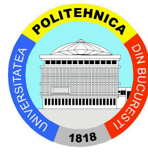
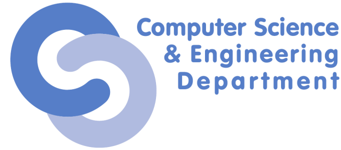
Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare





TEZĂ DE DOCTORAT

REZUMAT

Tehnici De Optimizare A Folosirii Memoriei in Sisteme De Calcul Moderne

Conducător Științific:
Prof. Dr. Ing. Răzvan-Victor Rughiniș

Autor:
Ing. Razvan Nitu

București, 2023

Cuprins

1 Introducere **1**
1.1 Contribuțiile Tezei 3
1.2 Structura Tezei 4

2 Stadiul Curent al Literaturii de Specialitate **6**
2.1 Tehnici de Management al Memoriei 6
2.2 Siguranța Memoriei în Nucleul Sistemului de Operare Linux 7
2.3 Tehnici de preluare 8

3 Performanța Memoriei - Adăugarea Suportului pentru Numărarea Referințelor in D **11**

4 Siguranța Memoriei - Îmbunătățirea Siguranței Aplicațiilor Folosind Limbajul D **13**

5 Arhitectura Memoriei - Optimizarea Prefetching-ului pentru Aplicații Existente **16**

6 Concluzii **18**
6.1 Listă de Publicații 19

Capitolul 1

Introducere

Progresele recente în arhitectura computerelor au permis procesoarelor să mărească continuu capacitatea la care execută instrucţiuni. Figura 1.1 evidențiază ritmul în care puterea de calcul a crescut de-a lungul anilor. După cum se poate observa, performanța memoriei nu a fost capabilă să țină pasul cu puterea de procesare, iar acest decalaj se adâncește pe măsură ce trec anii. În consecință, procesoarele puternice nu sunt exploatate la toate capacitățile lor din cauza blocajelor de date, ceea ce duce la înfometarea procesorului. Problema nu este reprezentată de lățimea de bandă a memoriei, ci mai degrabă de limitările fizice ale memoriei DRAM. În consecință, rezolvarea acestei probleme prin crearea de controlere de memorie mai bune sau memorie DRAM mai bună nu va elimina decalajul.

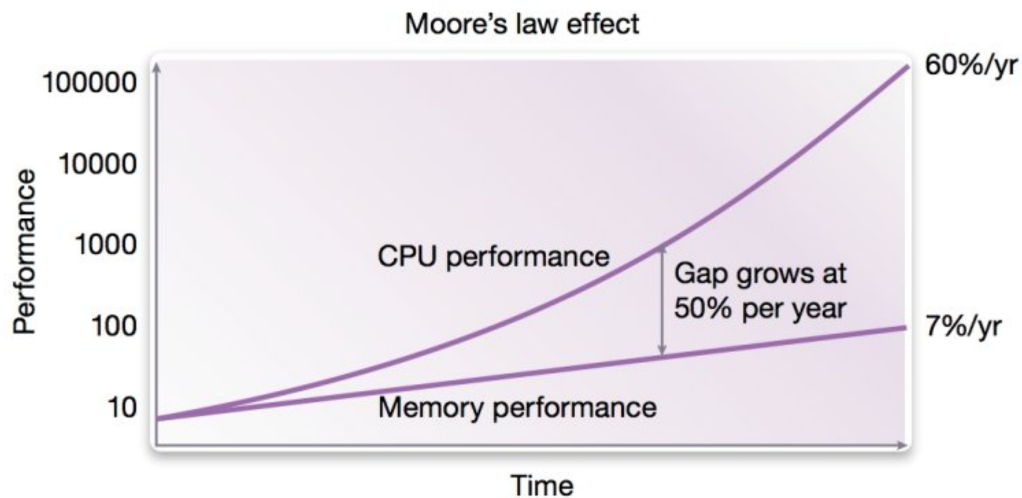


Figura 1.1: Puterea de procesare vs. performanta memoriei [1]

Sistemele de calcul moderne se înrăutățesc problema din cauza diferitelor straturi de abstracție care sunt stivuite deasupra memoriei fizice. Un sistem tipic utilizează un sistem de operare care gestionează memorie fizică și un alocator de biblioteci de rulare, care depinde de obicei de limbajul de programare care este utilizat, așa cum este prezentat în figura ??. Rezultatul este că operațiunile de memorie - care asigură alocarea, dealocarea sau utilizarea efectivă a memoriei - sunt multiplicate și trec prin mai multe indirectări, afectând în cele din urmă performanța generală.

Aceste abstractizări sunt puse în aplicare deoarece tehnologia computerelor a evoluat până la un punct în care performanța nu mai este singurul aspect critic al sistemelor. Securitatea,

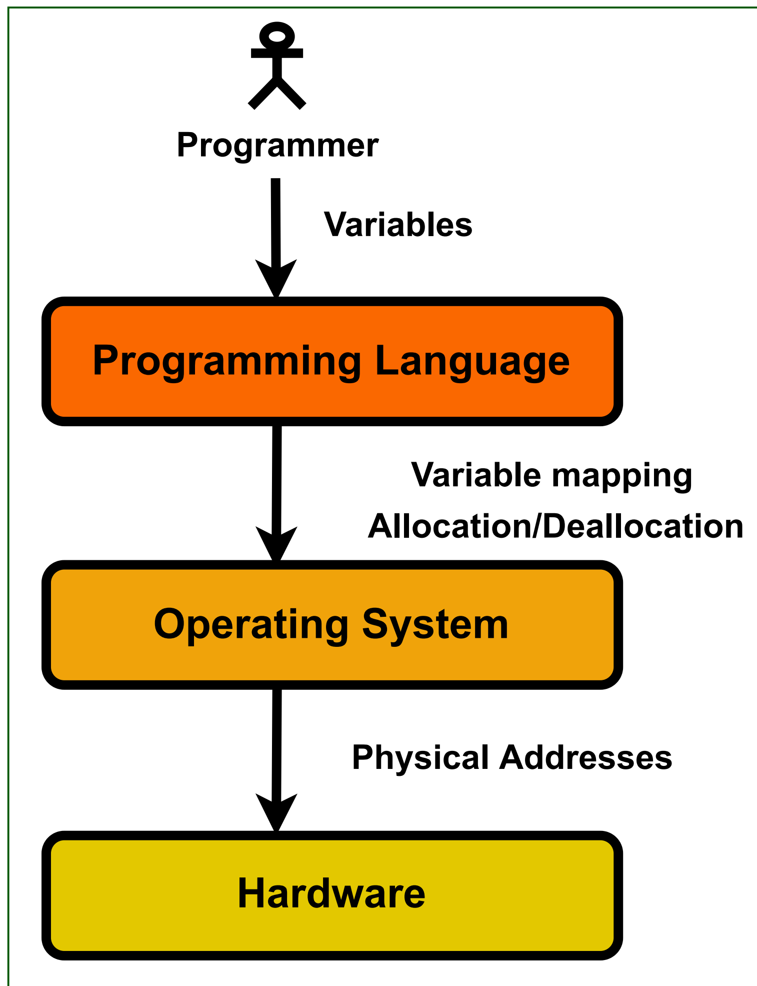


Figura 1.2: Niveluri de abstractizare în care se lucrează cu memoria.

mentenabilitatea și ușurința dezvoltării au devenit factori majori în adoptarea sau evitarea unei anumite tehnologii. În consecință, există compromisuri specifice aplicației realizate la alegerea unei anumite tehnologii pentru un anumit caz de utilizare.

În acest context, cercetătorii s-au străduit să îmbunătățească toate etapele în care memoria este manipulată: hardware, nivel de sistem de operare, nivel de limbaj de programare.

Din punct de vedere hardware, memoria cache a fost folosită pentru a salva datele care sunt reutilizate frecvent, astfel încât accesările ulterioare vor suferi o penalizare mai mică de latență. În plus, se adaugă prefetcher-uri pentru a încerca prezicerea accesărilor viitoare de memorie și aducerea datele în cache, înainte ca accesul real să aibă loc. Alte îmbunătățiri includ: instrucțiuni de preîncărcare care oferă utilizatorului posibilitatea de a spune procesorului ce adresă de memorie va fi utilizată în viitor, încărcături temporale care ocolesc ierarhia de memorie cache și controlere de memorie programabile. Cu toate acestea, soluțiile hardware sunt limitate deoarece acestea încearcă să remedieze problema pentru toate aplicațiile potențiale care ar putea fi rulate pe acesta. În schimb, în aceasta teză am pro-

pus o soluție nouă de preîncărcare a datelor in cache care se bazează pe analiza aplicației pentru a putea prelua perfect toate elementele de date necesare.

Sistemele de operare nu trebuie doar să asigure utilizarea optimă a resurselor sistemului, ci trebuie să asigure și aspectul de siguranță. În consecință, sistemele de operare utilizează module de memorie virtuală pentru a asigura sandboxing-ul procesului împreună cu alocatorii care implementează strategia de alocare. Cu toate acestea, dat fiind faptul că majoritatea sistemelor de operare sunt scrise în C, care este un limbaj de programare nesigur, nucleul în sine devine o problemă de securitate. În munca noastră, demonstrăm că nucleul Linux poate fi îmbunătățit în ceea ce privește siguranța printr-o tranziție incrementală la un limbaj de programare care este verificat mecanic pentru siguranță, D.

Limbajele de programare au evoluat până la punctul în care pot simplifica majoritatea sarcinilor programatorului: gestionarea memoriei poate fi facută automat, au fost puse în aplicare construcții expresive pentru a se ocupa de cele mai stricte nevoi, diferite instrumente ajută la menținerea și îmbunătățirea calității codului etc. Cu toate acestea, cele mai multe dintre acestea vin cu un cost. De exemplu, limbajele de programare care dispun de un colector de gunoi ridică povara gestionării manuale a memoriei, dar impun un consum suplimentar de resurse. Verificatorul de împrumut de ultimă generație de la Rust oferă garanții de securitate la un cost redus al resurselor, cu toate acestea, chiar și programele banale trebuie să specifice logica complexă a proprietății valorii. Ca alternativă, limbajul de programare D oferă posibilitatea de a verifica mecanic programe de siguranță asemănătoare C-ului, dar se bazează pe colectorul de gunoi pentru managementul memoriei. Perspectiva noastră este că utilizarea caracteristicilor de siguranță ale lui D împreună cu tehnica numărarii referințelor aduce avantajele atât ale gestionării automate a memoriei, cât și ale performanței memoriei.

Numai prin îmbunătățirea tuturor etapelor abstracției memoriei va fi posibil (1) să se micșoreze decalajul dintre puterea de procesare în creștere a CPU-urilor și performanța stagnantă a memoriei DRAM și (2) să se îndeplinească criteriile așteptate pentru siguranța memoriei și ușurința în utilizare. În această teză, deschidem calea către utilizarea unui limbaj sigur din punct de vedere al lucrului cu memoria, limbajul de programare D la toate nivelurile de abstractizare a memoriei, luând în considerare toate aspectele legate de memorie: performanță, siguranță și ușurință în utilizare. În plus, am propus o alternativă hardware la tehnicile de preîncărcare existente, care deschide calea către reducerea decalajului dintre accesul la calcul și la memorie.

1.1 Contribuțiile Tezei

Pe scurt, teza aduce următoarele contribuții:

- Am proiectat și implementat caracteristica de construcție a copiei pentru limbajul de programare D.
- Am proiectat și implementat capacitatea de a întrerupe tranzitivitatea calificatorilor

de tip din D prin introducerea unui nou cuvânt cheie *__mutable*.

- Am folosit caracteristicile menționate mai sus pentru a implementa structuri de date a căror referințe sunt numărate, demonstrând că acestea reprezintă o alternativă mai bună, în ceea ce privește performanța, la colectorul de gunoi.
- Am efectuat un audit de securitate al limbajului de programare D, evaluând nivelul de garanție pe care îl oferă subsetul sigur al limbajului. Am identificat 3 defecte în verificarea siguranței și am propus soluții pentru remedierea acestora.
- Am portat un driver de dispozitiv pentru kernelul Linux la D și am demonstrat că este posibil să integrăm un limbaj sigur pentru memorie în kernel-ul Linux fără pierderi de performanță.
- Am îmbunătățit biblioteca compilatorului D astfel încât instrumente ulterioare de analiză a codului pot fi dezvoltate plecânde de la implementarea efectivă a limbajului.
- Am îmbunătățit un instrument existent, *dpp*, care traduce automat fișierele antet C în D. Am demonstrat că acest instrument este utilizabil împreună cu fișierele antet ale nucleului Linux.
- Am realizat un sondaj, categorizând majoritatea instrumentelor de prefetching existente și evidențiind beneficiile și limitările acestora. Viitorii cercetători pot consulta sondajul nostru pentru a înțelege diferitele aspecte luate în considerare atunci când evaluează o tehnică de preluare preliminară.
- Am propus un cadru matematic pentru a înțelege capacitățile de prefetching ale unei aplicații date care rulează un hardware cu caracteristici date. Arhitecții de computere pot folosi aparatul nostru matematic pentru a înțelege blocajele unui sistem, în timp ce dezvoltatorii de software îl pot folosi pentru a înțelege blocajele aplicațiilor lor.
- Am propus o nouă tehnică de prefetching prin utilizarea unui FPGA pentru a accelera procesul.

1.2 Structura Tezei

Teza este structurată astfel:

Capitolul 2 prezintă cele mai relevante lucrări care au legătură cu sau au influențat această teză. Este organizat sub formă de secţiuni, fiecare tratând un anumit nivel al abstracţiilor memoriei pe care această teză le îmbunătățește. Prin urmare, cele 3 secțiuni sunt reprezentate de: tehnici de gestionare a memoriei, siguranța memoriei pentru nucleul Linux și tehnicile de preîncărcare.

Capitolul 3 prezintă contribuțiile noastre în ceea ce privește adăugarea suportului pentru numărărea referințelor D. Acesta detaliază toate aspectele tehnice ale implementării construcției copiei și *__mutable*. În cele din urmă, prezintă evaluarea implementărilor noastre

ale unor structuri de date care folosesc tehnica numărării referințelor pentru managementul memoriei, prezentând avantajele pe care le oferă față de colectarea gunoiului.

Capitolul 4 prezintă contribuțiile noastre în ceea ce privește îmbunătățirea aspectelor de siguranță a memoriei ale aplicațiilor. Inițial prezintă constatările noastre in urma unui audit de securitate pe limbajul de programare D. Continuăm prin îmbunătățirea interfaței de compilare pentru limbajul de programare D, care permite dezvoltatorilor să creeze instrumente care ajută la îmbunătățirea siguranța memoriei aplicațiilor. Vă prezentăm apoi călătoria noastră pentru a îmbunătăți siguranța memoriei nucleului Linux prin portarea unui driver către D. În cele din urmă, prezentăm îmbunătățirile pe care le-am adus instrumentului *dpp* pentru a permite utilizarea acestuia în contextul fișierelor kernel-ului Linux.

Capitolul 5 prezintă sondajul pe care l-am realizat pentru a înțelege mai bine tehnicile de preîncărcare existente. În acest proces, am identificat dimensiunile majore care pot fi utilizate pentru a clasifica astfel de tehnici. În plus, prezentăm cadrul matematic pe care l-am dezvoltat si care poate fi utilizat pentru a înțelege și optimiza mai bine capacitățile de preîncărcare ale unei aplicații date care rulează pe un anumit hardware. În cele din urmă, prezentăm noua noastră abordare de preîncărcare folosind un FPGA și evaluarea performanței acestuia.

Capitolul 6 încheie munca noastră și prezintă lista publicațiilor pe care se bazează scrierea curentă.

Capitolul 2

Stadiul Curent al Literaturii de Specialitate

În acest capitol vom evidenția care au fost îmbunătățirile majore care au fost propuse și implementate în comunitatea de cercetare în ceea ce privește optimizarea memoriei. Având în vedere gama largă de subiecte care trebuie discutate, am restrâns-o la domeniile care sunt deosebit de relevante pentru îmbunătățirile aduse de această teză.

În consecință, prezentul capitol este structurat în 3 subcapitole, fiecare evidențiind principalele lucrări de cercetare care au influențat contribuțiile acestei teze. Prin urmare, Secțiunea 2.1 discută care sunt tehnicile majore de alocare a memoriei și oferă o privire de ansamblu asupra limbajului de programare D, Secțiunea 2.2 evidențiază alternative la abordarea noastră de a face nucleul Linux mai sigur și limitările lor și Secțiunea 2.3 prezintă tehnicile majore de preîncărcare care sunt folosite în arhitecturile moderne de calculatoare, subliniind compromisurile care se fac.

2.1 Tehnici de Management al Memoriei

De la începutul capacității de a aloca direct memorie, managementul memoriei a devenit unul dintre cele mai complexe eforturi de programare. Prima și cea mai primitivă formă de alocare a memoriei este reprezentată de tehnica de gestionare manuală a memoriei. În această formă de gestionare a memoriei, utilizatorul are sarcina de a solicita și elibera memoria. O sarcină care este mai ușor de spus decât de făcut. Această formă de gestionare a memoriei a condus la nenumărate forme de bug-uri: folosire memoriei după ce a fost eliberată, dublă eliberare a memoriei, utilizare fără alocare, indicatori suspendați, memorie care nu este niciodată eliberată etc.

În consecință, au fost necesare forme mai avansate de management al memoriei, care să ridice complexitatea de pe umerii utilizatorilor. Rezultatul a fost implementarea limbajelor de programare colectate de gunoi sau a sistemelor de numărat referințe. Prima este o abordare grea în care este necesară o dimensiune considerabilă a resurselor de sistem, dar este extrem de ușor de utilizat (practic, complet transparent), în timp ce a doua este ușoară, dar vine cu limitări: nu acceptă referințe circulare și este într-o anumită măsură. invazivă.

2.1.1 Limbajul de Programare D

D este un limbaj imperativ, cu scop general, de programare a sistemelor. D își propune să îndeplinească cerințele unui limbaj de programare full stack, sub mantra „One language to rule them all". În consecință, D are suport pentru verificări mecanice de siguranță [2], programare funcțională [3], meta-programare [4], programare paralelă, programare orientată pe obiecte etc. Inspirat puternic din limbaje populare precum C, C++, Java și Python, D se bazează pe caracteristicile care există în alte limbaje: sintaxă în stil C și management manual al memoriei, clase similare Java și colectare de gunoi, sistem similar de șabloane C++, cum ar fi programarea funcțională similare Scheme-ului,etc. Deși unele dintre caracteristici se exclud reciproc (de exemplu: gestionarea manuală a memoriei și colectarea gunoiului), utilizatorul poate alege dintre diferitele opțiuni prin comutatoarele din linia de comandă.

2.2 Siguranța Memoriei în Nucleul Sistemului de Operare Linux

Îmbunătățirea siguranței kernel-ului Linux și a driverelor acestuia este punctul central al comunității de securitate profesională și de cercetare. Există diferite abordări, de la analiza statică a codului kernel-ului Linux [5–7] până la fuzzing [8–11] până la verificarea timpului de rulare și utilizarea instrumentelor [12, 13].

A fost abordată și ideea de a folosi limbaje de programare care implementează diferite caracteristici de siguranță a memoriei pentru a face codul kernel-ului Linux mai sigur.

2.2.1 Limbajul de Programare Rust

Disponibilitatea recentă a Rust ca limbaj de programare în kernel-ul Linux [14,15] deschide calea pentru adăugarea de cod scris într-un limbaj de programare securizat. Acest lucru este compatibil cu propria noastră abordare de a folosi D pentru a scrie cod în nucleul Linux. Deși garanțiile de siguranță a memoriei ale limbajului Rust sunt superioare în comparație cu D, integrarea acesteia în kernel-ul Linux este o sarcină foarte complicată. Ca dovadă, munca necesară pentru adăugarea suportului pentru Rust în nucleul Linux a fost realizată de 173 de persoane (prezente în jurnalul de modificări de comitere [16]) pe parcursul a 18 luni. Aceasta a inclus doar implementarea infrastructurii necesare pentru a integra codul Rust în kernel. Nu implementează niciun driver de dispozitiv sau nicio parte a nucleului Linux în Rust. Prin comparație, munca noastră a fost realizată de 3 persoane pe parcursul a 4 luni, inclusiv faza inițială de explorare a infrastructurii Linux, precum și portarea fișierelor de antet kernel. Durata reală de portare a driverului de dispozitiv a necesitat doar 2 până la 3 săptămâni. Cititorul ar trebui să ia în considerare că, între timp, s-au avansat lucrări pentru a automatiza portarea fișierelor de antet kernel în D [17], reducând astfel timpul necesar pentru integrarea driverelor de dispozitiv D la minimum. În plus, efortul de a integra Rust în nucleu a necesitat modificări ale compilatorului pentru

a se adapta codului ezoteric întâlnit, în timp ce munca noastră nu necesită nicio modificare a compilatorului.

2.3 Tehnici de preluare

Preluarea este o tehnică standard care a fost folosită în multe moduri diferite și în multe situații. Prezentăm pe scurt lucrări relevante și elaborăm, în continuare, abordarea noastră.

„Prefetching” poate fi implementat în hardware, software, precum și o combinație de hardware și software. Tabelul 2.1 grupează tehnici similare de preluare în categorii și evidențiază atributele relevante ale fiecărei tehnici.

Tehnici Hardware

Tehnicile **Hardware prefetching** necesită o unitate fizică specializată care se ocupă de monitorizarea acceselor la memorie și în mod automat generează solicitări de preluare preliminară. Această unitate este de obicei cuplată strâns la unitatea de execuție, în mod normal un nucleu de procesor. Acest lucru permite un nivel scăzut de latență de comunicare între nucleu și unitatea de prefetch. Unitățile hardware tind să nu suporte altceva decât o metoda prefetch care poate să nu fie optimă pentru toți algoritmii sau toate aplicațiile. În munca noastră și în această lucrare, arătăm că latența nu este un factor crucial pentru performanță, permițând un accelerator pentru a efectua în mod eficient preîncărcarea. Acest lucru ne permite, de asemenea, să implementăm tehnici mai complicate si superioare de prefetching.

2.3.1 Tehnici Software

Tehnicile de **prefetching software** se bazează pe indicii de prefetching sau instrucțiuni de preluare preliminară care sunt introduse în codul sursă. Acestea generează instrucțiuni de preîncărcare care sunt executate înainte de încărcarea efectivă. Aceste instrucțiuni sunt luate imediat și, prin urmare, nu blocheaza conducta de instrucțiuni. Această abordare are avantajul că nu necesită hardware suplimentar, deoarece majoritatea arhitecturilor implementează o formă de instrucțiuni de preluare anticipată. Cu toate acestea, tehnicile de preîncărcare software suferă de două neajunsuri majore: (1) introducerea cu precizie a instrucțiunilor de prefetching sunt dificile și (2) accese care au un ”load” in calculul adresei vor bloca în continuare conducta de instrucțiuni și, prin urmare, necesită calcul suplimentar care maschează prefatch-ul.

2.3.2 Helper Threads

Helper threads [36–40, 47–50] abordează preîncărcarea prin extragerea statică a codului pentru încărcări delincvente și rularea acestuia pe un context de fir de rezervă. Această abordare poate viza în mod optim orice acces prin creșterea numărului de helper threads. În plus, abordarea este suficient de flexibilă pentru a fi implementată atât în hardware [36–

Tabela 2.1: Tehnici de prefetching

Tehnica	SW	HW	Tipar	Analiza de predicție
Stride - hardware [18–23]	✗	✓	Simple stream	Dynamic at runtime
Pointer fetching - hardware [24–26]	✗	✓	Pointer chase	Dynamic at runtime
Indirect - hardware [27, 28]	✗	✓	Indirect	Dynamic at runtime
History based [29–32]	✗	✓	Complex stream	Dynamic at runtime
Run-ahead [33–35]	✗	✓	All	Dynamic at runtime
Helper threads [36–40]	✓	✓	All	Static
Software prefetching [41] [42]	✓	✗	Stride	Static
Software prefetching [43] [44]	✓	✗	Stride	Dynamic upfront
Software prefetching [45]	✓	✗	Indirect	Static
Software prefetching [46]	✓	✗	All	Dynamic upfront
Helper threads [47–50]	✓	✗	All	Static
Pointer fetching [51–54]	✓	✓	Pointer chase	Static
Programmable prefetcher [55]	✓	✓	All	Static
FPGA prefetching (this work)	✓	✓	All	Dynamic upfront

40] cât și în software [47–50]. Cu toate acestea, chiar și folosirea unui singur fir suplimentar vine la o penalizare de energie sporită pe nucleele de înaltă performanță. În plus, accesele care necesită încărcări în calcularea adresei lor se vor bloca și în absența unei cozi de evenimente hardware, sincronizarea încărcărilor devine costisitoare atât în termeni de implementare, cât și de performanță. În schimb, abordarea noastră necesită sincronizare minimală și rularea nucleelor de prefatch pe FPGA ar trebui să fie mai ieftină din punct de vedere al consumului de energie.

2.3.3 Hardware Programabil

Hardware-ul programabil folosește unități hardware specializate care sunt capabile să ruleze calcularea adresei specifice unei instrucțiuni. Jones și colab. au propus un prefetcher programabil conceput special pentru sarcini de lucru grafice care vizează anumite traversări [56]. Yi și colab. au proiectat un prefetcher hibrid care vizează accese indirecte la memorie [57]. Mai multe abordări au țintit structuri de date cu liste [51–54]. O abordare mai generală a fost dezvoltată de Jones și colab. [55] care utilizează mai multe nuclee mici

pentru a rula nuclee prefetch care sunt indicate în software. Această lucrare a dovedit o accelerare semnificativă pentru aplicații cu sarcină de lucru intensivă, totuși, designul nu este capabil să trateze modelul de urmărire a pointerului și dimensiunea nucleului de preluare prealabilă este limitată la doar câteva instrucțiuni, în timp ce [46] raportează nucleele de preluare anticipată care necesită până la 80 de instrucțiuni. Abordarea noastră este similară în esență cu [55], totuși, cu diferența că FPGA-ul este reconfigurabil. Aceasta oferă oportunitatea de a instanția hardware-ul minim necesar pentru a obține performanțe maxime. În plus, oferim un cadru matematic care ajută programatorii în instanțierea hardware-ul minim pentru preîncărcarea optimă.

Capitolul 3

Performanța Memoriei - Adăugarea Suportului pentru Numărarea Referințelor in D

Pe măsură ce se dezvoltă zilnic tot mai multe produse software, riscurile de securitate impuse de bazele de cod în creștere cresc. În 2019, Microsoft a raportat că cauza a 70% dintre erorile de securitate erau legate de memorie [58]. Costurile suportate din cauza defectelor de securitate și a exploatării acestora sunt de ordinul miliardelor [59], IBM System Science Institute afirmând că corecțiile costă de 100 de ori mai mult decât costurile de dezvoltare [60].

Pentru a ajuta la atenuarea riscului, limbajele de programare a sistemelor sigure pentru memorie, cum ar fi D și Rust, sunt din ce în ce mai mult adoptate de dezvoltatori. Un domeniu semnificativ în care sunt de dorit limbaje sigure pentru memorie este reprezentat de Internetul lucrurilor (IoT). Dispozitivele IoT au devenit o țintă populară de către atacatori pe care să le compromită și să le folosească ca armată de botnet IoT [61] care este folosită pentru a transporta atacuri împotriva întreprinderilor, guvernelor și chiar a unor țări întregi [62] [63]. Pentru a satisface nevoile dispozitivelor IoT, limbajele de programare folosite trebuie să producă și programe rapide și să fie luate în considerare resursele.

D este un limbaj de programare modern, la nivel de sistem, care își propune să ofere atât performanțe ridicate și siguranța memoriei într-un mod simplu, intuitiv și expresiv. Deși este un limbaj imperativ, D oferă concepte de stil funcțional, cum ar fi funcțiile pure și calificative de tip tranzitive. În plus, poate interopera cu codul C și C++ din fabrică, astfel oferind o cale simplă de migrare pentru codul moștenit.

D oferă un colector de gunoi (GC) [64] pentru funcțiile încorporate care utilizează memoria heap, cum ar fi array-urile dinamice și clasele, dar acceptă și gestionarea manuală a memoriei prin pointeri și *malloc/free*. Prin urmare, pentru situațiile în care colectorul de gunoi este inadecvat din cauza deficitului de resurse (memorie mică, număr mic de unități de calcul sau ambele) sau constrângeri în timp real, utilizatorii au posibilitatea de a implementa o strategie de alocare personalizată. Rețineți că, în acest scenariu, ușurința de utilizare este sacrificată pentru performanță, deoarece utilizatorul trebuie să gestioneze manual memoria. Asta s-a dovedit a fi un efort complex, consumator de timp și predispus la erori [65].

O a treia opțiune este reprezentată de contorizarea automată a referințelor (ARC) sub forma unei soluții de bibliotecă [66]. ARC este nu consuma multe resurse, deoarece de obicei stochează un câmp suplimentar de contor pentru fiecare instanță alocată. În ceea ce privește calculul, costul general adăugat constă în operaţii simple de adunare sau scădere. În plus, utilizarea ARC este aproape transparentă pentru utilizator: un obiect trebuie declarat ca fiind numărat de referințe și totul va fi gestiona în fundal. Asigurarea suportului pentru o astfel de opțiune este importantă, deoarece oferă flexibilitate maximă în termenii strategiilor de alocare: pentru majoritatea cazurilor, GC ar trebui să fie suficient; pentru cu constrângeri severe, în care GC nu poate fi suportat, trebuie utilizat ARC; pentru situatii extreme, unde nici macar ARC nu este suficient pentru a atinge garanțiile de performanță, ar trebui folosită gestionarea manuală a memoriei.

Capitolul 4

Siguranța Memoriei - Îmbunătățirea Siguranței Aplicațiilor Folosind Limbajul D

Siguranța memoriei a devenit o preocupare majoră pentru aplicațiile actuale. Odată cu migrarea mai multor aspecte ale vieții noastre de zi cu zi în online câștigăm eficiență, dar ne expunem atacatorilor cibernetici.

Hackerii vizează vulnerabilitățile dintr-un sistem și le exploatează. Astfel de acțiuni pot avea ca rezultat: obținerea de privilegii escaladate, cauzarea refuzului serviciului, criptarea datelor sensibile și solicitarea de răscumpărare, ștergerea datelor importante etc. Atacurile cu succes asupra companiilor sau persoanelor fizice duc de obicei la pierderi substanțiale, atât din punct de vedere financiar, cât și din perspectiva reputației.

Deși companiile investesc cantități tot mai mari de resurse în securitatea cibernetică [67], rezultatele nu sunt cele așteptate. Acest fenomen are o explicație simplă și constă în faptul că **securitatea este tratată retroactiv, nu proactiv**.

Atacurile cibernetice se pot clasifica în două mari categorii: ingineria socială și exploatarea vulnerabilităților.

Atacurile de inginerie socială apar atunci când un atacator încearcă să obțină o parolă sau privilegii ridicate, păcălind pe cineva într-o organizație că este cineva care nu este [68]. Exemple de atacuri de inginerie socială sunt: phishing, tailgating, shoulder surfing, ransomware etc. Principalul vector al atacurilor în atacurile de inginerie socială este omul.

Exploatarea vulnerabilităților se bazează pe faptul că există o greșeală sau un defect în implementarea unei aplicații, a unui sistem, a unui protocol etc. În timp ce ingineria socială atacă omul, exploatarea vulnerabilităților atacă programul.

Atât ingineria socială, cât și exploatarea vulnerabilității se bazează pe greșelile umane: în prima, un om identifică în mod greșit atacatorul ca fiind altcineva, în timp ce în cel urmă, un om face o greșeală tehnică atunci când implementează un program. Indiferent de câte resurse sunt investite, oamenii vor face greșeli. Acesta este motivul de bază pentru care îmbunătățirile de securitate au avut un succes limitat: ele abordează întotdeauna simptomul si nu rădăcină problemei.

Recent, au fost adoptate abordări mai bune pentru a îmbunătăți situația cu ingineria soci-

ală. Autentificarea cu doi factori [69] este un exemplu despre modul în care măsurile pot fi luate în mod proactiv: solicitați utilizatorului să treacă prin mai mulți pași de autentificare (parolă, biometric, parole unice etc.). Această situație presupune că utilizatorul va greși și va comunica parola unui atacator și necesită un pas suplimentar de autentificare.

Pentru exploatarea vulnerabilităților, cercetătorii propun instrumente formale de verificare pentru programe. Cu toate acestea, este greu să evaluezi corectitudinea oricărui program în mod automat, fără intervenția utilizatorului. Cu toate acestea, au fost făcuți pași în crearea standardelor și a instrumentelor formale de verificare care depind de program.

Din perspectiva limbajului de programare, siguranța este definită ca corectitudine în ceea ce privește utilizarea memoriei. Aceasta înseamnă că un program nu ar trebui să aibă pointeri atârnați, folosiți după eliberare, pointeri către cadre de stivă expirate etc. Unele limbaje sacrifică performanța pentru siguranța memoriei nepermițând utilizarea pointerilor (Java, Python, Javascript etc.), cu toate acestea, acest lucru înseamnă de obicei că este nevoie ca un colector de gunoi să urmărească durata de viață a variabilelor. La cealaltă extremă, avem limbajul de programare C care permite utilizarea foarte liberă a pointerilor care provoacă în cele din urmă introducerea a numeroase vulnerabilități. În ultimii ani, limbajele de programare precum Rust și D au câștigat atenție datorită abordării lor de mijloc, în care siguranța memoriei poate fi verificată mecanic de către compilator (adică, dacă un program se compilează, atunci nu există nicio situație în care să corupă memoria - astfel eliminând sursa principală a vulnerabilităților programului) fără a fi nevoie de un colector de gunoi.

Cu toate acestea, adoptarea unor astfel de limbaje tinere ridică provocări multiple:

- ușurința integrării cu programele existente. Nu vrem să reinventăm roata, așa că vrem să valorificăm bibliotecile existente scrise în alte limbaje.
- ușurință de utilizare. Programatorii vor să-și facă treaba, nu să petreacă ore în luptă cu limbajul, prin urmare un limbaj sigur pentru memorie nu trebuie să impună programatorului o complexitate suplimentară.
- scule existente. Programatorii nu adoptă limbaje dacă nu au un ecosistem puternic care oferă instrumente de dezvoltare.

În această lucrare, ne propunem să creștem utilizarea limbajelor sigure pentru memorie, arătând că D este un bun substitut pentru limbajele nesigure pentru memorie, cum ar fi C, datorită integrării sale bune cu familia C de limbaje de programare, a asemănării sale cu limbajele moderne precum Java și Python. Pentru aceasta implementăm modificări care îmbunătățesc starea actuală a instrumentelor.

Ca atare, principalele contribuții ale acestui capitol sunt:

- Efectuăm un audit de securitate pentru limbajul de programare D, identificând 2 defecte în sistemul său de analiză care pot duce la acceptarea unui cod nesigur. Propunem și implementăm soluții care elimină problemele descoperite.
- Analizăm instrumentele existente de dezvoltare software din ecosistemul D și ex-

tragem o interfață comună care poate fi folosită pentru compilatorul ca bibliotecă. Implementăm această interfață, deschizând astfel ușa pentru crearea mai multor instrumente pentru dezvoltatori D.

- Arătăm că limbajul de programare D poate fi folosit pentru a înlocui limbajul de programare C în scenarii de performanță și siguranță, cum ar fi sistemul de operare Linux. Implementăm și integrăm un driver de dispozitiv D în kernel-ul Linux.
- Îmbunătățim un instrument existent care generează automat fișiere antet D din sursele C pentru a fi utilizabile în interiorul nucleului Linux. Acest lucru ușurează integrarea codului D în nucleu.

Capitolul 5

Arhitectura Memoriei - Optimizarea Prefetching-ului pentru Aplicații Existente

Pe măsură ce diferența de viteză dintre procesoarele moderne și sistemul de memorie este în continuă creștere [70, 71], blocajul accesării memoriei în mașinile Von-Neumann de astăzi devine punctul dureros care inspiră diferite tehnici de optimizare, cum ar fi stocarea în cache [72] și preîncărcarea [73–76].

Preluarea este o tehnologie fundamentală pentru majoritatea sisteme de înaltă performanță astăzi [77–81]. Scopul preluării este de a recupera, într-un în timp util, date dintr-o memorie cu latență mare, de obicei DRAM și plasarea în memoria cache cu acces rapid. O caracteristică cheie a unui prefetcher este că își propune să preia datele necesare *înainte ca unitatea de calcul să îl acceseze și să îl folosească*. Preluarea poate reduce semnificativ timpul pe care un CPU trebuie să îl aștepte când accesează date.

Prefetcher-urile existente implementate în hardware [18–32] oferă o funcționare cu funcție fixă și nu se pot schimba în mod fundamental pentru a se adapta la aplicație, limitând performanța realizabilă.

Susținem că viitorii prefetcher-uri trebuie să fie configurabili pentru a suporta diferite strategii, eventual în măsura în care sunt configurate de software. Modelele de acces la memorie sunt bine cunoscute ca fiind dependente de aplicație, ceea ce face ca prelevarea să fie dificilă să fie efectuată atât în mod precis, cât și în timp util. De exemplu, diferite distanțe de preluare preliminară, de ex. cât de departe trimite prefetcher-ul cererile, poate duce la o variație de până la 10 ori a beneficiilor de performanță [82]. Prin urmare, susținem că preluarea preliminară trebuie să fie determinată de comportamentul dinamic al aplicației [46, 55, 82].

Acest lucru deschide un spațiu mare de proiectare pentru dezvoltator: ce strategie de preîncărcare ar trebui selectată? Când ar trebui trimisă o solicitare de preluare anticipată? Merită să păstrăm strategia actuală sau există un beneficiu să treci la una nouă (în timp ce se ține cont de potențialele cheltuieli generale ale acestei schimbări)? Ce parametri ar trebui selectați dacă strategia de preluare preliminară este parametrizabilă? Până acum, astfel de întrebări nu au putut fi abordate în mod sistematic.

În această lucrare, propunem un nou cadru analitic care, bazat pe măsurători ale execu-

ției aplicației, poate sugera strategii de preîncărcare aproape optime. După cunoștințele noastre, aceasta este prima lucrare care abordează preîncărcarea în acest fel. Metodologia noastra oferă două rezultate. Primul, pentru o anumită strategie de preîncărcare, analiza noastră oferă un program optimizat de preluări care este atât exact și oportun. Se poate folosi planul de preluare anticipată pentru a îmbunătăți performanța aplicației. În experimentele noastre, arătăm că accelerarea obţinută în timp ce utilizarea programului de preluare prealabilă generată este la optim sau aproape, văzând accelerări între 1,16× și 2,05×.

În al doilea rând, o estimare a performanței aplicației care folosește programul de preîncărcare menționat este calculat. Această estimare poate fi utilizată pentru a alege între diferite strategii de preluare anticipată. În experimentele noastre, diferența dintre accelerația estimată și cea măsurată este mai mică de 5%.

Înainte de munca noastră, preîncărcarea a fost privită ca o cutie neagră. Dezvoltatorii au folosit tehnici de încercare și eroare pentru dezvoltarea prefetcherilor în speranța de a îndeplini obiectivele de performanță. În schimb, cadrul nostru aduce transparență preluării prin furnizarea de instrumente analitice pentru un dezvoltator să înțeleagă capacitățile de preluare anticipată, sau limitări, ale unei aplicații care rulează pe un anumit sistem. În plus, oferă posibilitatea de a obține informațiile necesare pentru a selecta cea mai bună soluție dintr-un coș de opțiuni.

Mai jos, enumerăm principalele contribuții ale acestei lucrări.

- Am propus un cadru matematic de înțelegere și anticipare a performanței unui prefetcher.
- Dezvoltăm o metodologie de evaluare a capabilităților de preîncărcare ale unei aplicații pentru a permite dezvoltatorilor să evalueze adecvarea acestuia pentru o anumită configurație hardware.
- Descriem modul în care paralelismul la nivel de memorie (MLP) pentru preîncărcarea poate reduce decalajul la performanța optimă.
- Evaluăm acuratețea analizei noastre în contextul helper threads, software prefetching și FPGA prefetching.

Capitolul 6

Concluzii

Această teză a prezentat câteva îmbunătățiri cu privire la modul în care memoria este gestionată la diferite niveluri de abstractizare. Am început prin a evidenția importanța limbajelor moderne de programare sigure pentru memorie. Am prezentat programarea în D ca fiind următorul pas logic pentru dezvoltatorii care scriu cod în C
C++, evidențiind punctele sale forte în ceea ce privește expresivitatea și siguranța memoriei. Apoi am prezentat îmbunătățirile noastre la limbajul D care permite dezvoltatorilor să folosească D fără costuri de performanță în ceea ce privește utilizarea memoriei, oferind astfel o alternativă la subsetul principal al limbajului care folosește colectorul de gunoi.

Apoi am continuat să demonstrăm că D poate fi integrat cu succes cu nucleul Linux prin portarea și integrarea unui driver de dispozitiv, crescând astfel siguranța memoriei acestuia. Implementarea noastră este la egalitate cu performanța codului scris în C, demonstrând astfel că trecerea progresivă la un limbaj sigur pentru memorie, fără pierderi de performanță, este posibilă.

În cele din urmă, am oferit un cadru analitic pentru a înțelege capacitățile de preluare preliminară ale unei aplicații. Cadrul nostru poate să fie utilizat de arhitecții de computer și dezvoltatorii de software pentru a-și optimiza sistemele în ceea ce privește preîncărcarea. În plus, am demonstrat că preîncărcarea programabilă este posibilă prin descărcarea preîncărcării în FPGA-uri.

Prin urmare, am prezentat contribuții la toate nivelurile stivei de abstractizare a memoriei.

Diferența dintre computere și performanța accesului la memorie va crește pe măsură ce trece timpul. În plus, securitatea va deveni o preocupare care are o importanță egală (dacă nu mai mare) ca performanța memoriei. Ca atare, este imperios necesar ca operațiile de lucru cu memoria să fie optimizate pentru performanță și securitate la toate nivelurile de abstractizare. Această teză deschide calea către o lume în care operațiile de memorie care sunt ușor de exprimat de către utilizator sunt rulate într-un mediu securizat, pe lângă hardware personalizabil, specific aplicației.

6.1 Listă de Publicații

1. **Răzvan Nițu**, Eduard Stăniloiu, Cristian Crețeanu, Răzvan Rughiniș, *”Building an Interface for the D Compiler Library”*, 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet), Iași.
2. **Răzvan Nițu**, Eduard Stăniloiu, Cristian Done, Răzvan Rughiniș, *”Security Audit for the D Programming Language”*, 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet), Iași.
3. Eduard Stăniloiu, **Răzvan Nițu**, Robert Aron, Răzvan Rughiniș, *”Extending Client-Server API Support for Memory Safe Programming Languages”*, 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet), Iași
4. Eduard Stăniloiu, **Răzvan Nițu**, Cristian Becerescu, Răzvan Rughiniș, *”Automatic Integration of D Code With the Linux Kernel”*, 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet), Iași.
5. **Răzvan Nițu**, Eduard Stăniloiu, Răzvan Deaconescu, Răzvan Rughiniș, *”Adding Support for Reference Counting in the D Programming Language”*, Proceedings of the 17th International Conference on Software Technologies (ICSOFT), Lisbon, 2022.
6. Eduard Stăniloiu, **Răzvan Nițu**, Răzvan Deaconescu, Răzvan Rughiniș, *”A New Collection Framework For the D Programming Language”*, accepted for publication at U.P.B. Scientific Bulletin Series C, Bucharest, Romania, 2022.
7. **Răzvan Nițu**, Eduard Stăniloiu, Răzvan Deaconescu, Răzvan Rughiniș, *”Designing Copy Construction for the D Programming Language”*, accepted for publication at U.P.B. Scientific Bulletin Series C, Bucharest, Romania, 2022.
8. Eduard Stăniloiu, **Răzvan Nițu**, Alexandru Militaru, Răzvan Deaconescu, *”Safer Linux Kernel Modules Using the D Programming Language”*, accepted for publication in IEEE Access, 2022.
9. **Răzvan Nițu**, Lingfeng Pei, Trevor E, Carlson, A Cross-Prefetcher Schedule Optimization Methodology, IEEE Access, 2022.
10. Giorgiana Violeta Vlăsceanu, Caraman Ghenadie, **Răzvan Nițu**, Costin-Anton Boiangiu, *”A voting method for image binarization of text-based documents”*, 2022 21st RoEduNet Conference: Networking in Education and Research (RoEduNet), Bucharest.

Bibliografie

[1] A. Jaspe-Villanueva, “Scalable exploration of 3d massive models,” Ph.D. dissertation, 11 2018.

[2] R. Nitu, E. Staniloiu, C. Done, and R. Rughinis, “Security audit for the d programming language,” in *2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. IEEE, 2021, pp. 1–6.

[3] S. L. Peyton Jones and P. Wadler, “Imperative functional programming,” in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’93. New York, NY, USA: ACM, 1993, pp. 71–84. [Online]. Available: http://doi.acm.org/10.1145/158511.158524

[4] K. Czarnecki, K. Østerbye, and M. Völter, “Generative programming,” in *European Conference on Object-Oriented Programming*. Springer, 2002, pp. 15–29.

[5] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, “{DR}.{CHECKER}: A soundy analysis for linux kernel drivers,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1007–1024.

[6] R. Johnson and D. Wagner, “Finding User/Kernel pointer bugs with type inference,” in *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, Aug. 2004. [Online]. Available: https://www.usenix.org/conference/13th-usenix-security-symposium/finding-userkernel-pointer-bugs-type-inference

[7] D. Dawson, N. Hawes, C. Hoermann, N. Keynes, and C. Cifuentes, “Finding bugs in open source kernels using parfait,” 2009.

[8] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, “Difuze: Interface aware fuzzing for kernel drivers,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[9] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, “Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2541–2557.

[10] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, “Razzer: Finding kernel race bugs through fuzzing,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[11] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, “{kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 167–182.

[12] K. Lu, A. Pakki, and Q. Wu, "Automatically identifying security checks for detecting kernel semantic bugs," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 3–25.

[13] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *NDSS*, 2016.

[14] "Rust for Linux," https://github.com/Rust-for-Linux, accessed: 2022-04-17.

[15] "Rust in the linux kernel: Good enough," https://thenewstack.io/rust-in-the-linux-kernel-good-enough/, accessed: 2022-04-17.

[16] "Linux kernel commit to add rust support," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b, accessed: 2022-10-22.

[17] E. Staniloiu, R. Nitu, C. Becerescu, and R. Rughinis, "Automatic integration of d code with the linux kernel," pp. 1–6, 2021.

[18] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," ser. ASPLOS V, 1992.

[19] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[20] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[21] F. Dahlgren and P. Stenstrom, "Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors," in *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture*, 1995.

[22] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006.

[23] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," ser. ICS '09, 2009.

[24] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," ser. ASPLOS X, 2002.

[25] J. Collins, S. Sair, B. Calder, and D. M. Tullsen, "Pointer cache assisted prefetching," ser. MICRO 35, 2002.

[26] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," ser. ISCA '99, 1999.

[27] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, “Imp: Indirect memory prefetcher,” ser. MICRO-48, 2015.

[28] M. Cavus, R. Sendag, and J. J. Yi, “Array tracking prefetcher for indirect accesses,” in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018.

[29] S. Pakalapati and B. Panda, “Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 118–131.

[30] A. Jain and C. Lin, “Linearizing irregular memory accesses for improved correlated prefetching,” ser. MICRO-46, 2013.

[31] D. Joseph and D. Grunwald, “Prefetching using markov predictors,” 1999.

[32] Y. Solihin, Jaejin Lee, and J. Torrellas, “Using a user-level memory thread for correlation prefetching,” in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002.

[33] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, “Runahead execution: an alternative to very large instruction windows for out-of-order processors,” 2003.

[34] M. Hashemi, O. Mutlu, and Y. N. Patt, “Continuous runahead: Transparent hardware acceleration for memory intensive workloads,” 2016.

[35] O. Mutlu, H. Kim, and Y. N. Patt, “Techniques for efficient processing in runahead execution engines,” in *32nd International Symposium on Computer Architecture (ISCA’05)*. IEEE, 2005, pp. 370–381.

[36] S. ChappellRobert, StarkJared, P. KimSangwook, K. ReinhardtSteven, and N. PattYale, “Simultaneous subordinate microthreading (ssmt),” *ACM Sigarch Computer Architecture News*, 1999.

[37] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt, “Microarchitectural support for precomputation microthreads,” ser. MICRO 35, 2002.

[38] J. D. Collins, D. M. Tullsen, Hong Wang, and J. P. Shen, “Dynamic speculative precomputation,” in *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, 2001.

[39] J. D. Collins, Hong Wang, D. M. Tullsen, C. Hughes, Yong-Fong Lee, D. Lavery, and J. P. Shen, “Speculative precomputation: long-range prefetching of delinquent loads,” in *Proceedings 28th Annual International Symposium on Computer Architecture*, 2001.

[40] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham, “Dynamic helper threaded prefetching on the sun ultrasparc cmp processor,” ser. MICRO 38, 2005.

[41] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, “Compiler-directed data prefetching in multiprocessors with memory hierarchies,” in *Proceedings of the 4th International Conference on Supercomputing*, ser. ICS ’90, 1990.

[42] S. W. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti, "A compiler-directed data prefetching scheme for chip multiprocessors," ser. PPoPP '09, 2009.

[43] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The performance of runtime data cache prefetching in a dynamic optimization system," ser. MICRO 36, 2003.

[44] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn, "Profile-guided post-link stride prefetching," ser. ICS '02, 2002.

[45] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," ser. CGO '17, 2017.

[46] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," ser. ASPLOS '20, 2020.

[47] D. Kim and D. Yeung, "Design and evaluation of compiler algorithms for pre-execution," 2002.

[48] D. Kim and D. Yeung, "A study of source-level compiler algorithms for automatic construction of pre-execution code," *ACM Trans. Comput. Syst.*, 2004.

[49] Changhee Jung, Daeseob Lim, Jaejin Lee, and Y. Solihin, "Helper thread prefetching for loosely-coupled multiprocessor systems," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, 2006.

[50] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," ser. ASPLOS XVI, 2011.

[51] H. Al-Sukhni, I. Bratt, and D. A. Connors, "Compiler-directed content-aware prefetching for dynamic data structures," in *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[52] C.-L. Yang and A. Lebeck, "A programmable memory hierarchy for prefetching linked data structures," 2002.

[53] S. Choi, N. Kohout, S. Pamnani, D. Kim, and D. Yeung, "A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching," *ACM Trans. Comput. Syst.*, 2004.

[54] N. Kohout, Seungryul Choi, Dongkeun Kim, and D. Yeung, "Multi-chain prefetching: effective exploitation of inter-chain memory parallelism for pointer-chasing codes," in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[55] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," ser. ASPLOS '18, 2018.

[56] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," ser. ICS '16, 2016.

[57] M. Cavus, R. Sendag, and J. J. Yi, "Informed prefetching for indirect memory accesses," *ACM Trans. Archit. Code Optim.*, 2020.

[58] C. Cimpanu, "Microsoft: 70 percent of all security bugs are memory safety issues," *URL: https://www. zdnet. com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues*, 2019.

[59] A. Bannister, "Substandard software costs us economy $2tn through security flaws, legacy systems, abandoned projects," *URL: https://portswigger.net/daily-swig/substandard-software-costs-us-economy-2tn-through-security-flaws-legacy-systems-abandoned-projects*, 2021.

[60] M. Dawson, D. N. Burrell, E. Rahim, and S. Brewster, "Integrating software assurance into the software development life cycle (sdlc)," *Journal of Information Systems Technology and Planning*, vol. 3, no. 6, pp. 49–53, 2010.

[61] S. Dange and M. Chatterjee, "Iot botnet: the largest threat to the iot network," in *Data Communication and Networks*. Springer, 2020, pp. 137–157.

[62] D. McMillen, "Internet of threats: Iot botnets drive surge in network attacks," *URL: https://securityintelligence.com/posts/internet-of-threats-iot-botnets-network-attacks/*, 2021.

[63] Z. Whittaker, "Mirai botnet attackers are trying to knock an entire country offline," *URL: https://www.zdnet.com/article/mirai-botnet-attack-briefly-knocked-an-entire-country-offline/*, 2016.

[64] P. Lee, *Topics in Advanced Language Implementation*. MIT Press, 1991.

[65] D. E. Knuth, "The art of computer programming. volume 1: Fundamental algorithms. volume 2: Seminumerical algorithms," *Bull. Amer. Math. Soc*, 1997.

[66] J. H. McBeth, "Letters to the editor: on the reference counter method," *Communications of the ACM*, vol. 6, no. 9, p. 575, 1963.

[67] D. Kosutic and F. Pigni, "Cybersecurity: investing for competitive outcomes," *Journal of Business Strategy*, vol. ahead-of-print, 10 2020.

[68] F. Salahdine and N. Kaabouch, "Social engineering attacks: A survey," *Future Internet*, vol. 11, no. 4, p. 89, 2019.

[69] K. Reese, T. Smith, J. Dutson, J. Armknecht, J. Cameron, and K. Seamons, "A usability study of five Two-Factor authentication methods," in *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 357–370. [Online]. Available: https://www.usenix.org/conference/soups2019/presentation/reese

[70] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[71] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[72] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, “Make the most out of last level cache in intel processors,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.

[73] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Bingo spatial data prefetcher,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 399–411.

[74] S. Pakalapati and B. Panda, “Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 118–131.

[75] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Evaluation of hardware data prefetchers on server processors,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–29, 2019.

[76] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, “Path confidence based lookahead prefetching,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[77] D. Suggs, M. Subramony, and D. Bouvier, “The AMD “Zen 2” processor,” *IEEE Micro*, vol. 40, no. 2, pp. 45–52, 2020.

[78] K. Viswanathan, “Disclosure of hardware prefetcher control on some Intel® processors.” [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html

[79] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, “Knights landing: Second-generation intel xeon phi product,” *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[80] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, “Power4 system microarchitecture,” *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, 2002.

[81] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, “The ibm blue gene/q compute chip,” *IEEE Micro*, vol. 32, no. 2, pp. 48–60, 2012.

[82] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, “Apt-get: Profile-guided <i>timely</i> software prefetching,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys ’22. New York, NY, USA:

Association for Computing Machinery, 2022, p. 747–764. [Online]. Available: https://doi.org/10.1145/3492321.3519583