University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

PHD THESIS

SUMMARY

Memory Optimization Techniques For Modern Computer Systems

Scientific Adviser:

Prof. Dr. Ing. Răzvan-Victor Rughiniș

Author:

Ing. Razvan Nitu

Bucharest, 2023

Cuprins

Chapter 1

Introduction

The recent advances in computer architecture have enabled processors to continuously increase the capacity at which they execute instructions. Figure 1.1 highlights the pace at which compute performance has increased over the years. As it can be observed, memory performance has not been able to keep up with the processing power and this gap deepens as years pass by. As a consequence, powerful processors are not leveraged to their entire capabilities due to data stalls, leading to CPU starvation. The problem is not represented by memory bandwidth, but rather by the physical limitations of DRAM memory. As a consequence, solving this issue by creating better memory controllers or better DRAM memory is not going to entirely close the gap.

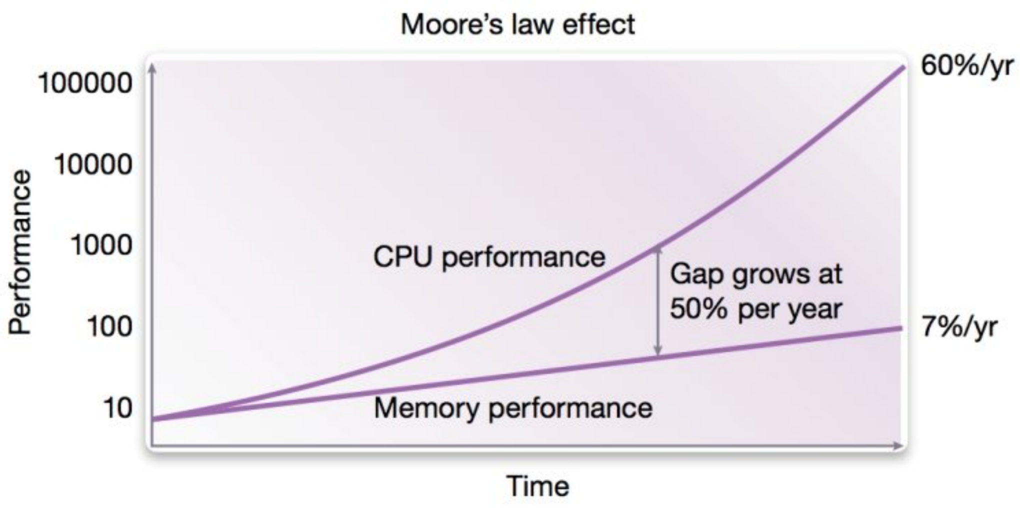

Figure 1.1: Processing power vs. memory performance [1]

To make matters worse, modern computing systems add to the problem due to the different layers of abstractions that are being stacked above the physical memory. A typical system uses an operating system that manages the physical memory and a runtime, library allocator that is typically dependent on the programming language that is used, as showcased in Figure 1.2. The result is that memory operations - that ensure allocation, deallocation or actual use of memory - are multiplied and go through multiple indirections, ultimately affecting overall performance.

These layers are put in place because the computer technology has evolved to a point where performance is not the single critical aspect of systems. Security, maintainability

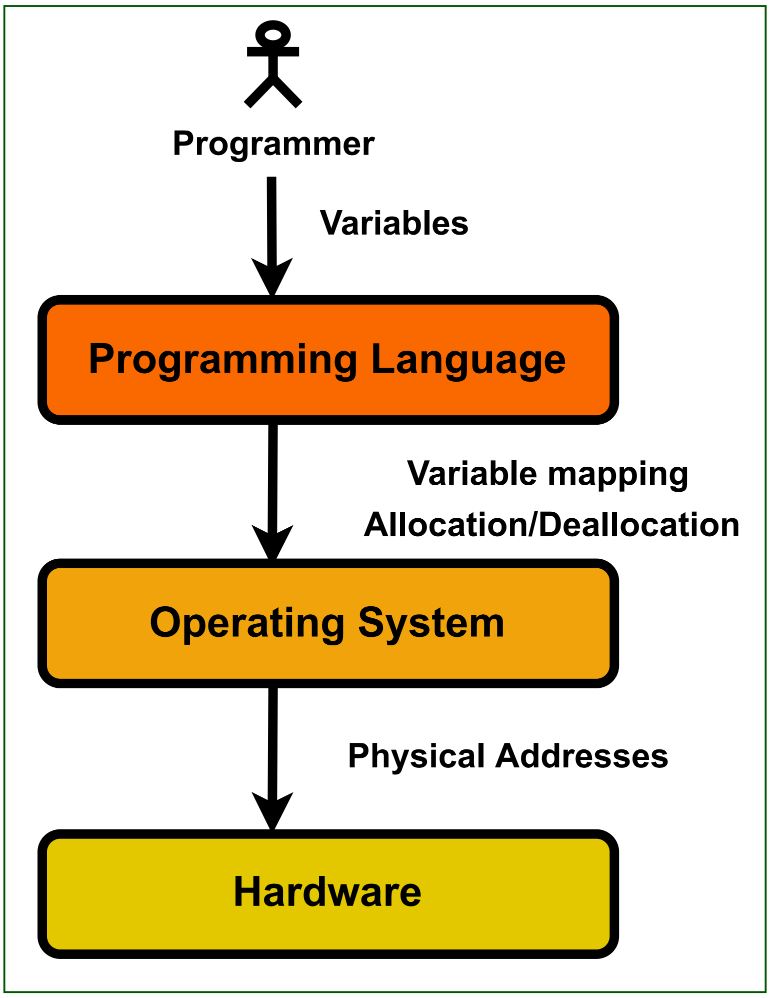

Figure 1.2: Abstraction layers on top of DRAM memory

and ease of development have become major factors in the adoption or avoidance of a particular technology. As a consequence, application specific trade-offs are being made when choosing a particular technology for a particular use case. Nevertheless, performance remains the guiding factor and the ultimate deciding factor.

In this context, researchers have strived to improve all stages where memory is being handled: hardware, operating system level, programming language level.

From a hardware perspective, cache memory has been employed to save data that is reused frequently so that subsequent accesses will suffer from a smaller latency penalty. Additionally, prefetchers are added to try and predict future memory accesses and fetch the data in the cache, before the actual access takes place. Other improvements include: preload instructions that offer the user the possibility to tell the processor which memory address are going to be used in the future, temporal loads that do not go through the cache hierarchy and programmable memory controllers. However, hardware solutions are limited because they try to fix the problem for all potential applications that might be run on it. In contrast, we propose a novel prefetching solution that relies on analysing the application at hand to be able to perfectly prefetch all the necessary data items.

Operating systems not only have to ensure optimal use of system resources, but also need to ensure the safety aspect. As a consequence, operating systems use virtual memory modules to ensure process sandboxing along with allocators that implement the allocation strategy. However, given the fact that most operating systems are written in C which is an unsafe programming language, the kernel itself becomes a security issues. In our work, we demonstrate, that the Linux kernel can be improved with regards to safety by an incremental transition to a programming language that is mechanically checked for safety, D.

Programming languages have evolved to the point where they can simplify most of the programmer's tasks: memory management may be handled by runtime implementations, expressive constructs have been put in place to take care of the most stringent needs, third party tools help in maintaining and improving the quality of the code etc. However, most of these come with a cost. For example, garbage collected programming language lift the burden of manual memory management, however, impose additional resource consumption. Rust's state of the art borrow checker offers security guarantees at low resource cost, however, even trivial programs need to specify the complex logic of value ownership. As an alternative, the D programming language offers the possibility to mechanically check for safety C-like programs, but relies on the garbage collector for lifetime management. Our perspective is that using D's safety features alongside reference counted data structures brings the advantages of both automatic memory management and memory performance.

Only by improving all stages of memory abstractions will it be possible to (1) close the gap between the increasing processing power of CPUs and the stagnating performance of DRAM memory and (2) meet the expected criteria for memory safety and ease of use. In this thesis, we pave the way to the usage of a memory safe language, the D programming language at all levels of memory abstraction, by taking into consideration all aspects of dealing with memory: performance, safety and ease of use. Additionally, we propose a hardware alternative to the existing prefetching techniques the paves the way to closing the gap between compute and memory access.

1.1 Thesis Contributions

To summarise, the thesis makes the following contributions:

- We have designed and implemented the copy construction feature for the D programming language.
- We have designed and implemented the ability to break the transitivity of the D type qualifiers by introducing a new keyword *__mutable*.
- We have used the above mentioned features to implement referenced counted data structures, proving that they represent a better alternative, with regards to perfor-

mance, to the garbage collector.

- We have performed a security audit of the D programming language, assessing the level of guarantee that the safe subset of the language offers. We have identified 3 flaws in the safety typesystem and have proposed solutions to fixing them.
- We have ported a Linux kernel device driver to D and have proved that it is possible to integrate a memory safe language in the Linux kernel without loss of performance.
- We have improved the D compiler library so so that subsequent tools may be implemented to enrich the language ecosystem.
- We have improved an existing tool, *dpp* that automatically translates C header files to D. We have proved that this tool is usable in conjunction with the Linux kernel header files.
- We have made a survey, categorizing most of the existing prefetching tools and highlighting their benefits and limitations. Future researchers may consult our survey to understand the different aspects taken into consideration when evaluating a prefetching technique.
- We have proposed a mathematical framework to understand the prefetching capabilities of a given application that runs a hardware with given characteristics. Computer architects may use our framework to understand the bottlenecks of a system, whereas software developers may use it to understand the bottleneck of their applications.
- We have proposed a novel prefetching technique by using an FPGA to offload prefetch requests.

1.2 Thesis Structure

The thesis is structured as follows:

Chapter 2 presents the most relevant works that are related or have influenced this thesis. It is organized in the form of sections, each treating a specific level of the memory abstractions that this thesis improves upon. Therefore, the 3 sections are represented by: memory management techniques, memory safety for the Linux kernel and prefetching techniques.

Chapter 3 presents our contributions in terms of adding reference counting to the D programming language. It details all of the technical aspects of implementing the copy construction and *__mutable* features. Finally, it presents the benchmarks for our implementations of some data structures that employ reference counting, showcasing the advantages it offers over garbage collection.

Chapter 4 presents our contributions with regards to improving the memory safety aspects of applications. It first presents our findings from our security audit over the D programming language. We continue by improving the compiler interface for the D programming language, which enable developers to create tools that aid in enhancing the memory safety of applications. We then present our journey to improve the memory safety of the Linux kernel by porting a device driver to D. Finally, we present the improvements that we have made to the *dpp* tool to enable it to be used in the context of the Linux kernel files.

Chapter 5 presents the survey that we have accomplished to better understand the existing prefetching techniques. In the process, we have identified the major dimensions that may be used to categories such techniques. Furthermore, we present a mathematical framework that may be used to better understand and optimize the prefetching capabilities of a given application that runs on a particular hardware. Finally, we present our novel approach of prefetching using an FPGA and its performance evaluation.

Chapter 6 concludes our work and showcases the list of publications that the current writing is based upon.

Chapter 2

State of the Art

In this chapter we will highlight what were the major improvements that have been proposed and implemented in the research community in terms of memory optimizations. Given the broad range of topics that need to be discussed, we have narrowed it down to the fields that are particularly relevant the improvements that are being made by this thesis.

As a consequence the present chapter is structured in 3 sub-chapters, each highlighting the major research papers that have influenced the particular memory abstraction that is being discussed. Therefore, Section 2.1 discusses what are the major memory allocation techniques and provides an overview over the D programming language, Section 2.2 highlights alternatives to our approach of making the Linux kernel safer and their limitations and Section 2.3 presents the major prefetching techniques that are employed modern computer architectures emphasizing the trade-offs that are being made.

2.1 Memory Management Techniques

Since the inception of the ability to directly allocate memory, memory management has become one of the most complex endeavors of programming. The first and most primitive form of memory allocation is represented by manual memory management technique. In this form of memory management, the user is tasked with demanding and freeing memory. A task that is easier said than done. This form of memory management has led to countless forms of bugs: use after free, double free, use without allocation, dangling pointers, memory that is never freed etc.

As a consequence, more advanced forms of memory management were required, ones that would lift the complexity of the shoulders of users. The result was the implementation of garbage collected programming languages or reference counted systems. The former is a heavyweight approach where a considerable size of system resources are necessary but is extremely easy to use (practically, completely transparent), whereas the latter is lightweight but comes with limitations: it does not support circular references and is to a certain degree invasive.

2.1.1 The D programming language

D is an imperative, general purpose, systems programming language. D aims to fulfill the requirements of a full stack programming language, under the mantra ”One language to rule them all". As a consequence, D has support for mechanical safety checks [2], functional programming [3], meta-programming [4], parallel programming, object oriented programming etc. Heavily inspired from popular languages like C, C++, Java and Python, D builds upon the features that exist in other languages: C-style syntax and manual memory management, Java-like classes and garbage collection, similar C++ template system, Scheme-like functional programming etc. Although some of the features are mutually exclusive (for example: manual memory management and garbage collection), the user may choose from the different options via command line switches.

2.2 Memory Safety for the Linux Kernel

Improving the safety of the Linux kernel and its drivers is the constant focus of the professional and research security community. There are different approaches ranging from static analysis of the Linux kernel code [5–7] to fuzzing [8–11] to the use of runtime checks and/or instrumentation [12, 13].

The idea of using programming languages that implement different memory safety features in order to make the Linux kernel code safer has also been tackled.

2.2.1 The Rust Programming Language

The recent availability of Rust as a programming language in the Linux kernel [14,15] paves the way for adding code written in a secure programming language. This is compatible with our own approach of using D to write code in the Linux kernel. Although the memory safety guarantees that Rust offers are superior when compared to D, integrating it in the Linux kernel is a very complicated task. As evidence, the work required to add support for Rust in the Linux kernel was done by 173 people (present in the commit changelog [16]) over the course of 18 months. This included solely the implementation of the infrastructure required to integrate Rust code in the kernel. It does not implement any device driver or any parts of the Linux kernel in Rust. By comparison, our work was done by 3 people over the course of 4 months, including the initial exploratory phase of the Linux infrastructure as well as the porting of the kernel header files. The actual porting time of the device driver required only 2 to 3 weeks. The reader should consider that, in the meantime, work has been advanced to automate the porting of kernel header files to D [17], thus reducing the required time to integrate D device drivers to a minimum. In addition, the effort to integrate Rust in the kernel has required compiler changes to accommodate the esoteric code encountered, whereas our work does not necessitate any compiler changes.

2.3 Prefetching techniques

Prefetching is a standard technique that has been used in many different ways and in many situations. We briefly outline relevant work and elaborate, further, our approach.

Prefetching can be implemented in hardware, software as well as a combination of hardware and software. Table 2.1 groups similar prefetching techniques into categories and highlights the relevant attributes of each technique.

Hardware Techniques

Hardware prefetching techniques require a specialized physical unit that handles the monitoring of memory accesses and automatically generates prefetch requests. This unit is commonly tightly coupled to the execution unit, normally a processor core. This allows for low latency communication between the core and the prefetch hardware unit. The hardware units tend not to support anything but a general prefetch method which may not be optimal for all algorithms or applications. In our work and in this paper, we show that latency is not crucial for performance allowing a loosely and program controlled acceller-ator to carry out prefetching effectively. This also allows the prefetchers in our approach to implement specialized and more complicated prefetch methods.

Table 2.1: Prefetching Techniques

Approach	**SW**	**HW**	**Pattern**	**Prefetch Schedule Analysis**
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

2.3.1 Software Techniques

Software prefetching techniques rely on prefetch hints or instructions that are inserted in the source code. These generate pre-load instructions that are executed before the actual load. These instructions are committed immediately and therefore do not stall the pipeline. This approach has the advantage that it does not require extra hardware since most architectures implement a form of prefetch instruction. However, software prefetching techniques suffer from two major shortcomings: (1) accurately inserting the prefetch instructions is difficult and (2) accesses that involve multiple long latency loads are still going to stall the pipeline and therefore require extra computation that masks the prefetch.

2.3.2 Helper Threads

Helper threads [36–40,47–50] tackle prefetching by statically extracting the code for delinquent loads and running it on a spare thread context. This approach can optimally target any access pattern by increasing the number of helper threads. Furthermore, it is flexible enough to be implemented both in hardware [36–40] and software [47–50]. However, even using a single extra thread comes at a an increased energy penalty on high performance cores. Moreover, accesses that require loads in their address computation will stall and in the absence of a hardware event queue the synchronization of loads becomes costly in terms of both implementation and performance. In contrast, our approach does not require any synchronization and running prefetch kernels on the FPGA should be cheaper in terms of energy consumption.

2.3.3 Programmable Hardware

Programmable hardware techniques employ specialized hardware units that are able to run specific address computation instructions. Jones et al. have proposed a programmable prefetcher specifically designed for graph workloads that targets specific traversals [56]. Yi et al have designed a hybrid prefetcher that targets indirect memory accesses [57]. Several approaches have targeted linked list data structures [51–54]. A more general approach has been developed by Jones et al. [55] that uses multiple small in order cores to run prefetch kernels that are indicated in software. This work has proven significant speed-ups for load-intensive applications, however, the design is not able to deal with the pointer chase pattern and the prefetch kernel size is limited to only a few instructions whereas [46] reports prefetch kernels that require up to 80 instructions. Our approach is similar in essence with [55], however, the difference is that the FPGA is reconfigurable. This offers the opportunity to instantiate the minimum hardware that is necessary to obtain maximal performance. Furthermore, we provide a mathematical framework that aids the programmers in instantiating the minimal hardware for optimal prefetching.

Chapter 3

Memory Performance - Adding Reference Counting in D

As more and more software products are developed daily, the security risks imposed by the growing code bases increase. In 2019, Microsoft reported that the cause for 70% of security bugs were memory related [58]. The costs incurred due to security flaws and their exploitation are in the billions [59], with the IBM System Science Institute stating that patching costs 100 times more than the development costs [60].

To help mitigate the risk, memory safe systems programming languages, such as D and Rust, are increasingly adopted by developers. A significant area where memory safe languages are desirable is represented by the Internet of Things (IoT). IoT devices have become a popular target for attackers to compromise and use as an IoT Botnet army [61] that is used to carry attacks against businesses, governments and even entire countries [62] [63]. In order to satisfy the needs of IoT devices, the programming languages used must also produce fast programs and be resource considerate.

D is a modern, systems-level programming language that aims to provide both high performance and memory safety in a simple, intuitive and expressive manner. Although it is an imperative language, D provides functional style concepts such as pure functions and transitive type qualifiers. In addition, it is able to inter-operate with C and C++ code out of the box, thus providing a simple migration path for legacy code.

D provides a garbage collector (GC) [64] for built-in features that use heap memory, such as dynamic arrays and classes, but also supports manual memory management via raw pointers and *malloc*/*free*. Therefore, for situations where the garbage collector is unsuitable due to resource scarcity (small memory, small number of computation units or both) or real-time constraints, users have the possibility to implement a custom allocation strategy. Note that in this scenario ease of use is sacrificed for performance, since the user needs to manually manage memory. This has proven to be a complex, time consuming and error prone endeavour [65].

A third option is represented by automatic reference counting (ARC) in the form of a library solution [66]. ARC is lightweight in terms of resource utilization, since it typically stores an extra counter field for each allocated instance. In terms of computation, the added overhead consists of simple addition or subtraction operations. In addition, the usage of ARC is almost transparent to the user: an object needs to be declared as being reference counted and everything will be taken care of behind the scenes. Providing support for such an option is important because it offers maximum flexibility in terms of allocation strategies: for the majority of cases, the GC should be sufficient; for constrained scenarios where the GC cannot be supported, ARC is to be used; for extreme situations, where not even ARC is sufficient to attain the performance guarantees, manual memory management should be employed.

Chapter 4

Memory Safety - Improving The Safety of Applications By Using the D programming Language

Memory safety has become a major concern for present day applications. With the advent of migrating more aspects of our daily lives into online environments we gain time efficiency, however, we expose ourselves to cyber attackers.

Hackers target vulnerabilities in a system and exploit them. Such actions may result in: gaining escalated privileges, causing denial of service, crypting sensitive data and asking for ransom, deleting important data etc. Successfully attacks on companies or individuals, typically result in substantial losses, from both a financial and a reputation perspective.

Although companies are investing increasing amounts of resources into cybersecurity [67], the results are not expected ones. This phenomenon has a simple explanation and lies in the fact **security is treated retroactively, not proactively**.

Cyberattacks come in 2 broad categories: social engineering and vulnerability exploitation.

Social engineering attacks occur when an attacker tries to obtain a password or elevated privileges by tricking someone into an organization that they are someone they are not [68]. Examples of social engineering attacks are: phishing, tailgating, shoulder surfing, ransomware etc. The main vector of attack in social engineering attacks is the human.

Vulnerability exploitation relies on the fact that there is some mistake or flaw in the implementation of an application, system, protocol etc. While social engineering attacks the human, vulnerability exploitation attacks the program.

Both social engineering and vulnerability exploitation rely on human mistakes: in the former, a human wrongfully identifies the attacker as being someone else, whereas in the later, a human makes a technical mistake when implementing a program. No matter how much resources are invested, humans will make mistakes. This is the underlying reason why security enhancements have had limited success: they always tackle the symptom that the root case.

Recently, better approaches have been adopted to improve the situation with social engineering. Two factor authentication [69] is an example on how measures can be taken proactively: have the user go through multiple steps of authentication (password, biometrically, one-time-passwords etc.). This situation assumes that the user will make a mistake and communicate his password to an attacker and requires an additional step of authentication.

For vulnerability exploitation, researchers are proposing formal verification tools for programs. However, it is hard to assess the correctness of any given program automatically, without user intervention. Nevertheless, steps have been made in creating standards and formal verification tools that are program dependent.

From a programming language perspective, safety is defined as a program being correct in terms of memory usage. This means that a program should not have dangling pointers, use after free, pointers to expired stack frames etc. Some languages trade-off performance for memory safety by not allowing the use of pointers (Java, Python, Javascript etc.), however, this typically means that there is the need for a garbage collector to track the lifetime of variables. At the other extreme, we have the C programming language which allows liberal use of pointers that ultimately cause the introduction of numerous vulnerabilities. In recent years, programming languages such as Rust and D have gain attention due to their middleground approach where memory safety may be mechanically checked by the compiler (i.e. if a program compiles, then there is no situation where it may corrupt memory - thus eliminating the main source of program vulnerabilities) without the need of a garbage collector.

However, the adoption of such young languages poses multiple challenges:

- ease of integration with existing programs. We don't want to reinvent the wheel so we want to leverage existing libraries written in other languages.
- ease of use. Programmers want to get work done, not spend hours in fighting the language, therefore a memory safe language does not need to impose extra complexity on the programmer.
- existing tooling. Programmers don't adopt languages if they don't have a strong ecosystem that provides development tools.

In this work, we aim at increasing the usage of memory safe languages by showing that D is a good substitute for memory unsafe languages such as C due to its good integration with the C family of programming languages, its similarity with existing modern languages such as Java and Python and we implement changes that improve the current state of tooling.

As such the main contributions of this chapter are:

- We perform a security audit for the D programming language, identifying 2 flaws in its type system that may lead to unsafe code being accepted. We propose and implement solutions that eliminate the discovered issues.
- We analyze existing D third party software development tools and extract a common interface that may be used for the compiler as a library. We implement this interface, thus opening the door to the creation of more D developer tools.

- We show that the D programming language may be used to replace the C programming language in performance and safety constrained scenarios such as the Linux operating system. We implement and integrate a D device driver into the Linux kernel.
- We improve an existing tool that automatically generates D header files from C sources to be usable inside the Linux kernel. This eases the integration of D code in the kernel.

Chapter 5

Memory Architecture - Improving Prefetching For Applications

As the speed gap between modern processors and the memory system is ever increasing [70, 71], the bottleneck of memory accessing in today's Von-Neumann machines becomes the pain-point that inspires various optimizing techniques such as caching [72] and prefetching [73–76].

Prefetching is a fundamental technology of most high-performance systems today [77–81]. The goal of prefetching is to retrieve, in a timely manner, data from a high latency memory, typically DRAM, and place it in fast-to-access cache memory. One key feature of a prefetcher is that it aims to fetch the data that is needed *before the computation unit accesses and uses it*. Prefetching can significantly reduce the time a CPU needs to wait when accessing data.

Existing prefetchers implemented in hardware [18–32] provide fixed-function operation and can not fundamentally change to adapt to the application, limiting attainable performance.

We argue that future prefetchers need to be configurable to support different strategies, possibly to the extent that they are configured by software. Memory access patterns are well known to be application dependent, which makes prefetching hard to be performed in a both accurately and timely manner. For example, different prefetching distances, i.e. how far ahead the prefetcher sends requests, can lead to up to 10$\times$ performance benefits variation [82]. Therefore, we argue that prefetching needs to be driven by dynamic application behavior [46, 55, 82].

This opens up a large design space for the developer: Which prefetching strategy should be selected? When should a prefetch request be sent out? Is it worthwhile to keep the current strategy or is there a benefit to switch to a new one (while considering the potential overhead of this change)? Which parameters should one select if the prefetch strategy is parameterizable? Until now, such questions have not been possible to address in a systematic way.

In this work, we propose a novel analytical framework that, based on measurements of application execution, can suggest close-to-optimal prefetcher strategies. Our framework provides two results. First, for a given prefetching strategy, the framework outputs an optimized schedule of prefetches that is both accurate and timely. The prefetching plan can be used to improve the application performance. In our experiments, we show that the speed-up obtained while using the generated prefetch schedule is at or near optimal, seeing speed-ups between 1.16× and 2.05×.

Second, a performance estimate of the application that uses the mentioned prefetching schedule is computed. This estimate can be used to select between different prefetching strategies. In our experiments, the difference between the estimated and the measured speed-up is less than 5%.

Prior to our work, prefetching has been viewed as a black box. Developers have been using trial-and-error techniques for developing prefetchers hoping to meet the performance targets. In contrast, our framework brings transparency to prefetching by providing the analytical tools for a developer to understand the prefetching capabilities, or limitations, of an application that runs on a given system. Additionally, it offers the possibility to obtain the information required to select the best solution from a basket of options.

Below, we list the main contributions of this work.

- We propose a mathematical framework to both understand and predict potential prefetcher performance. The framework abstracts the technique of prefetching and is general enough to cover most prefetching scenarios.
- We develop a methodology of evaluating the prefetching capabilities of an application to allow developers to evaluate its suitability for a given hardware configuration.
- We describe how memory-level parallelism (MLP) for prefetching can close the gap to optimal performance.
- We evaluate the accuracy or our framework in the context of helper threads, software prefetching and FPGA prefetching.

Chapter 6

Conclusions

This thesis has presented several improvements regarding how memory is handled at different abstraction levels. We began by highlighting the importance of modern memory safe programming languages in the context of a programmers every day job. We have presented the D programming as a logical next step for developers coming from a C
C++ background, highlighting its strengths in terms of expressiveness and memory safety. We then presented our improvements to the main language that enables developers to use D at no performance cost in terms of memory usage, thus providing an alternative to the main garbage collected subset of the language.

We then proceeded to demonstrate that D can be successfully integrated with the Linux kernel by porting and integrating a device driver, thus increasing its memory safety. Our implementation is on par with the performance of the C written device driver, thus demonstrating that incremental transitioning to a memory safe langauge, with no loss of performance, is possible.

Finally, we have provided an analytical framework to understand the prefetching capabilities of an application. Our framework may be used by computer architects and software developers to optimize their systems with respect to prefetching. In addition, we have demonstrated that programmable prefetching is possible by offloading prefetching to FPGAs.

Therefore, we have presented contributions on all levels of the memory abstraction stack.

The gap between computing and memory access performance is only going to get wider as time goes by. In addition, security will become a concern that is of equal (if not greater) importance than memory performance. As such, it is critical that memory operations all optimized for performance and security at all levels of abstraction. This thesis paves the way to a world where memory operations that are easily expressible by the user are run in a secure environment, on top of customizable, application-specific hardware.

6.1 List of Publications

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 Niţu**, Costin-Anton Boiangiu, *"A voting method for image binarization of text-based documents"*, 2022 21st RoEduNet Conference: Networking in Education and Research (RoEduNet), Bucharest.

Bibliography

[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