POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

OPTIMIZING DATA-INTENSIVE APPLICATIONS FOR MODERN HARDWARE PLATFORMS

Doctoral Dissertation of:
Alberto Scolari

Supervisor:
Prof. Marco D. Santambrogio

Tutor:
Prof. Pier Luca Lanzi

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

2018 – XXXI Cycle

Abstract

DATA-INTENSIVE applications have become widespread in the years, especially in cloud-like environments. Among them, Data Analytics (DA) and Machine Learning (ML) applications are particularly important categories that deeply impacted business and science in the last decade, and are expected to have an even higher impact in the upcoming years. In the latest years, we also saw hardware platforms evolving along different directions to overcome the limitations of Dennard's scaling and the end of Moore's law. While heterogeneity is coming into play for several applications, Central Processing Units (CPUs) have also evolved towards a growing number of cores, specialized Single Instruction, Multiple Data (SIMD) units, high memory hierarchies and, in general, a more complex and diverse set of features. On the other side, also data-intensive applications became more complex, to the extent that a current ML model may comprise tens of diverse operators to compute a single prediction value, while taking in input data of multiple types like text, number vectors and images. Oftentimes these applications are structured as "data pipelines" and go through many steps like input parsing, data pre-processing, analysis and possibly loading the output to some "data sink" at the end. Mastering this complexity to achieve the best implementation and deployment of an application in a given setting (hardware platform, software stack, co-located applications, etc.) is becoming a key issue to achieve best usage of the infrastructure and cost-effectiveness. This problem is especially hard with heterogeneous platforms, whose hardware features may not suit some parts of an application to be accelerated or may require a long redesign effort. Here, where the inevitable complexity of applications determines the diversity of operations, CPUs are still central, as they provide the flexibility and the maturity to efficiently run most of these workloads with sufficient performance even for today's needs, while being easier to program than other architectures. Moreover, their general-purpose

design naturally fits the diversity of data-intensive applications.

This work explores the performance headroom that lies unused in modern CPUs with data-intensive applications. This headroom encompasses several dimensions, each one with specific problems and solutions. A first problem to solve is the performance isolation of co-located applications on the CPU, which has to take in account the sharing of the Last Level Cache (LLC). Here, we propose and evaluate a mechanism for partitioning the LLC that works on recent server-like CPUs and requires software-only modifications in the Operating System (OS), thus not impacting hardware nor applications. This solution proves to be effective with a diverse set of benchmarks and allows meeting Quality of Service (QoS) goals even in a contentious, hardly predictable environment. The second problem is the optimization of data-intensive applications, which can be composed of multiple, diverse computational kernels. This work explores the limitations of current solutions, revolving around a *black-box* approach: application kernels are individually optimized and run, disregarding their characteristics and their sequence along the data path; indeed, a simple case study shows that even a manual, naïve solution can achieve noticeable speedups with respect to the current state of the art. Building on these findings, we generalize them into a *white-box* approach for applications optimization: while applications should be written as sequences of high-level operators, as current development frameworks already do, this sequence should also be exposed to the system where these applications run. By looking at this structure during the deployment of the application, the system can optimize it in an *end-to-end* fashion, tailoring the implementation to the specific sequence of operators, to the hardware characteristics and to the overall system characteristics, and running it with the most appropriate settings; in this way the application can make the best use of the CPU and provide higher QoS. Such a re-thinking of current systems and frameworks towards a white-box also allows a cleaner support of heterogeneous accelerators. Indeed, the high-level description that we advocate allows the system to transparently map some operations to more specialized accelerators if need be. However, optimized solutions need to keep a sufficient degree of flexibility to cover the diversity of computational kernels. As an example, we explore a case study around Regular Expression (RE) matching, which is a ubiquitous kernel in data-intensive applications with limited performance on CPUs, and we propose an architecture that enhances previous work in terms of performance and flexibility, making it a good candidate for the integration with existing frameworks.

Overall, this work proposes several solutions for the main issues around modern CPUs and data-intensive applications, breaking some common abstractions and advocating for an appropriate description level of those applications. The solutions proposed here leverage this level of description that enables various optimizations, providing novel guidelines in order to make the best use of the architecture. From this work, several research directions arise, especially around extending these abstractions and the related solutions to work with heterogeneous devices, whose usage for the masses calls for more automated optimization strategies and prediction models.

Contents

*

List of Tables

List of Figures

CHAPTER 1

Introduction

In the era of *Big Data* , the availability of large data sources to harvest information from led to an explosion of the demand for computing power [1] . This demand was met and sustained with technological innovations along different directions. The first direction was already well-known when Big Data applications started to appear in the beginning years of the 21st century, and was due to the steady growth of transistors density enabled by Moore's law, which enabled cramming more computing power inside a single chip. Similarly, other advancements enabled by lithographic processes along the hardware stack (DRAM, Flash, network, ...) fostered the *scale-up* trends of single servers. Since scale-up could still not meet the demands, the research and the industry explored orthogonal *scale-out* solutions by connecting commodity servers through high-bandwidth network technologies like multi-Gigabit Ethernet or InfiniBand. Unlike the long-lasting Moore's law, these solutions quickly showed their limits along multiple dimensions: reliability, energy consumption, Total Cost of Ownership (TCO), complexity of programming, etc. Therefore, further scale-out is very hard to achieve [2] , and also the scale-up trend of Moore's law is -

[1] It is beyond the scope of this work to give or adopt a precise definition of "Big Data", but we will use the common understanding of this term: therefore, we will generally refer to Big Data to indicate the availability of large amounts of data that can be cheaply retrieved for analysis, to extract useful information for business and science. For more precise definitions of this term, the reader can refer to [172] and [52], while for its profound consequences on business and science at scale the reader can refer to [101] and [104].

[2] Solutions exist to further increase scale-out capabilities of some applications, but require rethinking the entire application and sometimes even the algorithms at its basis, like the *parameter server* [152] architecture.

now unsurprisingly - coming to an end [48, 149].

Part of the research is exploring disruptive solutions by increasing the heterogeneity of computing resources, so that specialization via accelerators can fill the gap between the quickly increasing demand and the slowly increasing offer [149, 99]. However, this path challenges existing hardware/software abstractions and disrupts the programming models we are accustomed to, with many research efforts still investigating how to ease design and implementation of applications with heterogeneous devices for the masses. These efforts span several hot applicative fields, proposing different solutions for each one: despite the body of literature already available, what are the most promising directions is still not clear.

This is true also for those specific fields that Big Data enabled and that currently attract most of the attention of research and industry, like Data Analytics (DA) and Machine Learning (ML). Here, application patterns are more varied than in other fields (like scientific computing, essentially based on algebraic kernels), which makes it difficult to devise unified programming and optimization approaches even for these more restricted domains and even if considering Central Processing Units (CPUs) only, as the multiplicity of frameworks for ML programming shows. This diversity causes further complexity when heterogeneity comes into play, as those applications may stress different parts of the hardware architecture (storage, memory, floating-point units, control paths, ...) and it is hard to design specialized accelerators while retaining coverage of many applicative patterns. Therefore, many of these applications still run on CPUs, whose architectures have been optimized along multiple dimensions (memory bandwidth, caches for locality, vectorized floating-point units, ...) over the course of the decades and are still "good enough" for many of today's needs. Thus, while heterogeneity is very lively in the research, optimizing these applications to make *best* use of modern CPUs is an attractive alternative for the present times, and an equally interesting research challenge. This chapter starts from these observations to motivate the focus of this work and its goals.

1.1 The impact of Data Analytics and Machine Learning

The Big Data availability enabled several applicative trends. Among them is the so-called DA, which is in general the "ability to extract useful information from large datasets", in order to provide insights to support the decision-making process of an organization. Therefore, DA can be also seen as the set of business processes and techniques to achieve this goal. Although techniques of information extraction for decision-making predate the Big Data phenomenon, it is only with Big Data that the extracted information is statistically more reliable and oftentimes more insightful, as "hidden patterns" become visible only when large amounts of data are involved, which in turn requires powerful hardware and software tools.

However, these achievements cannot be obtained by looking at single datasets with

classical statistical techniques, but by correlating information from multiple data sources of different types ([101] shows some examples of this heterogeneity), some of them being often semi-structured or un-structured. This is probably the biggest difference between DA and previous statistical methods, which cannot cope with heterogeneous data sources. This difference also explains why DA became a popular topic of discussion, research and business right after the rise of the Big Data trend, and why DA is often referred to also as "Big Data" Data Analytics (DA). In its report on the potential benefits of harvesting information from Big Data sources [101], McKinsey estimates value gains in the order of magnitude of hundreds of billions of US dollars in five important sectors (Healthcare, Public Sector, Retail, Manufacturing, Telecommunications), coming from cost reductions and performance improvements. Therefore, it is no wonder how the economical interests and the research challenges of integrating heterogeneous data sources at scale stimulated the research in DA.

Similarly, also ML was enabled by Big Data availability, which allows building statistically reliable models to predict the value of unseen data instances [3] . The first difference between DA and ML is the computational process of analyzing data, which for ML is called *training* and is directed towards the maximization of a utility function on the basis of statistical considerations [8, Chapter 1]. Therefore, this analysis is often computationally more complex and typically requires even more computing power than DA algorithms [4] . In supervised learning [8, Chapter 1], the output of training is not even valuable *per se*, but is used to predict the value of future unseen data [5] , thus being mathematically modelled as a function: it is this function that, for each data instance it then sees, returns the final information that is of interest to the user. Therefore, while DA has changed the decision-making processes from a high-level perspective, ML has also changed the business applications from the design to their implementation internals, and many of today's "intelligent", popular, user-facing services like web search and speech recognition are powered by ML models and thus require much more computing power than before to run. Hence, these application patterns, here collectively referred to as *data-intensive*, are now common "bricks" for business and science and have become fundamental to manage information in our society [6] .

[3] Here, we refer mainly to *supervised learning*, and in particular to *prediction* and *regression* tasks, as they are the most common usages of ML, especially at the business level. However, the scope this work is more general and encompasses all ML branches.

[4] The reader can think at the training of a linear classifier or of a Neural Network (NN).

[5] When the problem has a high dimensionality this output may even be little to no insightful, unlike DA results: the reader can again think at the weights of a linear classifier or of a NN, whose interpretability is per se a research challenge [7, 29]

[6] We are deliberately excluding some categories of application that can be also described as data-intensive, like database workloads: this category, indeed, already received a lot of attention over the decades and the research is very mature around it. Furthermore, database workloads are mostly memory-bounded and less diverse than the applications we cover here, and thus stress only specific parts of the architecture. The applications we cover here are inherently more complex and mixed.

1.2 Trends of computing architectures for data-intensive applications

As from the introduction of this chapter, the growing demand of compute power was partially met by the growth Moore's law allowed. However, the failing of Dennard's scaling stopped the increase in frequency and limited the "raw" computing power a single core can provide, causing the additional transistors to be employed for providing more cores or more specialized units, like Single Instruction, Multiple Data (SIMD) units [69, 97]. These enhancements, unlike those in the first years of the 21st century (with CPU frequencies steadily increasing), caused higher software development costs, which could only partially be avoided by improving compilation techniques (like auto-vectorizing compilers) and tooling in general.

Nowadays, applications willing to leverage the full power of the CPU they run on should be designed for large multi-threading settings and should use specialized libraries (an example is Intel MKL [67]) and be appropriately compiled and optimized [7] , taking in account all the characteristics of the target CPU like the resources on the datapath and the entire memory hierarchy. This makes optimization steps more complex, especially when multiple computational kernels and patterns are used together in the same applications and the performance bottlenecks do not depend on a single kernel but on multiple steps of the compute chain. Data-intensive applications, which work with multiple heterogeneous data sources at the same time, exhibit these issues more likely than previous applications, as in different steps they may stress different hardware resources (memory bandwidth, caches, datapath control, SIMD units, etc.) and thus need very different optimization techniques. This work investigates these aspects and provides novel guidelines to lead and automate the optimization phases of this class of applications.

Combined with the ubiquity of multi-core architectures, this diversity exacerbates complexity and unpredictability issues: since several resources are shared within multi-cores, uncontrolled contention often occurs among threads, especially among applications with different characteristics, making attainable performance less predictable. Existing solutions to these phenomena are only partial, further complicate the hardware interface and deeply interact with software optimizations [8] . Therefore, for software optimizations to be effective on multi-core CPU predictability and isolation are additional requirements to guarantee, and very lively research problems. A major example of contended resources are multi-cores shared caches like the Last Level Cache (LLC), for which a large body of software optimizations exists (like the already mentioned [23]), which are yet unaware of co-locations. These phenomena are particularly relevant where co-location patterns are hardly predictable, and modern CPUs offer an increased number of cores with more

[7] An example of powerful optimizations for certain classes of kernel are polyhedral transformations [23], which are though very application-specific and may heavily change the operation-per-loaded-byte ratio.

[8] For example, using Intel's SIMD instructions on a core affects not only the core frequency but also the frequency of other cores [176, 87].

and more co-location possibilities. Furthermore, modern applications are usually hosted in cloud computing environments [63], where the user typically has (and wants) little to no control over the hardware resources and the co-locations. In particular, Platform-as-a-Service (PaaS) cloud services abstract physical resources and let users focus solely on the development and testing of the applications, managing the infrastructure automatically: these services leverage co-location to be highly profitable for the provider, which sometimes provides users Quality of Service (QoS) guarantees to make the service more commercially appealing. In such a scenario, applications of different users may intertwine in using the shared hardware resources and contentious patterns may arise and change over time, as the users' and applications' mix changes; here, ensuring a given level of performance and QoS may become impossible, or highly depend on the hardware platform (e.g. the number of cores, the cache management strategies, etc.— all aspects out of the user's control). Therefore, modern infrastructures need some form of resource reservation at least for important applications, with proper interfaces to interact with users giving details about their requirements. Achieving this isolation is an orthogonal problem with respect to the one discussed in the previous paragraph, and the second goal of this thesis.

Nevertheless, some kernels have performance requirements that CPUs are not able to meet; the above-mentioned enhancements of CPUs of the recent years provided little to no benefit to these kernels, which are hardly parallelizable and are not compute-heavy, but rather control-heavy and data-dependant, thus insensitive to larger SIMD units or better speculation. One noticeable example are Regular Expressions (REs), whose usage is very common in data-intensive applications that work with textual data: both DA and ML applications often use REs to filter text inputs and search specific text patterns within them. In such scenarios, where applications change with high frequency, flexibility is a key requirement and is as important as performance for the applicability of the solution in the cloud-like scenarios where data-intensive applications run [9]. To meet the requirements, accelerators may be used like Field Programmable Gate Arrays (FPGAs): these platforms, among others, offer a high degree of flexibility and satisfiable performance for many use cases, and fit very well the characteristics of RE kernels. Therefore, this thesis will also explore the offloading of RE matching to a dedicated FPGA solution and how this can be achieved while keeping highest flexibility to the workload changes.

Once mechanisms for optimizations and resource reservation are available, they can be combined together to provide a higher degree of control over the final performance of applications, both by making it leverage the hardware resources with highest efficiency and by protecting it from unanticipated contention, combining the workload information from developers with the performance goals of users and deciding the optimizations to

[9] As an anticipation of section 2.3 for the unfamiliar reader, some works embed the RE structure to be matched into hardware, e.g. for network traffic analysis, where some data patterns are well known (IP addresses, sub-networks, packets headers, etc.); the resulting circuit can be deployed as an Application-Specific Integrated Circuit (ASIC) into Intrusion Detection Systemss (IDSs) and firewalls for maximum performance and energy efficiency. This inflexible solution does clearly not fit quickly changing workloads like those this thesis deals with.

perform and the degree of isolation to enforce.

1.3 Framework-as-a-Service, a playground for data-intensive applications

With applications steadily transitioning to the cloud, an increasing type of service offer is Framework-as-a-Service (FaaS), sometimes referred to as *serverless*, which has common elements with PaaS and Software-as-a-Service (SaaS). Like PaaS, FaaS services are highly programmable and configurable, offering Application Programming Interfaces (APIs) for the most used programming languages (like Java, Python, C#). Like SaaS, they hide all the details about the underlying hardware/software architecture and take care of automatically scaling and provisioning resources at a fine granularity, the reason why the APIs they expose are very high-level. FaaS offers are usually billed with a fine-grained, pay-per-use policy, and allow users to build quickly evolving and scalable applications for a variety of scenarios. Common scenarios are the "classical" batch data analytics, for example with Microsoft Azure Data Lake Analytics [15], event-driven, streaming data analytics like Amazon Lambda [14] or more recent services for ML like Microsoft Azure Machine Learning service [16]. Here, users can train and deploy ML with the most common tools and frameworks and the infrastructure takes care of provisioning the resources for the phase at hand (for example, the Graphic Processing Units (GPUs) for training); this automation allows the user to focus mostly on the training phase, where the model is "designed and built" in an iterative fashion. These services, sometimes referred to as Machine-Learning-as-a-Service (MLaaS), are spreading together with the usage of ML models within commercial applications: in addition to the already cited Microsoft's service [16], also Google Cloud AutoML [54] and AWS SageMaker [10] can be cited.

They run multiple data-intensive applications simultaneously at a large scale, with the possibility to provide users QoS guarantees to make the offer more appealing to the market, as required by applications like real-time information services. Since they hide the underlying infrastructure to the end users, on one side they can aggressively leverage co-location to make the service profitable for the provider, but on the other side they have to carefully provision resources to guarantee performance and QoS with little information from users. Therefore, they need both high optimization capabilities for the applications written against their APIs as well as predictability and isolation; in some cases, they can also offer heterogeneous solutions to accelerate workloads to meet the highest users' demands [10] , but these capabilities should be properly abstracted away from the user under a flexible, high-level APIs. For all these reasons, FaaS services face all the challenges discussed before, and are thus a reference platform for this work and for future developments.

[10]This is the case, for example, for Google Tensor Processing Unit (TPU), which are offered to users [55], or for the GPUs in the main MLaaS offers listed before.

1.4 Organization of this work

Following the topics introduced in the previous section, this thesis is organized as follows. Chapter 2 provides the necessary background, discussing the state-of-the-art works around the topics introduced in section 1.2 and the unsolved technological and research challenges, thus motivating the work in this thesis, and also shows the scenarios that can mainly benefit from it. The following chapters detail the research efforts around the goals in section 1.2, and provide more in-depth discussions over the specific problem each one tackles, the related literature and the proposed solution. Chapter 3 discusses the isolation of applications in the CPU LLC, showing an isolation mechanism that works on a large variety of CPUs, including those in recent servers, without hardware changes, and is completely integrated in the Operating System (OS). Instead, chapter 4 investigates the performance of several ML applications and explores the potential room for performance improvements by using both FPGAs and CPUs, underlining the main limitations of current ML systems and insights for the following chapter. chapter 5 builds on these observations to propose a more general optimization framework for ML workloads on CPUs, which achieves noticeable performance benefits along several dimensions that can help final users meet QoS requirements and decrease the TCO of the service, especially in a cloud setting. The findings in this chapter can be generalized also to DA applications and outline multiple research directions to embrace a broader class of applications and more heterogeneous computing resources. From there, chapter 6 explores a solution to accelerate some widely used functionalities like REs in order to satisfy performance goals that CPUs cannot meet, and is a case study for those kernels that already need heterogeneous solutions while preserving flexibility. Finally, chapter 7 concludes this thesis with a discussion of its achievements, limitations and of possible future work directions.

CHAPTER 2

Background and limitations of state-of-the-art

This chapter overviews the background concepts this work is based upon. Section 2.1 introduces the structure of modern CPUs and in particular the datapath and the cache hierarchy, highlighting the main challenges in terms of programmability, performance and efficiency. Building on these concepts, section 2.2 discusses how these challenges emerge in the context of ML prediction systems, whose workloads exhibit peculiar characteristics, higher diversity and complexity and faster evolution than "classical" DA. Finally, section 2.3 explains the importance of REs as a very common kernel within a wide range of DA and ML workloads, discussing how the limitations of CPUs pave the way to heterogeneous solutions, especially based on FPGAs.

2.1 Multi-core CPUs and their challenges

In the recent years, the advancements of the lithographic technology for multi-core architectures enabled a steady increase of the number of cores and in their complexity. Overall, these changes greatly increased the complexity of the hardware, therefore impacting the programmability and the possibility to achieve an efficient usage of the architecture, especially with memory-intensive applications. The issues due to these advancements are along two orthogonal directions: the first issue is the *contention* arising from multiple applications sharing the computational resources, while the second issue arises from the

inherent complexity of the core and hampers *achieving optimal performance* with a broad range of data-intensive applications. The following sections revise these directions and highlight the main challenges and solutions the research proposed so far.

2.1.1 Co-location of data-intensive applications

Co-location of multiple applications on a single multi-core is common on almost every modern platform, from smartphones to multi-socket servers. However, co-located applications contend for the multi-core shared resources, like the on-chip interconnection bandwidth, the memory controller, I/O ports and, in particular, the LLC [49]. This last component is particularly important to ensure the performance of applications, which contention may severely degrade. The clusters powering modern cloud computing are particularly subject to these issues, as the management policy generally tries to co-locate workloads on the same multi-core, in order to pack the workload on a smaller number of servers. Cloud applications served by these clusters are often bound to provide QoS guarantees to customers, with penalties for the service provider in case of QoS violation. In these environments, contention is a particularly important issue and performance unpredictability has consequences at the business level. Therefore, the literature is particularly rich of works addressing the LLC contention issue from multiple sides, and a comprehensive review (which should span several decades of scientific literature) is beyond the scope of this work, which will overview the recent, relevant solutions.

To understand the key challenges and properly categorize each work, we introduce two aspects related to addressing contention, the *policy* and the *mechanism*. The policy is the decisional process to mitigate or eliminate contention, which depends on information about running applications (gathered from profiling, manual tagging of important applications or various performance metrics) and is implemented into an algorithm running in hardware or in software: this algorithm starts from the input information about the current status and properly computes the amount of hardware resources (CPU bandwidth, LLC partitions sizes, etc.) to be assigned to each application. Once this resources assignment is computed, the underlying mechanism applies it depending on the specific hardware/software architecture: common mechanisms are those in the OS scheduler to change the CPU bandwidth of applications, software-based approaches to partition the LLC (more in the following paragraphs) rather than dedicated hardware interfaces to this aim [66]. Therefore, all research works can be categorized by these two aspects, and we will abide by this distinction in the remainder of this section and in chapter 3. Both aspects have been extensively studied, and it suffices here to overview the main solutions to the challenges current systems face.

To design policies to improve LLC sharing, a first way is modelling the LLC usage characteristics of applications, typically using profiling information. This information may come from stand-alone profiling and tracing of the locality characteristics like reuse

(or stack) distance [31, 178] or from more high level information like execution time or job latency, like in [102, 179]. A second strategy to control contention is to detect sub-optimal situations at runtime and react appropriately, usually by changing the resource assignment via the scheduler [188] or by throttling batch applications in favor of latency-sensitive applications [103]. Overall, the research around solutions to LLC contention is still very lively, and even recent hardware enhancements like LLC monitoring and partitioning facilities [62] did not fully solve the problems and led to further developments [128]. Indeed, LLC management policies depend on many environmental factors like the type, recurrence, number and diversity of applications, the control over the software stack (application/hypervisor/OS) and the underlying hardware, and the state-of-the-art is still far from a generally applicable solution for all scenarios or, at least, for well-established set of solutions known to work well for common scenarios.

While the policy depends on environmental factors that are hardly generalizable, the mechanism allowing the partitioning is usually tied to more controllable, lower levels of the hardware/software stack, in particular the OS and the CPU architecture. Here, the crucial points are essentially the tuning knobs the hardware exposes to the OS, with most architectures allowing no external control on data placement within the LLC and some architectures, either commercially available or from research works, offering some degree of control. For example, the recent Intel Haswell CPU family introduced Cache Allocation Technology [66], which exposes registers for a *way partitioning* [6] mechanism [1] entirely implemented in hardware. Although hardware solutions usually achieve lower overhead over software solutions, limiting the cache ways accessible to an application also limits the maximum associativity the application can leverage, which penalizes applications with good locality especially with increasing core counts and thus more partitions to apply [2]. Instead, set partitioning preserves the full associativity of the cache, but requires working at the level of data memory addresses (LLCs usually employ physical addresses), which affect the target set in cache placement algorithms. The classical way to achieve set partitioning in the LLC is a technique called *page coloring* [24], which dates back from the 1990 and essentially consists in selecting the physical pages to be allocated to an application on the basis of the LLC partition the application should reside in; since page allocation is performed within the OS without hardware intervention, page coloring requires only software modifications and has extensively been used in research. Chapter 3 discusses this technique in more depth and reviews the relevant literature about page coloring and LLC partitioning mechanisms in general. However, starting from the *Sandy Bridge* family (2009) up to the latest years, Intel achieved utter prevalence in the server market where data-intensive applications run and introduced a hash function into the LLC set selection algorithm, making "classical" page coloring ineffective. Hence, the challenge

[1] way partitioning consists in choosing on which LLC ways each core can place its data into

[2] note that in the latest years the core count has increased while the associativity has not, as it directly impacts the latency of the critical path of the tag lookup and the hardware costs with more comparators required

arose to adapt this technique to the new hardware powering datacenters, while keeping the software overhead low and allowing scalability to the large memory sizes of modern servers. All these challenges and the proposed solutions are addressed in chapter 3.

2.1.2 Optimizing data-intensive applications

The recent decade saw the addition of multiple features into the core design: the main trends are increasing the speculation capabilities to saturate the memory bandwidth and hide latencies and to add dedicated instructions for compute-intensive tasks. Speculation features of CPUs increased, as the increasing density of components allowed incorporating more effective predicting logic into a single core, both for data prefetching and for branch prediction. On one hand, branch prediction logic, which is highly integrated within the core pipeline and therefore highly tied to each vendor's design, has evolved to the point of using solutions based on NNs, like in the newest AMD processors [11]. On the other hand, data prefetchers, initially based on detecting fixed data-access strides [51, 26], lately became usually more effective in using the memory bandwidth [111], although several DA workloads with sparse data access (for graph or algebraic applications) still exhibit poor performance [19]. On the side of compute-intensive workloads, the most noticeable changes revolve around newer SIMD features like AVX instructions on x86 CPUs [69], whose width and complexity further increased and now have 512 bits operands and support for low-precision arithmetic for neural operators [97].

Coupled with out-of-order execution of modern CPUs, which implement vendor-specific evolutions of Tomasulo's algorithm [162], all these features make performance complex to predict and to achieve, forcing programmers to optimize their code through several phases of profiling. For example, the state-of-the-art performance investigation techniques for CPU extend the Roofline model with a Tomasulo-based simulator of the micro-architecture internals that simulates instructions execution and computes the impact of each micro-architectural feature on the execution time [27]; although very insightful, these techniques still require manual work and are hardly automatable to a broad range of applications. To avoid manual tuning, industry and research provided several solutions to leverage modern CPUs capabilities. In particular, dedicated libraries like Intel MKL [67] and optimizing compilers like Intel ICC and the open-source compiler GCC are the de-facto standard tools for such tasks, but the research showed that even for well-known kernels these solutions cannot explore the entire design space and produce the best solution. For example, [154] partially explores the parameter space of a Fast Fourier Transform implementation and finds the best implementation for given ranges of parameters, reaching or even exceeding the performance of MKL and similar solutions. Similarly, [68] specializes the *GEMM* implementation for the given parameters at runtime.

Higher level abstractions on hardware like managed languages face the same problems, but offer more solutions that fit in the higher software stack. While optimizing and

vectorizing compilation capabilities are natively available with the runtime [73], some approaches propose dedicated interfaces to allow programmers explore and offer custom implementations of vectorized primitives [156]. Even with these solutions, problems remain when applications grow complex and have multiple, diverse computational steps, like data pipelines typical of DA and ML: here, optimized implementations of the computational kernel classically remain separated into different function calls, with the need to materialize intermediate results in memory. This materialization requires memory bandwidth to write and read memory, a resource that has not scaled equally with the computational power of CPUs and of other platforms [100]. To overcome these issues, works like Weld [126] propose a Domain-Specific Language (DSL) and a runtime for cross-libraries optimizations, enabling fusion of kernels and removing result materialization steps, as in-memory DataBase Management Systems (DBMSs) like MonetDB [65] already do from nearly three decades, since they have full upfront knowledge of the kernels characteristics. Recently, also specific fields like image processing and NNs saw end-to-end approaches for optimizations like Halide [134] and TVM [33], which start from a custom representation of the computation (either with a DSL or with a Direct Acyclic Graph (DAG) of tensor operations), perform end-to-end optimizations on the whole compute graph and generate optimized code for various platforms.

Despite this growing body of research, only a few areas have been covered, and more general solutions are missing. Furthermore, the usage of DSLs, though recently advocated with Domain Specific Architectures [77], goes towards the opposite direction of further specialization, and a proliferation of DSLs can force developers to learn multiple programming abstractions and toolsets that change over time. An example of this issue is the ML context, where multiple abstractions and toolsets exist for various types of NNs like Convolutional Neural Network (CNN), Deep Neural Network (DNN), etc. or very specific ML classes of models, while a general approach is missing that allows optimizing all ML operators and all pre-processing steps along data-intensive ML pipelines, as the following section explains.

2.2 Challenges of implementing Machine Learning Prediction-Serving systems on modern CPUs

The increased computing power of modern architectures allowed the growth of applications based on ML, which have a large variety of patterns to pre-process and transform data into some kind of prediction. This variety of patterns is especially visible during *inference* (also called *scoring*), where the input data "flow" through several operators to finally produce the output; in this phase, we experience the issues introduced in section 2.1.2. To have an idea of the variety of patterns and transformations needed to perform ML inference, it suffices to think of the wide variety of applications and scenarios ML models are employed in.

Nowadays, several “intelligent" services such as Microsoft Cortana speech recognition, Netflix movie recommender or Gmail spam detector depend on ML model inference capabilities, and are currently experiencing a growing demand that in turn fosters the research on their acceleration. In general, we can identify two typical scenarios with homogeneous characteristics in terms of deployment strategies and requirements:

- in the *web service* scenario a user-facing application, typically distributed and web-based, relies on an ML prediction to provide the output, as in the case of search or recommendation engines; the performance of the application tightly depends on the *latency* of the prediction
- in the *batch scenario*, the user runs periodic prediction tasks on large datasets, typically using multiple prediction algorithms with common pre-processing steps, as in the case of market prediction workloads; throughput is the key performance metric in these cases

While efforts exist that try to accelerate specific types of models - e.g., Brainwave [110] for DNN - many models we actually see in production are often generic and composed of several (tens of) different transformations.

Many workloads run in so-called *prediction-serving* systems, which are cloud services that highly abstract physical resources, ease the maintenance burden and provide (semi-)automated scaling capabilities. These systems are designed to be used by data science experts, who train prediction models according to their specific workflows based on some high-level framework and deploy them to such services; here, models are *operationalized* and made available to web or batch applications requesting predictions. . Indeed, while data scientists prefer to use high-level declarative tools such as Internal Machine Learning Toolkit (IMLT) for better productivity, operationalized models require low latency, high throughput, and highly predictable performance. By separating the platform where models are authored (and trained) from the one where models are operationalized, we can make simplifying assumptions on the latter:

- models are pre-trained and do not change during execution[3];
- models from the same cloud user likely share several transformations, or, in case such transformations are stateful (and immutable), they likely share part of the state; in some cases (for example a tokenization process on a common English vocabulary) the state sharing may hold even beyond single users

These assumptions together motivate the research of common patterns among multiple models: indeed, computationally-heavy and similar transformations with immutable state can be thoroughly optimized and even offloaded to dedicated hardware with only a one-time, setup cost for setting the state. Once the most common sequences of transformations

[3] With the exception of online learning, here not discussed for simplicity.

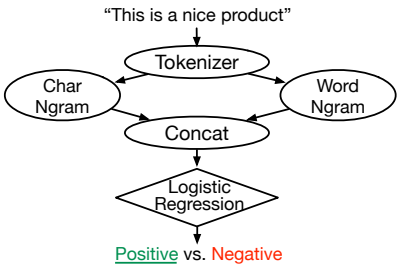

Figure 2.1: *A Sentiment Analysis (SA) pipeline consisting of operators for featurization (ellipses), followed by a ML model (diamond).* Tokenizer *extracts tokens (e.g., words) from the input string.* Char *and* Word Ngrams *featurize input tokens by extracting n-grams.* Concat *generates a unique feature vector which is then scored by a* Logistic Regression *predictor. This is a simplification: the actual DAG contains about 12 operators.*

are identified in a workload, multiple prediction models can benefit from their acceleration. Chapter 4 explores this approach by accelerating some ML models to both CPU and FPGA, showing large room for improvements over current approaches.

Building on these findings, chapter 5 tackles the more general case: there, ML models are authored as *pipelines* of transformations, which are DAGs of operators comprising *data transformations* and *featurizers* (e.g., string tokenization, hashing, etc.), and ML models (e.g., decision trees, linear models, SVMs, etc.). Many ML frameworks such as Google TensorFlow (TF) [158], Facebook Caffe2 [28], Scikit-learn [129], or Microsoft ML.Net [112] allow data scientists to declaratively author these pipelines of transformations to train models from large-scale input datasets. Figure 2.1 shows an example pipeline for text analysis whereby input sentences are classified according to the expressed sentiment. ML is usually conceptualized as a two-steps process: first, during *training* model parameters are estimated from large datasets by running computationally-intensive iterative algorithms (an example is the back-propagation algorithm [9, 175]); successively, trained pipelines are used for inference to generate predictions through the estimated model parameters.

When trained pipelines are served for inference, the full set of operators is deployed altogether to pre-process and featurize the raw input data before ML model prediction rendering. However, pipelines have different system characteristics based on the phase in which they are employed: for instance, at training time ML models run complex algorithms to scale over large datasets (e.g., linear models can use gradient descent in one of its many flavors [140, 135, 148]), while, once trained, they behave as other regular featurizers and data transformations; furthermore, during inference pipelines are often surfaced for direct users' servicing and therefore require low latency, high throughput, and graceful degradation of performance in case of load spikes. Therefore, we focus on the inference phase, which shows all the challenges overviewed in section 2.1.2. During training, most of the time is spent in actually finding the optimal parameters for the output model, which

involves the exploration of large solution spaces usually via a gradient-descent algorithm: here, the cost of pre-processing steps can be amortized via batching inputs, while most of the training time is actually spent in the space exploration. Instead, in prediction-serving latency requirements often prevent batching, and operators have to be run one at a time, and possibly be optimized altogether to take full advantage of the CPUs they run on.

Existing prediction-serving systems, such as Clipper [**clipper2**, 40], TF Serving [159, 122], Rafiki [170], ML.Net [112] itself, and others [130, 137, 113, 115] focus mainly on ease of deployment, where pipelines are considered as *black boxes* and deployed into *containers* (e.g., Docker [46] in Clipper and Rafiki, *servables* in TF Serving). Under this strategy, only "pipeline-agnostic" optimizations such as caching, batching and buffering are available, while it is not possible to apply in-depth optimizations across operators. Indeed, we found that black box approaches fell short on several aspects in addition to code optimizations. For instance, current prediction services are profitable for ML-as-a-service providers only when pipelines are accessed in batch or frequently enough, and may be not when models are accessed sporadically (e.g., twice a day, a pattern we observed in practice) or not uniformly. Also, increasing model density in machines, thus increasing utilization, is not always possible for two reasons: first, higher model density increases the pressure on the memory system, which is sometimes dangerous—we observed machines swapping or blocking when too many models are loaded. As a second reason, co-location of models may increase tail latency especially when seldom used models are swapped to disk and later re-loaded to serve only a few users' requests. And, even if kept in memory, black-box data and task replication leads to higher runtime overhead, poor locality and uncontrolled co-location of tasks. Interestingly enough, model pipelines often share similar structures and parameters inasmuch as A/B testing and customer personalization are often used in practice in large scale "intelligent" services; operators could therefore be shared between "similar" pipelines. Sharing among pipelines is further justified by how pipelines are authored in practice: ML pipelines are often produced by fine tuning pre-existing or default pipelines and by editing parameters or adding/removing steps like featurization, etc. In chapter 5 we will leverage these insights to go beyond a black-box approach and show that a wide-box approach can allow models optimizations along multiple dimensions.

2.3 In-depth Data Analytics acceleration: the case study of Regular Expressions

Although a white-box approach can provide substantial benefits to accelerate data-intensive applications, some widespread kernels are at the base of applications with ever-increasing performance demands. A major example of them are REs, which are widely applied in a number of fields that range from genome analysis to text analytics and IDSs, to name a few. The requirements of these applications and the large amounts of data to be analysed through REs make the performance of RE matching solutions crucial, and keep fostering

the research on this problem. As an example, IDSs may require near-real-time analysis of network packets at the network rate, which reaches several tens of Gb/s in bandwidth. REs are also used in many business applications, from the pre-processing steps of some ML pipelines to database queries. To sustain the demand of these applications, CPU-based solutions are not enough, and even the availability of more cores with newer silicon generations does not allow meeting the performance demands. Therefore, previous IDS solutions employ specialized hardware [106] or use more flexible alternatives like GPUs [166]. While the latter solutions retain good programmability and flexibility, they usually come at the cost of low energy efficiency with respect to the former solutions. Indeed, RE matching solutions in the literature in general have been struggling between performance and energy efficiency on one side and flexibility on the other side. Here, we are reviewing the main works in the literature of REs acceleration, to show the main directions the research has taken and the various shortcomings that motivate our work in this field, which demonstrates how dedicated accelerators, implemented in FPGA or even ASIC, are necessary for some data-intensive applications.

A large body of applications and of literature around REs revolves around "classical" packet inspection systems, which are at the base of security-oriented solutions for networked systems and are still a very live research topic. These applications analyze network traffic by inspecting the packet content at various levels of the ISO/OSI stack, from level 1 up to level 4 (traditionally called "packet analysis") or even to level 7 (called "deep packet inspection"). In the case of packet analysis, the patterns to match against are usually simple, being typically limited to part of the IP address (to match, for example, malicious sub-networks) and to port numbers. Instead, deep packet inspection requires more time-consuming pattern matching within the payload of packets, for example by looking for REs that describe URLs to detect spam contents or security exploits. In order to address the issues in this domain, many solutions have been developed, which historically make up the bulk of literature in accelerating RE matching. Indeed, section 6.2 reviews the main approaches of this corpus in the related chapter.

In recent years, RE matching found novel applications through genome analysis and its derivations, where REs are used to find common patterns among genomic sequences for a variety of goals. One large body of work, that has already emerged thanks to the advancements in genomic sequencing, is finding patterns for classification purposes, e.g. proteins [187] or generic genome sequences [125, 167]. Another fast-growing area that exploits RE matching for genomic applications revolves around the study of diseases and their treatment. Several works, for example, look for common patterns as genetic markers that are correlated to health issues of various types (like [108, 30]). Pushing this approach forward, so-called "personalized medicine" for, e.g., cancer treatment [71, 144] is gaining momentum as a promising application for pattern matching, where genomic inputs directly come from patients under treatment to diagnose possible diseases and find the best cure. Recent successful breakouts of these techniques [34] showed very promising results, and

suggest further developments of healthcare- and genomic- related applications that require powerful matching solutions for genomic data.

Coming to the implementation of dedicated solutions, the literature shows a large body of works using FPGAs and ASICs. In recent years, FPGAs have been employed in many different fields thanks to their flexibility and proved to be best with respect to energy efficiency in a wide variety of applications, to the extent that a large research body is exploring heterogeneous architectures that combine different technologies such as CPU, GPU and FPGA in the same system. Moreover, their reconfiguration capabilities make FPGAs compelling for realizing high-performing and energy-efficient systems that also need some degree of flexibility for deployment, trading off the best aspects of other architectures (including ASICs) in those scenarios. In particular, matching RE is usually a control-heavy activity where advanced speculative features and vector-wide extensions of modern CPUs typically fall short: indeed, CPUs mainly rely on their high frequency and on an increasing number of cores to achieve acceptable performance, at the cost of low energy efficiency and high heating dissipation. Similarly, RE matching does not typically fit GPU characteristics, which are not suited to control-heavy scenarios. Instead, FPGAs often achieve good performance and energy efficiency with control-heavy applications, as they allow embedding and thoroughly optimizing the control into the synthesizable logic.

Among others, a previous research work [127] explored how to use FPGAs for RE matching while offering a higher degree of flexibility than previous works. Its basic idea is to translate an RE into a set of dedicated instructions executed on a "dedicated CPU" built into FPGA, highly optimized for this task and highly configurable to be adapted to multiple scenarios. In chapter 6 we build on the approach to propose a more flexible and performing solution for RE matching.

CHAPTER 3

Performance predictability through isolation in Last Level Cache

This chapter describes our work on achieving performance isolation in the LLC of modern CPUs. Section 3.1 describes the problem in more detail, showing the issues in the context of modern CPUs. Section 3.2 provides the background technical knowledge of this chapter, especially the base software techniques, while section 3.3 discusses the state-of-the-art work in LLC isolation. Section 3.4 explains the impact of modern CPU architectures on the state-of-the-art techniques and the approach we used to design the system, while section 3.5 shows the evaluation. Finally, section 3.6 concludes the chapter with the achievements and the future work deriving from them.

3.1 Introduction

Contention on the shared LLC can have a fundamental, negative impact on the performance of applications executed on modern multi-cores. The increasing number of cores and the variability of workloads running in modern environments further exacerbate contention phenomena, hampering scalability and the fulfillment of QoS guarantees. Manufacturers of multi-cores tackled some of these issues with architectural changes. For example, Intel tried to solve contention at the port level by deep changes to the LLC structure.

With the Sandy Bridge family, Intel split the LLC in multiple parts, called *slices*, each one with dedicated resources, that cores can access via a dedicated ring interconnection network [90]. To prevent multiple cores from accessing the same slice simultaneously and to avoid bottleneck effects, Sandy Bridge spreads accesses by means of a hash function computed on the physical address of the data to be retrieved; instead, the access within a slice leverages the physical address with the usual scheme. This deep change, however, does not solve contention within sets. Cores can still access any slice uniformly, and co-located applications are still likely to experience strong contention when accessing sets within a slice. Intel, with the new Haswell platform, provides a solution based on way partitioning [66], which comes at the cost of limiting the associativity available to each application. To overcome this limitation, other approaches are possible to decrease contention within sets, as discussed in section 3.3, but are currently confined to research and no well-established solution exists.

A software-only approach to address LLC contention issues is based on *page coloring*, a technique that leverages the physical memory to control the mapping of data into the LLC: by controlling physical addresses of data, page coloring can partition the LLC among applications and increase the predictability of performance. The key assumption of page coloring is that the cache is physically addressed, so that controlling the physical memory of an application allows controlling also the cache space devoted to the application. However, recent multi-core architectures (e.g., Intel Sandy Bridge and later) switched from a physical addressing scheme to a more complex scheme that involves a hash function. Therefore, traditional page coloring is ineffective on these recent architectures, as discussed in discussed in section 3.4.

Here, we extend page coloring to work on these recent architectures by proposing a mechanism able to handle their hash-based LLC addressing scheme. Just as for traditional page coloring, the goal of this new mechanism is to deliver performance isolation by avoiding contention on the LLC, thus enabling predictable performance. We implement this mechanism in the Linux kernel and we evaluate it using several benchmarks from the SPEC CPU 2006 [61] and PARSEC 3.0 [20] suites, delivering performance isolation to concurrently running applications by enforcing partitioning of a Sandy Bridge LLC, which traditional page coloring techniques are not able to handle.

Overall, this work brings three main contributions.

1. We provide a methodology to reverse engineer the hash function of Sandy Bridge CPUs. The proposed methodology leverages hardware performance counters available in modern architectures and is based on assumptions that previous work demonstrated to be consistent across multiple families. Unlike previous methods, our methodology is robust to noise in input data and identifies the exact hash function. Moreover, although experimented with only one CPU model, we believe it is general enough to be applicable also to recent models by Intel.

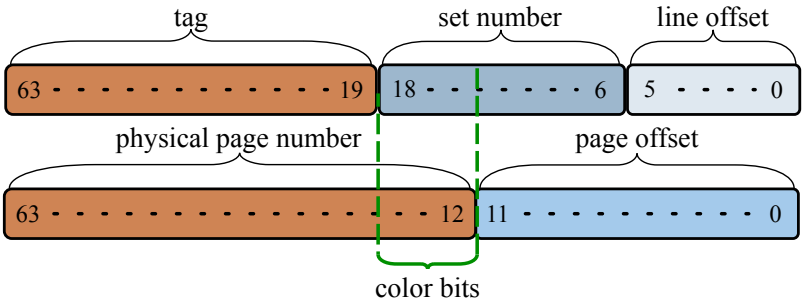

Figure 3.1: *Bit fields of a physical memory address*

2. Our solution uses the information on the hash to generalize page coloring to hash-based caches, allowing the system administrator to choose the size and the position of the LLC partition even in presence of hash-based addressing.

3. We generalize the notion of "page color" within the Linux memory allocator, thus achieving scalability and efficiency with a more general design than previous works.

We implement our page coloring scheme in the Linux kernel and validate it on real hardware with workloads from the SPEC CPU2006 and PARSEC 3.0 [20] benchmark suites.

3.2 Background

This section introduces the fundamental concepts for this work. Primarily, section 3.2.1 introduces *Page coloring*, the state-of-the-art technique to partition CPU caches that requires software-only modifications. These modifications mainly affect the physical memory allocator of the OS, which is the Buddy allocator in Linux and is introduced in section 3.2.2.

3.2.1 Page coloring

Well-known in the literature, page coloring [24] is the mechanism at the base of techniques for software cache partitioning, and is based on how modern shared caches map data to cache lines. These caches use the physical address to map data, and the allocation position can thus be controlled via the physical memory allocator of the OS. Some bits used to select the cache set are typically in common with the address of the physical page, and can be controlled via the OS to reserve cache space for a given application.

For example, fig. 3.1 shows the parameters of a real CPU, namely an Intel Xeon E5 1410, where the LLC is the third layer of cache. There, bits 12 to 18 are in common between the set number and the physical page number, and are called *color bits*, while a configuration of them is called *page color*. In the example, 7 color bits are present, so that the LLC can be split in at most 128 partitions. Nevertheless, fig. 3.3 shows that bits 12 to

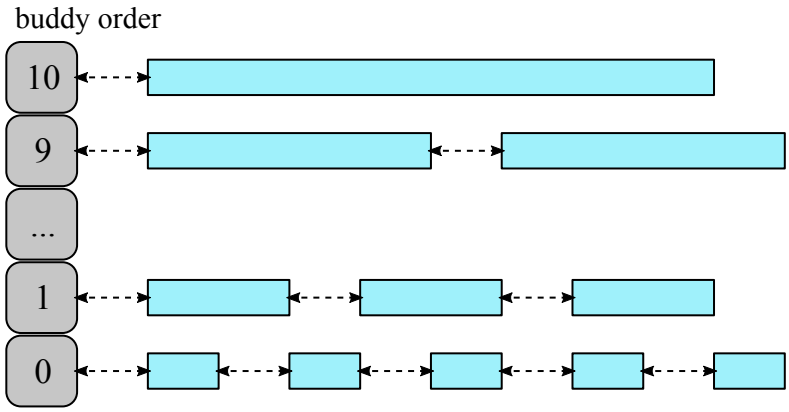

Figure 3.2: *Data structure of the original Buddy algorithm, with one list of buddies per order*

14 are used both as color bits and as set bits for Level 2 (L2) cache addressing. Using these bits for LLC partitioning would also partition the per-core L2 caches of accessing cores, which is undesirable. Therefore, only bits 15 to 18 are finally available for partitioning, finally allowing 16 partitions.

To enforce the partitioning, the OS allocates pages on a per-application basis. Each application is assigned a suitable number of colors, and the OS uses those colors only for the application memory. Therefore, other co-located applications cannot interfere with the data accesses of the target application, which is able to exploit the cache space reserved with full associativity.

The main disadvantage of page coloring comes when re-partitioning (called "recoloring") occurs: in this case, memory pages must be copied to new locations of different colors, with the consequence of a high overhead for data copy (in the order of 1us per page). Hence, page coloring has mainly been investigated as a static "technique", and the works in the literature try to limit recoloring overhead by copying pages only when really needed.

3.2.2 Buddy algorithm

The Buddy allocator [86] divides the physical memory into *buddies*, which are contiguous memory areas of different size. A parameter called *order* characterizes each buddy, whose size is 2^{order} the size of a page. Buddies are aligned to a memory address that is a multiple of their size, hence being aligned to $order$ bits beyond the number of page bits (typically 12): therefore, a buddy of order 0 is aligned to a 12 bits boundary, a buddy of order 1 is aligned to a 13 bits boundary, etc. This alignment constraint allows identifying each buddy through the memory address of its first byte. Each buddy is strictly coupled to either the previous or the following buddy of the same size, depending on the least significant non-

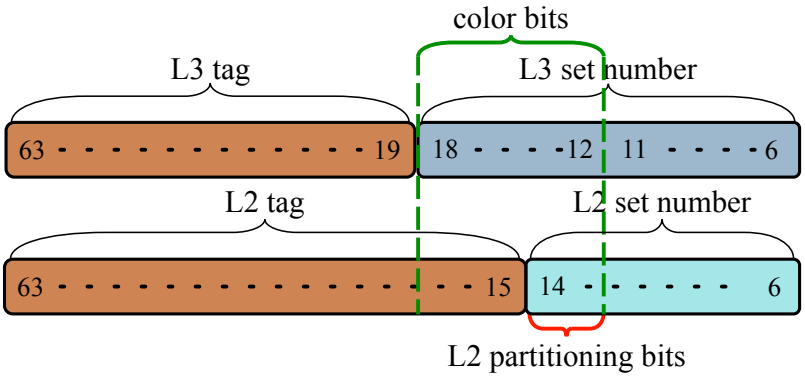

Figure 3.3: *Overlap of L2 set bits with color bits*

aligned bit: indeed, to find the coupled buddy from a given one, it is sufficient to invert this bit.

Figure 3.2 shows the data structure the algorithm uses to achieve efficiency, storing buddies of different orders in different lists. Therefore, when subsystems of the OS request a memory area of a certain size, the allocator rounds it by excess to the closest power-of-two number of pages and returns the buddy of that size. In case no suitable buddy is present, the Buddy algorithm *splits* a buddy of higher order in two parts, stores the second half to the list of free buddies of lower order and returns the first half. Conversely, the algorithm maintains scalability and efficiency by *merging* two contiguous free buddies, taking advantage of the "companion" of buddies. On every buddy being freed, the Buddy algorithm checks whether the coupled buddy is also free and groups them in a single, higher-order buddy that is inserted in the proper list, eventually iterating the merging procedure until the maximum order is reached.

3.3 State of the Art

A wide literature is available addressing contention within the LLC, which proposes solutions of very different nature. In summarizing the wide body of available literature, two main aspects should be distinguished: techniques and policies. On one side, the *techniques* are the mechanisms that allow controlling the cache. Many works, for example, propose hardware techniques that affect the internals of the cache, often modifying the Least Recently Used (LRU) priority assignment to cache lines. On the other side, the *policy* is the algorithm and the metrics that drive the mechanism, deciding the actions to be applied through the mechanism. For example, a partitioning policy may gather several metrics for each core and decide the size of each cache partition according to predefined goals (like fairness or user's QoS requirements). Both the mechanism and the policy can be implemented either in hardware or in software. This distinction is fundamental in this review,

and the related literature can be classified by means of these two aspects.

In the following review, we cover hardware techniques (section 3.3.1) and software techniques (section 3.3.2). Hardware techniques require modifications to the cache functioning, and may implement in hardware both the mechanism and the policy. However, a hardware technique can still offer interfaces to the software level in order to tune its behavior to high-level, software-defined goals. Instead, software techniques do not require hardware modifications, and are all based on page coloring to control where applications data are placed inside the cache. section 3.3.3 reviews the techniques to unveil the hash function of Intel's CPUs, from which we learned the assumptions that are at the base of our reverse engineering technique.

3.3.1 Hardware techniques

Overall, hardware techniques are diverse: some change the implementation of the LRU algorithm while others allow more explicit control over the data placement. A first mechanism for cache partitioning is called *way partitioning*, usually employing a bitmask to indicate which ways each core can access inside the cache sets. In case of eviction, the LRU policy works only on the cache lines assigned to the core causing the eviction, so that each set is effectively partitioned among the cores. Some special-purpose architectures like Octeon [121] or some prototype multi-cores [37] adopt this LLC partitioning mechanism, which is also finding room in modern, commercial architectures. However, way partitioning decreases the associativity available to a core, as it allows accessing only a subset of the lines, offering a smaller set of candidates for eviction with respect to the case with full associativity. Several policies have been proposed to compute the number of ways for partitioning. Among them, Utility-based Cache Partitioning (UCP) [132] is the base for several works in the literature: it defines the benefit each application can have from receiving more ways based on the derivative of the miss rate curve. Works like [57] allow way partitioning while increasing the spatial locality with aggressive prefetching, based on samples from several sets.

Other works, instead, change the LRU policy to tackle phenomena like thrashing or pollution. [150] consider multi-thread applications and focus on fairness among cores by penalizing the core with highest Instructions Per Cycle (IPC) in favor of the others: a modified LRU policy evicts a cache line of the high-performing core in case of miss from another. Since LRU is an history-based policy, it is unable to predict the usage of new cache lines. Therefore, low-reuse cache lines may evict high-reuse cache lines (*cache pollution*), and high-reuse cache lines can continuously evict each other (*cache thrashing*). [147] enhance the LRU policy with information about the frequency of recently evicted lines, which is stored in a novel hardware structure and allows deciding whether to store the incoming line or bypass it to the core. Other works attempt to predict the reuse of incoming blocks to affect the replacement policy, for example by storing the history of incoming

blocks [91] or with an address-mapped table of saturating counters [83]. Other works, instead, choose at runtime a certain policy based on *set dueling* [133], which consists in sampling the behavior of several cache sets where a fixed policy is applied [72]. Recently, [82] proposed two replacement policies based on the observation that the LLC should capture read-reuse patterns, while lines from write-only patterns can usually be evicted safely. [82] partitions each LLC set in two regions, the clean and dirty regions, and uses the clean region for read-only lines, while the dirty regions contains written lines and is resized according to information from set dueling.

Vantage [142], instead, is a more disruptive approach based on the *z-cache* model [141]. A zcache maps the incoming data to a line by means of a hash function, achieving high high associativity. Leveraging this feature, Vantage partitions the LLC by introducing two regions that capture high-reuse and low-reuse lines respectively, and that can be resized dynamically. [169] introduce the concept of *futility*, which describes the "uselessness" of a cache line w.r.t. the others. A cache should always present a broad set of "futile" candidates for eviction, even in the presence of partitioning, and [169] scales futility based on the insertion and eviction rates of each application's partition.

3.3.2 Software techniques

Many software techniques have been proposed to alleviate interference at the cache level. Most of these techniques assume a usual LLC addressing scheme, without a hash function, (like Intel's Nehalem's scheme).

Some works classify applications based on their sensitivity to co-located applications and on the contention they cause to other applications, usually by a micro-benchmark that exercises a tunable pressure in LLC and memory [43, 102]. In an initial learning phase the pressure on the memory subsystem is varied to devise the application profile,which is then used to classify the application and avoid dangerous co-locations. Pushing on this approach, [179] monitor latency-sensitive applications at runtime, continuously adapting the co-location to the application phases. Instead, other techniques directly manage the LLC and are all based on page coloring, but apply it for different goals and with different policies. For example, some tackle pollution by limiting the cache space I/O buffers can use, either assigning a fixed number of colors [85] or identifying pages accessed sequentially (typical of I/O buffers) and mapping them to few colors [45]. Other techniques, similarly, find polluting memory pages through the miss rate and migrate them to a small number of colors [153].

To precisely partition the LLC, some approaches perform application profiling, and typically choose the number of colors for each application based in low-level metrics such as miss rate and stall rate [157]. Other works like [186] or [95] profile the applications characteristics at runtime and perform several actions like recoloring hot pages only. Nonetheless, this profiling can be heavy (for example, [186] hot page recognition

requires traversing the OS page table). [93] evaluate several policies and metrics to guide the partitioning and find similar results in term of overhead. [92] investigate how to identify application phases by means of miss rate curves and IPC curves, and apply recoloring to adapt partitions to phases.

Several works employed a theoretical approach to model applications' LLC characteristics. [143] modeled the bandwidth usage and the performance variation due to co-location, using profile data that capture applications' characteristics and runtime phases across different time windows, validating their approach on an Intel Nehalem architecture. [25] evaluate several policies of fairness with a static partitioning approach on an Ivy Bridge architecture, and provide a theoretical framework to find the optimal LLC partitioning and sharing scheme with respect to these policies. Thanks to this approach, the policies they evaluate have general applicability.

Other works explore page coloring techniques on newer architectures. [84] focus on real-time systems, developing a partitioning and sharing scheme that considers tasks with given deadlines for completion. This work starts from profiling information about the stand-alone worst case execution time with different LLC partitions and devises a scheme for partitioning and sharing, and a time schedule that meets the applications' deadlines. To enforce the partitioning, [84] implements a page coloring technique on a Sandy Bridge CPU, but does not deal with the of the hash function, thus leaving the LLC partitions spread across all the slices. Furthermore, [84] employs all the bits from 12 to 17 as color bits, thus also partitioning the L2 cache (as in fig. 3.4). [181] propose two novel recoloring policies that consider also time sharing of cores and QoS requirements, focusing instead on server environments. The first policy recolors a number of pages proportional to the memory footprint, but proves to be sub-optimal since it often recolors rarely used pages. Instead, the second policy tracks page hotness by sampling the OS page tables and remaps them to different colors to better distribute the accesses. [181] also validates the proposed solutions on a low-end Sandy Bridge architecture. However, the presence of the hash-based mapping is not considered in the design phase.

Finally, page coloring finds applicability also with Virtual Machiness (VMs): in this scenario, the a priori knowledge of the memory footprint of VMs can be used as a hint for the LLC partition size. [76] use page coloring within the Xen hypervisor to show that also VMs benefit from LLC partitioning. Proceeding in this direction, [171] adds a dynamic re-coloring mechanism that moves the most used pages, not to stop the VMs during page copy.

3.3.3 Reverse engineering Intel's hash function

Effort has been devoted to reconstruct the hash function of Intel Sandy Bridge processors, mostly for security purposes: indeed, knowing the LLC hash function allows an attacker to perform a side-channel attack, e.g. by probing hot code areas like cryptographic libraries

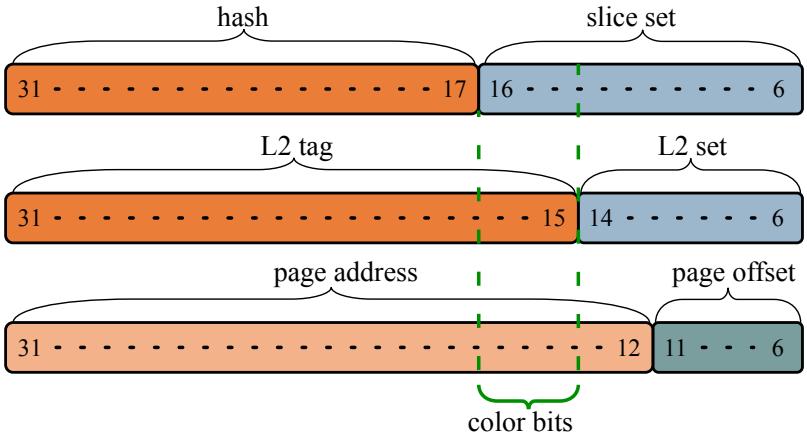

Figure 3.4: *Overlap of L2 set bits with color bits within Sandy Bridge*

and checking the load time.

Disregarding the way they unveiled Intel's hash functions, this body of research shows that the hash is based on the XOR operator, and that it depends on the number of slices of the LLC and not on the specific architecture: indeed, with 2^n slices the hash function consists of n different hashes that output a single bit, and that XOR certain bits. Hence, reconstructing the hash function basically reduces to finding which bits are XORed in each hash.

[64] discovered the hash function of an Intel Core i7-2600 multi-core, by finding conflicting addresses that map to the same LLC set. However, in [64] the authors manually compare the conflicting memory addresses to find repeated patterns that may hint the hash function. Instead, [70] adopt an analytic approach, solving a linear equation system in order to find the possible hashes: in their formulation, each address is a constant matrix of coefficients that multiplies a vector of unknowns, which represent the coefficients of the hash function (1 for the bit being used, 0 otherwise). Since the final solution depends on the unknown labels of the slices, [70] provides multiple solutions for each CPU, without identifying the real solution for the CPU under test. Moreover, [70] handles noise by filtering addresses with intermediate access latency, assuming that the remaining ones are reliable and can thus be used for the linear system formulation. Doing so, however, requires knowledge of the CPU latencies and depends on the specific model. Finally, [173] also investigates the hash function of a 4 cores and a 6 cores Sandy Bridge CPUs, but does not provide a mean to represent them as formulas or as algorithms, using instead mapping tables of considerable size.

3.4 Approach and design

With the introduction of a hash-based LLC addressing, page coloring becomes less effective in Sandy Bridge multi-cores. Figure 3.4 shows the typical memory layout for a Sandy Bridge processor, and in particular how the physical address is used to map LLC, L2 and physical memory. In fig. 3.4, "slice set" indicates the bits used to select the set within the LLC slice, while the rest of the bits are used to compute the hash that selects the target slice. As visible in fig. 3.4, the overlap between slice bits and L2 set bits leaves only 2 bits available for page coloring, thus with a granularity of only 4 partitions. Using also the L2 set bits causes partitioning of the L2 cache, which is an undesirable performance bottleneck since the L2 cache is per-core. Figure 3.5 shows the LLC miss rate (blue curve), the slowdown with respect to the full-LLC execution (red line) and the L2 cache miss rate (green line) of 8 applications from the SPEC CPU2006 suite [61]. In this scenario, we used the bits 12-16 for partitioning, and varied the number of colors from 2 (corresponding to 0.625MB of LLC and half L2 cache) up to 32 colors (full LLC and full L2). As a special case, bzip needs at least 1.88MB of LLC, since its memory footprint corresponds to 6 colors, thus using the whole L2 cache. In fig. 3.5, 5 out of 8 applications are sensitive to the LLC (we exclude bzip from this count for the aforementioned reason), and their slowdown is visibly correlated to the amount of L2 cache. For this reason, we chose not to partition the L2 cache, renouncing to the L2 set bits of fig. 3.4. Furthermore, since the two remaining color bits are within the slice bits and the hash function spreads accesses among all the cores, a single partition spans across 4 slices. This is undesirable, as it prevents a fine-grained control over the LLC placement and can increase the traffic on the on-chip ring bus.

To achieve a deeper control over the hash-based LLC, the knowledge of the hash function is fundamental. Section 3.4.1 describes the assumptions and the steps to reconstruct this information using the performance monitoring features available in modern architectures, focusing in particular on Intel's Sandy Bridge architecture. With the knowledge of the hash function, section 3.4.2 redefines the notion of page color to adapt it to a hash-based scheme, generalizes it to the memory areas of various sizes that the Buddy allocator handles and leverages this notion to achieve an efficient and scalable implementation of a color-aware memory allocation algorithm.

3.4.1 Reconstruction of the hash function

Despite the hash function of Sandy Bridge processors is undocumented, information is available from previous work [64, 70, 173], whose limits are shown in section 3.3.3. For the hash reconstruction we used a similar approach to [64], although using a more structured flow. In our reconstruction, we initially target an Intel Xeon E5 1410 CPU with 4 cores and 6GB of Random Access Memory (RAM). From the available literature, we can

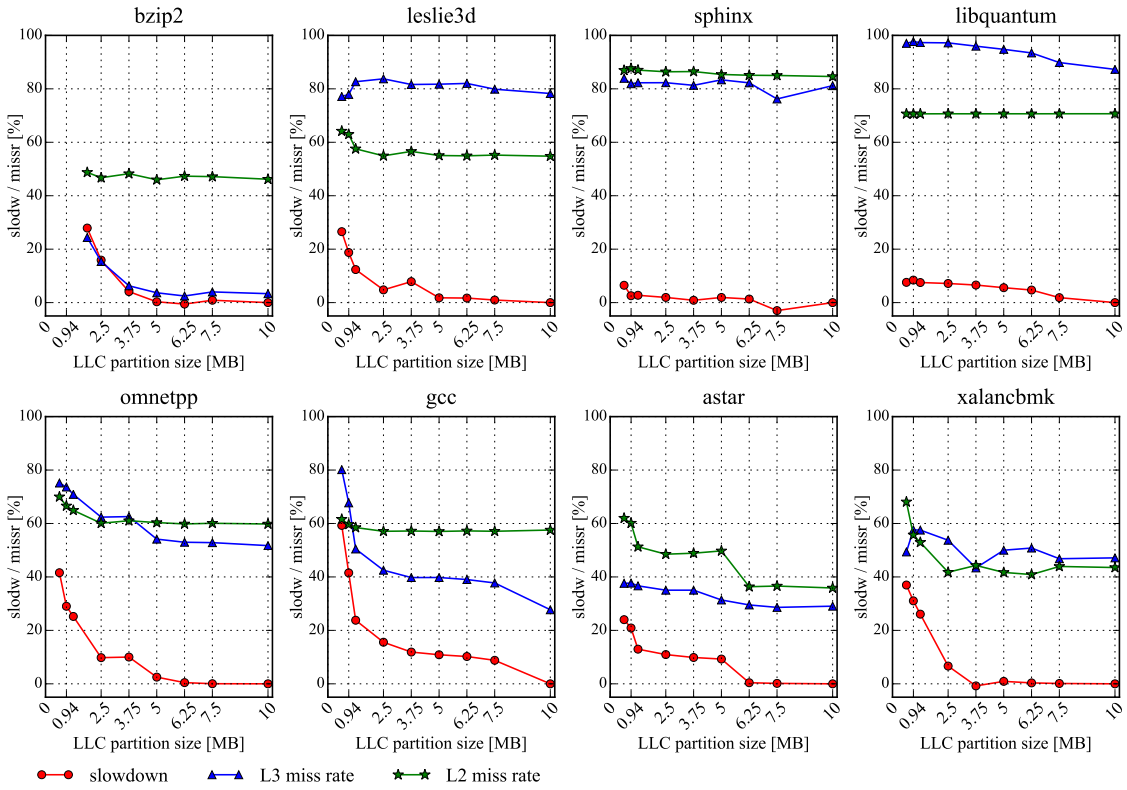

Figure 3.5: *SPEC cache profiles when using bits 12-16 for partitioning*

confidently make the following assumptions over the hash function of our CPU:

1. since the CPU is a 64 bits CPU with four cores, the hash function has the form $h() : \{0,1\}^{64} \longrightarrow \{0,1\}^2$, and is hence composed of two distinct scalar hash functions: $h(a) = h_2(a)h_1(a)$, where a is the input memory address; two independent functions allows designers to combine known hashes to evenly address all four slices

2. the scalar hash functions are computed by XORing certain bits of the memory address: this implementation has minimum area and power overhead with current lithography and incurs minimal latency, and shows very good evenness in practice

3. since our machine has 6GB of RAM, only bits 17 to 32 are used to compute the hash (as from fig. 3.4); the assumption of bit 17 as the lowest bit is in accordance with the literature, while the choice of bits 32 as highest bit is due to the amount of RAM

Although these assumptions are tailored to our specific CPU model, they can easily be generalized to any model having a number of cores that is a power of 2, as also previous works suggest. Instead, CPUs with a number of cores not being a power of 2 likely have non-linear hash functions, and are left as future work.

The reconstruction consists in finding which bits each function XORs. To this aim, [64] finds patterns from conflicting addresses that suggest which hash uses each bit, and then finds the hash functions by manually looking at these patterns. Here, we employ a more general approach that consists in using an Integer Linear Programming (ILP) model.

Collection of conflicting memory addresses To reconstruct h_2 and h_1, we collect memory addresses that collide with a given address a. Using the miss counter, the test accesses a (called *probe address*) first and then a sequence of addresses with distance $stride = 2^{17}B$: this stride ensures that the set the address is mapped to is the same of a, while the slice can vary because of the unknown hash. For example, if a is mapped to set 0 of slice 1, the accessed memory positions are all mapped to set 0 of an unknown slice. After traversing l memory location (with l increasing from 1), the test reads address a and checks whether the miss count has increased. If it is, the miss has been caused by the last address read, which is recorded as a *collider*: this location has evicted a, and has therefore the same hash. Otherwise, the test increases l and restarts reading the sequence of memory location, plus the new one at the end of the traversal. With this procedure, we collected many couples < probe address, colliders > in the form $< a, \{a_1, a_2, a_3, ..., a_n\} >$. Overall, we collected more than 2M colliding addresses distributed among 2K probe addresses.

Hash reconstruction as a clustering problem Thanks to assumption 1, the problem of reconstructing the hash can be reduced to a clustering problem. In the hash function h the bits are used in three possible *configurations*:

1. bits used only in h_1

2. bits used only in h_2

3. bits used in both h_1 and in h_2

For each configuration, we have to find a corresponding cluster of bits. To find bits in the same cluster, we look for colliders of the same probe having Hamming distance of 2. Since these colliders have the same hash, we infer that the two changing bits do not change the overall hash. Therefore, given the behavior of the XOR operator, these two bits must be used in the same way (only in h_1, only in h_2 or in both), and therefore belong to the same cluster (1, 2 or 3, respectively). For each couple of colliders at Hamming distance 2, we found the two different bits i and j, and counted how many times such couple appears across the whole dataset. This count represents the *similarity* $s_{i,j}$ of bits i and j: a high value of $s_{i,j}$ means that i and j are likely to belong to the same cluster, and so to be in the same configuration. Using only similarity, however, would make any clustering algorithm to trivially fit all bits in the same cluster, thus maximizing the total similarity.

Hence, we need another parameter to indicate when two bits should be in separate clusters. Like for similarity, we can estimate the *dissimilarity* $d_{i,j}$ of any two bits i and j to be in the same cluster. We can consider two couples probe-colliders $< a, \{a_1, a_2, a_3, ..., a_n\} >$ and $< b, \{b_1, b_2, b_3, ..., b_n\} >$ whose probes a and b have Hamming distance 1, thus with different hashes. If any two colliders a_k and b_l have Hamming distance 2, the two differing bits i and j must be in different clusters. Otherwise, they would cause the XOR chain of the hash to "flip" twice, resulting in the same hash (which is impossible since they have the same hash of their probes). The number of occurrences of each couple i, j is its dissimilarity $d_{i,j}$. Ideally, for any two bits i, j with $i \neq j$, it should hold that $s_{i,j} > 0$ if and only if $d_{i,j} = 0$. Nonetheless, the collected measures are affected by noise, which we attribute due to the hardware performance counters not being designed for reporting a single miss. For example, if the cache miss happens right after reading the collider a_k, the miss count might be updated with some unpredictable delay and the increased count be visible only after reading address a_{k+l}, which is mistakenly reported as a polluter. However, in our measurements we notice the similarity and dissimilarity values to be distributed in two groups of values: one group with high count values (thousands) and one with much lower counts. We assumed this last group to be due to measurement noise, and we applied a threshold to filter out these values. This is a key difference with respect to [70], which handles noise by filtering input data on the base of architecture-dependent latency values.

Using similarity and dissimilarity values, we can compute the best clustering by means of an ILP model. Introducing the binary variable $x_{i,j}$ that represents whether bits i and j are in the same cluster, we can write the objective function as in eq. (3.1), maximizing the

intra-cluster similarity and the inter-cluster dissimilarity.

$$\text{maximize} \sum_{i=17}^{31} \sum_{j=i+1}^{32} [x_{i,j} \times s_{i,j} + (1 - x_{i,j}) \times d_{i,j}] \tag{3.1}$$

For the clustering to be meaningful, we also have to force the transitivity property of clustering: if bits i is in the same cluster of bit j and j is in the same cluster of k, then bits i and k must also be in the same cluster. Therefore, we add the constraints in Equation (3.2).

$$\begin{aligned} &\forall i, j, k \mid 17 \leq i < j < k \leq 32 : \\ &x_{i,k} \geq x_{i,j} + x_{j,k} - 1, \qquad x_{i,j} \geq x_{i,k} + x_{j,k} - 1, \qquad x_{j,k} \geq x_{i,j} + x_{i,k} - 1 \end{aligned} \tag{3.2}$$

Using an ILP solver, we found as optimal solution the following three clusters of bits

$$\begin{aligned} c_1 &= 18, 25, 27, 30, 32 \\ c_2 &= 17, 20, 22, 24, 26, 28 \\ c_3 &= 19, 21, 23, 29, 31 \end{aligned}$$

These clusters are the same as those in [64] although the multi-core is different.

Configuration choice via LLC access latencies The next step is determining to which *configuration* a cluster corresponds to: for example, c_1 can correspond to h_1 (configuration 1), c_2 to h_2 and c_3 be the configuration of common bits; or any other combination, each one corresponding to a different hash function. There are, hence, six possible hashes. To find the one of our multi-core, we leveraged the different access latencies from cores to slice: if a core (numbered from 0 to 3) accesses the slice of its same number, the latency must be minimal, because each core is directly connected to its slice and does not pass through the ring bus. To obtain this measurement, we wrote a simple test that accesses sequential memory regions and measures the access latency. This measurement is possible by using a specific performance counter available on our multi-core that measures the memory latency in case of L2 miss. Throughout the measurements, the minimum value (that appeared consistently) was 18 cycles, which we assumed as the one indicating access to the local LLC slice. Finally, we repeated the test on each core i with all the possible hashes, and stored, for each core, the hashes that return i when the access latency is 18 cycles. At the end, only the hash function in fig. 3.6 makes correct predictions for all the cores, and is assumed to be the real hash function.

Results with another multi-core To validate our reconstruction methodology, we applied it to a machine with an 8-cores Intel Xeon E5 E5-2690 with 256GB of RAM. After collecting roughly 2.5M memory addresses, from the clustering phase we obtained the following 7

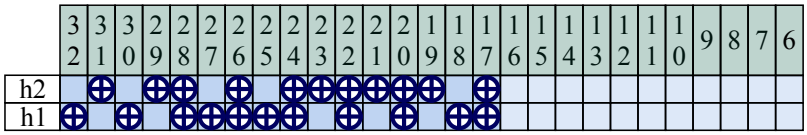

Figure 3.6: *Hash function of Intel Xeon E5 1410*

clusters:

$$c_1 = \{24, 17, 20, 28, 33\},\ c_2 = \{31, 34, 19, 37, 23\},\ c_3 = \{32, 25, 18\},\ c_4 = \{26, 35, 22\},$$
$$c_5 = \{27, 36, 30\},\ c_6 = \{21\},\ c_7 = \{29\}$$

which correspond to the 7 possible cases (shared among all hashes, shared between any two or exclusive). With these 7 cases, we should theoretically test the predictions for all cores of $7! = 5040$ possible hashes, corresponding to the permutations of the clusters. However, we may observe that the clusters have different sizes, and the bigger clusters are unlikely to be shared by all of the hashes. Indeed, this would limit the "entropy" among hashes, potentially causing access bottlenecks to few slices. On the opposite side, c_1 and c_2 are likely to be shared and not reserved for single hashes, since this would cause the one hash to have different entropy from the two using c_1 and c_2. Hence, these clusters are likely to be shared by couples. These assumptions limit the number of candidate hashes to 432 (3 candidates for the cluster shared by all cores, 4 candidates for the cluster shared by couples and 3 candidates for the reserved clusters), which makes the approach feasible in a reasonable time. For CPUs with a higher number of cores, this approach can be very time consuming, as it scales with the number of permutations of the clusters. However, future many-core CPUs will probably not rely on a ring-based architecture, and will have a completely different structure: therefore, we consider scalability issues to remain limited to a reasonable scale. [1] After testing, c_5 emerges as the shared cluster, and the three hash functions of the processor are

$$h_1 = c_5 \oplus c_1 \oplus c_3 \oplus c_7,\ h_2 = c_5 \oplus c_1 \oplus c_2 \oplus c_6,\ h_3 = c_5 \oplus c_2 \oplus c_3 \oplus c_4$$

where we use the $\oplus$ operator between clusters to indicate the XOR of all bits of the clusters.

3.4.2 Definition of color and structure of the color-aware Buddy allocator

With the knowledge of the hash function, we can redefine the color of a page by padding the two original color bits with the two hash bits, so that the color of a page with address

[1] It is possible to use branch and bound or backtracking strategies to find the final hash, constraining the research space over and over with missing addresses whose mapping slice is known (e.g. by checking the latency). However, this optimization is out of the scope of this publication, and is left for future work.

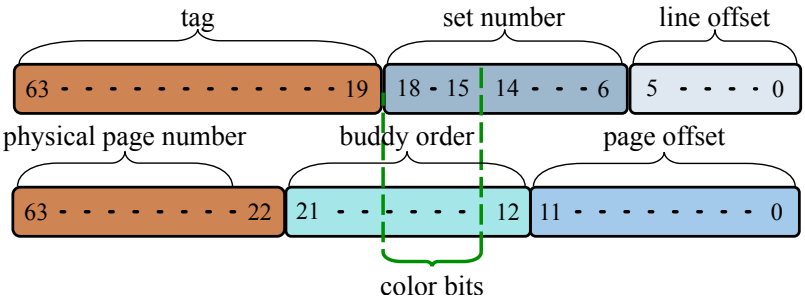

Figure 3.7: *Bits of buddy addresses forced to 0 and overlapping with color bits.*

$a = a_{63}...a_0$ becomes $c(a) = h_2(a)h_1(a)a_{16}a_{15}$. Due to this redefinition, the number of available LLC partitions is 16, a sufficient granularity. To make the lookup for a page of a specific color constant, we split the list of pages used by the Buddy algorithm into 16 sub-lists, one per color: in this way, the algorithm can select a sub-list in constant time, and remove the first page from the list (still in constant time). However, the notion of color is defined only for pages, while the Buddy algorithm manages physical memory in chunks, called *buddies*, of different sizes.

Indeed, each buddy is characterized by two parameters: the order, typically from 0 to 10, indicates its size, so that a buddy of order i is composed of 2^i physically contiguous pages; the address of the buddy is the address of the first memory page of the buddy, and is always aligned to the size of the buddy (its lower i bits, from bit 12 to bit $12 + i - 1$, are 0). In case no pages of a desired color are present, the algorithm should split a buddy of order 1 and check which page has the desired color. If, for example, the desired color is $c = 0011_2$, the algorithm should split a 1-order buddy, whose bit 12 is forced to 0 due to the memory alignment of buddies. Therefore, the color bits are not affected by the alignment, and the algorithm should look for a 1-order buddy of color c. To make this search constant, also the list of 1-order buddies should be split into 16 sub-lists, as for the pages. As from fig. 3.7, this holds until order 3, as there is no overlap between color bits and bits forced to 0 in the buddy address. In case the algorithm has to split a buddy of order 4, bit 15 is forced to 0, and the color of the buddy to look for is 0010_2: then, the algorithm should return the second half, whose bit 15 is 1. In this case, 8 configurations of the color bits are possible, since bit 15 is 0; thus, the list of order 4 should be split into 8 sub-lists. For higher colors, the available configurations of color bits further decrease by a factor of 2: 4 configurations for order 5, 2 for order 6 and 1 for orders 7 and higher.

To deal with the buddy lookup in a unique way, we generalize the notion of color to buddies of any order through the definition of the *mcolor*. For our testbed multi-core, the

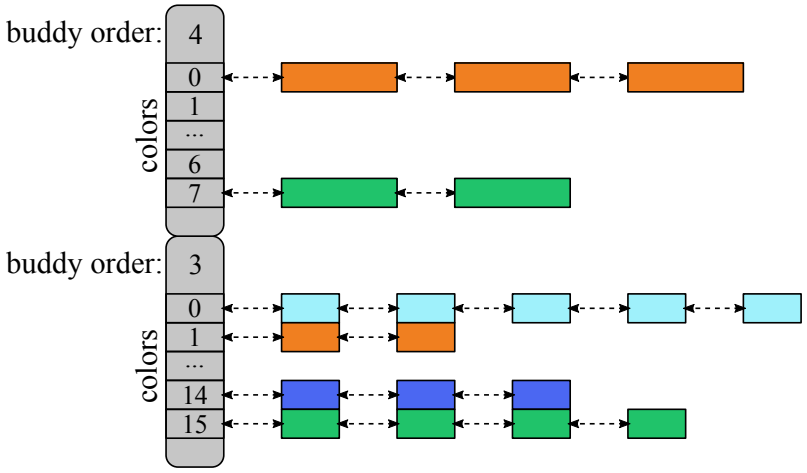

Figure 3.8: *Data structure of the color-aware Buddy algorithm*

mcolor can be defined, in base 2, as

$$mcolor(a,i) = \begin{cases} h_2(a)h_1(a)a_{16}a_{15} & \text{if } 0 \leq i \leq 3 \\ h_2(a)h_1(a)a_{16} & \text{if } i = 4 \\ h_2(a)h_1(a) & \text{if } i = 5 \\ (h_2(a)\ XOR\ h_1(a)) & \text{if } i = 6 \\ 0 & \text{otherwise} \end{cases}$$

This definition models our previous example and considers only the bits that can vary at each order, so that it can be used to query in which sub-list to look in at each order. It allows to compute the mcolor of each buddy, in order to insert it to the proper sub-list when it is freed. In particular, order 6 represents a special case, as it defines the *mcolor* as the negated equality between h_1 and h_2. This definition is due to the role of bit 17, which is forced to 0 in buddies of order 6. Figure 3.6 shows that this bit is used in both hash functions h_1 and h_2: therefore, if the hashes are equal, also the hashes of the two 5-order sub-buddies will be equal, since bit 17 either flips both the hashes (if it is 1) or none (if it is 0). Similarly, if the hashes are different, also the hashes of the sub-buddies will be different. If, for example, the desired color is 1010_2, it is possible to select either a 6-order buddy with hash 10_2 or one with hash 01_2 (both having different hashes): after the split, the algorithm will choose in the former case the first sub-buddy (whose bit 17 is 0, hence not flipping) and in the latter case the second sub-buddy, whose bit 17 being 1 will cause both hashes to flip. Finally, for orders greater than 6, the buddy contains all the possible colors, which map to 0.

Since the Buddy allocator receives requests of pages of specific colors, we need to efficiently compute the mcolor from the color in order to perform a fast lookup. Similarly

to the definition of the *mcolor*, we can map the requested color $c = c_3c_2c_1c_0$ to the *mcolor* of a given order i by means of the following function:

$$mcolor\_lookup(c, i) = \begin{cases} c_3c_2c_1c_0 & \text{if } 0 \leq i \leq 3 \\ c_3c_2c_1 & \text{if } i = 4 \\ c_3c_2 & \text{if } i = 5 \\ (c_3 \; XOR \; c_2) & \text{if } i = 6 \\ 0 & \text{otherwise} \end{cases}$$

With these definitions, it is possible to design a color-aware allocation algorithm based on the Buddy allocator that avoids lookup, maintaining efficiency and scalability.

Figure 3.8 sketches the structure of the modified Buddy algorithm. In place of a single list of buddies for each order, we modified the data structure at the base of the Buddy algorithm to have one list per *mcolor* for each order. Thus, at orders 0 to 3 there are 16 lists, order 4 has 8 lists, order 5 has 4, order 6 has 2 and orders from 7 on have a single list. Using one list per *mcolor* allows the algorithm to select a buddy of the desired *mcolor* in constant time. Consequently, also the procedures to select a colored page and to insert one are modified. Algorithm 3.1 shows the procedure to select a page of a desired color: if the page is present in the color list, it is returned immediately, otherwise a buddy of higher order is split. The procedure *SplitBuddy*, indeed, computes the target *mcolor* (line 9), looks for the buddy of correct *mcolor* eventually splitting a higher order buddy via a recursive call (lines 10-13), splits the buddy in two halves (line 15) and checks which half has the target *mcolor*, adding the other free half to the list of free buddies (16-21).

Listing 3.1 *Split and select procedures*

```
1  globaldata: list_head buddies[MAX_ORDER][MAX_COLORS]
2
3  procedure SplitBuddy(order ord, mcolor mcol): buddy
4     buddy first, second, current
5     mcolor firstc, secondc, targetc
6
7     if ord == MAX_ORDER return nil
8
9     targetc = McolorLookup(mcol, ord)
10    if ListIsEmpty(buddies[ord][targetc])
11       current = SplitBuddy(ord+1, mcol)
12       if current == nil return nil
13       else current = RemoveHead(buddies[ord][targetc])
14
15    first,second = ComputeHalves(current)
16    first_color = Mcolor(first, ord)
17    second_color = Mcolor(second, ord)
18    if firstc != targetc
19       Swap(first, second)
20       Swap(first_color, second_color)
21    InsertHead(buddies[ord-1][second_color], second)
22    return first
23 end procedure
24
25 procedure SelectPage(color col): buddy
```

```
26     if ListIsEmpty(buddies[0][col])
27        return SplitBuddy(1, mcol)
28     else
29        return RemoveHead(buddies[0][col])
30  end procedure
```

Similarly, the procedure to insert a freed buddy (algorithm 3.2) is based on the definition of *mcolor*, which is computed according to the definition (line 7). Then the insertion algorithm checks whether the "twin" buddy is also free: in case it is, it coalesces them in a single buddy of higher order and recursively calls the insertion procedure (lines 11 - 13); otherwise, it inserts the free buddy into the proper list (line 15).

Listing 3.2 *Insertion procedure*

```
1  globaldata: list_head buddies[MAX_ORDER][MAX_COLORS]
2
3  procedure InsertBuddy(buddy b, order ord)
4     buddy twin
5     mcolor mcol
6
7     mcol = Mcolor(b, ord)
8     twin = GetTwinBuddy(b, ord)
9     if ord < MAX_ORDER-1 AND BuddyIsFree(twin)
10       RemoveFromList(buddies[ord][Mcolor(twin, ord)])
11       b = CoalesceBuddy(b, twin, ord)
12       InsertBuddy(b, ord+1)
13    else
14       InsertHead(buddies[ord][mcol], b)
15 end procedure
```

Thanks to the definition of *mcolor* and, consequently, of the function *mcolor_lookup*, the splitting and insertion procedures do not perform any lookup to find a requested color, but execute in constant time independently from the number of buddies and, ultimately, from the size of the physical memory, maintaining the scalability and efficiency of the original algorithm.

The proposed allocator has been implemented in Linux 3.17, modifying the existing implementation of the Buddy allocator, which requires also considering the hierarchical memory distribution consisting of memory nodes and *zones*[2], as well as other heuristics to control memory fragmentation. For the purpose of a realistic implementation, we integrated our design into the existing codebase, modifying the routines that manage each zone. On top of those routines, the algorithms that select the node and the zone work as usual. To provide users with a suitable interface, a new *cgroup* [107] has been implemented to expose the LLC partitioning capabilities. This interface allows users to manually create

[2]in Linux terminology, a node corresponds to a physical memory node, while a zone is a partition of a node from which kernel subsystems allocate preferably: for example, the first B are the *Direct Memory Accesses (DMA) zone*, since old DMA controllers handling physical address of 24 bits can manage buffers only in this zone.

Table 3.1: *Selected SPEC CPU2006 tests*

Test	Input
libquantum	control
gcc	g23
omnetpp	omnetpp
leslie3d	leslie3d
xalancbmk	t5
sphinx	ctlfile
astar	rivers
bzip2	text

LLC partitions by specifying the colors of each partition and the applications that use the partition.

3.5 Experimental results

This section discusses the experimental setup to evaluate the proposed solution in section 3.5.1 and presents a first evaluation in section 3.5.2, showing how it is possible to control the LLC usage of stand-alone single core applications by means of LLC partitioning. Discussing the behavior of the selected applications, the section also devises application mixes to run in co-location, which are evaluated in section 3.5.2. Finally, section 3.5.4 evaluates the LLC partitioning of co-located multi-threaded benchmarks.

3.5.1 Methodology and testbed

To evaluate the effectiveness of the proposed solution, we selected 8 benchmarks from the SPEC CPU2006 suite [61]. These benchmarks have been selected as they have diverse profiles with respect to LLC usage and are mostly sensitive to the LLC size. They are also include many representative of DA applications, like text processing, speech recognition, and document parsing. Table 3.1 shows the selected applications along with their input sets, which are those that cause the longest runs. In a first phase, each application is profiled stand-alone, and the number of colors is varied from 1 to 16. Section 3.5.2 shows the applications profiles, describing its behavior with LLC partitions of any size. For LLC partitioning to be beneficial, these applications should ideally have the same profiles in co-location with others. Indeed, section 3.5.3 defines several workloads, where a *target* application is pinned to a core and isolated in LLC and 3 other applications are co-located on the other cores and contend for the rest of the LLC, acting as *polluters* to the application that benefits from isolation in LLC. Also with these setup, the target applications are profiled with several partition sizes. The aim of a variable partition size is to allow the final user to select the size that guarantees the requested QoS, which can be expressed, in

the case of the SPEC benchmarks, as the maximum tolerated slowdown with respect to the stand-alone execution.

The testbed machine has the following characteristics:

- Intel Xeon E5 1410, with 4 cores running at 2.80 GHz
- 6GB RAM DDR3 at 1333 MHz
- L1 instruction and data caches of 32KB each
- L2 unified cache of 256KB
- LLC of 10MB, composed of four slices
- Page size of 4KB
- 16 colors

To evaluate the benefits of LLC partitioning, we disabled advanced features that affect the LLC performance such as TurboBoost, the prefetchers and the power saving features of the kernel.

3.5.2 Application profiles

Figure 3.9 shows the profiles of the applications: the horizontal axis reports the size of the LLC partition (each color corresponds to 0.625MB of LLC), while the vertical axis reports the percentage of slowdown with respect to the execution with 10MB of LLC (red lines) and the miss rate (blue lines). The plots for gcc and bzip2 start from 1.88MB of LLC, due to their memory requirements: since partitioning the LLC limits the available physical memory (due to assigning less colors), the input sets of these two applications do not fit in less than 800MB, corresponding to a 1.88MB LLC partition.

Figure 3.9 also shows the behavior of applications with a "classical" page-coloring scheme with dotted lines: in this scenario, the partitioning mechanism is unaware of hashing ("w/o hash" in the legend) and uses bits 15-18 for partitioning, as in fig. 3.3 (where we don't use L2 partitioning bits). Figure 3.9 shows the effectiveness of our solution in controlling the usage of the LLC, highlighting also which applications are more sensitive to the amount of cache. The bars, which represent the standard error of the mean, in most of the plots are barely visible, showing that applications with a regular memory access pattern have a predictable behavior with varying LLC partition. In contrast, sphinx shows considerable variability, which is due to its unfriendly access pattern (caused by a search heuristic) and to the application itself repeatedly loading inputs (small speech samples). The figure shows that hash-unaware partitioning always performs better than hash-aware partitioning: in some cases, the hash-unaware miss rate at 1.25MB is comparable to the

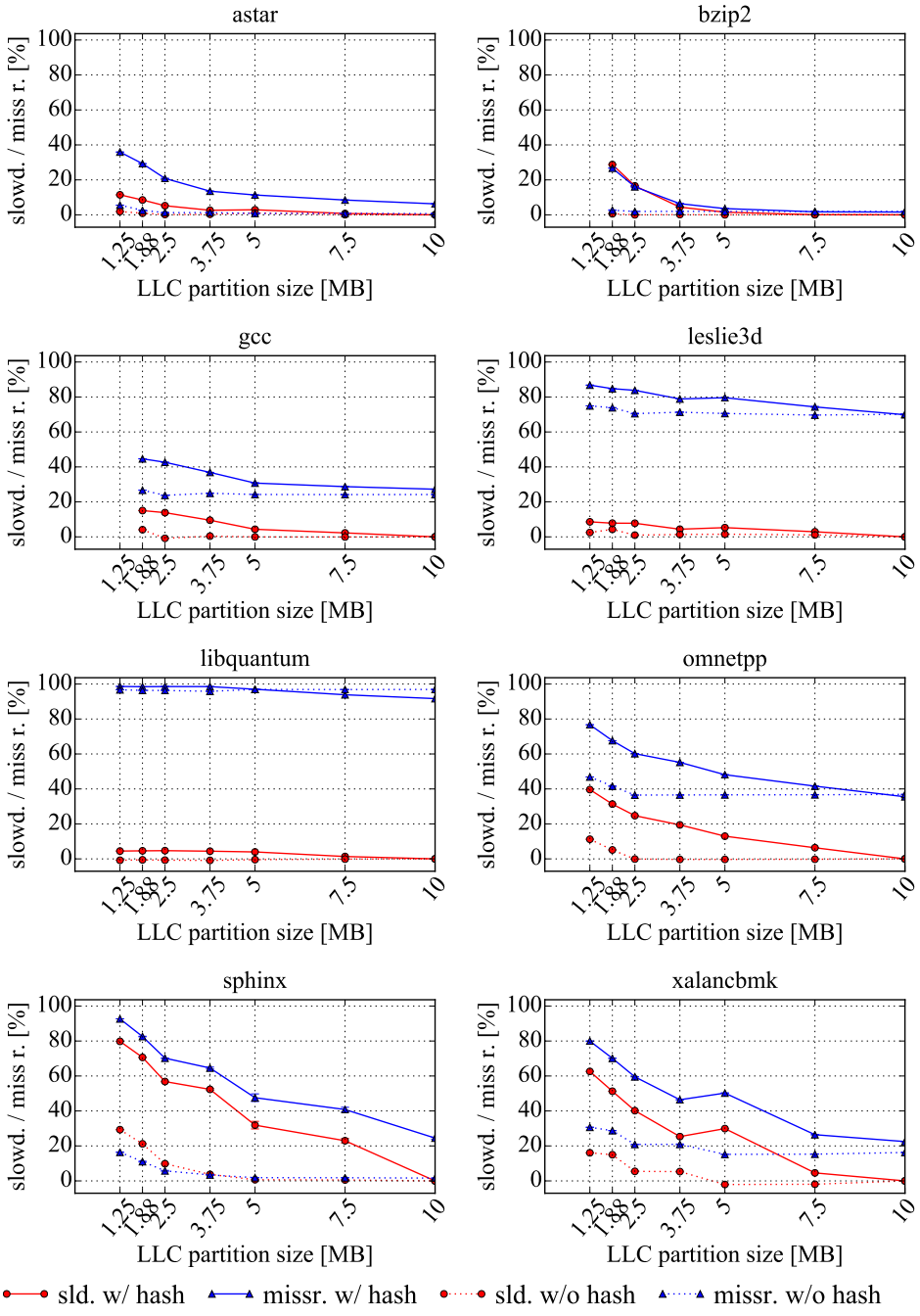

Figure 3.9: *Applications profiles with different cache partitions, with slowdown ("sld") and miss rate ("missr") hash-aware partitioning ("w/ hash") and hash-unaware partitioning ("w/o hash")*

Table 3.2: *Classification of applications*

Classification	Applications
sensitive	bzip2, omnetpp, xalancbmk, sphinx
insensitive	libquantum, leslie, astar, gcc

Table 3.3: *Test workloads*

Workload	Target	Polluters
W1	bzip2	xalancbmk, leslie3d, gcc
W2	omnetpp	sphinx, libquantum, astar
W3	xalancbmk	omnetpp, libquantum, gcc
W4	sphinx	bzip2, leslie3d, astar

hash-aware miss rate at 5MB or more. This phenomenon is due to the underlying functioning of the LLC: although hash-unaware partitioning uses bits 15-18, only bits 15-16 are effective. Indeed, bits 17-18 are used in the hash, which varies depending also on the higher bits that change from page to page. Therefore, when the allocator chooses a page only on the base of bits 15-18, its hash is uniformly distributed on the whole range, resulting in more LLC space allocated than the space requested. For example, setting only color 0 keeps bits 15-16 to 0 while the slice varies, resulting in the allocation of a quarter of *each* slice. Hence, hash-unaware curves in Figure 3.9 have larger plateaus than hash-aware curves. Instead, based on hash-aware profiles, applications have been classified according to their sensitivity to the partition size. Applications whose slowdown with the least amount of cache is equal or greater than 30% are defined to be *sensitive*, while the others are *insensitive*. Table 3.2 shows how the eight reference benchmarks are classified.

3.5.3 Co-location profiles

To evaluate isolation capabilities in co-location, we devised one workload for each sensitive application, called *target* application, which runs co-located with three other applications chosen randomly, called *polluters*. To have a diverse mix, the first polluter is chosen from the sensitive applications while the other two polluters are insensitive applications. Table 3.3 shows the four workloads, with the target and the polluters. Throughout all the tests we run the entire application with the biggest input, and we immediately restart the polluters whenever they terminate.

Figure 3.10 presents the results from the workload runs: the continuous lines show the slowdown (red) and the miss rate (blue) of the target application with hash-aware partitioning, while the dotted lines show slowdown and miss rate with hash-unaware par-

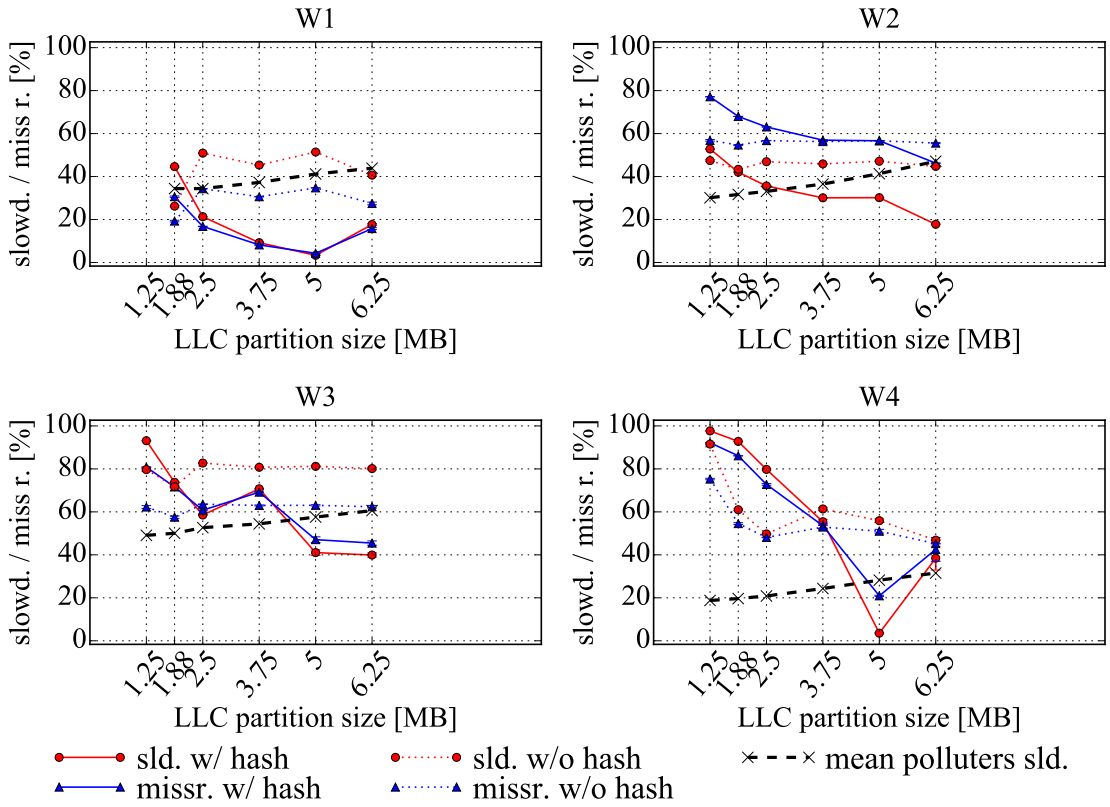

Figure 3.10: *Profiles of the workloads in Table 3.3, with different cache partitions: continuous lines refer to the target with hash-aware partitioning, dotted lines refer to the target with hash-unaware partitioning and dashed lines refer to the polluters*

Table 3.4: *Test workloads with limited I/O activity*

Workload	Target	Polluters
X1	bzip2	xalancbmk, leslie3d, *astar*
X2	omnetpp	sphinx, libquantum, *xalancbmk*
X3	xalancbmk	omnetpp, libquantum, *leslie3d*
X4	sphinx	bzip2, leslie3d, *sphinx*

titioning. Furthermore, the dashed black lines show the harmonic means of the polluters' slowdowns with hash-aware partitioning. To avoid the polluters from swapping to disk, the target applications receive at most 3.75GB of RAM, and the three polluters share the remaining 2.25GB: because of the memory partition effect, 3.75GB of RAM corresponds to the 6.25MB of LLC. The profiles show that hash-aware partitioning is more effective in controlling the performance for a varying LLC partition, while with hash-unaware partitioning the targets suffer from higher contention in the LLC (as from the miss rate line). Hash-unaware partitioning is more effective in the case with 1.88MB of LLC, since the LLC space devoted to the target is higher than 1.88MB due to bits 17-18 not limiting the LLC space. Instead, with higher amounts of LLC bits 15-16 can assume any value, so that any set within any slice is possible, and this mechanism is ineffective.

Comparing fig. 3.10 to fig. 3.9, the targets show a less regular behavior with respect to the stand-alone profiles. Instead, the polluters have, on average, less variations, since two of them are cache-insensitive, but exercise a noticeable pressure on the LLC, affecting the target performance. In particular, W1 has a small LLC footprint, and partitioning is able to keep the application's data in the LLC. W2, instead, has a larger memory footprint and a cache-friendly access pattern, and benefits from having large LLC space. W3 and W4 have irregularities that are due to the I/O activity of the polluters. Since the targets (xalancbmk and sphinx) have higher duration than the polluters, which are immediately restarted, multiple I/O bursts occur during the execution of the target. During these bursts, the kernel I/O subsystem employs page of any available color (kernel colors are always unrestricted to prevent hotspots in the kernel execution), mapping buffers to the colors allocated to the target. This phenomenon, exacerbated by the limited availability of memory, is well visible in W4, whose target (sphinx), has the longest execution time. Therefore, we devised 4 other workloads, reported in table 3.4, whose applications have similar execution times. These workloads derive from those in table 3.3 by replacing the polluter having the least duration with the highlighted one. For example, in the case of W4, all the polluters have very different duration from the target sphinx, and the only possible replacement was another instance of sphinx in lieu of astar. Figure 3.11 plots the resulting profiles similarly to fig. 3.10 (continuous lines for the target with hash-aware partitioning, dotted lines for the target with hash-unaware partitioning, dashed black lines for the polluters' slowdown

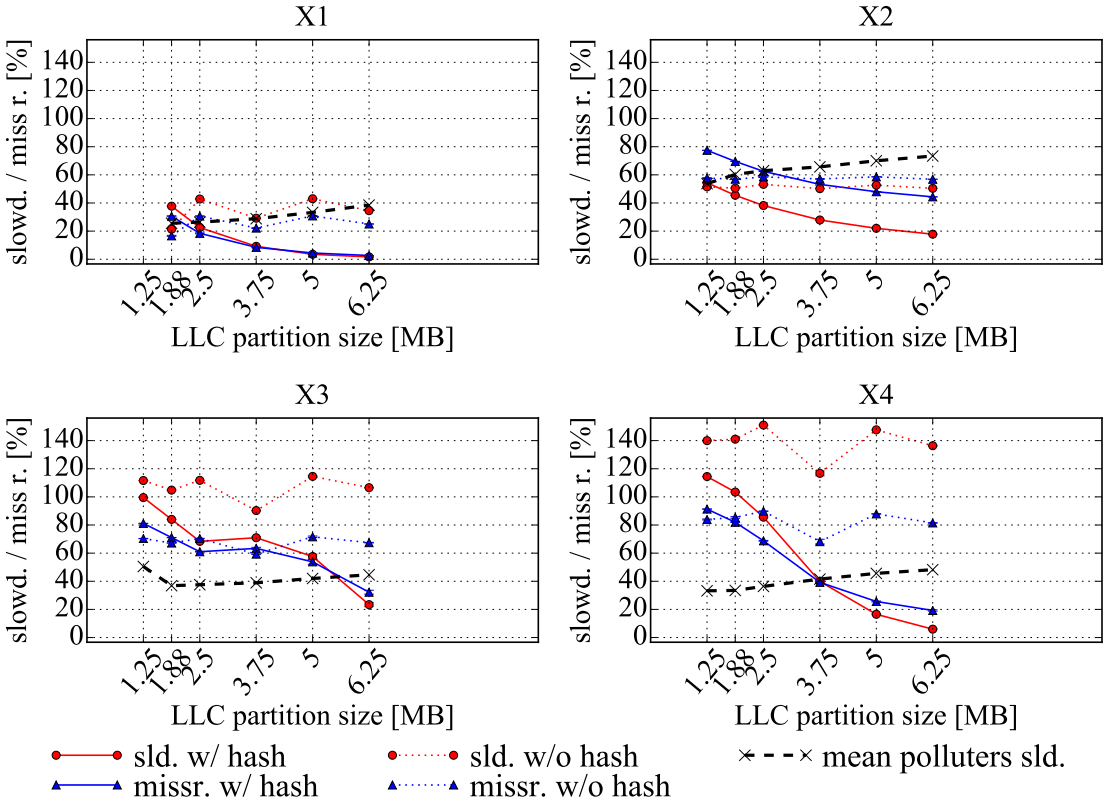

Figure 3.11: *Profiles of the workloads in Table 3.4, with different cache partitions*

Table 3.5: *PARSEC co-located workloads*

Workload	Target	Polluter
P1	bodytrack	swaptions
P2	ferret	facesim
P3	freqmine	raytrace
P4	vips	blackscholes

with hash-aware partitioning), showing indeed more regular curves that are closer to the stand-alone profiles.

3.5.4 Multi-threaded co-location profiles

To further evaluate the effectiveness of our hash-aware partitioning scheme, we perform a similar evaluation with multi-threaded applications from the PARSEC 3.0 suite [20]. For this evaluation, we co-locate two applications from the PARSEC suite: for each pair, the first application is selected as target, while the second acts as polluter and contends the LLC to the target. Both applications perform complete runs and have 4 threads each to maximize contentiousness (with standard OS time-sharing of CPU cores), and are given the PARSEC native input; to deal with different execution times, the polluter application is immediately restarted as soon as it terminates. The application pairs, named from P1 to P4, are shown in table 3.5. Figure 3.12 shows the profiles of the co-located pairs, where the dotted lines represent the stand-alone execution (not present in fig. 3.9) and the continuous lines the execution in co-location. Here, while the miss rate values in co-location are close to those stand-alone, the slowdown is significantly impacted by contention on computational bandwidth and on memory access.

3.6 Conclusions and future work

This chapter proposed a technique for LLC partitioning based on page coloring that is able to work also on modern hash-mapped caches and was recognized as state-of-the-art in the scientific literature [146]. The proposed design aims at maintaining the scalability and efficiency of the Linux Buddy allocator, while allowing the selection of a memory pages of any given color. Our approach is based on the knowledge of the LLC hash function, which is reconstructed by means of widely available performance counters. The technique presented to reconstruct this information is based on assumptions that are reasonably valid across multiple architectures. Therefore, the validation of our approach to more architectures (lake the most recent Haswell and Skylake) is a possible future work, as well as the evaluation of the allocator design on a broader range of configurations in order to test its efficiency and scalability capabilities.

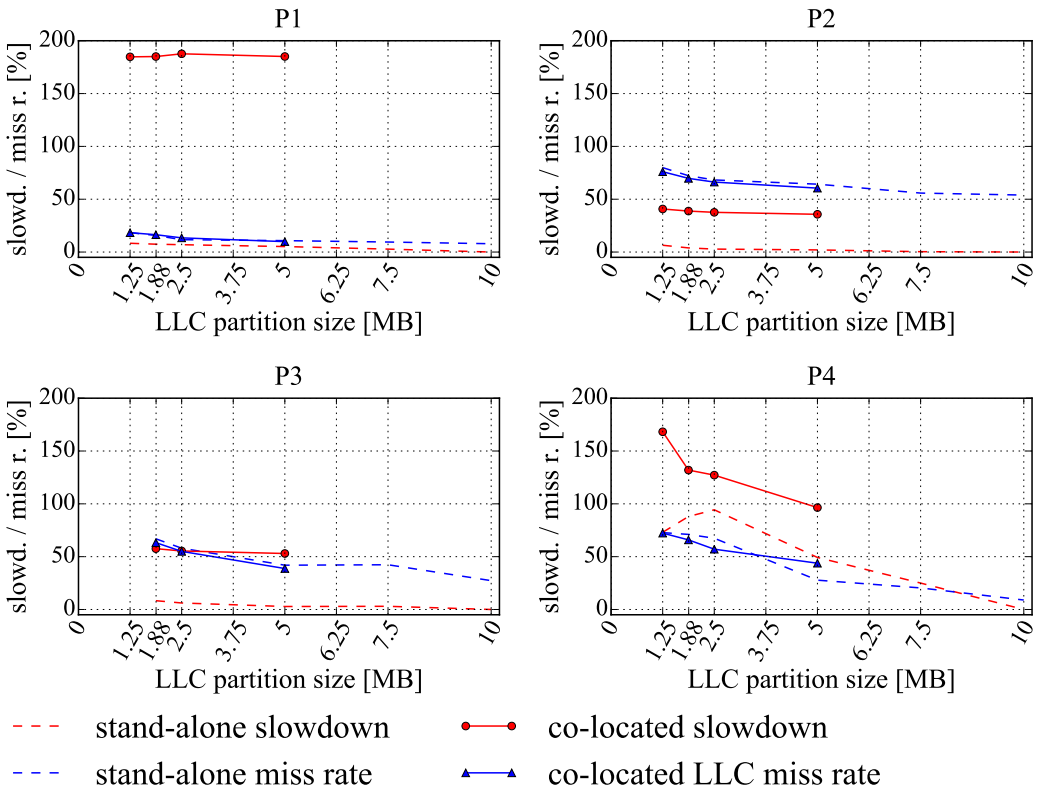

Figure 3.12: *Profiles of the workloads in Table 3.5, with different LLC partitions*

In the testbed environment, this technique was effective in controlling the usage of the LLC of selected applications running, even if some limitations emerged. However, while this technique is hindered by pollution due to OS buffers, orthogonal research [153] provides solution that can be integrated to mitigate this issue.

As a completion of this work, a policy that drives our partitioning technique is the most natural extension. This policy can, for example, also take in account the on-chip traffic among slices and further enforce performance isolation, possibly using re-coloring policies to adapt to a dynamic workload.

Finally, this solution needs a precise indication of the memory usage of the application. Common applications and development environments do not take this aspect under thorough control (e.g. many managed languages like Java automatically allocate large memory pools, which are increased and shrinked at runtime out of user's control), and tend to largely overbook capacity when pre-provisioning is necessary, as in PaaS or Infrastructure-as-a-Service (IaaS) environments. These limitations can be overcome in FaaS environments, where the high-level of task description allows the underlying implementations to be carefully designed to control and predict the memory capacity used, thus giving more reliable insights to achieve effective isolation.

CHAPTER 4

Investigation of Machine Learning workloads

Among newer fields, ML is playing an increasingly important role in industry and in research, which extensively focused on training accurate ML models. Once trained, these models are deployed for inference, where they usually run in commodity servers equipped with recent hardware, which may more and more often comprise accelerators based on FPGAs in addition to multi-core CPUs. Like DA applications are often structured as Embed, Transform, Load (ETL) pipelines, ML models are often composed by sequences of transformations, also structured in pipelines or in a DAG. While this design makes it easy to decompose and accelerate single model components at training time, predictions requires low latency and high performance predictability whereby end-to-end runtime optimizations and acceleration is needed to meet such goals. These optimizations have to take in account the characteristics of modern hardware to meet such requirements, while at the same time using resources efficiently as needed for applications to scale and be cost-effective. This chapter discusses the problem starting from a production-like model, and showing how a redesign of the model pipelines for efficient execution over CPUs and FPGAs can achieve performance improvements of several folds. Section 4.1 discusses the problem in more detail and section 4.2 overviews the related works. Section 4.3 explains the use case this work starts from and its characteristics, while section 4.4 shows the implementation of this use case for both CPU and FPGA, evaluated in section 4.5. Finally,

section 4.6 draws the final conclusions and highlights the key insights.

4.1 Introduction

While prior research focused on accelerating specific ML kernels [79, 94] or predictions for complex neural networks models [110], to our knowledge no research exists on the accelerating general ML prediction pipelines. Most of the input pre-processing, which is often a computationally demanding phase, still occurs on standard CPUs, as it is consists of very variegated steps. Therefore, to our knowledge there currently exists no end-to-end acceleration approach for generic pipelines, neither for CPU nor for heterogeneous devices like FPGAs, despite some of these initial steps (for example string tokenization and hashing) have characteristics that make them amenable for acceleration. In particular, the issues already known from DA applications and highlighted in section 2.1.2, such as data materialization in intermediate steps, operations fusion for better optimizations, etc., are still unsolved in most branches of ML.

Training and prediction pipelines generally have different system characteristics, where the latter are surfaced for direct users access and therefore require low latency, high throughput and high predictability. Accelerating predictions often requires redesigning these pipelines, either to allow CPU compilers better optimize the code or, with specialized hardware such as FPGA, to efficiently utilize the accelerator's capabilities. This work sheds some light on the problem, showing how some common operations of prediction pipelines can benefit from redesign for both CPU optimizations and hardware acceleration. To compose the ML pipelines, we will use an IMLT used in Microsoft for research purposes, which is mainly written in C#. Although unreleased, this toolkit has tested, production-quality performance and has an API that is similar in spirit to that of the main ML frameworks.

Like increasingly many of the off-the-shelf ML libraries, IMLT expresses ML pipelines as a DAG of *transformations*, which are mostly implemented in C# inside the toolkit itself. Figure 4.1 shows an example of a model pipeline used for sentiment analysis (a more detailed description of each single transformation composing the pipeline will be introduced in section 4.3). IMLT employs a pull-based execution model that lazily materializes input vectors, and tries to reuse existing data buffers for intermediate transformations. This largely decreases the memory footprint and the pressure on garbage collection, and avoids relying on external dependencies (like NumPY [120] for Scikit) for efficient memory management. Conversely, this design forces memory allocation along the data path, therefore making model scoring time hard to predict. This in turn results in difficulties in providing meaningful Service Level Agreements (SLAs) by MLaaS providers.

Starting from models authored using IMLT, we decouple the high-level user-facing API of the tool from the physical (execution) plane, with the goal of keeping data materialization minimal. This decoupling allows us to:

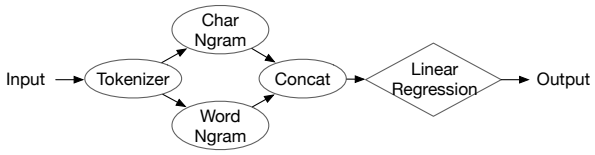

Figure 4.1: *A model pipeline for sentiment analysis.*

1. optimize how transformations are executed, for instance by compiling several transformations together into one efficient execution unit;

2. distribute different parts of the computation to heterogeneous devices such as CPU and FPGA (and, in the future, possibly to different machines) while optimizing the execution for each target device.

Using a production-like model we show that, compared to IMLT, our design improves the latency by several orders of magnitude on CPU and that generic ML prediction pipelines can also benefit from acceleration via FPGA. Interestingly, we find that a tuned CPU implementation outperformed the FPGA implementation; to fully exploit hardware acceleration, a redesign of prediction pipelines is not enough inasmuch as more hardware-friendly operations are not used at training, with proper redesign at the system level to adapt to FPGA characteristics.

4.2 Related Work

Two systems for managing ML prediction-serving have been introduced recently in the academia and the industry: Clipper [40] and TF Serving [160]. Clipper targets high performance online ML prediction while making model deployment easy. Clipper does not consider models as composed by complex DAG of transformations, but instead runs each pipeline as a single functional call in a separate container process and routes prediction requests via Remote Procedure Call (RPC) to the appropriate container. Users can deploy models learned by different frameworks, but this flexibility comes at the cost of losing the control over the execution inside the pipeline which instead relies on the target framework to run the model. Clipper's optimizations thereby focus on models as black boxes: Clipper caches results for popular queries and controls the batch size adaptively to achieve high throughput while achieving low latency SLA. Hence, those optimizations lose the chance to utilize hardware acceleration.

TensorFlow (TF) Serving is a library for serving ML models in TF framework. TF Serving supports models that are trained by TF and users can also define custom models or operations as *Servable*, which is the unit of scheduling and lifecycle management. TF Serving batches multiple prediction requests as Clipper, but the execution of pipelines is more flexible, allowing users to define custom *Servable* for the part of pipelines. While

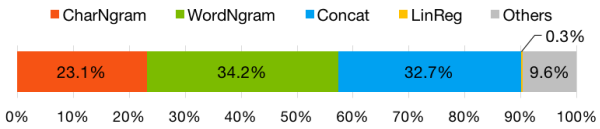

Figure 4.2: *Execution breakdown of the example model*

*Servable*s enable the use of hardware accelerators and make execution faster, there is no framework-level support for fine-grained control over the pipeline execution, as we aim instead.

4.3 Case Study for Accelerating Prediction Pipelines

To make our claims more concrete, throughout this work we will use a Sentiment Analysis (SA) prediction model as a motivating example, which starts from raw sentences and predicts a classification label for each sentence. The model approximately works as follow: the input sentences (strings) are tokenized into words and chars after an initial normalization step. Then two feature vectors are generated as bag-of-words composed by the n-ngrams extracted from word and chars, respectively. The two vectors are then normalized and concatenated into a unique feature vector which is then run through a simple linear regression model. Figure 4.1 contains the DAG of transformations composing the sentiment analysis example. It contains several common transformations, namely:

1. text normalization, like de-capitalization
2. tokenization: the text is split in words and in characters
3. n-gram extraction by grouping words, or characters, together
4. bag-of-words, to featurize the ngrams
5. concatenation of the features from words and the characters
6. normalization
7. linear regression prediction using the features from the words and the characters

Figure 4.2 shows the execution breakdown of this model when scored in IMLT, where the prediction (the LinReg transformation) takes a very small amount of time compared to other components (0.3% of the execution time). Instead, the n-gram transformations take up almost 60% of the execution time, while another 33% is spent in concatenating the feature vectors, which essentially consists in allocating memory and copying data. In summary, most of the execution time is lost in preparing the data for scoring, not in computing the prediction according to the regression ML operator ‘Ã t the heart” of this

model. In our workloads, we observed that this situation is common to many models, because many tasks employ simple prediction models like linear regression functions, both because of their prediction latency and because of their simplicity and understandability. In these scenarios, we can conclude that the acceleration of an ML pipeline cannot focus solely on specific transformations, as in the case of neural networks where the bulk of computation is spent in scoring, but needs a more systematic approach that considers the model as a whole.

4.4 Considerations and System Implementation

Following the observations of section 4.3, we argue that acceleration of generic ML prediction pipelines is possible if we optimize models end-to-end instead of single transformations: while data scientist should focus on high-level "logical" transformations, the system for the operationalization of the model for scoring (ideally, the runtime within the FaaS infrastructure) should focus on execution units making optimal decisions on how data is processed through model pipelines. We call such execution units *stages*, borrowing this term from the database and system communities. In our context, a stage is a sequence of one or more transformations that are executed together on a device (either a CPU or an FPGA, in this work), and represents the "atomic" unit of the computation. No data movement from a computing device to another occurs within a stage, while communication may occur between two stages running on different devices, like the PCIe communication when offloading to FPGA. Within a stage, multiple optimizations are possible, from high-level optimizations (like fusing operators together through data pipelining) to low-level, hardware-related ones (like vectorization for CPUs or task pipelining for FPGAs). The notion of stage is particularly important with regards to FPGA acceleration because it allows accounting for the cost of communication: grouping multiple transformations into a single stage allows trading off the data movement overhead, and makes it possible to devise models of the communication and the execution to be used for scheduling. Moreover, as we observe that certain sequences of transformations appear very frequently,they can be accelerated together and offloaded to dedicated hardware.

In the example of fig. 4.2 we distinguish three stages:

- In the first stage the input is tokenized and the number of occurrence of the char n-grams found in the sentence is returned as a sparse vector together with its cartesian norm.
- In the second stage, another feature vector is computed out of the bag-of-words representation of the input tokens (coming from the previous stage).
- In the third stage, weighting and normalization occur, and finally the exponentiation is computed; within this stage, it is possible to vectorize the weighting operation and

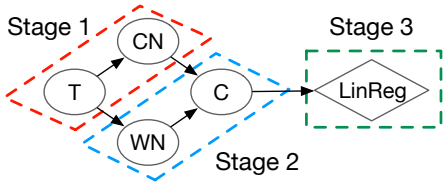

Figure 4.3: *Stages for the sentiment analysis example pipeline*

to change the representation of the vector (from sparse to dense) if this makes the computation more efficient

We optimized these three stages both for CPU execution and for FPGA execution, and give more details on the specific implementations in section 4.4.1 and section 4.4.2, respectively. The model DAG with related stage subdivision is pictorially described in fig. 4.3.

4.4.1 CPU Implementation

As a framework written in C# language, IMLT heavily depends on reflection, virtual buffers, and virtual function calls. Users do not need to worry about data types, memory layout, and how the virtual functions are implemented. In this way, users can build their custom pipelines easily, however, we observed that IMLT has to sacrifice performance at scoring time in order to provide such level of generality.

We ran an experiment where the model described in the previous section is first loaded in memory and then scored multiple times. Even when using the same input record, we noticed that large variability in the execution time exists: the first scoring is regularly around 540 times slower than the remaining executions. We refer to each case as *cold* and *hot*, respectively. Although the Just In Time (JIT) compiler makes subsequent requests run faster, one cannot in general guarantee the target SLA, especially with the multiple sources of unpredictability that lie in the hardware/software stack. Nonetheless, our optimization goal is then to make the prediction time faster and more predictable, to explore possible solutions for the more general use-cases.

In both the hot and the cold scenario, we identified that major bottlenecks were created by

1. the JIT compiler;
2. the module inferring data type with reflection;
3. virtual function calls;
4. allocation of memory buffers on demand.

To address the first two problems, in the CPU implementation we removed those overheads by rewriting the pipeline code: since the model pipelines are pre-trained, all the type information and data path information can be resolved at compile time. This removes the overhead of dynamic type binding and code compilation. Moreover, we pre-compiled the pipeline into bytecode, which reduces the cost of reflection and virtual function call. Furthermore, we created a native image of the pre-compiled code and its dependencies. In this way we avoided all unpredictable performance due to just-in-time compilation. Finally, we can benefit from information extracted at training time to improve memory management at scoring time: since the data types in the pipelines are fixed after training, we can estimate the amount of required memory before execution, and create pools of memory buffer which are then accessed when stages are executed. This information not only help the runtime optimize memory allocations, but also helps it compute a more precise estimation of the needed memory, which is a fundamental information for environment that leverage co-location to improve TCO and can also be used to improve isolation via the mechanisms discussed in chapter 3.

4.4.2 FPGA Implementation

In the FPGA implementation, the first stage can greatly benefit from the hardware characteristics. Indeed, the whole stage can be easily parallelized and its sub-phases can be pipelined with each other. In particular, the text normalization can be done immediately before the tokenization in a pipelined fashion, and the tokenization of a long string can be split into multiple parallel tokenizations by allowing some overlap of input characters between consecutive tokenizer units. A tokenizer unit should recognize common punctuation marks, the beginning and end of words, the occurrence of English contracted negations (like "don't") and emit a corresponding Murmur hash [114] for each token. To achieve an efficient implementation, we devised a Finite State Machine (FSM) recognizer automaton that is able to take in input one character per clock cycle, recognize the tokens observed and record the corresponding hashes. The design of the tokenizer allows trading between performance and area: if area is limited, a tokenizer can run over more characters and record more tokens at the expense of a higher latency; otherwise, it is possible to limit this latency by using many tokenizers in parallel, with at most one tokenizer per character that can emit at most one hash. To perform efficient tokenization on FPGA, we limited the maximum number of characters per word to a fixed threshold, chosen here as 28 characters. This value allows to recognize all the common words of the English language (and of other languages)

Once the hashes are available (output of stage 1), stage 2 associates each word to a unique number for n-gram extraction, with a dictionary lookup that goes to the off-chip memory. Although this operation is expensive, we argue it is no more expensive than on CPU because of the similar hardware for off-chip RAM memory access, and because of the

low locality of this transformation that makes caches ineffective. To make this operation as efficient as possible, we increased the number of buckets of the model dictionary in order to limit collisions (which are handled by serializing the data in the buffer) to a pre-defined number, so that the FPGA logic can fetch a fixed amount of data in a burst fashion for every lookup. In pipelines with dictionary lookup, we insert each n-gram identifier in an array, with a parallel lookup to avoid double insertions and increment the counts. This whole phase could be simplified and sped up by using the hash itself as the n-gram identifier and allowing collisions, but this requires retraining the model; as from the assumptions in section 4.3, we assume the model is fixed and we have no control on it, and leave this work for the future.

Regarding the FPGA implementation of the third stage, for the weights lookup we used the n-gram identifier to access the value in the off-chip memory, again at the cost of an off-chip memory access. This is due to the size of the weights vector, which cannot fit in the on-chip memory. As before, the sparsity of this computation requires memory accesses on both the FPGA and the CPU.

4.5 Experimental Evaluation

In this section we experimentally evaluate the performance gain of our implementations described in section 4.4 with respect to the baseline IMLT. For the CPU implementation, the experiments were run over a Windows 10 Pro machine with an Intel Xeon CPU E5-2620 with 2 processors at 2.10GHz, and 32 GB of RAM. Regarding the FPGA implementation, we used the SDAccel prototyping platform by Xilinx, which abstracts the communication, and we used Xilinx' High Level Synthesis (HLS) tools to generate the accelerator logic. As a device, we used an ADM-PCIE-KU3 board by Alpha Data, with 2 DDR3 memory channels and PCIe x8 connection to the CPU. While the area did not cause major limitations to the design, the number of RAM channels limited the number of parallel dictionary and weights lookups, despite still providing comparable bandwidth compared to a CPU for reading the input sentence and for the dictionary bursts. fig. 4.4 shows the results of our experiments over the sentiment analysis model of section 4.3 in terms of performance, reporting the scoring latency of IMLT, of the CPU implementation and of the FPGA implementation. We report the speedup over the prediction latency on both the hot and cold scenarios for the CPU implementation. For the cold scenario we scored one single record, while for the hot scenario we first use one record to worm up the model and then we average the scoring latency of a batch of 9 records. We repeated the experiment 5 times and we report the average speedup. All the results have been normalized with respect to the IMLT prediction latency in the hot scenario, which is our baseline. Figure 4.4 shows that our CPU implementation achieves 3.3 times improvement over IMLT in the hot scenario, while in the cold scenario it is still 20 times slower, but with an improvement of 87.5x over IMLT cold. These improvements are due to the up-front

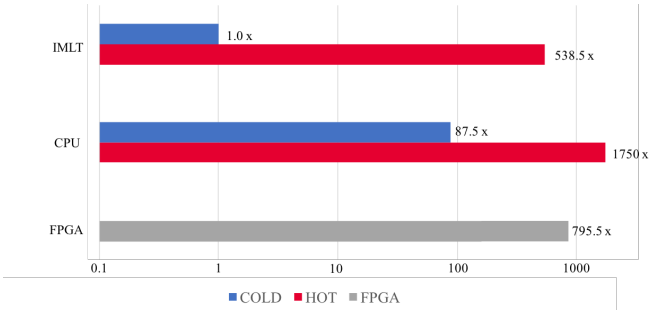

Figure 4.4: *Performance improvement achieved by CPU and FPGA implementations*

memory allocation and to the more optimized code, which avoids memory copy and data conversions among transformations within the same stage. Instead, the FPGA implementation (which does not suffer from warm-up delays and is thus shown in a single scenario) achieves a 1.48x speedup over IMLT hot but a 2.3x slowdown over the CPU implementation in the hot case. This result is due to the high penalty of off-chip RAM access, which is higher due the lower operating frequency and to design delays between memory bursts (due to HLS scheduling).

4.6 Conclusions and directions to harvest untapped performance

By exploring guidelines for the acceleration of generic ML prediction pipelines, this chapter showed that it is fundamental to avoid the large runtime overheads associated to managed environments and to high-level, single-operation models descriptions to achieve orders-of-magnitude speedups and "untapped" performance levels. To better devise hardware-specific optimizations, we introduced the notion of stages, observing that ML transformations usually occur in common sequences that can be organized into atomic execution and scheduling units. Based on this, we implemented a case study in CPU and FPGA, showing noticeable speedups over the initial baseline, thus paves the way to more research in optimizing ML prediction pipelines. This work was published in an international conference [145], and serves as the basis for the design principles adopted in chapter 5.

The identification and generation of stages is a very challenging work direction. In the FPGA case, operations fusions and optimizations techniques can be explored and applied, possibly with the aid of HLS tools. While a careful design space exploration is fundamental to identify appropriate performance/area trade-offs, higher gains come from appropriately setting the transformations at training time, for example by avoiding the dictionary lookup and allowing the hash conflicts; similarly, caching the model weights on the on-chip memory can enable low-latency, parallel lookups. Modeling these characteristics and properly sizing those stages in order to match a common throughput allows

maximizing the final performance.

For CPU implementations, a given pipeline can be generated automatically starting from the code of its transformations and from training time statistics, to accurately predict the resources usage. In our vision, a cost model is essential to perform the stage identification and thus to drive the optimization process. Despite compiler optimizations are a widely studied and mature field, they are usually limited by the complexity of the code base at hand: instead, the stage division limits the scope of the compiler, which can be more effective. Furthermore, once code-generated, stages can be pre-compiled and linked into a native binary, avoiding the cost of JIT compilation.

All these capabilities, once properly generalized, will enable a runtime living in a FaaS (or more specifically MLaaS) environment to effectively optimize ML and DA applications written as pipelines, provision resources appropriately and co-locate them while enforcing isolation in order to guarantee QoS

CHAPTER 5

Improving Machine Learning applications in the Cloud

While chapter 4 discusses in-depth an example ML application accelerated for CPU and FPGA, this section broadens the spectrum of target applications to generic ML pipelines in a cloud settings, supporting a large class of optimizations that target commodity servers with modern CPUs. ML models design, based on pipelines of transformations, allows to efficiently execute single model components at training-time; however, prediction-serving (introduced in section 2.2) has different requirements such as low latency, high throughput and graceful performance degradation under heavy load. Current prediction-serving systems consider models as black boxes, whereby prediction-time-specific optimizations are ignored in favor of ease of deployment, as chapter 4 suggested. This section presents PRETZEL, a prediction serving system introducing a novel white box architecture enabling both end-to-end and multi-model optimizations. Using production-like model pipelines, our experiments show that PRETZEL is able to introduce performance improvements over different dimensions; compared to state-of-the-art approaches PRETZEL is on average able to reduce 99th percentile latency by 5.5× while reducing memory footprint by 25×, and increasing throughput by 4.7×.

The remainder of the chapter is organized as follows: section 5.1 identifies a set of limitations affecting current black box model serving approaches; the outcome of the dis-

cussed limitations is a set of design principles for white box model serving, described in 5.2. Section 5.3 introduces the PRETZEL system as an implementation of the above principles, validated in section 5.4 with a set of experiments, while section 5.5 lists the limitations of current PRETZEL implementation and future work. To better compare the advancements of PRETZEL in light of its results, section 5.6 overviews the related work, while section 5.7 finally concludes the chapter.

5.1 Model Serving: State-of-the-Art and Limitations

Nowadays "intelligent" services depend more and more on ML scoring capabilities and are currently experiencing a growing demand [39], fostering the research in prediction-serving systems for cloud settings [**clipper2**, 159, 122, 40], where trained models from data science experts are operationalized.

Data scientists prefer to use high-level declarative tools such as ML.Net, Keras [81] or Scikit-learn for better productivity and easy operationalization. These tools provide dozens of pre-defined operators and ML algorithms, which data scientists compose into sequences of operators (called *pipelines*) using high-level APIs (e.g., in Python). ML.Net [112], the ML toolkit used in this chapter, is a C# library that runs on a managed runtime with garbage collection and JIT compilation, and derives from the IMLT used in the previous chapter. Unmanaged C/C++ code can also be employed to speed up processing when possible. Internally, ML.Net operators consume data vectors as input and produce one (or more) vectors as output [1] . Vectors are immutable whereby multiple downstream operators can safely consume the same input without triggering any re-execution. Upon pipeline initialization, operators composing the model DAG are analyzed and arranged to form a chain of function calls which, at execution time, are JIT-compiled to form a unique function executing the whole DAG on a single call. Although ML.Net supports NN models, in this work we only focus on pipelines composed by featurizers and classical ML models (e.g., trees, logistic regression, etc.).

Pipelines are first trained using large datasets to estimate models' parameters. ML.Net models are exported as compressed files containing several directories, one per pipeline operator, where each directory stores operator parameters in either binary or plain text files (the latter to ease human inspection). ML.Net, as most systems, aims to minimize the overhead of deploying trained pipelines in production by serving them into black box containers, where the same code is used for both training and inference. Figure 5.1 depicts a set of black box models where the invocation of the function chain (e.g., `predict()`) on a pipeline returns the result of the prediction: throughout this execution chain, inputs are pulled through each operator to produce intermediate results that are input to the following operators, similarly to the well-known Volcano-style iterator model of databases

[1] Note that this is a simplification. ML.Net in fact support several data types. We refer readers to [4] for more details.

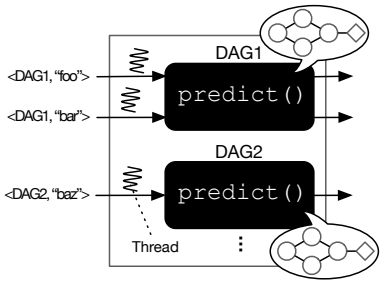

Figure 5.1: *A representation of how existing systems handle prediction requests. Each pipeline is surfaced externally as a black box function. When a prediction request is issued (`predict()`), a thread is dispatched to execute the chain as a single function call.*

[56]. To optimize the performance, ML.Net (and systems such as Clipper [40] among others) applies techniques such as handling multiple requests in batches and caching the results of the inference if some predictions are frequently issued for the same pipeline. However, these techniques assume no knowledge and no control over the pipeline, and are unaware of its internal structure. Despite being regarded as a good practice [189], the black box, container-based design hides the structure of each served model and prevents the system from controlling and optimizing the pipeline execution and from knowing its resource consumption in a fine-grained manner. Therefore, under this approach, there is no principled way neither for sharing optimizations between pipelines, nor to improve the end-to-end execution of individual pipelines. More concretely, we observed the following limitations in current state-of-the-art prediction serving systems.

Memory Waste: Containerization of pipelines disallows any sharing of resources and runtimes [2] between pipelines, therefore only a few (tens of) models can be deployed per machine. Conversely, ML frameworks such as ML.Net have a known set of operators to start with, and featurizers or models trained over similar datasets have a high likelihood of sharing parameters. For example, transfer learning, A/B testing, and personalized models are common in practice; additionally, tools like ML.Net suggest default training configurations to users given a task and a dataset, which leads to many pipelines with similar structure and common objects and parameters. To better illustrate this scenario, we pick a Sentiment Analysis (SA) task with 250 different versions of the pipeline of fig. 2.1 trained by data scientists at Microsoft.

Figure 5.2 shows how many different (parameterized) operators are used, and how often they are used within the 250 pipelines. While some operators like linear regression (whose weights fit in ~15MB) are unique to each pipeline, and thus not shown in fig. 5.2, many other operators can be shared among pipelines, therefore allowing more aggressive packing of models: Tokenize and Concat are used with the same parameters in all

[2] One instance of model pipeline in production easily occupies 100s of MB of main memory.

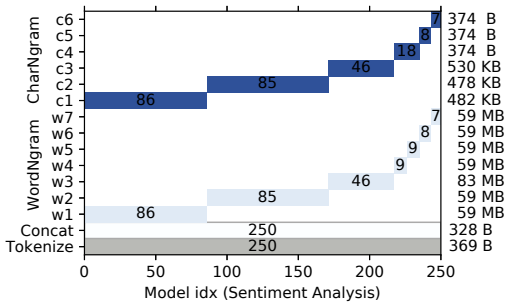

Figure 5.2: *How many identical operators can be shared in multiple SA pipelines. CharNgram and WordNgram operators have variations that are trained on different hyper-parameters. On the right we report operators sizes.*

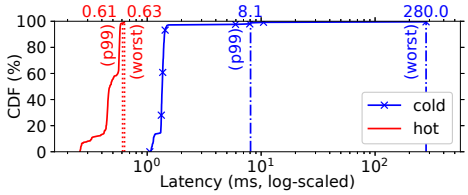

Figure 5.3: *Cumulative Distribution Function (CDF) of latency of prediction requests of 250 DAGs. We denote the first prediction as* cold*; the* hot *line is reported as average over 100 predictions after a warm-up period of 10 predictions. We present the 99th percentile and worst case latency values.*

pipelines; Ngram operators have only a handful of versions, where most pipelines use the same version of the operators. This suggests that the resource utilization of current black box approaches can be largely improved. **Prediction Initialization:** ML.Net employs a pull-based execution model that lazily materializes input feature vectors, and tries to reuse existing vectors between intermediate transformations. This largely decreases the memory footprint and the pressure on garbage collection at training time. Conversely, this design forces memory allocation along the data path, thus making latency of predictions sub-optimal and hard to predict. Furthermore, since DAGs are typically authored in a high-level language with generics, at prediction time ML.Net deploys pipelines as in the training phase, which requires initialization of function chain call, reflection for type inference and JIT compilation. While this composability conveniently hides complexities and allows changing implementations during training, it is of little use during inference, when a model has a defined structure and its operators are fixed. In general, the above problems result in difficulties in providing strong tail latency guarantees by ML-as-a-service providers. Figure 5.3 describes this situation, where the performance of *hot* predictions over the 250 sentiment analysis pipelines with memory already allocated and

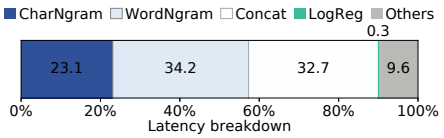

Figure 5.4: *Latency breakdown of a SA pipeline: each frame represents the relative wall clock time spent on an operator.*

JIT-compiled code is more than two orders of magnitude faster than the worst *cold* case version for the same pipelines.

To drill down more into the problem, we found that 57.4% of the total execution time for a single cold prediction is spent in pipeline analysis and initialization of the function chain, 36.5% in JIT compilation and the remaining is actual computation time.

Infrequent Accesses: In order to meet milliseconds-level latencies [182], model pipelines have to reside in main memory (possibly already warmed-up), since they can have MBs to GBs (compressed) size on disk, with loading and initialization times easily exceeding several seconds. A common practice in production settings is to unload a pipeline if not accessed after a certain period of time (e.g., a few hours). Once evicted, successive accesses will incur a model loading penalty and warming-up, therefore violating SLA. **Operator-at-a-time Model:** As previously described, predictions over ML.Net pipelines are computed by pulling records through a sequence of operators, each of them operating over the input vector(s) and producing one or more new vectors. While (as is common practice for in-memory data-intensive systems [117, 168, 12]) some interpretation overheads are eliminated via JIT compilation, operators in ML.Net (and in other tools) are "logical" entities (e.g., linear regression, tokenizer, one-hot encoder, etc.) with diverse performance characteristics. Figure 5.4 shows the latency breakdown of one execution of the SA pipeline of fig. 2.1, where the only ML operator (linear regression) takes two orders of magnitude less time with respect to the slowest operator (WordNgram). It is common practice for in-memory data-intensive systems to pipeline operators in order to minimize memory accesses for memory-intensive workloads, and to vectorize compute intensive operators in order to minimize the number of instructions per data item [42, 190]. ML.Net operator-at-a-time model [190] (as other libraries missing an optimization layer, such as Scikit-learn) is therefore sub-optimal in that computation is organized around logical operators, ignoring how those operators behave together: in the example of the sentiment analysis pipeline at hand, linear regression is commutative and associative (e.g., dot product between vectors) and can be pipelined with Char and WordNgram, eliminating the need for the Concat operation and the related buffers for intermediate results. As we will see in the following sections, PRETZEL's optimizer is able to detect this situation and generate an execution plan that is several times faster than the ML.Net version of the pipeline.

Coarse Grained Scheduling: Scheduling CPU resources carefully is essential to serve highly concurrent requests and run machines to maximum utilization. Under the black box approach:

1. a thread pool is used to serve multiple concurrent requests to the same model pipeline;
2. for each request, one thread handles the execution of a full pipeline sequentially [3], where one operator is active at each point in time;
3. shared operators/parameters are instantiated and evaluated multiple times (one per container) independently;
4. thread allocation is managed by the OS;
5. (5) load balancing is achieved "externally" by replicating containers when performance degradation is observed.

We found this design sub-optimal, especially in heavily skewed scenarios where a small amount of popular models are scored more frequently then others: indeed, in this setting the popular models will be replicated (linearly increasing the resources used) whereas containers of less popular pipelines will run underutilized, therefore decreasing the total resource utilization. The above problem is currently out-of-scope for black box, container-based prediction serving systems because they lack visibility into pipelines execution, and they do not allow models to properly share computational resources. These considerations lead us to the conclusion that in order to achieve better throughput and resources utilization, better scheduling decisions have to be implemented at the (shared) operator level instead of at the DAG level.

After highlighting the major inefficiencies of current black box prediction serving systems, we discuss a set of design principles for white box prediction serving.

5.2 White Box Prediction Serving: Design Principles

The limitations discussed in section 2.2 and section 5.1 of existing black box systems inspired us for developing PRETZEL, a system for serving predictions over trained pipelines (originally authored in ML.Net) that borrows ideas from the Database and System research. Recalling the observations in section 2.2, trained pipelines often share operators and parameters, such as weights and dictionaries used within operators, especially during featurization [185]. To optimize these workloads, we propose a *white box* approach for model serving whereby end-to-end and multi-pipeline optimization techniques are applied to reduce resource utilization while improving performance. Specifically, in PRETZEL

[3] Certain pipelines allow multi-threaded execution, but here we evaluate only single-threaded ones to estimate the per-thread efficiency.

deployment and serving of model pipelines follow a two-phase process. During an *off-line phase*, statistics from training and state-of-the-art techniques from in-memory data-intensive systems [42, 190, 22, 80, 117] are used in concert to optimize and compile operators into *model plans*. Model plans are white box representations of input pipelines such that PRETZEL is able to store and re-use parameters and computation among similar plans. In the *on-line phase*, memory (data vectors) and CPU (thread-based execution units) resources are pooled among plans. When an inference request for a plan is received, an event-based scheduling [174] is used to bind computation to execution units.

In particular, a *white box approach* allows to optimize the execution of predictions both horizontally *end-to-end* and vertically *among multiple model pipelines*.

White Box Prediction Serving: Model containerization disallows any sharing of optimizations, resources, and costs between pipelines. By choosing a white box architecture, pipelines can co-exist on the same runtime; unpopular pipelines can be maintained up and warm, while popular pipelines pay the bills. Thorough scheduling of pipelines' components can be managed within the runtime so that optimal allocation decisions can be made for running machines to high utilization. Nevertheless, if a pipeline requires exclusive access to computational or memory resources, a proper reservation-based allocation strategy can be enforced by the scheduler so that container-based execution can be emulated.

End-to-end Optimizations: The operationalization of models for prediction should focus on computation units making optimal decisions on how data are processed and results are computed, to keep low latency and gracefully degrade with load increase. Such computation units should:

1. avoid memory allocation on the data path;
2. avoid creating separate routines per operator when possible, which are sensitive to branch mis-prediction and poor data locality [117];
3. avoid reflection and JIT compilation at prediction time.

Optimal computation units can be compiled Ahead-Of-Time (AOT) since pipeline and operator characteristics are known upfront, and often statistics from training are available. The only decision to make at runtime is where to allocate computation units based on available resources and constraints.

Multi-model Optimizations: To take full advantage of the fact that pipelines often use similar operators and parameters (fig. 5.2), shareable components have to be uniquely stored in memory and reused as much as possible to achieve optimal memory usage. Similarly, execution units should be shared at runtime and resources properly pooled and managed, so that multiple prediction requests can be evaluated concurrently. Partial results, for example outputs of featurization steps, can be saved and re-used among multiple similar pipelines.

5.3 The Pretzel System Design

Following the above guidelines, we implemented PRETZEL, a novel white box system for cloud-based inference of model pipelines. PRETZEL views models as database queries and employs database techniques to optimize DAGs and improve end-to-end performance (section 5.3.1). The problem of optimizing co-located pipelines is casted as a multi-query optimization and techniques such as view materialization (section 5.3.3) are employed to speed up pipeline execution. Memory and CPU resources are shared in the form of vector and thread pools, such that overheads for instantiating memory and threads are paid upfront at initialization time.

PRETZEL is organized in several components. A *data-flow-style language integrated API* called Flour (section 5.3.1) with related *compiler* and *optimizer* called Oven (section 5.3.1) are used in concert to convert ML.Net pipelines into *model plans*. An Object Store (section 5.3.1) saves and shares parameters among plans. A Runtime (section 5.3.2) manages compiled plans and their execution, while a Scheduler (section 5.3.2) manages the dynamic decisions on how to schedule plans based on machine workload. Finally, a FrontEnd is used to submit prediction requests to the system.

In PRETZEL, deployment and serving of model pipelines follow a two-phase process. During the *off-line phase* (section 5.3.1), ML.Net's pre-trained pipelines are translated into Flour transformations. Oven optimizer re-arranges and fuses transformations into model plans composed of parameterized logical units called *stages*. Each logical stage is then AOT-compiled into physical computation units where memory resources and threads are pooled at runtime. Model plans are registered for prediction serving in the Runtime where physical stages and parameters are shared between pipelines with similar model plans. In the *on-line phase* (section 5.3.2), when an inference request for a registered model plan is received, physical stages are parameterized dynamically with the proper values maintained in the Object Store. The Scheduler is in charge of binding physical stages to shared execution units.

Figure 5.5 and fig. 5.6 pictorially summarize the above descriptions; note that only the on-line phase is executed at inference time, whereas the model plans are generated completely off-line. Next, we will describe each layer composing the PRETZEL prediction system.

5.3.1 Off-line Phase

Flour

The goal of Flour is to provide an intermediate representation between ML frameworks (currently only ML.Net) and PRETZEL, that is both easy to target and amenable to optimizations. Once a pipeline is ported into Flour, it can be optimized and compiled (section 5.3.1) into a model plan before getting fed into PRETZEL Runtime for on-line scoring.

Flour is a language-integrated API similar to KeystoneML [155], RDDs [184] or LINQ [105] where sequences of *transformations* are chained into DAGs and lazily compiled for execution.

Algorithm 5.1 shows how the SA pipeline of fig. 2.1 can be expressed in Flour. Flour programs are composed by transformations where a one-to-many mapping exists between ML.Net operators and Flour transformations (i.e., one operator in ML.Net can be mapped to many transformations in Flour). Each Flour program starts from a `FlourContext` object wrapping the Object Store. Subsequent method calls define a DAG of transformations, which will end with a call to `Plan` to instantiate the model plan before feeding it into PRETZEL Runtime. For example, in lines 2 and 3 of algorithm 5.1 the `CSV.FromText` call is used to specify that the target DAG accepts as input text in CSV format where fields are comma separated. Line 4 specifies the schema for the input data, where `TextReview` is a class whose parameters specify the schema fields names, types, and order. The following call to `Select` in line 5 is used to pick the `Text` column among all the fields, while the call to `Tokenize` in line 6 is used to split the input fields into tokens. Lines 8 and 9 contain the two branches defining the char-level and word-level n-gram transformations, which are then merged with the `Concat` transform in lines 10/11 before the linear binary classifier of line 12. Both char and word n-gram transformations are parameterized by the number of n-grams and maps translating n-grams into numerical format (not shown in the Listing). Additionally, each Flour transformation accepts as input an optional set of statistics gathered from training. These statistics are used by the compiler to generate physical plans more efficiently tailored to the model characteristics. Example statistics are max vector size (to define the minimum size of vectors to fetch from the pool at prediction time, as in section 5.3.2), dense/sparse representations, etc.

We have instrumented the ML.Net library to collect statistics from training and with the related bindings to the Object Store and Flour to automatically extract Flour programs from pipelines once trained.

Listing 5.1 Flour *program for the SA pipeline. Parameters are extracted from the original ML.Net pipeline.*

```
1  var fContext = new FlourContext(objectStore, ...)
2  var tTokenizer = fContext.
3  CSV
4                  .
5  FromText(',')
6                  .
7  WithSchema<TextReview>()
8                  .
9  Select("Text")
10                 .
11 Tokenize();
12
13 var tCNgram = tTokenizer.
14 CharNgram(numCNgrms, ...);
15 var tWNgram = tTokenizer.
16 WordNgram(numWNgrms, ...);
```

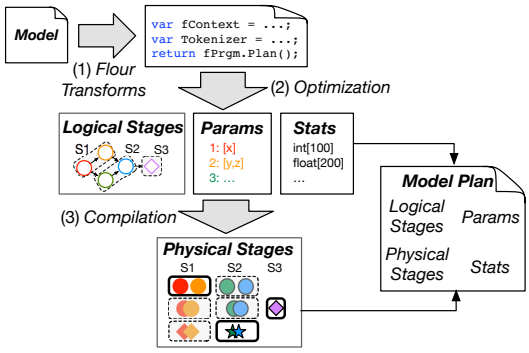

Figure 5.5: *Model optimization and compilation in* PRETZEL. *In (1), a model is translated into a* Flour *program. (2)* Oven Optimizer *generates a DAG of logical stages from the program. Additionally, parameters and statistics are extracted. (3) A DAG of physical stages is generated by the* Oven Compiler *using logical stages, parameters, and statistics. A model plan is the union of all the elements.*

```
17 var fPrgrm = tCNgram
18                 .
19 Concat(tWNgram)
20                 .
21 ClassifierBinaryLinear(cParams);
22
23 return fPrgrm.
24 Plan();
```

Oven

With Oven, our goal is to bring query compilation and optimization techniques into ML.Net.

Optimizer: When `Plan` is called on a Flour transformation's reference (e.g., `fPrgrm` in line 14 of algorithm 5.1), all transformations leading to it are wrapped and analyzed. Oven follows the typical rule-based database optimizer design where operator graphs (query plans) are transformed by a set of rules until a fix-point is reached (i.e., the graph does not change after the application of any rule). The goal of Oven Optimizer is to transform an input graph of Flour transformations into a stage graph, where each stage contains one or more transformations. To group transformations into stages we used the Tupleware's hybrid approach [42]: memory-intensive transformations (such as most featurizers) are pipelined together in a single pass over the data. This strategy achieves best data locality because records are likely to reside in CPU L1 caches [80, 117]. Compute-intensive transformations (e.g., vector or matrix multiplications) are executed one-at-a-time so that SIMD vectorization can be exploited, therefore optimizing the number of instructions per record [190, 22]. Transformation classes are annotated (e.g., 1-to-1, 1-to-n, memory-bound, compute-bound, commutative and associative) to ease the optimization process:

no dynamic compilation [42] is necessary since the set of operators is fixed and manual annotation is sufficient to generate properly optimized plans [4] .

Stages are generated by traversing the Flour transformations graph repeatedly and applying rules when matching conditions are satisfied. Oven Optimizer consists of an extensible number of *rewriting steps*, each of which in turn is composed of a set of rules performing some modification on the input graph: each rule is generally composed by a matching condition and graph rewriting logic. Each rewriting step is executed sequentially: within each step, the optimizer iterates over its full set of rules until an iteration exists such that the graph is not modified after all rules are evaluated. When a rule is active, the graph is traversed (either top-down, or bottom up, based on rule internal behavior; Oven provides graph traversal utilities for both cases) and the rewriting logic is applied if the matching condition is satisfied over the current node.

In its current implementation, the Oven Optimizer is composed of 4 rewriting steps:

InputGraphValidatorStep: This step comprises three rules, performing schema propagation, schema validation and graph validation. Specifically, the rules propagate schema information from the input to the final transformation in the graph, and validate that

1. each transformation's input schema matches with the transformation semantics (e.g., a WordNgram has a string type as input schema, or a linear learner has a vector of floats as input);
2. the transformation graph is well-formed (e.g., a final predictor exists).

StageGraphBuilderStep: It contains two rules that rewrite the graph of (now schematized) Flour transformations into a stage graph. Starting with a valid transformation graph, the rules in this step traverse the graph until a pipeline-breaking transformation is found, i.e., a Concat or an n-to-1 transformation such as an aggregate used for normalization (e.g., L2). These transformations, in fact, require data to be fully scanned or materialized in memory before the next transformation can be executed. For example, operations following a Concat require the full feature vector to be available, or a Normalizer requires the L2 norm of the complete vector. The output of the `StageGraphBuilderStep` is therefore a stage graph, where each stage internally contains one or more transformations. Dependencies between stages are created as aggregation of the dependencies between the internal transformations. By leveraging the stage graph, PRETZEL is able to considerably decrease the number of vectors (and as a consequence the memory usage) with respect to the operator-at-a-time strategy of ML.Net.

StageGraphOptimizerStep: This step involves nine rules that rewrite the graph in order to produce an optimal (logical) plan. The most important rules in this step rewrite the stage graph by

[4] Note that ML.Net does provide a second order operator accepting arbitrary code requiring dynamic compilation. However, this is not supported in our current version of PRETZEL.

1. removing unnecessary branches (similar to common sub-expression elimination);
2. merging stages containing equal transformations (often generated by traversing graphs with branches);
3. inlining stages that contain only one transform;
4. pushing linear models through Concat operations;
5. removal of unnecessary stages (e.g., when linear models are pushed through Concat operations, the latter stage can be removed if not containing any other additional transformation).

OutputGraphValidatorStep: This last step is composed of six rules. These rules are used to generate each stage's schema out of the schemas of the single internal transformations. Stage schema information will be used at runtime to request properly typed vectors. Additionally, some training statistics are applied at this step: transformations are labeled as sparse or dense, and dense compute-bound operations are labeled as vectorizable. A final validation check is run to ensure that the stage graph is well-formed.

In the example sentiment analysis pipeline of fig. 2.1, Oven is able to recognize that the Linear Regression can be pushed into CharNgram and WordNgram, therefore bypassing the execution of Concat. Additionally, Tokenizer can be reused between CharNgram and WordNgram, therefore it will be pipelined with CharNgram (in one stage) and a dependency between CharNgram and WordNgram (in another stage) will be created. The final plan will therefore be composed of two stages, versus the initial four operators (and vectors) of ML.Net.

Model Plan Compiler: Model plans have two DAGs: a DAG of *logical stages*, and a DAG of *physical stages*. Logical stages are an abstraction of the results of the Oven Optimizer; physical stages contain the actual code that will be executed by the PRETZEL runtime. For each given DAG, there is a 1-to-n mapping between logical to physical stages so that a logical stage can represent the execution code of different physical implementations. A physical implementation is selected based on the parameters characterizing a logical stage and available statistics.

Plan compilation is a two step process. After the stage DAG is generated by the Oven Optimizer, the Model Plan Compiler (MPC) maps each stage into its logical representation containing all the parameters for the transformations composing the original stage generated by the optimizer. Parameters are saved for reuse in the Object Store (section 5.3.1). Once the logical plan is generated, MPC traverses the DAG in topological order and maps each logical stage into a physical implementation. Physical implementations are AOT-compiled, parameterized, lock-free computation units. Each physical stage can be seen as a parametric function which will be dynamically fed at runtime with the proper data vectors and pipeline-specific parameters. This design allows PRETZEL runtime to share

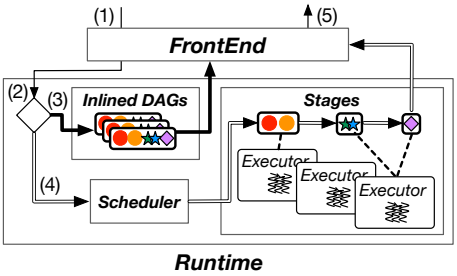

Figure 5.6: *(1) When a prediction request is issued, (2) the* Runtime *determines whether to serve the prediction using (3) the request/response engine or (4) the batch engine. In the latter case, the* Scheduler *takes care of properly allocating stages over the Executors running concurrently on CPU cores. (5) The* FrontEnd *returns the result to the Client once all stages are complete.*

the same physical implementation between multiple pipelines and no memory allocation occurs on the prediction path (more details in section 5.3.2). Logical plans maintain the mapping between the pipeline-specific parameters saved in the Object Store and the physical stages executing on the Runtime as well as statistics such as maximum vector size (which will be used at runtime to request the proper amount of memory from the pool). Figure 5.5 summarizes the process of generating model plans out of ML.Net pipelines.

Object Store

The motivation behind Object Store is based on the insights of fig. 5.2: since many DAGs have similar structures, sharing operators' state (parameters) can considerably improve memory footprint, and consequently the number of predictions served per machine. An example is language dictionaries used for input text featurization, which are often in common among many models and are relatively large. The Object Store is populated off-line by MPC: when a Flour program is submitted for planning, new parameters are kept in the Object Store, while parameters that already exist are ignored and the stage information is rewritten to reuse the previously loaded one. Parameters equality is computed by looking at the checksum of the serialized version of the objects.

5.3.2 On-line Phase

Runtime

Initialization: Model plans generated by MPC are registered in the PRETZEL Runtime. Upon registration, a unique pipeline ID is generated, and physical stages composing a plan are loaded into a system *catalog*. If two plans use the same physical stage, this is loaded only once in the catalog so that similar plans may share the same physical stages during execution. When the Runtime starts, a set of vectors and long-running thread pools

(called *Executors*) are initialized. Vector pools are allocated per Executor to improve locality [**Gamsa:1999:T\gls {ml}:296806.296814**]; Executors are instead managed by the Scheduler to execute physical stages (section 5.3.2) or used to manage incoming prediction requests by the FrontEnd. Allocations of vector and thread pools are managed by configuration parameters, and allow PRETZEL to decrease the time spent in allocating memory and threads during prediction time.

Execution: Inference requests for the pipelines registered into the system can be submitted through the FrontEnd by specifying the pipeline ID, and a set of input records. fig. 5.6 depicts the process of on-line inference. PRETZEL comes with a *request-response engine* and a *batch engine*. The request-response engine is used by single predictions for which latency is the major concern whereby context-switching and scheduling overheads can be costly. Conversely, the batch engine is used when a request contains a batch of records, or when the prediction time is such that scheduling overheads can be considered as negligible (e.g., few hundreds of microseconds). The request-response engine inlines the execution of the prediction within the thread handling the request: the pipeline physical plan is JIT-compiled into a unique function call and scored. Instead, by using the batch engine requests are forwarded to the Scheduler that decides where to allocate physical stages based on the current runtime and resource status. Currently, whether to use the request-response or batch engine is set through a configuration parameter passed when registering a plan. In the future we plan to adaptively switch between the two.

Scheduler

In PRETZEL, model plans share resources, thus scheduling plans appropriately is essential to ensure scalability and optimal machine utilization while guaranteeing the performance requirements. Since model inference is latency sensitive, the Scheduler is optimized towards local memory access, and attempts to run the stages according to where the input data reside, e.g. by running all the stages to serve a single prediction request on the same core.

The Scheduler coordinates the execution of multiple stages via a late-binding event-based scheduling mechanism similar to task scheduling in distributed systems [124, 184, 174]: each core runs an Executor instance whereby all Executors pull work from a shared pair of queues: one *low priority* queue for newly submitted plans, and one *high priority* queue for already started stages. At runtime, a scheduling event is generated for each stage with related set of input/output vectors, and routed over a queue (low priority if the stage is the head of a pipeline, high priority otherwise). Two queues with different priorities are necessary because of memory requirements. Vectors are in fact requested per pipeline (not per stage) and lazily fulfilled when a pipeline's first stage is being evaluated on an Executor. Vectors are then utilized and not re-added to the pool for the full execution of the pipeline. Two priority queues allow started pipelines to be scheduled earlier and

therefore return memory quickly.

Reservation-based Scheduling: Upon model plan registration, PRETZEL offers the option to reserve memory or computation resources for exclusive use. Such resources reside on different, pipeline-specific pools, and are not shared among plans, therefore enabling container-like provision of resources. Note however that parameters and physical stage objects remain shared between pipelines even if reservation-based scheduling is requested.

5.3.3 Additional Optimizations

Sub-plan Materialization: Similarly to materialized views in database multi-query optimization [59, 35], results of installed physical stages can be reused between different model plans. When plans are loaded in the runtime, PRETZEL keeps track of physical stages and enables caching of results when a stage with the same parameters is shared by many model plans. Hashing of the input is used to decide whether a result is already available for that stage or not. We implemented a simple LRU strategy on top of the Object Store to evict results when a given memory threshold is met.

Note that sub-plan materialization is different than the result caching implemented in other systems such as Clipper: result caching is a single-pipeline optimization by which the results of predictions are saved and re-used in case the same prediction for the same input is requested; sub-plan caching is instead a multi-pipeline optimization by which partial results are stored and re-used in case the same input is employed to score different (but similar) models.

External Optimizations: While the techniques described so far focus mostly on improvements that other prediction serving systems are not able to achieve due to their black box nature, PRETZEL FrontEnd also supports "external" optimizations such as the one provided in Clipper and Rafiki. Specifically, the FrontEnd currently implements prediction results caching (with LRU eviction policy) and delayed batching whereby inference requests are buffered for a user-specified amount of time and then submitted in batch to the Runtime. These external optimizations are orthogonal to PRETZEL's techniques, so both are applicable in a complementary manner.

5.4 Evaluation

PRETZEL implementation is a mix of C# and C++. In its current version, the system comprises 12.6K lines of code (11.3K in C#, 1.3K in C++) and supports about two dozens of ML.Net operators, among which linear models (e.g., linear/logistic/Poisson regression), tree-based models, clustering models (e.g., K-Means), Principal Components Analysis (PCA), and several featurizers.

Scenarios: The goals of our experimental evaluation are to evaluate how the white box

approach performs compared to black box.

- *memory*: in the first scenario, we want to show how much memory saving PRETZEL's white box approach is able to provide with respect to regular ML.Net and ML.Net boxed into Docker containers managed by Clipper.
- *latency*: this experiment mimics a request/response pattern (like that in [138]) such as a personalized web-application requiring minimal latency. In this scenario, we run two different configurations:
  1. a micro-benchmark measuring the time required by a system to render a prediction;
  2. an experiment measuring the total end-to-end latency observed by a client submitting a request.
- *throughput*: this scenario simulates a batch pattern (like in [18]) and we use it to assess the throughput of PRETZEL compared to ML.Net.
- *heavy-load*: we finally mix the above experiments and show PRETZEL's ability to maintain high throughput and graceful degradation of latency, as load increases. To be realistic, in this scenario we generate skewed load across different pipelines. As for the *latency* experiment, we report first the PRETZEL's performance using a micro-benchmark, and then we compare it against the containerized version of ML.Net in an end-to-end setting.

Configuration: All the experiments reported in the work were carried out on a Windows 10 machine with 2 $\times$ 8-core Intel Xeon CPU E5-2620 v4 processors at 2.10GHz with HyperThreading disabled, and 32GB of RAM.

We used .Net Core version 2.0, ML.Net version 0.4, and Clipper version 0.2. For ML.Net, we use two black box configurations: a non-containerized one (one ML.Net instance for all models), and a containerized one (one ML.Net instance for each model) where ML.Net is deployed as Docker containers running on Windows Subsystem for Linux (WSL) and orchestrated by Clipper. We commonly label the former as just ML.Net; the latter as ML.Net + Clipper. For PRETZEL we AOT-compile stages using CrossGen [1]. For the end-to-end experiments comparing PRETZEL and ML.Net + Clipper, we use an ASP.Net FrontEnd for PRETZEL, and the Redis front-end for Clipper. We run each experiment three times and report the median.

Pipelines: Table 5.1 describes the two types of model pipelines we use in the experiments: 250 unique versions of SA pipeline, and 250 different pipelines implementing AC: a regression task used internally to predict how many attendees will join an event. Pipelines within a category are similar: in particular, pipelines in the SA category benefit from subplan materialization, while those in the AC category are more diverse and do not benefit

Table 5.1: *Characteristics of pipelines in experiments*

Type	Sentiment Analysis (SA)	Attendee Count (Attendee Count (AC))
Characteristic	Memory-bound	CPU-bound
Input	Plain Text (variable length)	Structured Text (40 dimensions)
Size	50MB - 100MB (Mean: 70MB)	10KB - 20MB (Mean: 9MB)
Featurizers	N-gram with dictionaries (∼1M entries)	PCA, KMeans, Ensemble of multiple models

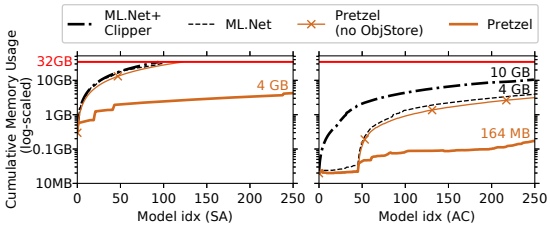

Figure 5.7: *Cumulative memory usage (log-scaled) of the pipelines in* PRETZEL*, ML.Net and ML.Net + Clipper. The horizontal line represents the machine's physical memory (32GB). Only* PRETZEL *is able to load all SA pipelines within the memory limit. For AC,* PRETZEL *uses one order of magnitude less memory than ML.Net and ML.Net + Clipper. The memory usage of* PRETZEL *without* Object Store *is almost on par with ML.Net.*

from it. These latter pipelines comprise several ML models forming an ensemble: in the most complex version, we have a dimensionality reduction step executed concurrently with a KMeans clustering, a TreeFeaturizer, and multi-class tree-based classifier, all fed into a final tree (or forest) rendering the prediction. SA pipelines are trained and scored over Amazon Review dataset [60]; AC ones are trained and scored over an internal record of events.

5.4.1 Memory

In this experiment, we load all models and report the total memory consumption (model + runtime) per model category. SA pipelines are large and therefore we expect memory consumption (and loading time) to improve considerably within this class, proving that PRETZEL's Object Store allows to avoid the cost of loading duplicate objects. Less gains are instead expected for the AC pipelines because of their small size. Figure 5.7 shows the memory usage for loading all the 250 model pipelines in memory, for both categories. For SA, only PRETZEL with Object Store enabled can load all pipelines [5] . For AC, all configurations are able to load the entire working set, however PRETZEL occupies only 164MBs: about 25× less memory than ML.Net and 62× less than ML.Net + Clipper. Given the nature of AC models (i.e., small in size), from fig. 5.7 we can additionally notice

[5]Note that for ML.Net, ML.Net + Clipper and PRETZEL without Object Store configurations we can load more models and go beyond the 32GB limit. However, models are swapped to disk and the whole system becomes unstable.

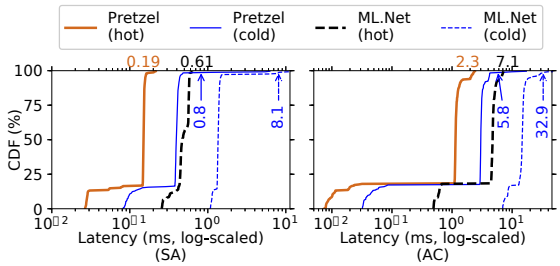

Figure 5.8: *Latency comparison between ML.Net and* PRETZEL. *The accompanying blue lines represent the cold latency (first execution of the pipelines). On top are the P99 latency values: the hot case is above the horizontal line and the cold case is annotated with an arrow.*

the overhead (around 2.5×) of using a container-based black box approach vs regular ML.Net.

Keeping track of pipelines' parameters also helps reducing the time to load models: PRETZEL takes around 2.8 seconds to load 250 AC pipelines while ML.Net takes around 270 seconds. For SA pipelines, PRETZEL takes 37.3 seconds to load all 250 pipelines, while ML.Net fills up the entire memory (32GB) and begins to swap objects after loading 75 pipelines in around 9 minutes.

5.4.2 Latency

In this experiment we study the latency behavior of PRETZEL in two settings. First, we run a micro-benchmark directly measuring the latency of rendering a prediction in PRETZEL. Additionally, we show how PRETZEL's optimizations can improve the latency. Secondly, we report the end-to-end latency observed by a remote client submitting a request through HTTP.

Micro-benchmark

Inference requests are submitted sequentially and in isolation for one model at a time. For PRETZEL we use the request-response engine over one single core. The comparison between PRETZEL and ML.Net for the SA and AC pipelines is reported in fig. 5.8. We start with studying *hot* and *cold* cases while comparing PRETZEL and ML.Net. Specifically, we label as cold the first prediction requested for a model; the successive 10 predictions are then discarded and we report hot numbers as the average of the following 100 predictions.

If we directly compare PRETZEL with ML.Net, PRETZEL is 3.2× and 3.1× faster than ML.Net in the 99th percentile latency in hot case (denoted by $P99_{hot}$), and about 9.8× and 5.7× in the $P99_{cold}$ case, for SA and AC pipelines, respectively. If instead we look at the difference between cold and hot cases relative to each system, PRETZEL again provides improvements over ML.Net. The $P99_{cold}$ is about 13.3× and 4.6× the $P99_{hot}$ in ML.Net,

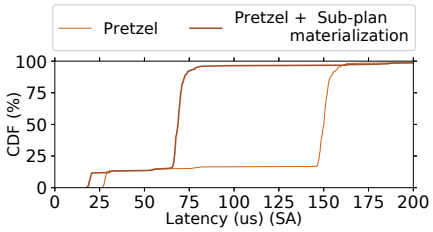

Figure 5.9: *Latency of* PRETZEL *to run SA models with and without sub-plan materialization. Around 80% of SA pipelines show more than 2× speedup. Sub-plan materialization does not apply for AC pipelines.*

whereas in PRETZEL $P99_{cold}$ is around 4.2× and 2.5× from the $P99_{hot}$ case. Furthermore, PRETZEL is able to mitigate the long tail latency (worst case) of cold scoring. In SA pipelines, the worst case latency is 460.6× off the $P99_{hot}$ in ML.Net, whereas PRETZEL shows a 33.3× difference. Similarly, in AC pipelines the worst case is 21.2× $P99_{hot}$ for ML.Net, and 7.5× for PRETZEL. The gap between two categories is due to the pipelines' characteristics: SA pipelines are memory-intensive and more likely to benefit caching objects. AC pipelines are compute-intensive, so the optimizations have less effect in the worst case.

To better understand the effect of PRETZEL's optimizations on latency, we turn on and off some optimizations and compare the performance.

AOT compilation: This options allows PRETZEL to pre-load all stage code into cache, removing the overhead of JIT compilation in the cold cases. Without AOT compilation, latencies of cold predictions increase on average by 1.6× and 4.2× for SA and AC pipelines, respectively.

Vector Pooling: By creating pools of pre-allocated vectors, PRETZEL can minimize the overhead of memory allocation at prediction time. When we do not pool vectors, latencies increase in average by 47.1% for hot and 24.7% for cold, respectively.

Sub-plan Materialization: If different pipelines have common featurizers (e.g., SA as shown in fig. 5.2), we can further apply sub-plan materialization to reduce the latency. Figure 5.9 depicts the effect of sub-plan materialization over prediction latency for hot requests. In general, for the SA pipelines in which sub-plan materialization applies, we can see an average improvement of 2.0×, while no pipeline shows performance deterioration.

End-to-end

In this experiment we measure the end-to-end latency from a client submitting a prediction request. For PRETZEL, we use the ASP.Net FrontEnd, and we compare against ML.Net + Clipper. We use the Clipper's default front-end that is built on Redis [136]. The end-to-

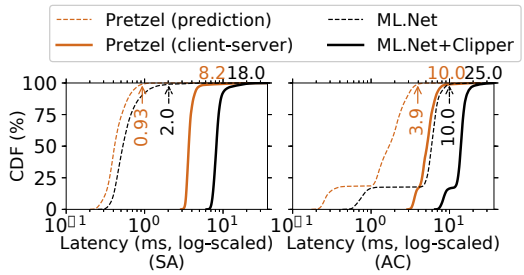

Figure 5.10: *The latency comparison between ML.Net + Clipper and* PRETZEL *with ASP.Net* FrontEnd*. The overhead of client-server communication compared to the actual prediction is similar in both* PRETZEL *and ML.Net: the end-to-end latency compared to the just prediction latency is 9× slower in SA and 2.5× in AC, respectively.*

end latency considers both the prediction latency (i.e., fig. 5.8) as well as any additional overhead due to client-server communication. As shown in fig. 5.10, the latter overhead in both PRETZEL and ML.Net + Clipper is in the milliseconds range (around 4ms for the former, and 9 for the latter). Specifically, with PRETZEL, clients observe a latency of 4.3ms at $P99$ for SA models (vs. 0.56ms $P99$ latency of just rendering a prediction) and a latency of 7.3ms for AC models (vs. 3.5ms). In contrast, in ML.Net + Clipper, clients observe 9.3ms latency at $P99$ for SA models, and 18.0ms at $P99$ for AC models.

5.4.3 Throughput

In this experiment, we run a micro-benchmark assuming a batch scenario where all 500 models are scored several times. We use an API provided by both PRETZEL and ML.Net, where we can execute prediction queries in batches: in this experiment we fixed the batch size at 1000 queries. We allocate from 2 up to 13 CPU cores to serve requests, while 3 cores are reserved to generate them. The main goal is to measure the maximum number of requests PRETZEL and ML.Net can serve per second.

Figure 5.11 shows that PRETZEL's throughput (queries per second) is up to 2.6× higher than ML.Net for SA models, 10× for AC models. PRETZEL's throughput scales on par with the expected ideal scaling. Instead, ML.Net suffers from higher latency in rendering predictions and from lower scalability when the number of CPU cores increases. This is because each thread has its own internal copy of models whereby cache lines are not shared, thus increasing the pressure on the memory subsystem: indeed, even if the parameters are the same, the model objects are allocated to different memory areas.

5.4.4 Heavy Load

In this experiment, we show how the performance changes as we change the load. To generate a realistic load, we submit requests to models by following the Zipf distribution

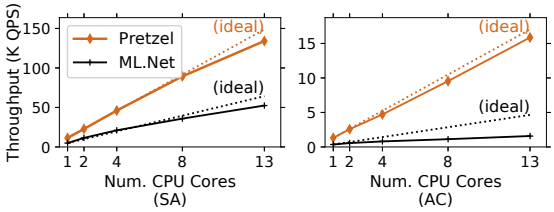

Figure 5.11: *The average throughput computed among the 500 models to process one million inputs each. We scale the number of CPU cores on the x-axis and the number of prediction queries to be served per second on the y-axis.* PRETZEL *scales linearly to the number of CPU cores.*

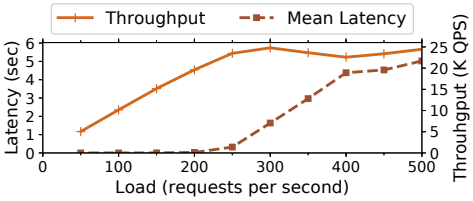

Figure 5.12: *Throughput and latency of* PRETZEL *under the heavy load scenario. We maintain all 500 models in-memory within a* PRETZEL *instance, and we increase the load by submitting more requests per second. We report latency measurements from latency-sensitive pipelines, and the total system throughput.*

($\alpha = 2$) [6] . As in section 5.4.2, we first run a micro-benchmark, followed by an end-to-end comparison.

Micro-benchmark

We load all 500 models in one PRETZEL instance. Among all models, we assume 50% to be "latency-sensitive" and therefore we set a batch size of 1. The remaining 50% models will be requested with 100 queries in a batch. As in the throughput experiment, we use the batch engine with 13 cores to serve requests and 3 cores to generate load. Figure 5.12 reports the average latency of latency-sensitive models and the total system throughput under different load configurations. As we increase the number of requests, PRETZEL's throughput increases linearly until it stabilizes at about 25k queries per second. Similarly, the average latency of latency-sensitive pipelines gracefully increases linearly with the load.

Reservation Scheduling: If we want to guarantee that the performance of latency-critical pipelines is not degrading excessively even under high load, we can enable reservation scheduling. If we run the previous experiment reserving one core (and related vectors) for one model, this does not encounter any degradation in latency (max improvement of 3

[6]The number of requests to the ith most popular models is proportional to $i^{-\alpha}$, where α is the parameter of the distribution.

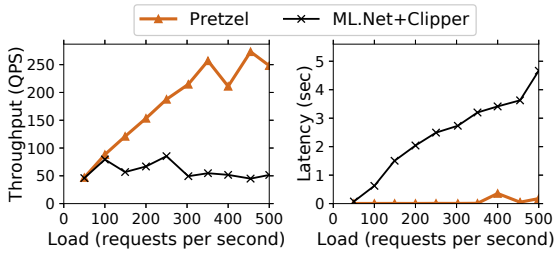

Figure 5.13: *Throughput and latency of* PRETZEL *and ML.Net + Clipper under the end-to-end heavy load scenario. We use 250 AC pipelines to allow both systems to have all pipelines in memory.*

orders of magnitude) as the load increases, while maintaining similar system throughput.

End-to-end

In this setup, we periodically send prediction requests to PRETZEL with the ASP.Net FrontEnd and ML.Net + Clipper. We assume all pipelines to be latency-sensitive, thus we set a batch of 1 for each request. As we can see in fig. 5.13, PRETZEL's throughput keeps increasing up to around 300 requests per second. If the load exceeds that point, the throughput and the latency begin to fluctuate. On the other hand, the throughput of ML.Net + Clipper is considerably lower than PRETZEL's and does not scale as the load increases. Also the latency of ML.Net + Clipper is several folds higher than with PRETZEL. The difference is due to the overhead of maintaining hundreds of Docker containers; too many context switches occur across/within containers.

5.5 Limitations and Future Work

Although the results achieved in section 5.4 are promising, PRETZEL has several limitations, in both the off-line and the on-line phases.

Off-line Phase: PRETZEL has two limitations regarding Flour and Oven design. First, PRETZEL currently has several logical and physical stages classes, one per possible implementation, which make the system difficult to maintain in the long run. Additionally, different back-ends (e.g., PRETZEL currently supports operators implemented in C# and C++) require all specific operator implementations. We are however confident that this limitation will be overcome once code generation of stages will be added, also targeting heterogeneous hardware, e.g., with hardware-specific templates [88]. Secondly, Flour and Oven are currently limited to pipelines authored in ML.Net, and porting models from different frameworks to the white box approach may require non-trivial work. On the long run our goal is, however, to target unified formats such as ONNX [123]; this will allow us to apply the discussed techniques to models from other ML frameworks as well.

On-line Phase: PRETZEL's fine-grained, stage-based scheduling may introduce additional overheads in contrasts to coarse-grained whole pipeline scheduling due to additional buffering and context switching. However, such overheads are related to the system load and therefore controllable by the scheduler. Additionally, we found Garbage Collection (GC) overheads to introduce spikes in latency. Although our implementation tries to minimize the number of objects created at runtime, in practice we found that long tail latencies are common. On white box architectures, failures happening during the execution of a model may jeopardize the whole system. We are currently working on isolating model failures over the target Executor. Finally, PRETZEL runtime currently runs on a single-node and is unaware of the hardware topology; an experimental scheduler adds Non Uniform Memory Access (NUMA)-awareness to scheduling policies. We expect this scheduler to bring benefits for models served from large instances (e.g., [47]). We expect in the future to be able to scale the approach over distributed machines, with automatic scale up/out capabilities.

5.6 Related Work

Prediction Serving: As from the Introduction, current ML prediction systems [40, 122, 130, 41, 137, 113, 170, 115] aim to minimize the cost of deployment and maximize code re-use between training and inference phases [189]. Conversely, PRETZEL casts prediction serving as a database problem and applies end-to-end and multi-query optimizations to maximize performance and resource utilization. Clipper and Rafiki deploy pipelines as Docker containers connected through RPC to a front-end. Both systems apply external model-agnostic techniques to achieve better latency, throughput, and accuracy. While we employed similar techniques in the FrontEnd, in PRETZEL we have not yet explored "best effort" techniques such as ensembles, straggler mitigation, and model selection. As from section 4.2, TF Serving deploys pipelines as *Servable*s, which are units of execution scheduling and version management. One Servable is executed as a black box, although users are allowed to split model pipelines and surface them into different Servables, similarly to PRETZEL's stage-based execution. Such optimization is however not automatic. LASER [2] enables large scale training and inference of logistic regression models, applying specific system optimizations to the problem at hand (i.e., advertising where multiple ad campaigns are run on each user) such as caching of partial results and graceful degradation of accuracy. Finally, runtimes such as CoreML [38] and Windows ML [177] provide on-device inference engines and accelerators. To our knowledge, only single operator optimizations are enforced (e.g., using target mathematical libraries or hardware), while neither end-to-end nor multi-model optimizations are used. As PRETZEL, TVM [163, 33] provides a set of logical operators and related physical implementations, backed by an optimizer based on the Halide language [134]. TVM is specialized on neural network models

and does not support featurizers nor "classical" models.

5.6.1 Optimization of ML Pipelines:

There is a recent interest in the ML community in building languages and optimizations to improve the execution of ML workloads [163, 116, 32, 161, 78]. However, most of them exclusively target NNs and heterogeneous hardware. Nevertheless, we are investigating the possibility to substitute Flour with a custom extension of Tensor Comprehension [165] to express featurization pipelines. This will enable the support for Neural Network featurizers such as word embeddings, as well as code generation capabilities (for heterogeneous devices). We are confident that the set of optimizations implemented in Oven generalizes over different intermediate representations.

Uber's Michelangelo [109] has a Scala DSL that can be compiled into bytecode which is then shipped with the whole model as a zip file for prediction. Similarly, H2O [58] compiles models into Java classes for serving: this is exactly how ML.Net currently works. Conversely, similar to database query optimizers, PRETZEL rewrites model pipelines both at the logical and at the physical level. KeystoneML [155] provides a high-level API for composing pipelines of operators similarly to Flour, and also features a query optimizer similar to Oven, albeit focused on distributed training. KeystoneML's cost-based optimizer selects the best physical implementation based on runtime statistics (gathered via sampling), while no logical level optimizations is provided. Instead, PRETZEL provides end-to-end optimizations by analyzing logical plans [42, 80, 117, 22], while logical-to-physical mappings are decided based on stage parameters and statistics from training. Similarly to the SOFA optimizer [139], we annotate transformations based on logical characteristics. MauveDB [44] uses regression and interpolation models as database views and optimizes them as such. MauveDB models are tightly integrated into the database, thus only a limited class of declaratively definable models is efficiently supported. As PRETZEL, KeystoneML and MauveDB provide sub-plan materialization.

5.6.2 Scheduling:

Both Clipper [40, 36] and Rafiki [170] schedule inference requests based on latency targets and provide adaptive algorithms to maximize throughput and accuracy while minimizing stragglers, for which they both use ensemble models. These techniques are external and orthogonal to the ones provided in PRETZEL. To our knowledge, no model serving system explored the problem of scheduling requests while sharing resource between models, a problem that PRETZEL addresses with techniques similar to distributed scheduling in cloud computing [124, 183]. Scheduling in white box prediction serving share similarities with operators scheduling in stream processing systems [17, 164] and web services [174].

5.7 Conclusion

Inspired by the growth of ML applications and MLaaS platforms, we identified how existing systems fall short in key requirements for ML prediction-serving, disregarding the optimization of model execution in favor of ease of deployment. Conversely, this work casts the problem of serving inference as a database problem where end-to-end and multi-query optimization strategies are applied to ML pipelines. To decrease latency, we have developed an optimizer and compiler framework generating efficient model plans end-to-end. To decrease memory footprint and increase resource utilization and throughput, we allow pipelines to share parameters and physical operators, and defer the problem of inference execution to a scheduler that allows running multiple predictions concurrently on shared resources.

Experiments with production-like pipelines show the validity of our approach in achieving an optimized execution: PRETZEL delivers order-of-magnitude improvements on previous approaches and over different performance metrics. These results were accepted in the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI) [131].

As an immediate future work, we are considering to expand the Scheduling layer with runtime decision capabilities to achieve higher automation. For example, we can consider the size of each pipeline and the number of requests arriving to each pipeline to schedule their execution, controlling at runtime the resources available to each stage according to the current requirements. Furthermore, we can assign differentiated resources at the stage level, since some common stages may be shared among many DAGs and thus receive many more incoming requests than others. This mechanism will likely allow higher throughput than the base OS scheduler, which is unaware of the shared stages and of their different requirements.

To target FPGA-based systems with PRETZEL, as in chapter 4, the Scheduling layer can be augmented with FPGA control and decide to schedule some stages on such resources, taking in account the programming cost (in the order of hundreds of milliseconds) and the data-movement costs.

PRETZEL is a step forward towards the vision in section 1.3, and is designed to be the runtime powering an MLaaS infrastructure: since PRETZEL can control all aspects of the application execution, it is able to also predict and control the amount of resources needed, allowing the lower layers (like an OS modified as in chapter 3) to enforce isolation.

CHAPTER 6

The case study of RegEx acceleration on FPGA

As from section 2.3, REs are widely used to find patterns among data, like in genomic markers research for DNA analysis, deep packet inspection or signature-based detection for network intrusion detection system. They are also widely employed within DA and ML pre-processing phases, and therefore are first candidates to be accelerated. An accelerated implementation can then be exposed to applications, or can be chosen by a FaaS runtime like the one in chapter 5 to transparently offload some operations (or stages, in PRETZEL's design). Due to the diversity of applications, such an accelerator must retain flexibility in order to cover a wide range of usages.

This chapter investigates a novel and efficient RE matching architecture for FPGAs, based on the concept of matching core proposed in [127]. To detail the work, this chapter starts from the background introduced in section 2.3 to motivate the importance of efficient RE matching as discussed in section 6.1. Then, section 6.2 reviews the state of the art over RE matching techniques, and section 6.3 goes into details about the proposed solution, showing its design principles and the novelties with respect to previous approaches. Then, section 6.4 evaluates the proposed design on the target platforms, and projects those results to larger systems in order to investigate other applicative scenarios and to compare against other works. Finally, section 6.5 draws final conclusions over the proposed work and highlights potential future works in the same direction.

6.1 Approach and achievements

As from [127], RE can be software-compiled into sequences of basic matching instructions that a matching core runs on input data, and can be replaced to change the RE to be matched. Building on this approach, we investigate how accelerate REs while retaining programming flexibility, in order to cover a broad range of workloads. To achieve this goal, we build on [127] and overcome its shortcomings by expanding the capabilities of the matching core and by exploring the design of a multi-core architecture.

With respect to [127], this work makes two major improvements towards bringing acceleration capabilities to data-intensive applications with RE matching:

- we devised a better pre-processing mechanism for the execution of RE and a renewed design of the single-core with respect to [127].
- we designed a scalable multi-core architecture that is able to perform RE matching, achieving over 100x speedup with 16 cores over a software RE matcher compiled from GNU Flex [50] while running at 130MHz.
- we realized a prototype with a cross-platform design easily embeddable within heterogeneous architectures. By implementing an 8-core architecture running at 70MHz on a PYNQ-Z1 board we reached a maximum speedup of 37.1x with respect to the same software solution.

The system can easily be deployed to different platforms, like a Xilinx VC707 board for server-like scenarios and a Xilinx PYNQ-Z1 embedded platform, obtaining relevant results in terms of throughput and low area utilization on both of them, while making no changes to the architecture of the single-core. The proposed architecture scales up with the available resources and is customizable to multiple usage scenarios: our experiments, compared to a state-of-the-art software solution, reach speedups over 100x, while running at 130MHz, over a Flex-based matching application [50] running on an Intel i7 CPU at 2.8GHz.

6.2 Related work

Scientific literature addressed the RE matching problem in several ways, also leveraging different technologies based on CPUs, GPUs, FPGAs and ASICs. The vast majority of research works on pattern recognition are based on Deterministic Finite Automata, which encode in a fixed data structure the transitions between accepted characters. This simplicity comes at the cost of a memory usage that grows with the size and complexity of the RE, and - in the basic implementation - allows fetching only one character at a time. Research works that addressed the space issue are generally based on more efficient representations of the transitions.

For example, works like [89] and [96] find large room to group state transitions together, either by identifying common transitions between states or by grouping them according to the cluster of states they lead to. Similarly, [75] clusters states and groups transitions according to the final target cluster, and employs the clustering information to improve the locality of transitions accessing on an FPGA prototype. Some works focus on optimizing the DFAs representation to specifically target ASIC technologies. As an example, [98] encodes state transitions as rules, which group together similar transitions into a rule that defines a set of conditions for the current state and input values in order select the state, thereby compressing the DFA representation. Then, by mapping these rules into Static Random-Access Memory (SRAM) memory by means of proper hash functions, [98] achieved a very short and predictable memory lookup latency, and a relatively small logic footprint. With a radically different approach, [53] implements in ASIC a solution based on the Aho-Corasick string matching algorithm [5], overcoming its high memory requirements by splitting each character input in single bits. Therefore, each bit has only two possible transitions, which are easily stored in a lookup memory and evaluated in parallel. The matching strings that result from these lookups are then disjointed in a tree-like fashion, which is easily implementable in hardware.

To address the performance limitation of analyzing only one character at a time, the research also proposed several works that tackle this issue in multiple ways. Among DFA-based solutions, [180] parallelizes character processing by first computing all the possible transitions of input characters in parallel, and then by merging them in successive steps, all in a pipelined fashion. This parallelization scheme is independent from the DFA encoding, and [180] evaluates its approach with ClusterFA [74] DFA compression, achieving over an order of magnitude speedup on FPGA. In [106], the authors propose a solution based on Ternary Content Addressable Memorys (TCAMs), which encodes the transitions of the DFA to improve the lookup process.

Another body of work is still based on DFA, but leverages hardware parallelism to increase the matching performance, without however changing the DFA representation with respect to standard approaches. As an example, [13] uses Non-deterministic Finite Automaton (NFA) to tackle one of the main performance limitation of DFA approaches. Since the initial position of a matching string within a larger string is known only when the match occurs, conventional DFA-based solutions have to either repeat the matching process for each possible initial character or to save an unbounded number of possible initial positions, until a match occurs. To overcome these shortcomings, [13] activates a new DFA for a given RE whenever an initial character is found, finally selecting the first DFA advertising a match. While this solution greatly improves performance through parallelism, it does not scale well to complex REs with Kleene operators nor to a workload composed on multiple REs to match against. Instead, [166] leverages the characteristics of GPUs to match multiple REs in parallel, encoding each RE as a separate DFA.

Different approaches do not rely on the use of Finite State Automata but focus on

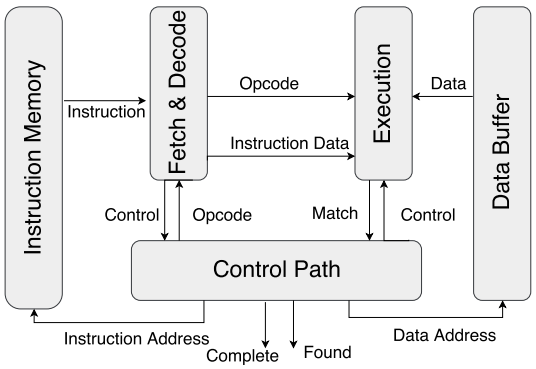

Figure 6.1: *High level of the single-core architecture*

achieving an efficient lookup process. For example, [3] proposes a hash-based encoding scheme for text patterns (not specifically REs) that generates a dictionary matching engine. Instead, [118] employs bitmap index structures to encode the strings to match against, which is very efficient on FPGA logic and allows looking up multiple characters at a time.

The approach that we are going to adopt has been developed in [127] and the idea is to treat REs as programs that are composed by instruction sequences executed on a customized processor over a stream of data. REs are compiled mich like a programming language, producing a sequence of instructions that will drive a dedicated CPU. This solution does not build on an FSA as most solutions present in the literature, but allows quick "reconfiguration" of the matching core by simply changing the instruction sequence, without changing the configuration of the FPGA fabric: indeed, instructions are stored in a dedicated on-chip Block Random Access Memory (BRAM), whose content can easily be written by the system's CPU.

6.3 Design Methodology

This section shows the design of the proposed architecture and how it can be deployed as single core and as multi-core. Section 6.3.1 shows the overall components of the system and how they interact to run the matching program, whose extensions with respect to [127] are highlighted in section 6.3.2. Section 6.3.3 shows the internal design of the single core and highlights the differences with respect to [127], and, finally, section 6.3.4 explains the multi-core implementation, computing the expected throughput as the number of cores scales.

Table 6.1: *Opcode encoding of the instructions.*

opcode	RE	Description
000000	NOP	No Operation
100000	(	enter subroutine
010000	AND	and of cluster matches
001000	OR	or of cluster matches
011000	.	match any character
000001	)*	match any number of sub-RE
000010	)+	match one or more sub-RE
000011	)|	match previous sub-RE or next one
000100	)	end of subroutine
000101	OKP	Open Kleene Parenthesis
000111	JIM	Jump If Match

6.3.1 RE matching flow and system components

The first step of the matching process is the compilation of the RE to match via a software compiler, whose output is the "program" that TiReX runs to match over the input string. Section 6.3.2 explains the format of the instructions of this program, which are highly customized for the RE matching scenario. This program is loaded in the Instruction Memorys (IMs) shown in fig. 6.1, from which the Fetch and Decode Units (FDUs) loads the single instructions to decode them, propagating the control signals and the instruction operands to the Execution Units (EUs) and the Control Paths (CPs). The EUs loads the input characters from the Data Buffer and searches for patterns in these data according to the signals received from the FDUs, emitting a match signal to the CPs in case the current input characters match the pattern encoded in the current instruction. Finally, the CPs exposes signals to indicate completion of the matching process and the presence of a match.

6.3.2 Instruction set extensions

Table 6.1 shows the TiReX instructions set, which is composed of the basic operators for RE matching and of parenthesis operators to optimize sub-procedures that match an inner RE. The opcode field, represented in table 6.1, is made of 6 bits, while 32 following bits allow to encode for *reference* characters as operands of the instruction; as from [127], the references are the characters to be matched with the instruction, and their number determines how many characters can be matched in parallel (for example, an OR matches if any of its four operands matches while an AND matches if all its four operands matches). As an improvement over [127], we introduced two main modifications to improve the matching performance by avoiding stall cycles or FDUs re-alignment delays in case of match of complex REs. The first modification is the addition of the *Open Kleene Parenthesis* OKP instruction: in case of an RE ending with a Kleene operator "*" or "+", the

compiler translates the opening parenthesis as a nested sub-RE delimited by the new OKP instruction and the final ")*" or ")*+" instruction. The OKP operator instructs the pipeline that the sub-match to be executed can be repeated when the instruction ")*" is reached, thus hinting the internal prefetching logic of the core (described in section 6.3.3) that a jump in the instruction flow may occur to the instruction following the OKP. Similarly to the OKP instruction, the JIM instruction (standing for *Jump IF Match*) instructs the architecture to jump ahead in case of match of a sub-RE chained in OR with other REs (like "(*RE1*)|(*RE2*)|(*RE3*)"). Indeed, in case a match occurs the following sub-REs can be ignored, and the core logic can directly jump to the instruction following the sequence of ORed REs. The OKP instruction uses the reference field to encode the address of the instruction to jump to in case of match.

These modifications have been inspired by the dominant applications RE matching is important for. Indeed, applicative fields like genome analysis and packet inspections typically look for repeated patterns in input data (Kleene operator) or for the occurrence of one among many patterns (chain of ORed REs). In these scenarios, where the sub-REs can be complex and long, we found large room for improvement over the previous approach, potentially saving many cycles of execution time.

6.3.3 Single-Core Architecture

The customized processor, completely designed in VHSIC Hardware Description Language (VHDL), is shown in fig. 6.2 and has a two stage pipeline architecture with an instruction memory and a data buffer, as in Figure 6.1, allowing to load instructions and data at the same clock cycle. The Fetch and Decode stage fetches the instructions from the instruction memory and decodes them, producing three output signals . The first is the opcode, that drives the computation of the following stage by instructing it over the pattern to match (like ANDed or ORed characters). The second output is the reference, which is the set of characters the input should be matched against. The third output is the valid reference, that is whether each character is valid or not in the instruction reference. Indeed, the instruction set allows leaving some characters as blank, for example to match an OR of only two characters (the default reference set being composed of four characters).

By instantiating three different, specialized FDUss (marked A to C in fig. 6.2), which work on different instructions, the core is able to avoid cycle losses in case of no match or in case of special instructions such as the JIM. The first FDUs keeps a copy of the very first program instruction and of its control signals, and is essential in case of mismatch; in such a case, the pipeline should restart the program execution, wasting one cycle to fetch and decode the initial instruction again. This unit is particularly important in case of long sequences of characters that do not match the RE: in this case, the pipeline would restart the execution very often, wasting a large percentage of cycles in stalling to restart the program execution. The second FDUs is dedicated to continuously prefetching the next

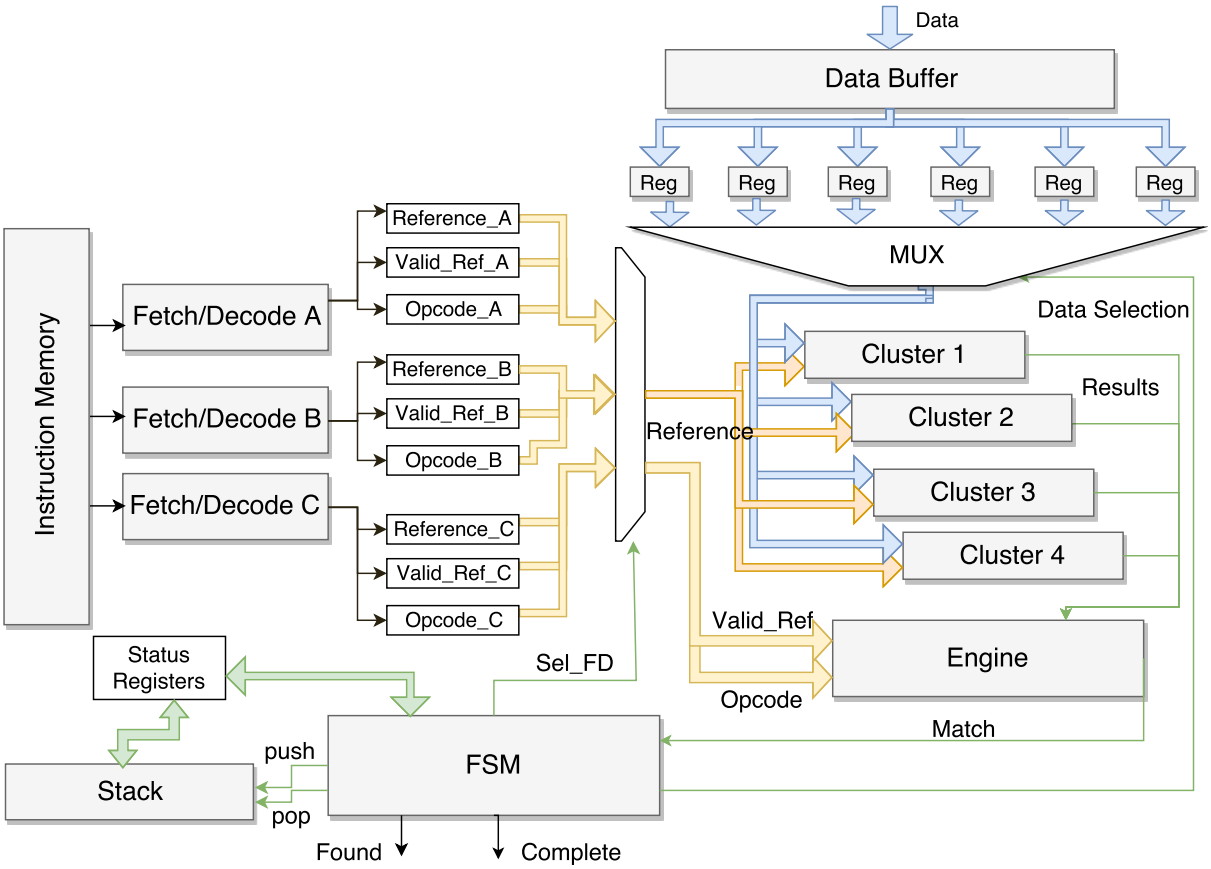

Figure 6.2: *Details of the core logic, with the pipeline components*

instruction, which is essential in case a match is found. Finally, the third FDUs leverages the compiler hints for prefetching, working on the instruction after the sequence of ORed REs (in case of JIM) or on the initial instruction of a sub-RE (in case of OKP).

The Execute stage takes as inputs the signals from the previous stage and a portion of data from the data buffer. The outputs of this component are the match signal and the data offset of the matched character sequence. The EUs performs the basic logic operations like AND or OR between the input data and the reference, according to the opcode and to the reference validity. To increase the number of characters processed in a single clock cycle, four cluster of comparators are present (numbered 1 to 4 in fig. 6.2), each one composed by four comparators. Since the EUs contains most of the combinational components of our design, properly sizing is crucial to the trade off between number of characters processed per clock cycle and the length of the synthesis critical paths, which both contribute to the overall performance. We found out that the adoption of four clusters of four comparators each yields good performance while keeping the critical path low enough to achieve frequencies in the order of 100MHz or more.

As a complement to the Execution logic, the data buffer is augmented with six output registers that store all the possible portions of data that can be requested by the Execution stage. Indeed, depending on the result of an instruction (match/not match) and on the number of characters the instruction is matching (from 1 to 4), different inputs can be necessary. In particular, the number of characters instruction i matches determines how many characters should be fetched for instruction $i+1$ (these numbers being equal). However, the logic to fetch a variable number of characters impacts on the critical path, and is avoided by using 5 different registers storing the possible inputs for instruction $i+1$ (the same 4 characters of instruction i in case of not match, one new character in case of one character matched, 2 new characters in case of 2 characters matching and so on). Finally, one more register is present in case the match fails and the RE is restarted.

On top of all the components, the CPs is responsible to control the overall matching process. It requests the instructions from the instruction memory, determines the right FDUs output for the current clock cycle, arbitrates the data buffer registers and fetches new data from an external data memory (implemented with a BRAMs) to refill the data buffer.

6.3.4 Multi-Core Architecture

In order to parallelize the RE recognition we decide to adopt a multi-core architecture. Depending on the application at hand, the matching process can be performed in two different modes:

1. each core is provided with the same set of instructions but with different portions of the input data; this is useful in scenarios like genome matching, where the amount of

data can largely exceed the number of interesting sequences and potential overlaps can easily be found in a following stage

2. the input data are the same for each core but the sets of instructions are different; this is useful in scenarios like malware detection, where the number of patterns to look for is potentially much larger than the input data

The design consists in multiple instantiations of the core and its data BRAM, resulting in a single private memory multi-core architecture.

In the first operational mode, the compiler splits the input data according to an heuristic that tries to ensure the absence of a matching string in the cut between two consequent chunks of data. The hypothesis behind this heuristic relies on a user-specified threshold that indicates the maximum length of a match, which can be set according to the different use cases of the system. Based on this parameter, the compiler splits the data in portions with overlapping regions whose length is determined by the threshold, and assigns each portion to a core. Equations 6.1, 6.2 and 6.3 formalize the adopted heuristic:

$$B_{size} = \frac{S_{data}}{N_{tc}} \tag{6.1}$$

$$\forall i \; EoD_i = B_{size} \cdot (i+1) + Tr \tag{6.2}$$

$$\forall i \; SoD_i = EoD_{i-1} - Tr \tag{6.3}$$

where B_{size} is the batch size to be loaded into the private memory of each core, S_{data} is the size of the data to be analysed and N_{tc} is the number of cores instantiated in the system. EoD_i refers to the End of Data of the *i-th* core and *Tr* is the user defined threshold of maximum match length. SoD_i instead is the Start of Data for the *i-th* core and is based on the EoD of the previous core. The only exception is of course the SoD of the first core which will be at the beginning of data.

Based on the approach we used, it is possible to obtain theoretical results in terms of throughput of our architecture. Although the performances are highly data dependent, the main parameter is affecting the minimum and maximum throughput values is the number many characters that can be compared for each clock cycle. Our design envisages three different time metrics depending on the state of the FSM:

$$T_{nm} = \frac{1/F}{NCluster} \tag{6.4}$$

$$T_{mc} = \frac{1/F}{ClusterWidth} \tag{6.5}$$

$$T_{mu} = \frac{1}{F} \tag{6.6}$$

Table 6.2: *Performance of the 16-core implementation on the VC707 compared to Flex*

Regular Expression	Flex (Intel i7 2.8GHz)	16-core (130 MHz)	Speedup
ACCGTGGA	271 μs	2.07 μs	130.9x
(TTTT)+CT	121 μs	4.54 μs	26.65x
(CAGT)|(GGGG) |(TTGG)TGCA(C|G)+	263 μs	3.36 μs	78.27x

Table 6.3: *Area Utilization of a single-core implementation on VC707*

VC707 Board	Slice LUTs	Slice Reg.	F7 Muxes
Used	1921	1775	261
Percentage	0.63%	0.29%	0.17%

Where *F* is the system frequency, T_{nm} ($\frac{ns}{char}$) is the throughput reached whenever the data is not matching the RE, T_{mc} represents the situation in which the core is in a matching state and the current instruction is a concatenation and T_{mu} is the last case for which the state is still in the matching state and the instruction is a chain of ORs.

6.4 Experimental Results

The architecture described in section 6.3 has been implemented on two different boards by Xilinx. The first platform is a VC707 board powered by a Virtex-7 FPGAs and the second platform is a Digilent PYNQ-Z1 board powered by a ZYNQ System on Chip comprising an ARM CPUs and a Xilinx FPGAs. On both platforms we used Xilinx Vivado 2016.4 for synthesis and implementation. The design of the VC707 includes a Microblaze soft processing unit, which serves as an interface to our core and feeds it with instructions and data. Instead, on the PYNQ platform the FPGAs is directly attached to and controlled from the ARM processor on the same die.

As a first result, the area utilization of a single-core of the architecture is very small. Indeed, on the VC707 section 6.3.4 shows that a single core uses less than 1% of the resources, while on the PYNQ section 6.4 shows an area utilization of less than 4% of them. Moreover, in both cases the bottleneck resource is the amount of LUTs, which are increasingly available in recent FPGAss thanks to the advancements of lithography. This bottleneck is due to the control-bounded nature of REs matching, which requires mostly logic to control the REs matching process.

We performed tests with multiple cores in order to use a significant portion of the available FPGAs area. Tables 6.2 and 6.4 show the results in terms of speedup with respect

Table 6.4: *Performance of the 8-core implementation on the PYNQ compared to Flex*

Regular Expression	Flex (Intel i7 2.8GHz)	8-core (70 MHz)	Speedup
ACCGTGGA	271 μs	7.2 μs	37.63x
(TTTT)+CT	121 μs	8.21 μs	14.73x
(CAGT)|(GGGG) |(TTGG)TGCA(C|G)+	263 μs	30.3 μs	8.67x

Table 6.5: *Area Utilization of a single-core implementation on PYNQ*

PYNQ Board	Slice LUTs	Slice Reg.	F7 Muxes
Used	1845	1775	261
Percentage	3.46%	1.66%	0.98%

to Flex [50] for the VC707 platform and the PYNQ platform, respectively. Flex is a tool that encodes an input REs to be matched into a full-fledged C program, which is based on an optimized Finite State Machine specifically tailored to the input REs. The program obtained from Flex was compiled with best optimizations and run on an Intel i7 CPUs with a peak frequency of 2.8 GHz. The input REss are shown in the leftmost column of table 6.2 and of table 6.4, and are inspired to DNA-matching applications of increasing complexity: they are are an example of recent DA workload that is is gaining increasing interest and market. As input text we used the first 16KB of the Homo Sapiens chromosome [21], which contains at most 3 matches for each REs. This small number of matches represents a worst-case scenario for REs matching applications, as the matching engine has to examine long input sequences which do not match, and has to frequently restart the matching process with new incoming characters, therefore stressing the control capabilities of the architecture. To leverage the available area, we deployed multiple cores on each platform, in order to parallelize the matching process for the same string: each core matches against the same REs and receives a portion of the input string, and the result from the first matching core is returned. To balance the overhead of distributing REs instructions and data to the cores and the input string, we synthesized 16 cores on the VC707 platform reaching a frequency of 130MHz. Similarly, we ran the same tests on a Digilent PYNQ-Z1 board. Since PYNQ has fewer resources than the VC707, we could instantiate up to 8 cores on PYNQ, reaching a frequency of 70MHz, and we ran the same tests we performed on the VC707 with the results in table 6.4. The VC707 16-core system at 130MHz achieved 16.64 *Gb/s* as worst throughput and 66.56 *Gb/s* in the best case, while the 8-core implementation on the PYNQ at 70MHz resulted in bitrates ranging from 4.48 *Gb/s* - 17.92 *Gb/s*.

Table 6.6: *Bitrate comparisons with the state-of-the-art*

Solution	Clock Frequency [*MHz*]	Bitrate [*Gb/s*]
VC707 16-core	130	16.64 - 66.54
PYNQ 8-core	70	4.48 - 17.92
[127]	318.47	10.19 - 18.18
[180]	-	3.39 - 29.59
[75]	150	230 - 430
[106]	-	10 - 19
[98]	2300	20 - 40
[53] (FPGA)	100	3.2
[53] (ASIC)	1000	256
[13]	250	1 - 16
[3]	250	12.16
[118]	100	11.98
[166]	-	16

Finally, table 6.6 includes the bitrates and the working frequencies (where available) of the main works reviewed in section 6.2 together with the results achieved by our work, which compare favourably against most of the works reviewed in terms of pure performance. In the cases where the results were not directly comparable, we computed the achievable bitrate from the time to match a character (extracted from the cited paper) and from the number of cores, which we assumed to be 8. In this way, we are implicitly assuming that the matching performance of reviewed works scale linearly with the number of cores, which is a reasonable assumption if the proposed solutions can do parallel matching on slices of the input (as we did). To compare against the PYNQ, we assumed 8 cores to be deployed.

$$B_x = \frac{1}{T_x} \cdot 8 \cdot 10^9 \cdot N_{tc} \tag{6.7}$$

Comparing the reviewed works against ours, we can note that the works that outperform ours are based on better hardware implementations, typically because they have higher operating frequencies (and are mostly ASIC) and they embed the REs into the control logic, thus renouncing to the flexibility our work retains.

6.5 Conclusions

This work presented a multi-core architecture tailored to pattern matching, implemented on a high-level FPGA and an System On Chip (SoC). Thanks to the approach adopted, the system achieved remarkable results both in terms of resource utilization and in terms of performance. The implementation on the PYNQ platform demonstrated the possibility to easily integrate the design in SoCs and Heterogeneous architectures. This work was

published and presented in the 25th edition of Reconfigurable Architectures Workshop (RAW).

Future works on improvements move along several directions. The first direction to work on to directly improve performance is to increase the frequency of the entire system, so as to get a proportional performance increase. With such an increase in throughput and broader applicative scenarios, the system should sustain an appropriate data rate to feed input characters to the matching engine, which will possibly require exploring new memory organizations, like data prefetchers or caches. Similarly, as many applications need matching multiple REs in parallel, a multi-core system requires an appropriate interconnection to feed the cores with data and collect match results: in this direction, Network-on-Chip solutions look a viable solution to explore in order to balance the performance with the area occupation and the routing issues. Finally, to expose the accelerators to users' applications, a standard interface is required, like a PCIe interface that applications can talk to; the integration with a PCIe subsystem would allow the deployment of our solution to environments like AWS F1 FPGA-based solutions, which are based on Xilinx SDAccel toolchain and are designed for cloud-like scenarios. This would be the starting point for embracing RE acceleration within a FaaS infrastructure and transparently offload some of the heaviest pre-processing stages of DA and ML applications.

CHAPTER 7

Conclusions and Future Work

This chapter concludes this thesis by reviewing its achievements in section 7.1 and by discussing limitations and future work in section 7.2.

7.1 Achievements

This work addressed the main limitations of data-intensive applications, especially for DA and ML, focusing the analysis on the recent CPUs and on cloud-like settings. It tackled the two main issues of this kind applications while running on modern hardware platforms: the co-location and resource sharing issue and the optimization of users' application in a transparent way. Chapter 3 addressed the first issue by introducing a general-purpose mechanism for LLC partitioning that requires software-only modifications. The following chapters chapter 4 and chapter 5 investigated the main issues of optimizing ML applications on modern hardware and proposed insights and solutions that are applicable to a large set of data-intensive applications, implementing the novel solutions in a prototype like PRETZEL. Then, chapter 6 explored a promising enhancement to PRETZEL by investigating an FPGA-based solution for RE matching to that can be deployed in cloud-like scenarios like FaaS and MLaaS; since it is based on an architecture that achieves the required performance while retaining large flexibility, the solution proposed in chapter 6 is thus suitable for a large class of applications.

In our vision, this works provides the foundations to build more efficient FaaS systems for data-intensive applications that can make *most* efficient use of CPUs, guarantee isolation and thus QoS and also flexibly scale to heterogeneous solutions for higher levels of performance. The key insight, from chapter 4, is the necessity of a white-box approach for this class of application, due to their structure composed of multiple diverse operations with different performance characteristics, as from section 2.1.2. Since the DAG of operations can be arbitrarily complex and diverse within data-intensive applications, only a white-box, deep visibility over the application internals can ensure that an automatic optimization approach is effective and general, while the internals being expressed with the granularity of high-level operators also ensures portability to different hardware and software platforms, like offloading to FPGA accelerators. Therefore, we envision this approach and these abstractions to become fundamental guidelines for future FaaS systems for data-intensive applications.

7.2 Limitations and future works

To realistically move towards the goal of a FaaS system for data-intensive applications that is able to meet SLAs employing the solutions this work proposes, the white-box approach and the operator-level of abstraction are the starting point. Indeed, operators characteristics are known upfront and can be modelled within an appropriate analytical framework, building on the considerations already discussed at the end of chapter 5: from here, operators' information can be combined together and extended, for example with locality information to devise the amount of LLC needed. Although this thesis laid the foundations for this future endeavour, combining the proposed solutions into a single system, further augmented to cover the large class of data-intensive applications, still requires a lot of research effort. The literature only partially covers some relevant aspects.

Works like Halide [134] already use operators' information to schedule the computation on multiple cores of a CPU and on multiple nodes of a cluster: a similar approach is needed for the broader class of data-intensive applications, as Halide is designed only for image processing. Therefore, and end-to-end optimization and implementation flow, embodied in a runtime system that is generic enough to cover data-intensive applications and heterogeneous resources is still ahead. In particular, the next step that is missing is an optimizer that covers all the relevant ML and DA operations. In our opinion, the Oven optimizer introduced in section 5.3 is the starting point, but needs more generalization. Indeed, so far it covers only a subset of very common operators in ML.Net; furthermore, the ever-expanding and ever-changing implementations of today's frameworks for ML and DA suggest optimizations strategies to have a solid theoretical and analytical base, as hand-written rules would not keep up with the developments of these fields. The most promising future direction is thus to take further inspiration from the Database community, as from PRETZEL's design guidelines, and explore porting more and more of their solutions to this

larger field. A possible difference lies in the higher diversity of data-intensive kernels: while SQL operators are limited in number and can be naturally described in terms of *Relational Algebra*, ML and DA operations are much more variegated. As a promising approach, we argue that many such operations can be described in terms of *Tensor Algebra*, as some works like [33] already do, despite limited to NNs. In our opinion, this research direction is very promising and can lead to a large body of work.

A second aspect this work shed limited light on is on how to leverage heterogeneity, enhancing the envisioned system to also consider very diverse implementations. The literature shows possible solutions that move towards different directions.

The first direction is a thorough modelling of the hardware behavior thanks to a deep knowledge of the platform, for example via the Extended Roofline Model [27], which can guide backend compilers towards the best implementation and predict the final performance also in presence of GPUs [119]. This would allow computing the resource and runtime costs of each implementation and use this information for scheduling and mapping inside evolutions of PRETZEL's Oven optimizer and Scheduler. However, especially FPGAs make the prediction very complex, as the Roofline model needs to explore a larger solutions space that is not known a priori [151]. While further efforts are needed to improve prediction capabilities on this specific front, the problem can be alleviated with Domain-Specific Architectures (DSAs), whose known architecture reduces the exploration space to the architecture parameters. Although DSAs are a promising solution for future heterogeneous platforms [1] , they are still a quickly evolving research field with no established guidelines nor tools: thus, predictions on possible research paths towards this direction are — in our opinion — very premature at the moment of writing.

The second direction is the one [33] takes, and relies on the prediction capabilities of ML models to select the best implementation based on previous experience. Indeed, universal approximators like NNs can be effective in predicting the non-linear behavior of complex hardware configurations (taking in account computing units, memory hierarchy, interconnection topology, bandwidth, etc.) starting from an input application, and the initial results of [33] are promising. This black-box approach can perhaps apply to those platform with a very complex, and sometimes not analytically describable, solution space to explore, where analytical models like the Roofline have limited effectiveness.

Since both the black box approaches to optimization and implementations and the analytical approach (possibly limited via DSAs-based templates) are still very hot research topics, it is not possible to identify the most promising research direction. The road towards efficient, heterogeneous systems for data-intensive applications is large and exciting.

[1] Hennessy and Patterson, in their lecture for the Turing award at ISCA2018 [77], also advocate for higher levels of abstraction to allow using DSAs for offloading to accelerators. The reader can note that the TiReX architecture of chapter 6 is essentially a DSA for REs.

List of abbreviations

A

AC	Attendee Count. 74–78, 80
AOT	Ahead-Of-Time. 65, 66, 70, 74, 77
API	Application Programming Interface. 6, 50, 60, 66, 67, 78, 82
ASIC	Application-Specific Integrated Circuit. 5, 17, 18, 86, 87, 96

B

BRAM	Block Random Access Memory. 88, 92, 93

C

CDF	Cumulative Distribution Function. 62
CNN	Convolutional Neural Network. 13
CP	Control Path. 89, 92
CPU	Central Processing Unit. I, II, VII, 2, 4, 5, 7, 9–13, 15–21, 24, 26–28, 30, 33, 38, 45, 49–51, 53–59, 64–66, 68, 71, 74, 75, 78, 79, 86, 88, 94, 95, 99, 100

D

DA	Data Analytics. I, 2, 3, 5, 7, 9, 12, 13, 38, 49, 50, 58, 85, 95, 97, 99–101

DAG Direct Acyclic Graph. 13, 15, 49, 50, 52, 54, 60, 62, 64, 66–68, 70, 71, 83, 100
DBMS DataBase Management System. 13
DFA Deterministic Finite Automaton. 86, 87
DMA Direct Memory Accesses. 37
DNN Deep Neural Network. 13, 14
DSA Domain-Specific Architecture. 101
DSL Domain-Specific Language. 13, 82

E
ETL Embed, Transform, Load. 49
EU Execution Unit. 89, 92

F
FaaS Framework-as-a-Service. 6, 47, 53, 58, 85, 97, 99, 100
FDU Fetch and Decode Unit. 89, 90, 92
FPGA Field Programmable Gate Array. 5, 7, 9, 15, 17, 18, 49–51, 53–57, 59, 83, 85–88, 94, 96, 97, 99–101
FSA Finite State Automaton. 87, 88
FSM Finite State Machine. 55, 93

G
GC Garbage Collection. 81
GPU Graphic Processing Unit. 6, 17, 18, 86, 87, 101

H
HLS High Level Synthesis. 56, 57

I
IaaS Infrastructure-as-a-Service. 47
IDS Intrusion Detection Systems. 5, 16, 17
ILP Integer Linear Programming. 30–32
IM Instruction Memory. 89
IMLT Internal Machine Learning Toolkit. 14, 50–52, 54, 56, 57, 60
IPC Instructions Per Cycle. 24, 26

J

JIT Just In Time. 54, 58, 60, 62, 63, 65, 72, 77

L

L2 Level 2. 22, 23, 26–28, 32, 39

LLC Last Level Cache. II, 4, 7, 10, 11, 19–28, 32, 34, 37–39, 41, 43, 45–47, 99, 100

LRU Least Recently Used. 23, 24, 73

LUT Look-Up Table. 94, 95

M

ML Machine Learning. I, 2, 3, 5–7, 9, 13–17, 49–53, 57–63, 66, 75, 80–83, 85, 97, 99–101

MLaaS Machine-Learning-as-a-Service. 6, 50, 58, 83, 99

MPC Model Plan Compiler. 70, 71

N

NFA Non-deterministic Finite Automaton. 87

NN Neural Network. 3, 12, 13, 60, 82, 101

NUMA Non Uniform Memory Access. 81

O

OS Operating System. II, 7, 10, 11, 21–23, 26, 45, 47, 64, 83

OSDI Operating Systems Design and Implementation. 83

P

PaaS Platform-as-a-Service. 5, 6, 47

PCA Principal Components Analysis. 73, 75

Q

QoS Quality of Service. II, 5–7, 10, 19, 23, 26, 38, 58, 100

R

RAM Random Access Memory. 28, 30, 32, 39, 43, 55–57

RAW Reconfigurable Architectures Workshop. 97
RE Regular Expression. II, 5, 7, 9, 16–18, 85–90, 92, 94–97, 99, 101
RPC Remote Procedure Call. 51, 81

S
SA Sentiment Analysis. 52, 61–63, 67, 74–78
SaaS Software-as-a-Service. 6
SIMD Single Instruction, Multiple Data. I, 4, 5, 12, 68
SLA Service Level Agreement. 50, 51, 54, 63, 100
SoC System On Chip. 96
SRAM Static Random-Access Memory. 87

T
TCAM Ternary Content Addressable Memory. 87
TCO Total Cost of Ownership. 1, 7, 55
TF TensorFlow. 15, 16, 51, 81
TPU Tensor Processing Unit. 6

V
VHDL VHSIC Hardware Description Language. 90
VM Virtual Machines. 26

W
WSL Windows Subsystem for Linux. 74

Bibliography

[1] *.Net Core Ahead of Time Compilation with CrossGen*. 2018. URL: https://github.com/dotnet/coreclr/blob/master/Documentation/building/crossgen.md.

[2] Deepak Agarwal et al. "LASER: A Scalable Response Prediction Platform for Online Advertising". In: New York, New York, USA, 2014. ISBN: 978-1-4503-2351-2. DOI: 10.1145/2556195.2556252. URL: http://doi.acm.org/10.1145/2556195.2556252.

[3] Kanak Agarwal and Raphael Polig. "A high-speed and large-scale dictionary matching engine for information extraction systems". In: *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. IEEE. 2013, pp. 59–66.

[4] Zeeshan Ahmed and et al. "Machine Learning for Applications, not Containers (under submission)". In: 2018.

[5] Alfred V. Aho and Margaret J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search". In: *Commun. ACM* 18.6 (1975), pp. 333–340. ISSN: 0001-0782. DOI: 10.1145/360825.360855. URL: http://doi.acm.org/10.1145/360825.360855.

[6] David H. Albonesi. "Selective Cache Ways: On-demand Cache Resource Allocation". In: *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 32. Haifa, Israel: IEEE Computer Society, 1999, pp. 248–259. ISBN: 0-7695-0437-X. URL: http://dl.acm.org/citation.cfm?id=320080.320119.

[7] Jay A Alexander and MC Mozer. "Template-based procedures for neural network interpretation". In: *Neural Networks* 12.3 (1999), pp. 479–498.

[8] Ethem Alpaydin. *Introduction to Machine Learning*. 2nd. The MIT Press, 2014. ISBN: 026201243X, 9780262012430.

[9] Ethem Alpaydin. "Multilayer Perceptrons". In: *Introduction to Machine Learning*. 2nd. The MIT Press, 2014. Chap. 11. ISBN: 026201243X, 9780262012430.

[10] *Amazon SageMaker*. URL: https://aws.amazon.com/sagemaker/.

[11] *AMD Takes Computing to a New Horizon with Ryzen Processors*. 2016. URL: https://www.amd.com/en/press-releases/amd-takes-computing-2016dec13.

[12] Michael Armbrust et al. “Spark SQL: Relational Data Processing in Spark”. In: Melbourne, Victoria, Australia, 2015. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742797. URL: http://doi.acm.org/10.1145/2723372.2742797.

[13] Kubilay Atasu et al. “Hardware-accelerated regular expression matching for high-throughput text analytics”. In: *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE. 2013, pp. 1–7.

[14] *AWS Lambda*. URL: https://aws.amazon.com/lambda/.

[15] *Azure Data Lake Analytics*. URL: https://azure.microsoft.com/en-in/services/data-lake-analytics/.

[16] *Azure Machine Learning service*. URL: https://azure.microsoft.com/en-in/services/machine-learning-service/.

[17] Brian Babcock et al. “Operator Scheduling in Data Stream Systems”. In: *The VLDB Journal* 13.4 (Dec. 2004), pp. 333–353. ISSN: 1066-8888. DOI: 10.1007/s00778-004-0132-6. URL: http://dx.doi.org/10.1007/s00778-004-0132-6.

[18] *Batch Python API in Microsoft Machine Learning Server*. 2018. URL: https://docs.microsoft.com/en-us/machine-learning-server/operationalize/how-to-consume-web.

[19] S. Beamer, K. Asanovic, and D. Patterson. “Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server”. In: *2015 IEEE International Symposium on Workload Characterization*. Oct. 2015, pp. 56–65. DOI: 10.1109/IISWC.2015.12.

[20] Christian Bienia et al. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’08. Toronto, Ontario, Canada: ACM, 2008, pp. 72–81. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454128. URL: http://doi.acm.org/10.1145/1454115.1454128.

[21] National Center for Biotechnology Information. *Homo Sapiens Chromosome Dataset*. ftp://ftp.ncbi.nlm.nih.gov/genomes/H_sapiens/CHR_01/hs_alt_CHM1_1.1_chr1.fa.gz.

[22] Peter A. Boncz, Marcin Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution.” In: 2005, pp. 225–237. URL: http://dblp.uni-trier.de/db/conf/cidr/cidr2005.html#BonczZN05.

[23] Uday Bondhugula et al. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *Acm Sigplan Notices*. Vol. 43. 6. ACM. 2008, pp. 101–113.

[24] Brian K. Bray, William L. Lunch, and Michael J. Flynn. *Page Allocation to Reduce Access Time of Physical Caches*. Tech. rep. Stanford, CA, USA, 1990.

[25] Jacob Brock et al. “Optimal Cache Partition-Sharing”. In: *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*. ICPP ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 749–758. ISBN: 978-1-4673-7587-0. DOI: 10.1109/ICPP.2015.84. URL: http://dx.doi.org/10.1109/ICPP.2015.84.

[26] S. Byna, Yong Chen, and Xian-He Sun. “A Taxonomy of Data Prefetching Mechanisms”. In: *Parallel Architectures, Algorithms, and Networks, 2008. I-SPAN 2008. International Symposium on*. May 2008, pp. 19–24. DOI: 10.1109/I-SPAN.2008.24.

[27] V. C. Cabezas and M. PÃ$\frac{1}{4}$schel. "Extending the roofline model: Bottleneck analysis with microarchitectural constraints". In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. Oct. 2014, pp. 222–231. DOI: 10.1109/IISWC.2014.6983061.

[28] *Caffe2*. 2017. URL: https://caffe2.ai.

[29] S. Chakraborty et al. "Interpretability of deep learning models: A survey of results". In: *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. Aug. 2017, pp. 1–6. DOI: 10.1109/UIC-ATC.2017.8397411.

[30] Vicki S Chambers et al. "High-throughput sequencing of DNA G-quadruplex structures in the human genome". In: *Nature biotechnology* 33.8 (2015), p. 877.

[31] Dhruba Chandra et al. "Predicting inter-thread cache contention on a chip multi-processor architecture". In: *11th International Symposium on High-Performance Computer Architecture*. Feb. 2005, pp. 340–351. DOI: 10.1109/HPCA.2005.27.

[32] Tianqi Chen et al. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *CoRR* (2015). arXiv: 1512.01274. URL: http://arxiv.org/abs/1512.01274.

[33] Tianqi Chen et al. "TVM: End-to-End Optimization Stack for Deep Learning". In: *CoRR* (2018). arXiv: 1802.04799. URL: http://arxiv.org/abs/1802.04799.

[34] The University of Chicago Medical Center. *History of Personalized Medicine Brings Future Hope to Lung Cancer Patients*. http://www.uchospitals.edu/specialties/cancer/patient-stories/victor-lung.html.

[35] Rada Chirkova and Jun Yang. "Materialized Views". In: *Foundations and Trends in Databases* 4.4 (2012), pp. 295–405. ISSN: 1931-7883. DOI: 10.1561/1900000020. URL: http://dx.doi.org/10.1561/1900000020.

[36] *Clipper*. 2018. URL: http://clipper.ai/.

[37] Henry Cook et al. "A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy - efficiency While Preserving Responsiveness". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: ACM, 2013, pp. 308–319. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485949. URL: http://doi.acm.org/10.1145/2485922.2485949.

[38] *Core ML*. 2018. URL: https://developer.apple.com/documentation/coreml.

[39] Dan Crankshaw and Joseph Gonzalez. "Prediction-Serving Systems". In: *Queue* 16.1 (Feb. 2018), 70:83–70:97. ISSN: 1542-7730. DOI: 10.1145/3194653.3210557. URL: http://doi.acm.org/10.1145/3194653.3210557.

[40] Daniel Crankshaw et al. "Clipper: A Low-latency Online Prediction Serving System". In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. Boston, MA, USA: USENIX Association, 2017, pp. 613–627. ISBN: 978-1-931971-37-9. URL: http://dl.acm.org/citation.cfm?id=3154630.3154681.

[41] Daniel Crankshaw et al. "The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox". In: *CIDR*. 2015.

[42] Andrew Crotty et al. “An Architecture for Compiling UDF-centric Workflows”. In: 8.12 (Aug. 2015), pp. 1466–1477. ISSN: 2150-8097. DOI: 10.14778/2824032.2824045. URL: http://dx.doi.org/10.14778/2824032.2824045.

[43] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-aware scheduling for heterogeneous datacenters”. In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 77–88.

[44] Amol Deshpande and Samuel Madden. “MauveDB: Supporting Model-based User Views in Database Systems”. In: Chicago, IL, USA, 2006. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142483. URL: http://doi.acm.org/10.1145/1142473.1142483.

[45] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. “SRM-Buffer: An OS Buffer Management Technique to Prevent Last Level Cache from Thrashing in Multicores”. In: *Proc. of EuroSys*. 2011.

[46] *Docker*. 2018. URL: https://www.docker.com/.

[47] *EC2 large instances and numa*. 2018. URL: https://forums.aws.amazon.com/thread.jspa?threadID=144982.

[48] H. Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. June 2011, pp. 365–376.

[49] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. “Managing Contention for Shared Resources on Multicore Processors”. In: *Commun. ACM* 53.2 (Feb. 2010), pp. 49–57. ISSN: 0001-0782. DOI: 10.1145/1646353.1646371. URL: http://doi.acm.org/10.1145/1646353.1646371.

[50] *Flex, the fast lexical analyzer generator*. 1987. URL: https://github.com/westes/flex.

[51] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. “Stride Directed Prefetching in Scalar Processors”. In: *Proceedings of the 25th Annual International Symposium on Microarchitecture*. MICRO 25. Portland, Oregon, USA: IEEE Computer Society Press, 1992, pp. 102–110. ISBN: 0-8186-3175-9. URL: http://dl.acm.org/citation.cfm?id=144953.145006.

[52] Amir Gandomi and Murtaza Haider. “Beyond the hype: Big data concepts, methods, and analytics”. In: *International Journal of Information Management* 35.2 (2015), pp. 137–144.

[53] Vaibhav Gogte et al. “HARE: Hardware accelerator for regular expressions”. In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–12.

[54] *Google Cloud AutoML*. URL: https://cloud.google.com/automl/.

[55] *Google CLOUD TPU*. URL: https://cloud.google.com/tpu/.

[56] G. Graefe. “Volcano: An Extensible and Parallel Query Evaluation System”. In: *IEEE Trans. on Knowl. and Data Eng.* 6.1 (Feb. 1994), pp. 120–135. ISSN: 1041-4347. DOI: 10.1109/69.273032. URL: http://dx.doi.org/10.1109/69.273032.

[57] S. Gupta and H. Zhou. “Spatial Locality-Aware Cache Partitioning for Effective Cache Sharing”. In: *2015 44th International Conference on Parallel Processing*. Sept. 2015, pp. 150–159. DOI: 10.1109/ICPP.2015.24.

[58] *H2O*. URL: https://www.h2o.ai/.

[59] Alon Y. Halevy. “Answering Queries Using Views: A Survey”. In: *The VLDB Journal* 10.4 (Dec. 2001), pp. 270–294. ISSN: 1066-8888. DOI: 10.1007/s007780100054. URL: http://dx.doi.org/10.1007/s007780100054.

[60] Ruining He and Julian McAuley. “Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering”. In: *WWW*. Montréal, Québec, Canada, 2016. ISBN: 978-1-4503-4143-1. DOI: 10.1145/2872427.2883037. URL: https://doi.org/10.1145/2872427.2883037.

[61] John L. Henning. “SPEC CPU2006 Benchmark Descriptions”. In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. ISSN: 0163-5964. DOI: 10.1145/1186736.1186737. URL: http://doi.acm.org/10.1145/1186736.1186737.

[62] Andrew Herdrich et al. “Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family”. In: *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 657–668.

[63] C. N Höfer and G. Karagiannis. “Cloud computing services: taxonomy and comparison”. In: *Journal of Internet Services and Applications* 2.2 (Sept. 2011), pp. 81–94. ISSN: 1869-0238. DOI: 10.1007/s13174-011-0027-x. URL: https://doi.org/10.1007/s13174-011-0027-x.

[64] R. Hund, C. Willems, and T. Holz. “Practical Timing Side Channel Attacks against Kernel Space ASLR”. In: *Security and Privacy (SP), 2013 IEEE Symposium on*. May 2013, pp. 191–205. DOI: 10.1109/SP.2013.23.

[65] S Idreos et al. “MonetDB: Two Decades of Research in Column-oriented Database”. In: (2012).

[66] Intel Corp. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. Tech. rep. Apr. 2015. URL: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf.

[67] *Intel Math Kernel Library*. URL: https://software.intel.com/mkl.

[68] *Intel Math Kernel Library Improved Small Matrix Performance Using Just-in-Time (JIT) Code Generation for Matrix Multiplication (GEMM)*. 2018. URL: https://software.intel.com/en-us/articles/intel-math-kernel-library-improved-small-matrix-performance-using-just-in-time-jit-code.

[69] *Introduction to Intel Advanced Vector Extensions*. 2011. URL: https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.

[70] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *Systematic Reverse Engineering of Cache Slice Selection in Intel Processors*. Cryptology ePrint Archive, Report 2015/690. http://eprint.iacr.org/. 2015.

[71] Sarah E Jackson and John D Chester. “Personalised cancer medicine”. In: *International journal of cancer* 137.2 (2015), pp. 262–266.

[72] Aamer Jaleel et al. “High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)”. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA ’10. Saint-Malo, France: ACM, 2010, pp. 60–71. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1815971. URL: http://doi.acm.org/10.1145/1815961.1815971.

[73] *JDK 9 Release Notes*. URL: http://www.oracle.com/technetwork/java/javase/9-notes-3745703.html.

[74] Lei Jiang, Jianlong Tan, and Yanbing Liu. *ClusterFA: a memory-efficient DFA structure for network intrusion detection*. 2012.

[75] Lei Jiang et al. “A fast regular expression matching engine for NIDS applying prediction scheme”. In: *Computers and Communication (ISCC), 2014 IEEE Symposium on*. IEEE. 2014, pp. 1–7.

[76] Xinxin Jin et al. “A Simple Cache Partitioning Approach in a Virtualized Environment”. In: *Proc. of ISPA*. 2009.

[77] *John Hennessy and David Patterson Deliver Turing Lecture at ISCA 2018*. URL: https://www.acm.org/hennessy-patterson-turing-lecture.

[78] Daniel Kang et al. “NoScope: Optimizing Neural Network Queries over Video at Scale”. In: 10.11 (Aug. 2017), pp. 1586–1597. ISSN: 2150-8097. DOI: 10.14778/3137628.3137664. URL: https://doi.org/10.14778/3137628.3137664.

[79] K. Kara et al. “FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off”. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2017, pp. 160–167. DOI: 10.1109/FCCM.2017.39.

[80] Alfons Kemper et al. “Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer”. In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 41–47. URL: http://sites.computer.org/debull/A13june/hyper1.pdf.

[81] *Keras*. 2018. URL: https://www.tensorflow.org/api_docs/python/tf/keras.

[82] S. Khan et al. “Improving cache performance using read-write partitioning”. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2014, pp. 452–463. DOI: 10.1109/HPCA.2014.6835954.

[83] M. Kharbutli, M. Jarrah, and Y. Jararweh. “SCIP: Selective cache insertion and bypassing to improve the performance of last-level caches”. In: *Applied Electrical Engineering and Computing Technologies (AEECT), 2013 IEEE Jordan Conference on*. Dec. 2013, pp. 1–6. DOI: 10.1109/AEECT.2013.6716445.

[84] Hyoseung Kim, Arvind Kandhalu, and Ragunathan (Raj) Rajkumar. “A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems”. In: *Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems*. ECRTS ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 80–89. ISBN: 978-0-7695-5054-1. DOI: 10.1109/ECRTS.2013.19. URL: http://dx.doi.org/10.1109/ECRTS.2013.19.

[85] JongWon Kim et al. “Explicit Non-reusable Page Cache Management to Minimize Last Level Cache Pollution”. In: *Proc. of ICCIT*. 2011.

[86] Kenneth C. Knowlton. “A Fast Storage Allocator”. In: *Commun. ACM* 8.10 (Oct. 1965), pp. 623–624. ISSN: 0001-0782. DOI: 10.1145/365628.365655. URL: http://doi.acm.org/10.1145/365628.365655.

[87] Vlad Krasnov. *On the dangers of Intel’s frequency scaling*. URL: https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/ (visited on 01/03/2019).

[88] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. “Generating code for holistic query evaluation.” In: *ICDE*. 2010. ISBN: 978-1-4244-5444-0. URL: http://dblp.uni-trier.de/db/conf/icde/icde2010.html#KrikellasVC10.

[89] Sailesh Kumar et al. “Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection”. In: *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’06. Pisa, Italy: ACM, 2006, pp. 339–350. ISBN: 1-59593-308-5. DOI: 10.1145/1159913.1159952. URL: http://doi.acm.org/10.1145/1159913.1159952.

[90] Oded Lempel. *2nd Generation Intel Core Processor Family: Intel Core i7, i5 and i3*. 2011. URL: http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.19.9-Desktop-CPUs/HC23.19.911-Sandy-Bridge-Lempel-Intel-Rev%5C%207.pdf.

[91] Lingda Li et al. "Optimal Bypass Monitor for High Performance Last-level Caches". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. Minneapolis, Minnesota, USA: ACM, 2012, pp. 315–324. ISBN: 978-1-4503-1182-3. DOI: 10.1145/2370816.2370862. URL: http://doi.acm.org/10.1145/2370816.2370862.

[92] Xiaofei Liao et al. "A Phase Behavior Aware Dynamic Cache Partitioning Scheme for CMPs". In: *International Journal of Parallel Programming* (2014), pp. 1–19.

[93] Jiang Lin et al. "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems". In: *Proc. of HPCA*. 2008.

[94] Xiang Lin, R.D. Shawn Blanton, and Donald E. Thomas. "Random Forest Architectures on FPGA for Multiple Applications". In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. GLSVLSI '17. Banff, Alberta, Canada: ACM, 2017, pp. 415–418. ISBN: 978-1-4503-4972-7. DOI: 10.1145/3060403.3060416. URL: http://doi.acm.org/10.1145/3060403.3060416.

[95] L. Liu et al. "Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random?" In: *IEEE Transactions on Computers* 65.6 (June 2016), pp. 1921–1935. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2462813.

[96] T. Liu et al. "An efficient regular expressions compression algorithm from a new perspective". In: *2011 Proceedings IEEE INFOCOM*. 2011, pp. 2129–2137. DOI: 10.1109/INFCOM.2011.5935024.

[97] *Lower Numerical Precision Deep Learning Inference and Training*. 2018. URL: https://software.intel.com/en-us/articles/lower-numerical-precision-deep-learning-inference-and-training.

[98] Jan van Lunteren and Alexis Guanella. "Hardware-accelerated regular expression matching at multiple tens of Gb/s2". In: *INFOCOM, 2012 Proceedings IEEE*. IEEE. 2012, pp. 1737–1745.

[99] C. Mack. "The Multiple Lives of Moore's Law". In: *IEEE Spectrum* 52.4 (Apr. 2015), pp. 31–31. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2015.7065415.

[100] Nihar R. Mahapatra and Balakrishna Venkatrao. "The Processor-memory Bottleneck: Problems and Solutions". In: *XRDS* 5.3es (Apr. 1999). ISSN: 1528-4972. DOI: 10.1145/357783.331677. URL: http://doi.acm.org/10.1145/357783.331677.

[101] James Manyika et al. "Big data: The next frontier for innovation, competition, and productivity". In: (2011).

[102] Jason Mars et al. "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations". In: *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2011, pp. 248–259.

[103] Jason Mars et al. "Contention Aware Execution: Online Contention Detection and Response". In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '10. Toronto, Ontario, Canada: ACM, 2010, pp. 257–265. ISBN: 978-1-60558-635-9. DOI: 10.1145/1772954.1772991. URL: http://doi.acm.org/10.1145/1772954.1772991.

[104] Andrew McAfee et al. “Big data: the management revolution”. In: *Harvard business review* 90.10 (2012), pp. 60–68.

[105] Erik Meijer, Brian Beckman, and Gavin Bierman. “LINQ: Reconciling Object, Relations and XML in the .NET Framework”. In: 2006.

[106] Chad R Meiners et al. “Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems”. In: *Proceedings of the 19th USENIX conference on Security*. USENIX Association. 2010, pp. 8–8.

[107] Paul Menage. *Control Group Linux documentation*. 2004. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[108] Angelika Merkel and Neil Gemmell. “Detecting short tandem repeats from genome data: opening the software black box”. In: *Briefings in bioinformatics* 9.5 (2008), pp. 355–366.

[109] *Michelangelo*. URL: https://eng.uber.com/michelangelo/.

[110] *Microsoft Brainwave project announcement*. URL: www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/.

[111] Sparsh Mittal. “A Survey of Recent Prefetching Techniques for Processor Caches”. In: *ACM Comput. Surv.* 49.2 (Aug. 2016), 35:1–35:35. ISSN: 0360-0300. DOI: 10.1145/2907071. URL: http://doi.acm.org/10.1145/2907071.

[112] *ML.Net*. 2018. URL: https://dot.net/ml.

[113] Akshay Naresh Modi et al. “TFX: A TensorFlow-Based Production-Scale Machine Learning Platform”. In: 2017.

[114] *MurMur hash website*. URL: https://sites.google.com/site/murmurhash/.

[115] *MXNet Model Server (MMS)*. 2018. URL: https://github.com/awslabs/mxnet-model-server.

[116] Graham Neubig et al. “DyNet: The Dynamic Neural Network Toolkit”. In: *ArXiv e-prints* (2017). URL: http://arxiv.org/abs/1512.01274.

[117] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: 4.9 (June 2011), pp. 539–550. ISSN: 2150-8097. DOI: 10.14778/2002938.2002940. URL: http://dx.doi.org/10.14778/2002938.2002940.

[118] Xuan-Thuan Nguyen et al. “Highly parallel bitmap-based regular expression matching for text analytics”. In: *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*. IEEE. 2017, pp. 1–4.

[119] Cedric Nugteren and Henk Corporaal. “The Boat Hull Model: Adapting the Roofline Model to Enable Performance Prediction for Parallel Computing”. In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’12. New Orleans, Louisiana, USA: ACM, 2012, pp. 291–292. ISBN: 978-1-4503-1160-1. DOI: 10.1145/2145816.2145859. URL: http://doi.acm.org/10.1145/2145816.2145859.

[120] *NumPY website*. URL: www.numpy.org/.

[121] *Octeon processors family by Cavium Networks*. http://www.cavium.com/newsevents_octeon_cavium.html. Sept. 2004.

[122] Christopher Olston et al. “TensorFlow-Serving: Flexible, High-Performance ML Serving”. In: *Workshop on ML Systems at NIPS*. 2017.

[123] *Open Neural Network Exchange (ONNX)*. 2017. URL: https://onnx.ai.

[124] Kay Ousterhout et al. "Sparrow: Distributed, Low Latency Scheduling". In: Farminton, Pennsylvania, 2013. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522716. URL: http://doi.acm.org/10.1145/2517349.2522716.

[125] Ross Overbeek et al. "WIT: integrated system for high-throughput genome sequence analysis and metabolic reconstruction". In: *Nucleic acids research* 28.1 (2000), pp. 123–125.

[126] Shoumik Palkar et al. "Weld: Rethinking the Interface Between Data-Intensive Applications". In: *CoRR* abs/1709.06416 (2017). arXiv: 1709.06416. URL: http://arxiv.org/abs/1709.06416.

[127] Marco Paolieri, Ivano Bonesana, and Marco Domenico Santambrogio. "ReCPU: A parallel and pipelined architecture for regular expression matching". In: *Vlsi-Soc: Advanced Topics on Systems on a Chip*. Springer, 2009, pp. 1–20.

[128] Ioannis Papadakis et al. "Improving QoS and Utilisation in modern multi-core servers with Dynamic Cache Partitioning". In: *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017*. Ed. by Carsten Clauss et al. Stockholm, Sweden, Jan. 2017, pp. 21–26. ISBN: 978-3-00-055564-0. DOI: 10.14459/2017md1344298.

[129] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12 (Nov. 2011), pp. 2825–2830. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1953048.2078195.

[130] *PredictionIO*. 2018. URL: https://predictionio.apache.org/.

[131] "PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/lee.

[132] Moinuddin K. Qureshi and Yale N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches". In: *Proc. of MICRO*. 2006.

[133] Moinuddin K. Qureshi et al. "Adaptive Insertion Policies for High Performance Caching". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA '07. San Diego, California, USA: ACM, 2007, pp. 381–391. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250709. URL: http://doi.acm.org/10.1145/1250662.1250709.

[134] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 519–530. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462176. URL: http://doi.acm.org/10.1145/2491956.2462176.

[135] Benjamin Recht et al. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: ed. by J. Shawe-Taylor et al. 2011. URL: http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf.

[136] *Redis*. URL: https://redis.io/.

[137] *Redis-ML*. 2018. URL: https://github.com/RedisLabsModules/redis-ml.

[138] *Request Response Python API in Microsoft Machine Learning Server*. 2018. URL: https://docs.microsoft.com/en-us/machine-learning-server/operationalize/python/how-to-consume-web-services.

[139] Astrid Rheinländer et al. "SOFA: An extensible logical optimizer for UDF-heavy data flows". In: *Inf. Syst.* 52 (2015), pp. 96–125. DOI: 10.1016/j.is.2015.04.002. URL: https://doi.org/10.1016/j.is.2015.04.002.

[140] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[141] Daniel Sanchez and Christos Kozyrakis. "The ZCache: Decoupling Ways and Associativity". In: *Proc. of MICRO*. 2010.

[142] Daniel Sanchez and Christos Kozyrakis. "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning". In: *Proc. of ISCA*. 2011.

[143] A. Sandberg et al. "Modeling performance variation due to cache sharing". In: *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. Feb. 2013, pp. 155–166. DOI: 10.1109/HPCA.2013.6522315.

[144] nature international weekly journal of science. *Personalized cancer vaccines show glimmers of success*. https://www.nature.com/news/personalized-cancer-vaccines-show-glimmers-of-success-1.22249.

[145] A. Scolari et al. "Towards Accelerating Generic Machine Learning Prediction Pipelines". In: *2017 IEEE International Conference on Computer Design (ICCD)*. Nov. 2017, pp. 431–434. DOI: 10.1109/ICCD.2017.76.

[146] Alberto Scolari, Davide Basilio Bartolini, and Marco Domenico Santambrogio. "A Software Cache Partitioning System for Hash-Based Caches". In: *ACM Trans. Archit. Code Optim.* 13.4 (Dec. 2016), 57:1–57:24. ISSN: 1544-3566. DOI: 10.1145/3018113. URL: http://doi.acm.org/10.1145/3018113.

[147] Vivek Seshadri et al. "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing". In: *Proc. of PACT*. 2012.

[148] Shai Shalev-Shwartz and Tong Zhang. "Stochastic Dual Coordinate Ascent Methods for Regularized Loss". In: *J. Mach. Learn. Res.* 14.1 (Feb. 2013), pp. 567–599. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2567709.2502598.

[149] J. M. Shalf and R. Leland. "Computing beyond Moore's Law". In: *Computer* 48.12 (Dec. 2015), pp. 14–23. ISSN: 0018-9162. DOI: 10.1109/MC.2015.374.

[150] Akbar Sharifi et al. "Courteous Cache Sharing: Being Nice to Others in Capacity Management". In: *Proceedings of the 49th Annual Design Automation Conference*. 2012.

[151] Bruno da Silva et al. "Performance Modeling for FPGAs: Extending the Roofline Model with High-level Synthesis Tools". In: *Int. J. Reconfig. Comput.* 2013 (Jan. 2013), 7:7–7:7. ISSN: 1687-7195. DOI: 10.1155/2013/428078. URL: http://dx.doi.org/10.1155/2013/428078.

[152] Alexander Smola and Shravan Narayanamurthy. "An architecture for parallel topic models". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 703–710.

[153] Livio Soares, David Tam, and Michael Stumm. "Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer". In: *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2008, pp. 258–269.

[154] Daniele G. Spampinato and Markus Püschel. "A Basic Linear Algebra Compiler". In: *International Symposium on Code Generation and Optimization (CGO)*. 2014, pp. 23–32.

[155] Evan R. Sparks et al. "KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics". In: 2017.

[156] Alen Stojanov et al. "SIMD Intrinsics on Managed Language Runtimes". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: ACM, 2018, pp. 2–15. ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168810. URL: http://doi.acm.org/10.1145/3168810.

[157] David Tam et al. "Managing Shared L2 Caches on Multicore Systems in Software". In: *Proc. of WIOSCA*. 2007.

[158] *TensorFlow*. 2016. URL: https://www.tensorflow.org.

[159] *TensorFlow Serving*. 2016. URL: https://www.tensorflow.org/serving.

[160] *TensorFlow Serving website*. URL: https://www.tensorflow.org/serving/.

[161] *TensorFlow XLA*. URL: https://www.tensorflow.org/performance/xla/.

[162] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". In: *IBM Journal of Research and Development* 11.1 (Jan. 1967), pp. 25–33. ISSN: 0018-8646. DOI: 10.1147/rd.111.0025.

[163] *TVM*. 2018. URL: https://tvm.ai/.

[164] Taegeon Um et al. "Scaling Up IoT Stream Processing". In: *APSys*. Mumbai, India, 2017. ISBN: 978-1-4503-5197-3. DOI: 10.1145/3124680.3124746. URL: http://doi.acm.org/10.1145/3124680.3124746.

[165] Nicolas Vasilache et al. "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions". In: *CoRR* (2018). arXiv: 1802.04730. URL: http://arxiv.org/abs/1802.04730.

[166] Giorgos Vasiliadis et al. "Regular expression matching on graphics hardware for intrusion detection". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2009, pp. 265–283.

[167] Jaak Vilo et al. "Mining for putative regulatory elements in the yeast genome using gene expression data." In: *Ismb*. Vol. 2000. 2000, pp. 384–394.

[168] Skye Wanderman-Milne and Nong Li. "Runtime Code Generation in Cloudera Impala". In: *IEEE Data Eng. Bull.* 37 (2014), pp. 31–37.

[169] Ruisheng Wang and Lizhong Chen. "Futility Scaling: High-Associativity Cache Partitioning". In: *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE. 2014, pp. 356–367.

[170] W. Wang et al. "Rafiki: Machine Learning as an Analytics Service System". In: *ArXiv e-prints* (Apr. 2018).

[171] Xiaolin Wang et al. "A dynamic cache partitioning mechanism under virtualization environment". In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*. IEEE. 2012, pp. 1907–1911.

[172] Jonathan Stuart Ward and Adam Barker. "Undefined By Data: A Survey of Big Data Definitions". In: *CoRR* abs/1309.5821 (2013). arXiv: 1309.5821. URL: http://arxiv.org/abs/1309.5821.

[173] Zhipeng Wei, Zehan Cui, and Mingyu Chen. “Cracking Intel Sandy Bridge’s Cache Hash Function”. In: *arXiv preprint arXiv:1508.03767* (2015).

[174] Matt Welsh, David Culler, and Eric Brewer. “SEDA: An Architecture for Well-conditioned, Scalable Internet Services”. In: 2001.

[175] P. J. Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (Oct. 1990), pp. 1550–1560. ISSN: 0018-9219. DOI: 10.1109/5.58337.

[176] WikiChip. *Frequency Behavior - Intel*. URL: https://en.wikichip.org/wiki/intel/frequency_behavior#Base.2C_Non-AVX_Turbo.2C_and_AVX_Turbo (visited on 01/03/2019).

[177] *Windows ML*. 2018. URL: https://docs.microsoft.com/en-us/windows/uwp/machine-learning/overview.

[178] Chi Xu et al. “Cache contention and application performance prediction for multi-core systems”. In: *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. Mar. 2010, pp. 76–86. DOI: 10.1109/ISPASS.2010.5452065.

[179] Hailong Yang et al. “Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers”. In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 607–618.

[180] Jiajia Yang et al. “PiDFA: A practical multi-stride regular expression matching engine based On FPGA”. In: *Communications (ICC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1–7.

[181] Ying Ye et al. “COLORIS: A Dynamic Cache Partitioning System Using Page Coloring”. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT ’14. Edmonton, AB, Canada: ACM, 2014, pp. 381–392. ISBN: 978-1-4503-2809-8. DOI: 10.1145/2628071.2628104. URL: http://doi.acm.org/10.1145/2628071.2628104.

[182] Jeong-Min Yun et al. “Optimal Aggregation Policy for Reducing Tail Latency of Web Search”. In: *SIGIR*. Santiago, Chile, 2015. ISBN: 978-1-4503-3621-5. DOI: 10.1145/2766462.2767708. URL: http://doi.acm.org/10.1145/2766462.2767708.

[183] Matei Zaharia et al. “Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling”. In: 2010. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755940. URL: http://doi.acm.org/10.1145/1755913.1755940.

[184] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *NSDI*. 2012.

[185] Ce Zhang, Arun Kumar, and Christopher Ré. “Materialization Optimizations for Feature Selection Workloads”. In: *ACM Trans. Database Syst.* 41.1 (Feb. 2016), 2:1–2:32. ISSN: 0362-5915. DOI: 10.1145/2877204. URL: http://doi.acm.org/10.1145/2877204.

[186] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. “Towards Practical Page Coloring-based Multicore Cache Management”. In: *Proc. of EuroSys*. 2009.

[187] Tao Zhao et al. “Genome-wide analysis and expression profiling of the DREB transcription factor gene family in Malus under abiotic stress”. In: *Molecular genetics and genomics* 287.5 (2012), pp. 423–436.

[188] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. “Addressing shared resource contention in multicore processors via scheduling”. In: *ACM Sigplan Notices*. Vol. 45. 3. ACM. 2010, pp. 129–142.

[189] Martin Zinkevich. *Rules of Machine Learning: Best Practices for ML Engineering*. URL: https://developers.google.com/machine-learning/rules-of-ml.

[190] Marcin Zukowski et al. "MonetDB/X100 - A DBMS In The CPU Cache." In: *IEEE Data Eng. Bull.* 28.2 (Aug. 25, 2005), pp. 17–22. URL: http://dblp.uni-trier.de/db/journals/debu/debu28.html#ZukowskiBNH05.