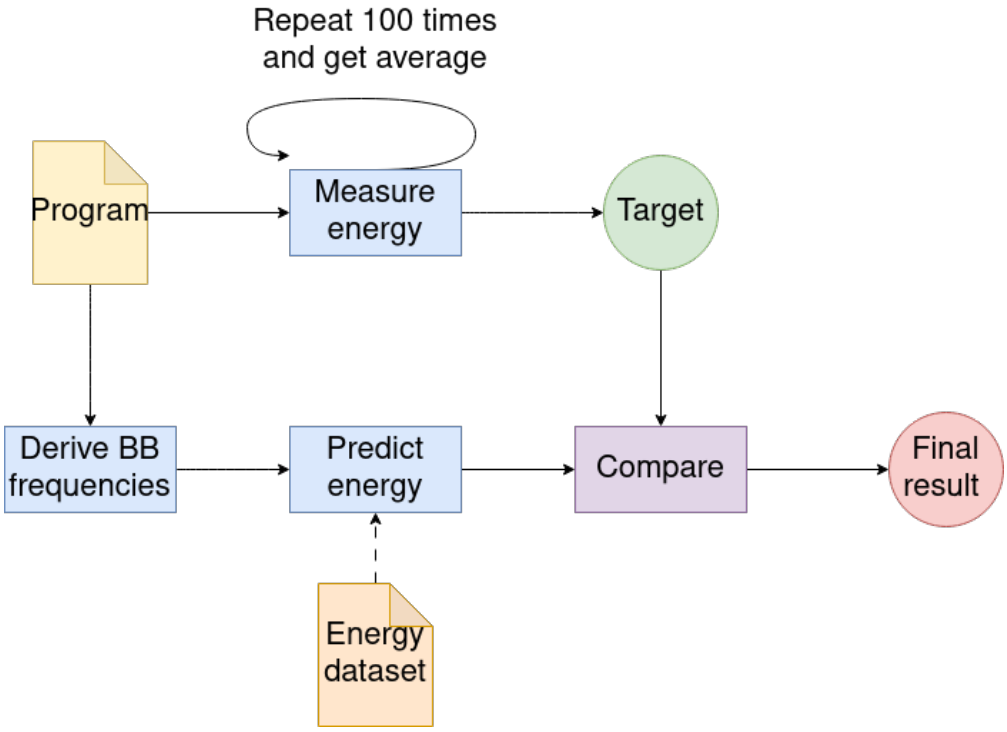


Figure A.6: Evaluation process based on whole-program error.

Execution context as inter-block effects

It is known that execution context heavily affects energy consumption. Via maintaining a 1-1 relationship as well as the same ordering between executed basic blocks and those stored by our method, execution context is made implicit. Future work aiming to utilize what this Chapter produces must be careful and model *sequences* instead of individual basic blocks.

A.3 Evaluation

Our claim is that the methodology described in Section A.2 yields a reliable dataset of energy consumption at the basic block granularity. To evaluate this claim, we must first define what reliability stands for. We thus borrow from the

Table A.1: Machines used for evaluation.

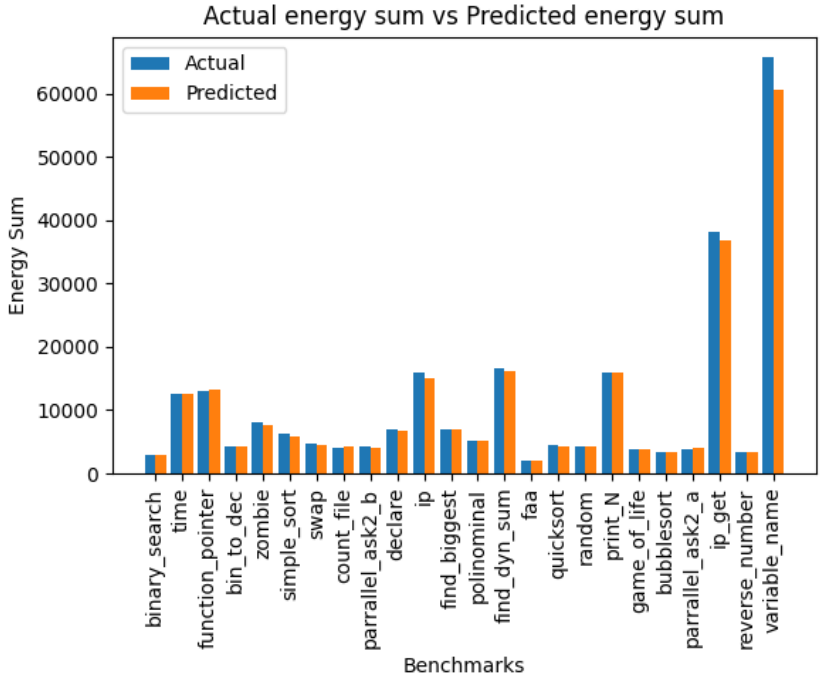
Machine	Spec	Value
A	Cores	12
	Clock frequency	2.2 GHz
	Main memory	16 GiB
	L1i cache	192 KiB
	L1d cache	192 KiB
	L2 cache	1.5 MiB
	L3 cache	9 MiB
B	Cores	8
	Clock frequency	3.4 GHz
	Main memory	32 GiB
	L1i cache	128 KiB
	L1d cache	128 KiB
	L2 cache	1 MiB
	L3 cache	8 MiB
Both	OS	Ubuntu 20.4 LTS
	Architecture	x86_64 (Skylake)
	Page size	4096 B

state of the art [92], and assume that a dataset is reliable to the degree that it achieves a *low whole-program error*.

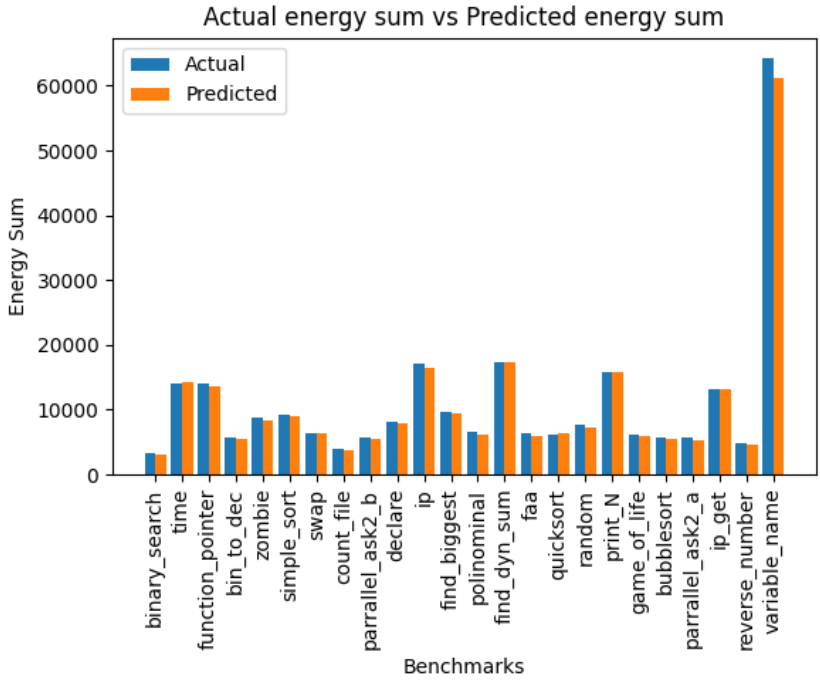
We form our evaluation process as illustrated in Figure A.6. After extracting our dataset from a set of programs, we aggregate duplicate entries via their mean energy cost. We revisit each program and execute it repeatedly to derive an average total energy measurement. In parallel, we trace it once more and count how many times each individual basic block was executed. We lookup our mean-aggregated dataset's contents and form an energy "prediction", by summing the corresponding frequency-energy products. The closer this prediction is to the actual total energy measured, the more reliable our dataset is.

A.3.1 Experimental setup

To run our experiments, we utilized two different machines described in Table A.1. A basic block energy dataset was formed by applying our method to 24 C single-threaded microbenchmarks from real-world workloads. We selected the *core* granularity exposed by RAPL, since it offered the highest refresh rate: the energy measurements in question thus reflect CPU costs exclusively.



(a) Machine A. Mean error is 3.43%.



(b) Machine B. Mean error is 2.66%.

Figure A.7: Main results. Vertical axes are measured in RAPL energy units. For the machines tested, each unit is 61μJ. ALEA [92], the current state-of-the-art in basic block energy accounting, reports a max error of 2%.

Our method can be applied as-is to larger applications. However, given its dependency to Intel PT execution traces, we opted for microbenchmarks in order to restrict trace file size. Since we operate on the basic block granularity, any potential shortcomings should be evident here too.

A.3.2 Results and discussion

The main result is depicted at Figure A.7. It is evident that our method achieves a very low whole-program error across all test cases. We complement our quantitative data with a qualitative comparison between the work presented in this Chapter and the state-of-the-art for basic block energy accounting, ALEA [92]:

Table A.2: A qualitative comparison of our tool versus the state-of-the-art.

Feature	**ALEA**	**This work**	**Practical consequences**
Basic measurement domain	Power	Energy	By leveraging RAPL for direct energy consumption data, we avoid the need to profile and integrate execution time. We also better utilize existing infrastructure.
Basic block identification	Disassembly	LLVM pass and execution trace traversal	We have already mentioned the problems that come with binary lifting. Our framework is more transparent and more *general*: no access to statically linked executables is assumed.
Instrumentation procedure	Manual	Automatic	By avoiding manual instrumentation, the proposed method is significantly more user-friendly and also less error prone.
Open-source availability	No	Yes	Our tool is exposed to the world for further experimentation, modification, optimization.
Reliability (whole-program error)	$\leq$2% everywhere	$\sim$2.5-3.5% mean	On average, our tool is of very similar effectiveness compared to the state-of-the-art.

The above points vouch for the greater transparency and flexibility supported by this Chapter's contributions, as well as our method's effectiveness in reliably capturing basic block energy consumption.

A.4 Conclusions

This Chapter presented a flexible, reliable methodology for performing basic block energy accounting. It leverages three commercially available tools: Intel's Running Average Power Limit and Processor Trace technologies, and the LLVM compiler toolchain. Our method is empirically shown to be of close effectiveness to the current state-of-the-art, while at the same time being more general in the sense of imposing fewer constraints to the user and automating crucial involved processes.

The main conclusion is that modern computing stacks expose all necessary equipment to form energy consumption datasets on the basic block level. On their own, these datasets have no practical value. The most obvious way forward is to build context-aware, i.e., sequential, models for use either by the compiler itself or higher in the abstraction hierarchy.

An ambitious direction would be to lower both the construction and the utilization of the models closer to the hardware, like Intel does with RAPL [22, 24]. Recall that the presented work *relies* on RAPL—a more independent and open-source approach would be extremely valuable in the RISC-V era.

Appendix B

Chapter 4 Addendum

B.1 Lemma 1

What follows is a verbatim copy of Lemma 1 by Buchsbaum et al. [13]. As in the main text, we represent parts of the proof that are of no concern to implementing the FU with "[...]".

LEMMA 1. *Given a set Y of unit-height jobs, all live at some fixed x-coordinate t, an integer box-height parameter H, and a sufficiently small $\epsilon > 0$, there exist a subset Y' of Y, $|Y - Y'| \leq 2H\lceil 1/\epsilon^2 \rceil$, a set B of boxes, each of height H, and a boxing of Y' into B such that at any x-coordinate u,*

$$L_B(u) \leq L_{Y'}(u) + 4\epsilon L_Y(u)$$

Proof. [...] partition the jobs of Y into *strips* [...]. The first two strips are defined as follows.

- Create a vertical strip consisting of the $H\lceil 1/\epsilon^2 \rceil$ jobs with the earliest starting x-coordinates (or fewer if there are not enough jobs)
- If any jobs remain, create a horizontal strip consisting of the $H\lceil 1/\epsilon^2 \rceil$ jobs that remain with the latest ending x-coordinates (or fewer if not enough jobs remain)

Define Y' to be the set of all jobs *not* in the first vertical or first horizontal strip. [...] Now partition the jobs of Y' as follows. As long as there are jobs remaining, repeat the following.

- Create a vertical strip consisting of the $H\lceil 1/\epsilon \rceil$ jobs that remain with the earliest starting x-coordinates (or fewer if there are not enough jobs left)
- If any jobs remain, create a horizontal strip consisting of the $H\lceil 1/\epsilon \rceil$ jobs that remain with the latest ending x-coordinates (or fewer if not enough jobs remain)

Now, for every vertical strip of Y', take the jobs in order of decreasing ending x-coordinate, in groups of size H (the last group of the last strip possibly smaller), and box them. Similarly, for every horizontal strip of Y', take the jobs in order of increasing starting x-coordinate, in groups of size H (the last group of the last strip possibly smaller), and box them. [...] □

We call the jobs in $Y - Y'$ *unresolved jobs*.

B.2 The Impossibility of Theorem 19

As above, we begin with a verbatim copy of Theorem 19 [13]:

THEOREM 19. *For all $\epsilon > 0$, there exists a polynomial-time $(2 + \epsilon)$-approximation algorithm for DSA.*

Proof. Consider some small positive δ to be determined. Let $X = X_s \cup X_l$, where X_s is the set of jobs of height less than $\delta^7 L$ and $X_l = X \setminus X_s$. Use Theorem 16 with error parameter δ to pack the jobs in X_s, yielding a $(1 + c'\delta)$-approximation for some constant c'. Apply the $(1 + \delta)$-approximation algorithm implied by Theorem 12 with the same δ to pack the jobs in X_l, which is possible because the load divided by the minimum height is at most $1/\delta^7$, which is certainly at most $C\ log_2 n / log_2 log_2 n$ for $C = 1/\delta^7$; this yields a $(1 + \delta)$ approximation. Choose δ so that $\delta(c' + 1) = \epsilon$. □

The impossibility of writing the above as a computer program function is evident. The parameter δ governs all steps, but is only determined in the end. Nevertheless, given the liberties we have taken with the rest of BA in order to make it functional, future attempts to "hack" Theorem 19 might prove fruitful.

Bibliography

[1] *Energy-Efficient Multi-mode Embedded Systems*. Springer US, Boston, MA, 2004, pp. 99–131.

[2] Agrawal, S., Ghosh, P., Kumar, G., and Radhika, T. Memory footprint optimization for neural network inference in mobile socs. In *2023 IEEE Women in Technology Conference (WINTECHCON)* (2023), pp. 1–6.

[3] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[4] Ali Zeinolabedin, S. M., Partzsch, J., and Mayr, C. Analyzing arm coresight etmv4.x data trace stream with a real-time hardware accelerator. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2021), pp. 1606–1609.

[5] Ameller, D., Ayala, C., Cabot, J., and Franch, X. How do software architects consider non-functional requirements: An exploratory study. In *2012 20th IEEE International Requirements Engineering Conference (RE)* (2012), pp. 41–50.

[6] and, J. H. Z. Significance testing of the spearman rank correlation coefficient. *Journal of the American Statistical Association 67*, 339 (1972), 578–580.

[7] and, S. S. Nonparametric statistics. *The American Statistician 11*, 3 (1957), 13–19.

[8] Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M.,

Hirsh, B., Huang, S., Kalambarkar, K., Kirsch, L., Lazos, M., Lezcano, M., Liang, Y., Liang, J., Lu, Y., Luk, C. K., Maher, B., Pan, Y., Puhrsch, C., Reso, M., Saroufim, M., Siraichi, M. Y., Suk, H., Zhang, S., Suo, M., Tillet, P., Zhao, X., Wang, E., Zhou, K., Zou, R., Wang, X., Mathews, A., Wen, W., Chanan, G., Wu, P., and Chintala, S. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2024), ASPLOS '24, Association for Computing Machinery, p. 929–947.

[9] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not. 35*, 11 (Nov. 2000), 117–128.

[10] Berger, E. D., Zorn, B. G., and McKinley, K. S. Reconsidering custom memory allocation. *SIGPLAN Not. 37*, 11 (Nov. 2002), 1–12.

[11] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), PACT '08, Association for Computing Machinery, p. 72–81.

[12] Bryant, R. E., and O'Hallaron, D. R. *Computer Systems: A Programmer's Perspective*, 3rd ed. Pearson, 2015.

[13] Buchsbaum, A. L., Karloff, H., Kenyon, C., Reingold, N., and Thorup, M. Opt versus load in dynamic storage allocation. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2003), STOC '03, Association for Computing Machinery, p. 556–564.

[14] Cai, Y., Kazman, R., Silva, C. V., Xiao, L., and Chen, H.-M. Chapter 6 - a decision-support system approach to economics-driven modularity evaluation. In *Economics-Driven Software Architecture*, I. Mistrik, R. Bahsoon, R. Kazman, and Y. Zhang, Eds. Morgan Kaufmann, Boston, 2014, pp. 105–128.

[15] Cavano, J. P., and McCall, J. A. A framework for the measurement of software quality. *SIGSOFT Softw. Eng. Notes 3*, 5 (Jan. 1978), 133–139.

[16] Ceurstemont, S. Controlling ai's growing energy needs. *Commun. ACM 68*, 3 (Feb. 2025), 15–17.

[17] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)* (2009), pp. 44–54.

[18] Chen, L., Sultana, S., and Sahita, R. Henet: A deep learning approach on intel® processor trace for effective exploit detection. In *2018 IEEE Security and Privacy Workshops (SPW)* (2018), pp. 109–115.

[19] Chen, T., Guo, Q., Temam, O., Wu, Y., Bao, Y., Xu, Z., and Chen, Y. Statistical performance comparisons of computers. *IEEE Transactions on Computers 64*, 5 (2015), 1442–1455.

[20] Chen, Y., Brahmakshatriya, A., Mendis, C., Renda, A., Atkinson, E., Sýkora, O., Amarasinghe, S., and Carbin, M. Bhive: A benchmark suite and measurement framework for validating x86-64 basic block performance models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)* (2019), pp. 167–177.

[21] Chrobak, Marek, and Ślusarek, Maciej. On some packing problem related to dynamic storage allocation. *RAIRO-Theor. Inf. Appl. 22*, 4 (1988), 487–499.

[22] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. Rapl: memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2010), ISLPED '10, Association for Computing Machinery, p. 189–194.

[23] De Greef, E., Catthoor, F., and De Man, H. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing 23*, 12 (1997), 1811–1837. Parallel processing and multimedia.

[24] Desrochers, S., Paradis, C., and Weaver, V. M. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems* (New York, NY, USA, 2016), MEMSYS '16, Association for Computing Machinery, p. 455–470.

[25] Evans, J. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada* (2006).

[26] Falessi, D., and Voegele, A. Validating and prioritizing quality rules for managing technical debt: An industrial case study. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)* (2015), pp. 41–48.

[27] Farach-Colton, M., Krapivin, A., and Kuszmaul, W. Optimal bounds for open addressing without reordering, 2025.

[28] Fernández-Sánchez, C., Garbajosa, J., Vidal, C., and Yagüe, A. An analysis of techniques and methods for technical debt management: A reflection from the architecture perspective. In *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics* (2015), pp. 22–28.

[29] Fernández-Sánchez, C., Garbajosa, J., and Yagüe, A. A framework to aid in decision making for technical debt management. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)* (2015), pp. 69–76.

[30] Fiedler, B., Meier, R., Schult, J., Schwyn, D., and Roscoe, T. Specifying the de-facto os of a production soc. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification* (New York, NY, USA, 2023), KISV '23, Association for Computing Machinery, p. 18–25.

[31] Fog, A. Instruction tables. *Technical University of Denmark* (2018).

[32] Fontana, F. A., Ferme, V., Zanoni, M., and Roveda, R. Towards a prioritization of code debt: A code smell intensity index. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)* (2015), pp. 16–24.

[33] Gao, S., Gao, C., Gu, W., and Lyu, M. Search-Based LLMs for Code Optimization . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)* (Los Alamitos, CA, USA, May 2025), IEEE Computer Society, pp. 254–266.

[34] Garey, M. R., and Johnson, D. S. *Computers and intractability*, vol. 174. freeman San Francisco, 1979.

[35] Ge, X., Cui, W., and Jaeger, T. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, Association for Computing Machinery, p. 585–598.

[36] Georgiou, S., Rizou, S., and Spinellis, D. Software development lifecycle for energy efficiency: Techniques and tools. *ACM Comput. Surv. 52*, 4 (Aug. 2019).

[37] Gergov, J. Approximation algorithms for dynamic storage allocation. In *Algorithms — ESA '96* (Berlin, Heidelberg, 1996), J. Diaz and M. Serna, Eds., Springer Berlin Heidelberg, pp. 52–61.

[38] Gergov, J. Algorithms for compile-time memory optimization. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* (USA, 1999), SODA '99, Society for Industrial and Applied Mathematics, p. 907–908.

[39] Gholami, A., Yao, Z., Kim, S., Hooper, C., Mahoney, M. W., and Keutzer, K. Ai and memory wall. *IEEE Micro 44*, 3 (2024), 33–39.

[40] Gholami, A., Yao, Z., Kim, S., Hooper, C., Mahoney, M. W., and Keutzer, K. Ai and memory wall. *IEEE Micro 44*, 3 (2024), 33–39.

[41] Glinz, M. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)* (2007), pp. 21–26.

[42] Grandoni, F., Mömke, T., and Wiese, A. A ptas for unsplittable flow on a path. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing* (New York, NY, USA, 2022), STOC 2022, Association for Computing Machinery, p. 289–302.

[43] Guo, C., Zhang, R., Xu, J., Leng, J., Liu, Z., Huang, Z., Guo, M., Wu, H., Zhao, S., Zhao, J., and Zhang, K. Gmlake: Efficient and transparent gpu memory defragmentation for large-scale dnn training with virtual memory stitching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2024), ASPLOS '24, Association for Computing Machinery, p. 450–466.

[44] Guo, S., and Zhao, H. Fuzzy best-worst multi-criteria decision-making method and its applications. *Knowledge-Based Systems 121* (2017), 23–31.

[45] Gálvez, W., Grandoni, F., Heydrich, S., Ingala, S., Khan, A., and Wiese, A. Approximating geometric knapsack via l-packings. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)* (2017), pp. 260–271.

[46] Haeupler, B., Hladík, R., Rozhoň, V., Tarjan, R., and Tětek, J. Universal optimality of dijkstra via beyond-worst-case heaps, 2024.

[47] Hähnel, M., Döbel, B., Völp, M., and Härtig, H. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev. 40*, 3 (Jan. 2012), 13–17.

[48] Haj-Yahya, J., Mendelson, A., Ben Asher, Y., and Chattopadhyay, A. *Power Management of Modern Processors.* Springer Singapore, Singapore, 2018, pp. 1–55.

[49] Hertz, M., Blackburn, S. M., Moss, J. E. B., McKinley, K. S., and Stefanović, D. Error-free garbage collection traces: how to cheat and not get caught. *SIGMETRICS Perform. Eval. Rev. 30*, 1 (June 2002), 140–151.

[50] Hunter, A., Kennelly, C., Turner, P., Gove, D., Moseley, T., and Ranganathan, P. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021), USENIX Association, pp. 257–273.

[51] Imanishi, A., and Xu, Z. A heuristic for periodic memory allocation with little fragmentation to train neural networks. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2024), ISMM 2024, Association for Computing Machinery, p. 82–94.

[52] Imanishi, A., and Xu, Z. A heuristic for periodic memory allocation with little fragmentation to train neural networks. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2024), ISMM 2024, Association for Computing Machinery, p. 82–94.

[53] Jacob, B., and Mudge, T. N. *Notes on calculating computer performance*. University of Michigan, Computer Science and Engineering Division . . . , 1995.

[54] Jarzębowicz, A., and Weichbroth, P. A qualitative study on non-functional requirements in agile software development. *IEEE Access 9* (2021), 40458–40475.

[55] Jato-Espino, D., Castillo-Lopez, E., Rodriguez-Hernandez, J., and Canteras-Jordana, J. C. A review of application of multi-criteria decision making methods in construction. *Automation in Construction 45* (2014), 151–162.

[56] Jayaseelan, R., Mitra, T., and Li, X. Estimating the worst-case energy consumption of embedded software. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)* (2006), pp. 81–90.

[57] Johnstone, M. S., and Wilson, P. R. The memory fragmentation problem: solved? In *Proceedings of the 1st International Symposium on Memory Management* (New York, NY, USA, 1998), ISMM '98, Association for Computing Machinery, p. 26–36.

[58] Katsaragakis, M., Papadopoulos, L., Konijnenburg, M., Catthoor, F., and Soudris, D. A memory footprint optimization framework for python applications targeting edge devices. *Journal of Systems Architecture 142* (2023), 102936.

[59] Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K., and Ou, Z. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst. 3*, 2 (Mar. 2018).

[60] Kierstead, H. A polynomial time approximation algorithm for dynamic storage allocation. *Discrete Mathematics 88*, 2 (1991), 231–237.

[61] Kierstead, H. A. The linearity of first-fit coloring of interval graphs. *SIAM Journal on Discrete Mathematics 1*, 4 (1988), 526–530.

[62] Kierstead, H. A., and Saoub, K. R. Generalized dynamic storage allocation. *Discrete Mathematics & Theoretical Computer Science Vol. 16 no. 3* (Nov 2014).

[63] Korf, R. E., Moffitt, M. D., and Pollack, M. E. Optimal rectangle packing. *Annals of Operations Research 179* (2010), 261–295.

[64] Kuszmaul, B. C. Supermalloc: a super fast multithreaded malloc for 64-bit machines. *SIGPLAN Not. 50*, 11 (June 2015), 41–55.

[65] Lamprakos, C., Xanthopoulos, P., Katsaragakis, M., Xydis, S., Soudris, D., and Catthoor, F. Futureproof static memory planning. *ACM Transactions on Programming Languages and Systems* (**submitted**).

[66] Lamprakos, C. P., Bouras, D. S., Catthoor, F., and Soudris, D. Reliable basic block energy accounting. In *Embedded Computer Systems: Architectures, Modeling, and Simulation* (Cham, 2023), C. Silvano, C. Pilato, and M. Reichenbach, Eds., Springer Nature Switzerland, pp. 193–208.

[67] Lamprakos, C. P., Marantos, C., Siavvas, M., Papadopoulos, L., Tsintzira, A.-A., Ampatzoglou, A., Chatzigeorgiou, A., Kehagias, D., and Soudris, D. Translating quality-driven code change selection to an instance of multiple-criteria decision making. *Information and Software Technology 145* (2022), 106851.

[68] Lamprakos, C. P., Papadopoulos, L., Catthoor, F., and Soudris, D. The impact of dynamic storage allocation on cpython execution time, memory footprint and energy consumption: An empirical study. In *Embedded Computer Systems: Architectures, Modeling, and Simulation* (Cham, 2022), A. Orailoglu, M. Reichenbach, and M. Jung, Eds., Springer International Publishing, pp. 219–234.

[69] Lamprakos, C. P., Xydis, S., Catthoor, F., and Soudris, D. The unexpected efficiency of bin packing algorithms for dynamic storage allocation in the wild: An intellectual abstract. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2023), ISMM 2023, Association for Computing Machinery, p. 58–70.

[70] Lamprakos, C. P., Xydis, S., Catthoor, F., and Soudris, D. The unexpected efficiency of bin packing algorithms for dynamic storage allocation in the wild: An intellectual abstract. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2023), ISMM 2023, Association for Computing Machinery, p. 58–70.

[71] Lamprakos, C. P., Xydis, S., Kourzanov, P., Perumkunnil, M., Catthoor, F., and Soudris, D. Beyond rss: Towards intelligent dynamic memory management (work in progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (New York, NY, USA, 2023), MPLR 2023, Association for Computing Machinery, p. 158–164.

[72] Lamprakos, C. P., Xydis, S., Kourzanov, P., Perumkunnil, M., Catthoor, F., and Soudris, D. Beyond rss: Towards intelligent dynamic memory management (work in progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (New York, NY, USA, 2023), MPLR 2023, Association for Computing Machinery, p. 158–164.

[73] Lamprou, I., Zhang, Z., de Juan, J., Yang, H., Lai, Y., Filhol, E., and Bastoul, C. Safe optimized static memory allocation for parallel deep learning. In *Proceedings of Machine Learning and Systems* (2023), D. Song, M. Carbin, and T. Chen, Eds., vol. 5, Curan, pp. 305–324.

[74] Lattner, C. Llvm and clang: Next generation compiler technology. In *The BSD conference* (2008), vol. 5, pp. 1–20.

[75] Lattner, C., and Adve, V. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), pp. 75–86.

[76] Leijen, D., Zorn, B., and de Moura, L. Mimalloc: Free list sharding in action. In *Programming Languages and Systems* (Cham, 2019), A. W. Lin, Ed., Springer International Publishing, pp. 244–265.

[77] Levental, M. Memory planning for deep neural networks. *arXiv preprint arXiv:2203.00448* (2022).

[78] Li, R., Wu, Q., Kavi, K., Mehta, G., Yadwadkar, N. J., and John, L. K. Nextgen-malloc: Giving memory allocator its own room in the house. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2023), HOTOS '23, Association for Computing Machinery, p. 135–142.

[79] Liétar, P., Butler, T., Clebsch, S., Drossopoulou, S., Franco, J., Parkinson, M. J., Shamis, A., Wintersteiger, C. M., and Chisnall, D. snmalloc: a message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2019), ISMM 2019, Association for Computing Machinery, p. 122–135.

[80] Liu, Y., Shi, P., Wang, X., Chen, H., Zang, B., and Guan, H. Transparent and efficient cfi enforcement with intel processor trace. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), pp. 529–540.

[81] Maas, M., Andersen, D. G., Isard, M., Javanmard, M. M., McKinley, K. S., and Raffel, C. Learning-based memory allocation for c++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 541–556.

[82] Maas, M., Andersen, D. G., Isard, M., Javanmard, M. M., McKinley, K. S., and Raffel, C. Learning-based memory allocation for c++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 541–556.

[83] Maas, M., Beaugnon, U., Chauhan, A., and Ilbeyi, B. Telamalloc: Efficient on-chip memory allocation for production machine learning accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (New York, NY, USA, 2022), ASPLOS 2023, Association for Computing Machinery, p. 123–137.

[84] Maas, M., Kennelly, C., Nguyen, K., Gove, D., McKinley, K. S., and Turner, P. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2021), ISMM 2021, Association for Computing Machinery, p. 28–38.

[85] Mahendra Pratap Singh, M. K. J. Evolution of processor architecture in mobile phones. *International Journal of Computer Applications 90*, 4 (March 2014), 34–39.

[86] Mairiza, D., Zowghi, D., and Nurmuliani, N. An investigation into the notion of non-functional requirements. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (New York, NY, USA, 2010), SAC '10, Association for Computing Machinery, p. 311–317.

[87] Mansi, M., and Swift, M. M. Characterizing physical memory fragmentation, 2024.

[88] Mendis, C., Renda, A., Amarasinghe, D., and Carbin, M. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *Proceedings of the 36th International Conference on Machine Learning* (09–15 Jun 2019), K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97 of *Proceedings of Machine Learning Research*, PMLR, pp. 4505–4515.

[89] Michael, M. M. Scalable lock-free dynamic memory allocation. *SIGPLAN Not. 39*, 6 (June 2004), 35–46.

[90] Moffitt, M. D. Minimalloc: A lightweight memory allocator for hardware-accelerated machine learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (New York, NY, USA, 2024), ASPLOS '23, Association for Computing Machinery, p. 238–252.

[91] Mömke, T., and Wiese, A. Breaking the barrier of 2 for the storage allocation problem. *CoRR abs/1911.10871* (2019).

[92] Mukhanov, L., Nikolopoulos, D. S., and De Supinski, B. R. Alea: Fine-grain energy profiling with basic block sampling. In *2015 International Conference on Parallel Architecture and Compilation (PACT)* (2015), pp. 87–98.

[93] Murphy-Hill, E., Parnin, C., and Black, A. P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering 38*, 1 (2012), 5–18.

[94] Navasca, C., Maas, M., Maniatis, P., Lim, H., and Xu, G. H. Predicting dynamic properties of heap allocations using neural networks trained on static code: An intellectual abstract. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2023), ISMM 2023, Association for Computing Machinery, p. 43–57.

[95] Neugebauer, R., and Mcauley, D. Energy is just another resource: energy accounting and energy pricing in the nemesis os. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems* (2001), pp. 67–72.

[96] Newell, A., and Pupyrev, S. Improved basic block reordering. *IEEE Transactions on Computers 69*, 12 (2020), 1784–1794.

[97] Oh, D.-J., Moon, Y., Ham, D. K., Ham, T. J., Park, Y., Lee, J. W., Ahn, J. H., and Lee, E. Maphea: A framework for lightweight memory hierarchy-aware profile-guided heap allocation. *ACM Trans. Embed. Comput. Syst. 22*, 1 (Dec. 2022).

[98] Osterweil, L. Strategic directions in software quality. *ACM Comput. Surv. 28*, 4 (Dec. 1996), 738–750.

[99] Ouni, A., Kessentini, M., Ó Cinnéide, M., Sahraoui, H., Deb, K., and Inoue, K. More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process 29*, 5 (2017), e1843. e1843 smr.1843.

[100] Pallister, J., Kerrison, S., Morse, J., and Eder, K. Data dependent energy modeling for worst case energy consumption analysis. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems* (New York, NY, USA, 2017), SCOPES '17, Association for Computing Machinery, p. 51–59.

[101] Paradiso, J., and Starner, T. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing 4*, 1 (2005), 18–27.

[102] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research 12*, 85 (2011), 2825–2830.

[103] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. a. P., and Saraiva, J. a. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (New York, NY, USA, 2017), SLE 2017, Association for Computing Machinery, p. 256–267.

[104] Pinto, G., and Castor, F. Energy efficiency: a new concern for application software developers. *Commun. ACM 60*, 12 (Nov. 2017), 68–75.

[105] Pisarchyk, Y., and Lee, J. Efficient memory management for deep neural net inference. *arXiv preprint arXiv:2001.03288* (2020).

[106] Powers, B., Tench, D., Berger, E. D., and McGregor, A. Mesh: compacting memory management for c/c++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2019), PLDI 2019, Association for Computing Machinery, p. 333–346.

[107] Robson, J. M. An estimate of the store size necessary for dynamic storage allocation. *J. ACM 18*, 3 (July 1971), 416–423.

[108] Robson, J. M. Bounds for some functions concerning dynamic storage allocation. *J. ACM 21*, 3 (July 1974), 491–499.

[109] Robson, J. M. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal 20*, 3 (01 1977), 242–244.

[110] Salajegheh, M. N. *Software techniques to reduce the energy consumption of low-power devices at the limits of digital abstractions.* PhD thesis, 2012. AAI3556283.

[111] Savage, J., and Jones, T. M. Halo: post-link heap-layout optimisation. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (New York, NY, USA, 2020), CGO '20, Association for Computing Machinery, p. 94–106.

[112] Scherer, M., Macan, L., Jung, V. J. B., Wiese, P., Bompani, L., Burrello, A., Conti, F., and Benini, L. Deeploy: Enabling energy-efficient deployment of small language models on heterogeneous microcontrollers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 43*, 11 (2024), 4009–4020.

[113] Schöne, R., Ilsche, T., Bielert, M., Velten, M., Schmidl, M., and Hackenberg, D. Energy efficiency aspects of the amd zen 2 architecture. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)* (2021), pp. 562–571.

[114] Sekiyama, T., Imamichi, T., Imai, H., and Raymond, R. Profile-guided memory optimization for deep neural networks, 2018.

[115] Stollon, N. *Nexus IEEE 5001.* Springer US, Boston, MA, 2011, pp. 169–193.

[116] Tillet, P., Kung, H. T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In

Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (New York, NY, USA, 2019), MAPL 2019, Association for Computing Machinery, p. 10–19.

[117] Tiwari, V., Malik, S., Wolfe, A., and Lee, M. T.-C. *Instruction Level Power Analysis and Optimization of Software*. Springer US, Boston, MA, 1996, pp. 139–154.

[118] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society s2-42*, 1 (1937), 230–265.

[119] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., Vijaykumar, A., Bardelli, A. P., Rothberg, A., Hilboll, A., Kloeckner, A., Scopatz, A., Lee, A., Rokem, A., Woods, C. N., Fulton, C., Masson, C., Häggström, C., Fitzgerald, C., Nicholson, D. A., Hagen, D. R., Pasechnik, D. V., Olivetti, E., Martin, E., Wieser, E., Silva, F., Lenders, F., Wilhelm, F., Young, G., Price, G. A., Ingold, G.-L., Allen, G. E., Lee, G. R., Audren, H., Probst, I., Dietrich, J. P., Silterra, J., Webber, J. T., Slavič, J., Nothman, J., Buchner, J., Kulick, J., Schönberger, J. L., de Miranda Cardoso, J. V., Reimer, J., Harrington, J., Rodríguez, J. L. C., Nunez-Iglesias, J., Kuczynski, J., Tritz, K., Thoma, M., Newville, M., Kümmerer, M., Bolingbroke, M., Tartre, M., Pak, M., Smith, N. J., Nowaczyk, N., Shebanov, N., Pavlyk, O., Brodtkorb, P. A., Lee, P., McGibbon, R. T., Feldbauer, R., Lewis, S., Tygier, S., Sievert, S., Vigna, S., Peterson, S., More, S., Pudlik, T., Oshima, T., Pingel, T. J., Robitaille, T. P., Spura, T., Jones, T. R., Cera, T., Leslie, T., Zito, T., Krauss, T., Upadhyay, U., Halchenko, Y. O., Vázquez-Baeza, Y., and 1.0 Contributors, S. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods 17*, 3 (Mar 2020), 261–272.

[120] von Neumann, J. First draft of a report on the edvac. *IEEE Annals of the History of Computing 15*, 4 (1993), 27–75.

[121] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. Dynamic storage allocation: A survey and critical review. In *Memory Management* (Berlin, Heidelberg, 1995), H. G. Baler, Ed., Springer Berlin Heidelberg, pp. 1–116.

[122] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. Dynamic storage allocation: A survey and critical review. In *Memory Management* (Berlin, Heidelberg, 1995), H. G. Baler, Ed., Springer Berlin Heidelberg, pp. 1–116.

[123] Zhao, K., Xue, K., Wang, Z., Schatzberg, D., Yang, L., Manousis, A., Weiner, J., Van Riel, R., Sharma, B., Tang, C., and Skarlatos, D. Contiguitas: The pursuit of physical memory contiguity in datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2023), ISCA '23, Association for Computing Machinery.

[124] Zhao, L., and Hayes, J. H. Rank-based refactoring decision support: two studies. *Innovations in Systems and Software Engineering 7*, 3 (Sep 2011), 171–189.

[125] Zhao, P., Zhang, H., Fu, F., Nie, X., Liu, Q., Yang, F., Peng, Y., Jiao, D., Li, S., Xue, J., Tao, Y., and Cui, B. Memo: Fine-grained tensor management for ultra-long context llm training. *Proc. ACM Manag. Data 3*, 1 (Feb. 2025).

[126] Zhao, P., Zhang, H., Fu, F., Nie, X., Liu, Q., Yang, F., Peng, Y., Jiao, D., Li, S., Xue, J., Tao, Y., and Cui, B. Memo: Fine-grained tensor management for ultra-long context llm training. *Proc. ACM Manag. Data 3*, 1 (Feb. 2025).

[127] Zuo, Z., Ji, K., Wang, Y., Tao, W., Wang, L., Li, X., and Xu, G. H. Jportal: precise and efficient control-flow tracing for jvm programs with intel processor trace. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2021), PLDI 2021, Association for Computing Machinery, p. 1080–1094.

Statement on the use of Generative AI

I did not use generative AI assistance tools during the research/writing process of my thesis.
The text, code, and images in this thesis are my own (unless otherwise specified). Generative AI has only been used in accordance with the KU Leuven guidelines and appropriate references have been added. I have reviewed and edited the content as needed and I take full responsibility for the content of the thesis.

List of Publications

Lamprakos, C. P., Marantos, C., Siavvas, M., Papadopoulos, L., Tsintzira, A.-A., Ampatzoglou, A., Chatzigeorgiou, A., Kehagias, D., and Soudris, D. Translating quality-driven code change selection to an instance of multiple-criteria decision making. *Information and Software Technology 145* (2022), 106851

Lamprakos, C. P., Bouras, D. S., Catthoor, F., and Soudris, D. Reliable basic block energy accounting. In *Embedded Computer Systems: Architectures, Modeling, and Simulation* (Cham, 2023), C. Silvano, C. Pilato, and M. Reichenbach, Eds., Springer Nature Switzerland, pp. 193–208

Lamprakos, C. P., Papadopoulos, L., Catthoor, F., and Soudris, D. The impact of dynamic storage allocation on cpython execution time, memory footprint and energy consumption: An empirical study. In *Embedded Computer Systems: Architectures, Modeling, and Simulation* (Cham, 2022), A. Orailoglu, M. Reichenbach, and M. Jung, Eds., Springer International Publishing, pp. 219–234

Lamprakos, C. P., Xydis, S., Kourzanov, P., Perumkunnil, M., Catthoor, F., and Soudris, D. Beyond rss: Towards intelligent dynamic memory management (work in progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (New York, NY, USA, 2023), MPLR 2023, Association for Computing Machinery, p. 158–164

Lamprakos, C. P., Xydis, S., Catthoor, F., and Soudris, D. The unexpected efficiency of bin packing algorithms for dynamic storage allocation in the wild: An intellectual abstract. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2023), ISMM 2023, Association for Computing Machinery, p. 58–70

Lamprakos, C., Xanthopoulos, P., Katsaragakis, M., Xydis, S., Soudris, D., and Catthoor, F. Futureproof static memory planning. *ACM Transactions on Programming Languages and Systems* (**submitted**)

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
9, Iroon Polytechniou St.
GR-15772 Athens

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
Celestijnenlaan 200A box 2402
B-3001 Leuven