Locality optimizations on irregular algorithms and data structures

Nuno Filipe Monteiro Faria

UMinho | 2011

Universidade do Minho
Escola de Engenharia

Nuno Filipe Monteiro Faria

Locality optimizations on irregular algorithms and data structures

Outubro de 2011

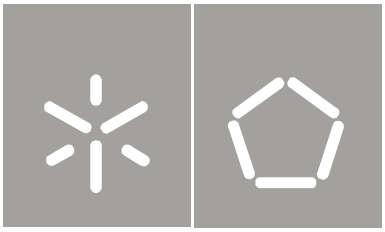
Universidade do Minho
Escola de Engenharia

Nuno Filipe Monteiro Faria

Locality optimizations on irregular algorithms and data structures

Outubro de 2011

Locality optimizations on irregular algorithms and data structures
Nuno Filipe Monteiro Faria
UMinho | 2011



Universidade do Minho
Escola de Engenharia

Nuno Filipe Monteiro Faria

Locality optimizations on irregular algorithms and data structures

Dissertação de Mestrado
Engenharia Informática

Trabalho efectuado sob a orientação do
Professor Doutor João Luís Sobral

Outubro de 2011

Acknowledgments

This dissertation would not have been possible without the aid of my parents and my sister, which have always endured and supported me through my life.

Second, I thank my advisor, Professor João Luís Sobral, for giving me the opportunity to join him and his team in research, for fostering my critical thinking, guiding me and sharing his experience and wisdom with me. I would also like to thank the professors I found along my master course for introducing me to the highly motivating theme of high performance computing, Professor João Luís Sobral, Professor Alberto Proença, Professor Rui Ralha and Professor António Pina.

Last but not least, thank you to all my friends and lab colleagues in university, Roberto Ribeiro, Rui Silva, Rui Gonçalves, Diogo Mendes, Nuno Silva, for all the brainstorm-ish discussions and opinions that allowed me to improve and push myself further - a few themes presented in this dissertation were discussed in some of those brainstorms. To my colleagues/room-mates through my graduate years with whom I shared unforgettable experiences, Nuno Silva, Pedro Silva, Emanuel Gonçalves, André Félix, Samuel Moreira, Roberto Ribeiro and Pedro Miranda - I believe I have made friends for life.

Nuno Filipe Monteiro Faria

The project that served as ground for this dissertation was funded under the agreement of the protocol between FCT and UTAustin

Parallel Programming Refinements for Irregular Applications (UTAustin/CA/0056/2008)

Otimizações de localidade em algoritmos e estruturas de dados irregulares

Resumo

Estruturas de grafos baseadas em apontadores têm sido amplamente discutidas por várias comunidades científicas que tencionam usar este tipo de estruturas. Muitas destas áreas têm a necessidade de usar e implementar algoritmos complexos e sofisticados, como por exemplo, encontrar a árvore de expansão de custo mínimo de um grafo. Estes algoritmos têm um comportamento típicamente irregular no que toca aos padrões de acesso à memória. Como tal, o objectivo é otimizar estas implementações de estruturas de grafos visando aspetos que melhorem a localidade dos acessos à memória. O impacto da latência de memória tem sido continuamente atenuado através do uso de memórias *cache* nas arquitecturas de computadores atuais - as técnicas de otimização para estruturas regulares (ex., matrizes) são bem conhecidas neste âmbito, contudo estruturas irregulares inserem-se numa classe de problemas para os quais ainda não existem consolidadas representações eficientes.

Nesta dissertação é efetuado um estudo sobre as optimizações possíveis em algoritmos como métodos de ordenação *heap-sort*: são apresentadas uma implementação padrão de um algoritmo *heap-sort* e uma versão amigável da cache, originalmente apresentada por Emde Boas. Juntamente com os problemas de esforço de memória, existem também os problemas de algoritmos intensivos em linguagens de alto nível. Frameworks *orientadas a objectos* são normalmente baseadas em conceitos, linguagens e mecanismos abstratos de alto-nível, o que pode criar inconsistências quando combinadas com aspetos habituais da computação avançada (contiguidade de elementos em memória, localidade, eliminar a redundância do código-fonte, etc.). Aspetos como a gestão de objetos em memória (introduzidos por políticas como *type-erasure* e *auto-boxing*) e conceitos abstratos relativamente ao encapsulamento (uso ineficiente de APIs, em conjunto com mecanismos de tipos genéricos) são identificados como sendo problemáticos na resolução de sobrecarga e introdução de refinamentos nas implementações. É notada uma clara incompatibilidade entre *aplicações irregulares intensivas* e metodologias *object-oriented*, nomeadamente em Java. Otimizações e alterações ao código-fonte são propostas em aplicações que sofrem destas limitações de desempenho, com o intuito de diminuir a sobrecarga que a abstração introduz nas implementações. É feito um conjunto de experiências com aplicações intensivas ao ordernar elementos com o *heap-sort* e com o algoritmo sobre grafos para encontrar a árvore de expansão de custo mínimo, para que se possam introduzir otimizações como novas distribuições de dados nas estruturas para melhorar os padrões de acesso à memória, mais amigáveis da *cache*. Padrões eficientes de acesso à memória é o principal interesse tendo em consideração aspetos sobre a localidade, quer seja utilizando primitivas eficientes de distribuições de dados em memória, ou diminuindo a complexidade do código-fonte dos algoritmos e estruturas. As otimizações propostas melhoram o comportamento de cache verificado em versões iniciais, assim como o tempo de execução. Aspetos baixo-nível como *misses* nas caches L1 e L2 salientam o carácter amigável da cache das distribuições de dados e as melhorias no número de *misses*; os menores *misses* na TLB mostram as melhorias na complexidade das implementações.

Locality optimizations on irregular algorithms and data structures

Abstract

Pointer-based graph structures has been discussed by the scientific community that aims to use such structures on several areas. Many of these areas have the need to implement complex and, sometimes, extremely sophisticated algorithms, like finding the minimum spanning tree of a graph. These algorithms are well known for being hard to efficiently execute on current multi-core machines due to irregular patterns of memory accesses. Thus, the objective is optimizing the implementation of graph structures aiming for better memory locality. Memory latency problems have been attenuated by using cache memories in nowadays computer architectures - the memory optimization techniques for regular data-structures (e.g., matrices) are well known, as for irregular data-structures insert themselves in areas where efficient representations are not yet well known.

Optimization algorithms like sorting problems are studied through the use of heap-sort methods: we present a standard implementation and an optimized version originally presented by van Emde Boas. Along with the problems of memory straining we refer also to the problems of intensive algorithms in modern high-level languages. Object-oriented frameworks usually operate with high-level concepts and abstract mechanisms that may not combine well with core features of high performance computing (element contiguity, locality, eliminating code redundancy, etc.). We identify aspects like inefficient object-memory management (introduced by type-erasure and auto-boxing) abstract concepts regarding encapsulation (inefficient API usage alongside with generic mechanisms) as being problematic issues in the resolution of overheads of data-intensive algorithms. A mismatch is clearly noticed between *irregular data-intensive applications* and *object-oriented* in Java. We propose a series of optimizations and changes to the source code of applications that suffer from bottlenecks related to these, in order to demean the setbacks imposed by abstractions. We perform tests on heap sorting and Prim's minimal spanning tree algorithm in order to introduce the improvements made in data layouts optimizing irregular memory accesses. Efficient memory access patterns are the main concern in this thesis and good cache locality on modern memory architectures, whether by using efficient sorting techniques or improving pointer-chasing complexity in algorithms and data structures, are the main goals. The optimizations proposed are able to decrease the cache misses of applications and, most important, execution time. We analyse the low-level instruction counts in order to accurately show that instruction complexity decreases; L1 and L2 cache miss lower counts prove the efficiency of cache-friendly layouts and show the miss behaviour improvement; TLB low miss counts verify the improvement in address and memory management.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We become what we behold.
We shape our tools and then our tools shape us.

Marshall McLuhan

Several strands in today's computational science scene use and depend on large scale data such as molecular dynamics through the form of neighbourhood lists, computational biology, geographic pin-pointing software (maps are basically processed graphs with pertinent data), etc.. These kinds of applications, holding massive data sets, require capable and large enough memory systems able to attain a reasonable coefficient between processing phases and I/O phases. However, since larger memory levels are usually slower there is an inborn memory latency that impacts program execution. Which is why optimized software solutions that decrease latency times in I/O phases are extremely welcome. Due to today's heterogeneity in devices (workstations, clusters, laptops, phones, tablets, etc.), simply increasing memory characteristics may not be feasible - it may be expensive, or even physically challenging. Recently, the size of these devices has been decreasing making cache memories smaller and increasing the need of cache-efficient applications.

From an architectural point of view, the processor-memory gap has been increasing in modern architectures composed of efficient and high frequency processors. Processors in modern computer architectures are extremely powerful and can compute data at an intense rate. However data is not always readily available for processors to access, systems sometimes have to search for data in memory and this searching operation implies transferring data through a slow bus. Modern memory architectures are usually a multi-level hierarchy, each level ranging from smallest and fastest to bigger and slower, in respect to the processing unit. The current memory architectures are extremely well prepared and the problems caused by memory stalls in an application, where the application needs to wait for data to be available to be read, are very well thought through. However, their throughput rate may not attend to current efficiency in processing units (CPUs,

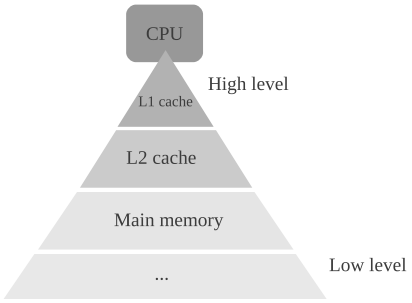


Figure 1.1: High and low memory levels in memory hierarchy

GPUs, etc.) due to their limitations in memory size and bandwidth. *Processor-in-memory* is a concept that refers to a processing unit (such as a CPU) tightly coupled with memory modules in the same chip, in order to reduce memory latency. This has enabled the appearance of cache memories that are fast memories integrated with the processor to store data that the processor is likely to use - searching for data elements in higher level[1] memory is fast - see Figure 1.1. CPU cache memories are much faster than, e.g., RAM memories, and when the processor searches for an element that is in cache this is called a *cache-hit*, on the other hand, if the element is not in cache and we have to fetch the data from the original memory device, we call this a *cache-miss*.

1.1 Data intensive and Irregular applications

A data-intensive application is, as the name implies, an application that needs to perform a large number of read/write operations from and onto memory in order to be able to proceed. When dealing with large chunks of data that do not fit into the small-fast memory levels (cache memory), data is stored in bigger and slower levels. Analysing in a finer grain how data-intensive applications perform in memory systems, memory stalls are caused by the lack of memory storage space and bandwidth responsible for bringing pertinent in-memory data chunks into close distance memory of the processing unit, and since pertinent data is not accessible the application must wait until it is loaded by the memory architecture from bigger-lower levels, stalling program execution.

Irregular applications are so due to the irregular access patterns performed over memory or, or due to a random factor that may condition and influence computations - let us focus in irregular data structures. The irregularity in data structures comes from the seemingly amorphous representation of data in memory - one cannot predict the memory access patterns as done in, for instance, a matrix multiplication problem where properties like cache alignment and tiling are easily predicted and managed to improve performance - the subject of data cache reuse and its positive impact in performance has been widely studied in [31]. Irregular data structures

[1] For clarity reasons, smaller/faster levels are considered *high level* and bigger/slower levels are considered *lower levels* in memory hierarchy.

are usually featured by a pointer-chasing nature, which means a data chunk brought into cache memory (resultant from a load from a memory reference) may not be usable and the system has to perform a load from higher and more expensive levels. As a visual aid, the reader can imagine memory accesses jumping back and forth throughout a data structure taking no advantages from the data bulk loads made into cache memories.

Irregular applications often relate to data intensive applications because they both strain memory architectures due to their data-dependent nature - an algorithm cannot proceed until memory data dependencies are resolved and data is loaded by processing units.

1.2 Locality of reference

In this dissertation locality of reference (considered only *locality* for simplicity) is widely discussed as being one of the main issues to optimize - locality is related to the storage and loading processes in memory systems where load operations try to bring pertinent chunks of data into high-level memory, i.e., the relevant data to an application, thus minimizing the number of data fetches on slower memory levels. There are two kinds of locality: temporal locality and spatial locality. Good temporal locality consists of referencing data that will probably be referenced in the future (a good example of temporal locality is the caching system of browsers, by keeping the information of the most recent pages visited in memory); for instance, if the *time interval* in which the same memory region is accessed twice is a *long time interval*, it is very likely that the memory region has already been evicted from cache memory. Good spatial locality occurs when data is referenced and nearby memory sections become more likely to be referenced in the near future, avoiding loading memory that is off-chip, i.e., in lower memory levels - a good example of this are loop reordering optimizations in matrix multiplication problems, mentioned in Section 2.4.1.

The three C's

Caches memories are defined by three main parameters that drive their basic functioning: total capacity, block size (or cache line size) and associativity (direct mapped, fully associative or set associative). An ideal combination of these parameters is hard to find because it deeply relies on the nature of applications being run. The authors in [22] performed an empirical study by benchmarking several different caches and distinguished three kinds of misses that might occur:

i. *compulsory misses*, caused by first reference to a memory region - unavoidable regardless of associativity and capacity.

ii. *capacity misses*, caused by the limited cache size solely - cache associativity is not considered.

iii. *conflict misses*, misses that could be avoided, caused by earlier evictions of pertinent data. Two sub-types are also considered: *mapping misses*, cannot be avoided due to particular cache associativity level, and *replacement misses* originated by replacement policies.

With the appearance of recent multiprocessor architectures has risen a new *C*, *coherence misses*, where concurrent writes and reads may injure a program's correctness and performance. The class of misses studied in this dissertation does not fall directly into any of these categories, although it is related to each one: pointer-chasing misses can be caused by the first reference to a memory region (compulsory), that was not loaded due to the sparsity and size of the considered data (capacity); pointer-misses can also be conflicting in the way that data reuse is hard in irregular memory access due to the amorphous nature of irregular structures (conflict misses) - data reuse is hard to predict.

1.3 Optimizing data layouts

As a way of improving memory performance, improving data layout representations in memory is a good way to achieve better locality in cache memories. Data layouts have impact on cache performance because they directly affect the memory access patterns for a given dataset, improving, for instance, cache-hit behaviour by reorganizing data elements for tighter adjacent memory addresses. Some of the most used data layouts consist in rearranging how elements and/or attributes in a data set are layed out in memory to achieve better alignment and spatial/temporal locality.

Of course, these optimizations sometimes strongly rely on the destined framework. Object-oriented frameworks show some possibly troublesome features in the scope of data intensive and irregular applications. Specifically auto-boxing and encapsulation can stand as a barrier to performance; some studies were made regarding the core features of data collections and their representation (layout) in memory in object-oriented environments.

1.3.1 Object-oriented and Memory management

Object-oriented (OO) frameworks appear as an undeniably useful resource providing software developers flexibility, abstraction and modularity. However, OO languages like Java and C#, implement virtual machines with their automatic memory management policies (garbage collection, allocating and freeing allocated memory). The problem with (i) auto-boxing is that it does not guarantee memory element contiguity in datasets with abstract data types; the problem with (ii) encapsulation is that it may originate overhead in cross-structure optimizations.

In the specific case of Java, in the Java Virtual Machine (JVM) and its Garbage Collector (GC) the programmer has little or no control over memory allocation; when writing data intensive algorithms one must be careful and consider the overhead of generic type mechanisms and its underlying pointer-resolving overhead in generic collections.

1.3.2 Parallelism motivations

Part of our interest in optimizing data layouts in irregular applications as also to do with the scalability problems of irregular data intensive algorithms in modern multi/many-core environments, in which the problems of irregular accesses are aggravated. *Conflict misses* are common in this scope and can be avoided by rearranging the data space distribution for better use of memory accesses, however *coherence* problems are strongly related to multi-core environments, which can be a setback on optimization methodologies applied. Primitives which allow analysing the regions of data able to be processed in parallel have already began to be considered [36] - a data-centric view of irregular algorithms, instead of looking at computational dependencies, is the main analysis. However, parallelism concepts are not considered in the presented work.

1.4 Main goals, Scope and Contributions

Goals

The main goal with this dissertation is to look at problems related to memory inefficiency in irregular data intensive applications, perform a study on memory locality optimizations by changing data layout instead of using common loop reordering methodologies. Also, the aim is to study the impacts of high level object-oriented languages and show that despite the software development facilities, combining HPC with abstractions may generate undesired overhead. Layout optimizations come as a suitable way to decrease cache miss counts and increase memory efficiency in modern multi-level memory architectures. Although abstractions may generate overheads there are possible solutions to demean them, for instance, improved virtual machine management or decreasing the complexity of the underlying abstraction layers in between optimized structures. We aim to present and study data layout arrangements that benefit performance through locality in overhead-prone environments of object-oriented.

Scope

The subjects in this dissertation are studied regarding locality optimizations, memory access patterns and pointer resolving complexity decrease. The main themes are:

- Object-oriented programming in scientific data intensive algorithms and related overheads.
- Cache-friendly heap sorting algorithm.
- Linked data structures, which are typically irregular data structures being characterized by a jump-pointer memory access pattern.
- Graph related algorithms, in the scope of irregular algorithms and data structures, namely Prim's Minimal Spanning Tree (MST) graph algorithm.
- Low level benchmark analysis; architectural bottlenecks and improvements.

Contributions

In-depth analysis of low-level information gathered in a series of benchmarks to instruction counts, clock cycles, level 1 (L1) and level 2 (L2) cache-hits and misses in typically irregular algorithms and data structures is provided considering the main problems found in high-level languages. We point to important cares to take in high performance programming in these abstraction-based frameworks.

1.5 Organization of the Dissertation

This dissertation is composed of four chapters including the current Chapter 1 and the concluding Chapter 4. In Chapter 2 is discussed the current state of the art of the theories, methodologies and technologies applied in the scope of the dissertation.

Chapter 3, *Optimizing data structures*, embodies the studies made, taken considerations, work and results. Section 3.1 explains the problems of object-oriented frameworks. Benchmark and OO considerations have impact on how the case studies for priority-queues and graphs are contextualized, so these concepts are opportunely recalled in further sections (Sections 3.2 and 3.3). Layout optimizations and API refinement compositions are introduced in Section 3.4. The benchmarking methodology, so the reader can have a better understanding of the results presented are discussed and presented in Section 3.5, *Benchmark methodology*.

In Chapter 4, the concluding remarks are given and future work is proposed.

Chapter 2

Background

The goal of this chapter is to discuss the state of the art for models able to represent the I/O complexity of memories, introducing the concepts of cache aware and cache oblivious as performance improvement opportunities, optimizations at JVM level for object-oriented frameworks and finally a section dedicated to graph related data structures, algorithms and current methodologies.

2.1 Cache-aware and cache-oblivious

Cache-aware algorithms are those that work optimally by regulating certain configurations that have direct impact on cache performance. These configurations must often be tuned for each platform since they perform better in some architectures than others. One simple example may be, the block size that many applications use in order to explicitly optimize cache alignment. As affirmed by Frigo in [20], cache-oblivious algorithms have no notion of specific cache parameters in order to make a given execution more scalable in a specific memory architecture. A cache-oblivious (COB) algorithm tends to perform well on any memory hierarchy model. Recent memory trends have deep variations of details such as the number of levels in its hierarchy or the memory size of each level, the data block size transferred between each level and associativity specifications. COB algorithms' objective is then to ignore the memory specifications and adapt with some leeway to the memory model the algorithm is running on. Another big advantage is in its modelling aspect: if we model a COB algorithm according, for instance, to a two-level memory model, the COB nature will not be lost when applied to hierarchies with more levels.

2.2 Memory models

Computational models are an important mathematical tool used to express the implementation complexity of an algorithm in a relatively general way, and there are many models available to

express the complexity of modern memory systems [51]. Memory models are discussed in some depth by many authors that aim at cache conscious implementations of data-structures to help understand how we can consider in computational and abstract terms how to take advantage from cache memories to maximize cache efficiency (temporal and spatial locality) [47]. The memory models discussed in some depth are: *random access machine* model, *external-memory model*, *hierarchical memory model* and the *ideal-cache model*.

2.2.1 RAM model

A Random Access Machine (RAM) is considered to be a hypothetical computer used to analyse and measure the running time of an application. The abstract computer considered in this model has following features : (i) each simple operation takes 1 time step (arithmetic, logical and jump operations); (ii) loops are not single-step operations given that the code inside a loop can influence the total program complexity - the complexity of loops depends on the number of iterations; and lastly, but most importantly, (iii) each memory access takes 1 time-step - the RAM model does not differentiate any level of memory hierarchy.

The main advantage in this model is its simplicity, however, this fact makes it most of the times unreliable in practical terms since it may ignore or violate important architecture features like: multiplication operations can take longer than sum operations; it does not represent modern memory systems properly not worrying about memory sizes, latencies or cache associativity.

2.2.2 External memory model

Typical external memory algorithms are a well known topic in problems in which data dimensions overcome the high level memory - the internal memory close to processing units which we consider to be internal when data is not transferred via slow BUS communication interfaces. External memory models are also known as *disk access models* or *I/O models*. According to Rønn et al. [47], this model can be seen as two consecutive memory layers communicating with each other: an internal and an external one. This model considers the following parameters:

- N, number of elements to process
- M, size of internal memory
- B, number of elements that can be transferred in a block, $1 \leq B \leq M$
- P, number of blocks (B) transferred simultaneously, $1 \leq P \leq \lfloor M/B \rfloor$

One of the disadvantages of this model is that these parameters are specific of each architecture which makes code portability a burden, as most cache-aware implementations are.

2.2.3 Hierarchical memory model

The hierarchical model is different from the external memory because it considers several memory layers, each layer size increases in powers of 2 as it distances itself from the CPU. This model is not enough regarding some details between memory levels, for instance cache associativity. In this model each memory access does not take 1 time-step nor a constant time-step: each memory access depends on a certain function which defines the existing number of memory layers and the size of each layer, $f(i)$.

2.2.4 Ideal-cache model

Algorithm analysis using this model considers, as in external model, the parameters B, M and N; the multiple transferring factor P is ignored. The underlying *cache-obliviousness* gender of algorithms sprays optimality in memory-bound algorithms considering any two contiguous memory levels, disregarding any specific architecture detail; also any portability problems raised by embedding architecture details in the code disappear, since cache-oblivious algorithms are unaware of B and M parameters.

The ideal-cache model is often referred by many authors as the model to consider for cache-oblivious algorithms and data structures. The ideal-cache model is a simplification of actual memory systems. With a number of assumptions made and justified by Frigo et al. [20] and mentioned in [47], this model can closely relate to modern memory hierarchies. An ideal cache considers the properties presented in the following assumptions.

Tall cache assumption

The tall cache assumption states that the block size is not greater than the number of blocks, $B \leq M$. A good visual aid for the reader is presented by Olsen et al. [39]: a tall cache is the opposite of a *wide* cache, where the block size covers the size of the whole memory level. According to Brodal, the tall cache assumption is needed to achieve the optimal sorting bound with the *Funnel-Sort* algorithm [7]. The tall cache assumption is representative of most memory architectures in current computers, therefore memory models that assume a tall cache existence tend to present accurate analysis.

A common cache size assumed by most authors for modern architectures is $M = \Omega(B^{\gamma+1})$, where γ is a constant factor $\gamma > 0$, and is usually set to $\gamma = 1$ [10, 39, 7], therefore, cache size is defined by $M = \Omega(B^2)$.

Optimal replacement assumption

A *least-recently-used* (LRU) replacement policy is used for cache data replacement, using a FIFO (first-in/first-out) strategy. According to Sleator, Tarjan [52] and Rønn [47], considering a cache size of M and a constant factor $c > 0$, any algorithm will incur in c times more cache misses than it would have when using an optimal cache with size $1 - \frac{1}{c}$. According to [47], for a constant

$c = 2$, an algorithm that causes $2Q$ caches misses on an LRU cache (with cache size M, and block size B), will cause a maximum of Q misses in an optimal cache of size $\frac{M}{2}$.

Rønn et al. [47], presented a fairly interesting example for this replacement problem. Before hand lets clear some necessary concepts for understanding the next presented case:

- B_S = block size
- M = memory size
- $\frac{M}{B_S}$ = cache line size

Recurring to the same example in [47], consider a cyclic scan over an array divided in b blocks, and each block stores $\#els_b = \frac{M}{B_S} + 1$ elements; an access to the first element of the first block will bring into cache a block of $\frac{M}{B_S}$ elements, leaving the last block element out of the loaded block. Since we consider an LRU policy, when accessing this last element, which is already out of the cache block, the first accessed element will be discarded (due to LRU eviction), however since this last element belongs to another block, a cache miss occurs and as a result another block is discarded - this will cause a cache miss per access until the algorithm is finished.

However, this worse case scenario rarely occurs when dealing with regular complexities, also, when compared with random data placement policies, the results are similar.

Two memory layers assumption

Since the ideal cache model considers only two memory layers, real and common memory systems operate with up to five levels, we need to ensure that this model is applicable on such architectures. Rønn et al. discusses two alternatives to justify the optimality of a two memory layer:

(i) data in a given memory layer i is also present at layer $i + 1$, being layer i the closest to CPU - this is called the *inclusion property*.

(ii) the *inclusion property* is maintained in between any two consecutive memory layers *for any sequence of accesses*; the data present in a given memory layer i is also present at memory layer $i + 1$, given the same memory access patterns. In theory, it is as if layer i works as the highest level (closest to the CPU) in the memory hierarchy.

Auto-replacement and full associativity assumption

The underlying idea behind auto-replacement is to automatically replace cache blocks in case of occurrence of cache misses. This is a concept easily understood from a programmer perspective: hardware processes high-level cache-misses and usually the operating system handles data transfers in lower levels like RAM or disk.

Full associativity allows for a memory system to choose freely where to store data loaded into cache levels - this property, although forcing programs to search in more cache memory addresses, can result in fewer cache misses.

2.2.5 Caches in multi-processor environments

Although we do not attend the problems of parallelism in this dissertation, it is important to note that cache systems in current parallel architectures may implement a series of features that may revolutionize more traditional approaches to cache efficiency: *inclusive* and *exclusive* caches. In inclusive cache environments if a chunk data is present in L1 cache it is forced to be present in deeper cache levels as well. This assumption is made for the ideal cache memory model presented in this chapter.

The appearance of exclusive caches goes against the assumptions made in the presented models - inclusion is expected for all considered memory levels. Multi-core processors like AMD Opteron^{TM}maintain the coherence of data between several cores by sharing a cache that serves all cores in the process of fetching data from non-cache levels [24, 1, 12]. L2 and L3 caches are exclusive to each core, in order to reduce the costs of synchronization between core-specific caches. In these cases, L2 and L3 caches work as victim-caches (holding data evicted from L1 cache), and all data fetches are directly aimed at L1 cache, skipping L2 and L3 that might have been causing coherence problems, or overhead-ish operations to avoid them.

Nevertheless, we initially assume for all models and analysis made in the dissertation, the inclusion property in multi-level cache systems.

2.3 Cache-aware/oblivious sorting

Sorting algorithms restrain memory architectures - we can look at different optimization scopes depending on its intended usage and data-type to sort: (i) optimization from a complexity optimization point-of-view or (ii) from a memory performance point-of-view.

Many authors provide several sorting primitives and alternatives to traditional sorting primitives, being the most popular cache-aware/oblivious variants of sorting primitives like *heap-sort* [49]: Distribution-Heap [3], Funnel-Heap [8], etc. One of the most interesting cache-oblivious implementations is the one proposed by van Emde Boas [15]. But before jumping into the cache oblivious/aware studies we study sorting implementations that do not concern about cache efficiency - some examples are: Fibonacci-Heap [19], Pairing-Heap [18], Violation-Heap [14].

In this section we provide an overview of the studies made around the efficient priority-queue (PQ) and heap implementations.

2.3.1 Efficient heap implementations

A sorting algorithm may apply various techniques, such as *divide-and-conquer* - let us focus on heap-sort techniques. A heap is a tree-based data structure that obeys the heap property (Definition 1). Many authors studied and developed efficient structures in this area, some of which are explained with some detail and discuss the main advantages/disadvantages between them.

Definition 1. ***Heap property -*** *the key of a child node must be higher or equal than the key of its parent (this can vary whether we consider a min-heap or a max-heap; in our case we consider a min-heap).*

Fibonacci-Heap

A Fibonacci-heap, developed by [19] is a binomial heap that follows a specific set of rules. A binomial heap may be composed of several binomial trees. The order k of a binomial-tree[1] inside a binomial heap can vary; if it has order $k = 0$ the tree is composed of a single node; if not, the binomial tree with order k has a root with k children, each of order corresponding to the set represented in equation 2.1, i.e., the order of each child would be $[k-1, k-2, \ldots, k-k]$.

$$\{k-i\}, i \in [1, 2, \ldots, k] \tag{2.1}$$

Binomial-heaps obey the *heap property* (Definition 1) and several other properties not considered here.

The Fibonacci-Heap data structure is a linked-list where each node stores pointers to four other nodes: parent, left and right sibling, and one of its children - the structure does not need to point out all children because that is implicit by making the linked-list circular. The main aspect to take into account here is optimizing operational complexity. The most commonly found operations in heaps are: *insert*, *find-min*, *delete*, *delete-min* (a combination of *find-min* and *delete*) and *update-key* (with the variations *increase-key* or *decrease-key*). For the Fibonacci-Heap, having a full understanding of how these operations work may be complicated, so we first provide an explanation of the main intention grounded to this heap.

Underlying idea in Fibonacci-Heaps and Amortized analysis

The Fibonacci-Heap's main feature is the amortized bound costs in operations *delete* and *decrease-key*. *Amortized analysis*[2] is focused on analysing an algorithm basing judgements in not only the immediate results but also in longer-term results that are based in a sequence of operations. The underlying idea is to "delay" work for later operations that will do it in less asymptotic time that it would if it had been done in a single operation. In this Fibonacci-Heap case, amortized analysis comes in, for instance, in *delete-min* operation: the work related to obeying the *heap-property* after each operation is successively delayed to later operations - even though we do not know the absolute correct position of every element in the heap, we know what is the *minimum/maximum*. The *insert* operation takes constant time $O(1)$; *decrease-key* takes amortized cost *O*(k)*, where k is a constant therefore *O*(1)*; *delete-min* takes *O(log n)*.

[1] For clarity and simplicity we may sometimes refer to a binomial tree with order k has B_k

[2] For simplicity, when referring to amortized costs in Big-O-notation we will add a "*" character.

Pairing-Heap

Pairing-Heaps were introduced in [18], and studied further in [53, 17, 42]. This heap is a multi-way heap ordered tree which is based also in *amortize-bounded operations* and it has better practical results when compared to other heaps such as Fibonacci-Heaps. The data structure used for this implementation is in general terms a pair composed by a root element and a possibly empty list of pairing sub-heaps. The heap property presented in Definition 1 is also present: all sub-heap roots must not be smaller than the root element.

Underlying idea in Pairing-Heaps

The heap property is maintained with *delete-min* operation which consists in two phases implicitly made by recursive calls (first phase) and their corresponding backward steps (second phase):

(i) a *left-to-right* pass, merging pairs of sub-heaps together (thus its designation),

(ii) and a *right-to-left* pass, merging the resulting list of sub-heaps.

Insert operation runs in asymptotic time of *O(1)*; *delete-min* takes *O(log n)*; and the *merge* operation takes *O(1)*.

2.3.2 Cache-efficient heap and priority-queues

From this scope we now take a look at the memory behaviour of cache-efficient heaps instead of just relying in complexity analysis and operational optimization. The heaps mentioned further also consider the memory models (see Section 2.2) and the analysis is done based on these.

The cache oblivious nature of algorithms in this section is crucial for the good results that have been achieved in most recent research material. Data structure engineering has a higher importance in cache-aware/oblivious - these may directly influence the performance of a given algorithm because memory issues such as cache alignment, cache replacement policies or size assumptions play an important role.

Funnel-Heap

The Funnel-heap, introduced as a priority-queue in [8], is based on binary-merging of values present in buffers existing in a queue. The data structure is simple and easily imagined as a binary tree of *buffers* and *binary mergers*, the two main elements for this priority-queue structure. This algorithm is a variant of *merge-sort* (divide-and-conquer paradigm) [11] and was based on *funnel-sort* [20]. The data structure presented in [20] has a good memory behaviour for its operations; the *insert* and *delete-min* consider a tall-cache (see Section 2.2.4) and have an amortized I/O cost per operation of $O(\frac{1}{B} log_{M/B} \frac{N}{B})$.

Distribution-Heap

The cache-oblivious PQ implementation presented in [3, 39] is partitioned by levels, each of which has buffers with sizes depending on the level its in. The authors of [3] divided their structure explanation in three parts: levels, buffers and data layout.

This structure consists of $\Theta(log(log(N)))$ levels that can extend from size N to a constant c. The i^{th} level has size $N^{(\frac{2}{3})^{i-1}}$ therefore level sizes would be the following: N, $N^{\frac{2}{3}}$, $N^{\frac{4}{9}}$, ..., c. Elements are stored in buffers from each level that do not only store elements but are also used to pass elements up and down the structure levels. The number and sizes of such buffers are, for level X:

- one up buffer u^X, with a maximum size of X elements,
- and at most $X^{\frac{1}{3}}$ down buffers, each with maximum size between $\frac{1}{2}X^{\frac{2}{3}}$ and $2X^{\frac{2}{3}}$ elements.

There are invariants defined by the authors to ensure the correctness of this heap throughout these various levels:

Invariant 1. *The down buffers in level X are sorted externally, but their internal elements may not be sorted.*

Invariant 2. *At level X, the elements in down buffers have smaller keys in up buffers.*

Invariant 3. *The down-buffer elements of level X have smaller keys than down buffer elements of the next higher level* $X^{\frac{3}{2}}$.

Up buffers store elements *on their way up* the heap; as for down buffers store elements *on their way down* the heap. The single-element *insert* and *delete-min* operations are not directly implemented. The operations that insert and remove elements from this heap are *push* and *pull*, which insert and remove a stream of elements (respectively) into/from a given level. In [39] the author implemented the single-element operations creating auxiliary *insertion/extraction* buffers.

Cache-oblivious B-Tree

A B-tree [5] with a block size B (according to memory parameters) gets to an *optimal search bound* of $O(1 + log_{B+1}(N))$ and allows insertions, removals, updates in amortized logarithmic asymptotic time. It is basically a regular binary tree data structure with the possibility of storing several elements per node as well as having several children. The cache oblivious B-tree developed in [5] uses the van Emde Boas layout [15], not the data structure itself, only the data layout. The objective was to create a multi-level structure that behaved well in more than two memory levels as well as consider several sizes for transfer blocks. As explained in [15] the van Emde Boas layout is a recursive data layout that allows for spatial locality benefits, which allied to the characteristics of B-trees and good asymptotic times the authors proved to have near-optimal efficiency on any multi-level memory architecture.

In order to maintain locality of reference, the authors in [5] developed the tree to be strongly weight-balanced and to do so they considered the following properties:

(i) *Descendent amortization* - under the assumption of whenever a node is rebalanced, their descendants are all *touched*; the amortized number of elements per insertion is *O(log N)*

(ii) *Strong weight balance* - for some constant d, every node at a given height h has $\Theta(d^h)$.

One of the main concerns in this research was the inefficient usage of memory space due to the strong weight balance property: it makes the PQ waste array storage space creating unused wholes in the B-tree data structure, therefore a packed-memory array is considered in their work.

2.4 Locality optimizations

In a general way, applications have been optimized regarding locality of reference through the use of two methodologies: (i) changing the order of nested loops [34, 2, 21] and (ii) through the use of data space transformations (changing array layouts) [25, 2, 33, 9]. One of the main concerns in parallel computing is, precisely, locality of reference. Locality optimizations allow for multi-processor architecture to make better use of data distribution. As stated in the early research in [2], in order to increase locality, skilled programmers manually altered data structure and algorithm specifications. An effort is clearly made to make compilers automatically generate such optimizations, specially in multi-processor environments where the performance gains are scalable [2]. For instance, the data layout of a parallel application through several processors can be optimized by automatically assigning memory regions to specific processing units (or the opposite, processing units to memory regions) in order to increase memory coalescing and decrease inter-processor data communication and synchronization [25, 2] (which results in unwanted latency). This is broadly explored by reordering nested loops to alter the access patterns for each (possibly parallel) iteration, basically to assign to each iteration work that operates in contiguous or the same memory addresses. However, problems like *legality* and data correctness of algorithms may not always allow loop reordering techniques. Data space transformations do not interfere with the specifications of an algorithm, however they can limit an application: if a data structure is transformed, all accesses to that structure must be revised; as for loop reordering, only the loop in question is affected [2]. In modern object-oriented frameworks, accesses to structures can be abstracted, however such abstractions are prone to overheads.

2.4.1 Algorithmic locality optimizations

Optimizing locality can go through changing the access patterns of the computations. Since loop-based applications are more susceptible to locality optimizations, changing the order and the amount of accessed data at different iterations can greatly improve performance.

Loop reordering

This optimization consists in inter-changing and interfering with the order of the loops to change the access patterns of the computations applied in the data structure. Another common consideration is to improve register usage, i.e., prevent unnecessary slow memory accesses if the value can be stored in fast memory, through the use of temporary variables [56]. Changing the order of loops can greatly improve the efficiency of memory accesses - spatial locality is improved - see Listings 2.1 and 2.2, which show a matrix multiplication algorithm. The improvement is in the order of access to rows and columns: the optimized version iterates through the second (B) matrix in a raster manner (row-major), which for row-major languages [3] means good cache alignment and spatial locality improvement, since the loaded row from B will be used, instead of the column from B - this means that each resulting matrix (M) element is not fully computed at each iteration, instead it is successively computed to take advantage of row-major accesses in matrix B.

Listing 2.1: Regular matrix multiplication implementation

```
1  for i = 1 to N
2    for j = 1 to N
3      for k = 1 to N
4        M[i,k] += A[i,j] * B[j,k]
```

Listing 2.2: Matrix multiplication implementation with loop-reordering

```
1  for i = 1 to N
2    for k = 1 to N
3      for j = 1 to N
4        M[i,k] += A[i,j] * B[j,k]
```

Loop tiling

Loop tiling (or blocking) consists in considering blocked versions of a given loop-based application that uses nested loops, in order to improve aspects like cache alignment and data reuse (through the form of *temporal locality*) and loading of pertinent adjacent memory (spatial locality). This form of optimization has been studied also as a way to identify independent data chunks and uncover parallelism [57, 56]. Tiling reduces the number of loads from lower memory levels, by taking advantage of the data already present in cache.

Listing 2.3: Matrix multiplication implementation with loop-tiling

```
1  for ii = 1 to N step s
2    for jj = 1 to N step s
3      for i = 1 to N
4        for j = ii to min(ii + s - 1, N)
5          for k = jj to min(jj + s - 1, N)
6            M[i,k] += A[i,j] * B[j,k]
```

[3] The majority of languages operate on memories in a row-major manner.

The main aspect optimized in tiled matrix multiplication algorithm in Listing 2.3 is temporal locality: the values that have already been accessed and computed, and will be needed for future computations are reused in inner loops. Ideally, the blocks implicitly created by partitioning computations correspond to blocks that fully fit in cache. A scheme of loop-tiling is shown in Figure 2.1 for multiplying matrices A and B, resulting in matrix M.

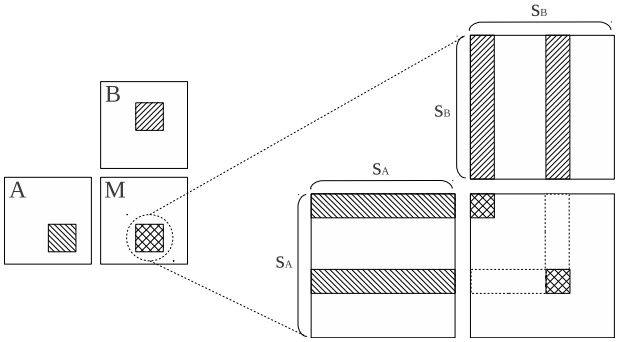


Figure 2.1: Matrix loop-tiling scheme

2.4.2 Data layout locality optimizations

The previous optimizations only interfere with the algorithmic part of a program - it is basically adapting an algorithm to the data structure by performing the necessary actions to better suit the algorithm to the data structure and hardware specifications. Data layout optimizations refer to an alternative concept which is interfering with the data layout of the considered structure and make the necessary data arrangements in memory in order to improve the access patterns performed by a given algorithm.

Tiling and recursive layouts

Tiling is a data rearrangement technique used to improve the cache efficiency by explicitly place elements in memory in such a way that cache accesses are maximized [45, 40, 15]. In [15] a recursive layout is defined for the tree in a heap-sort implementation: this layout consists in storing *down-going* children tree elements in adjacent memory regions so that memory accesses converge to the same memory regions. This kind of optimizations (data layout optimizations) relies mostly on the improvement of spatial locality - the arrangement of elements in contiguous memory addresses.

Inlining and marshalling layouts

Inline allocation of data structures consists in optimizing spatial locality by eliminating one level of indirection in pointer-chasing structures: the referenced data is brought into close memory

of the *reference* to avoid wholes in memory (preserving semantic order). This optimization is possible to be applied in compilers and object-oriented frameworks - where usually objects are considered memory references. This optimization gender has positive impact in data intensive applications where collections of objects are used. Many compilers and, in the case of object-oriented, garbage collectors, apply similar policies to eliminate some level of indirection between pointers and data, resulting in lower *pointer dereference costs*. For instance, object-online reordering in Jikes [23] increases spatial locality by performing compile-time analysis, adaptive sampling and object traversal in garbage collecting phases: it finds the *hot* spots (fields and methods) in an application and rearranges them in memory for improved spatial locality.

Marshalling may be perceived as the process of serialization, however backed by a different concept: if data is present in a pointer-chasing collection is sparsely layed out in memory, the process of marshalling compacts data elements together, not considering any specific order except that of the original order of the elements - this concept is used for example in computer communication policies to decrease the storage space of structures, but can also be used to influence data layout and access patterns.

Studies on the latency impact of data-intensive applications on modern memory architectures have been made in [4] - the authors study three kinds of memory access patterns and offer some views over *software pre-fetching* and *locality optimizations*. Both sides are singly analysed warning to the value of *latency tolerance* and *latency reduction* for software pre-fetching and locality optimizations respectively. According to their study, locality optimizations show better latency behaviour than software pre-fetching in low memory bandwidths, and the opposite for high memory bandwidths. The work at [4] denotes three specific types of structures that provide different approaches on memory access patterns: affine array accesses, indexed array accesses and pointer-chasing accesses.

Affine array accesses

Memory accesses with a constant and well known behaviour such as accessing multidimensional arrays, common on applications like dense-matrix linear algebra and finite-difference PDE (partial differential equations) solvers - no irregular data accesses.

Indexed array accesses

As seen further, this access pattern organizes data in an indexed manner so that less space is needed. However the cache performance on such patterns is poor due to irregular memory accesses. One does not know previously the data organization on memory. These irregular accesses on memory make an irregular algorithm, where the amount of computation is only uncovered at run-time.

Pointer-chasing accesses

Structures based on pointers, like linked lists and n-ary trees, are part of this memory access class. It shares some characteristics with the *indexed* concept, because the access to a specific portion of data must be resolved (at run-time) to access the data itself. The main characteristic in this type of access is that an access to an additional part of the data-structure can only be resolved when the *pointer* to that portion of data is resolved.

Software pre-fetching consists in explicitly placing fetching instructions on memory that is likely to be missing in cache. This has proved to be a good way to avoid memory stalls and some studies are made regarding this subject on memory access patterns (discussed earlier). The authors use pre-fetching for all three kinds of memory accesses to hide memory latency - locality optimizations are presented as a way to reduce latency by reordering computations and/or data layouts: (i) for affine array accesses, *tiling* is given as a good choice; (ii) for indexed accesses, data and data layout reordering can also be used to improve spatial locality at compile and run-time; (iii) for pointer-chasing programs the used technique is dynamic memory allocation to improve spatial locality, for instance, dynamic locality-conscious placement of parent and children nodes of a tree at run-time - a cache-conscious heap is introduced.

2.4.3 JVM level optimizations

The Jikes RVM[4] (Research VM) project is a research aimed Java virtual machine that introduces several optimizations. Studying the topics of multi-threading, multi-core heterogeneity and, in our interest, garbage collection strategies for improved performance, namely in the field of locality optimizations. In [23] et al., the authors state that although explicit memory management in C offers benefits in allocation performance, it has a hidden setback: *"it cannot move objects without violating language semantics"*, like in Java's GC. Online object reordering (OOR) is introduced as a way to inject locality in Java applications, by using copying collectors, finding hot (frequently used) field accesses and storing objects in memory using heat information. The authors in [23] concluded that their framework optimizations improved locality aspects with a negligible overhead.

Regarding one of the problems studied in this dissertation - OO related overheads (Chapter 3.1) - the need of high-level low-level programming is enhanced in [16], where the authors explain that underlying abstractions commonly used in Java may represent a source of overhead in program performance. As such, an optimized framework is implemented addressing the problems of data representation (primitive, compound and unboxed types) - featuring, for instance, layout control of fields/attributes inside a class.

[4] `http://jikesrvm.org/`

Object-online reordering in Jikes

In the interest of locality, object-online reordering (OOR) in Jikes virtual machine goes through three phases: static hot field access identification, dynamic hot field identification and garbage collection reordering [23].

(i) **static analysis**: at *compile-time*, the program code blocks are marked as *cold* or *hot*. This distinction is made through a series of heuristics mentioned in [23], like loop analysis and branch-prediction; for instance, loop code blocks are marked as hot since these are eligible to be run multiple times.

(ii) **dynamic hot field identification**: samples are periodically taken at *run-time* for methods and fields accesses - *heat* parameters are then redefined for each code block at each sample.

(iii) **garbage collection reordering**: at *GC-time*, hot blocks are traversed first, and an array with all heat values is created to map hot fields; finally, hot fields are enqueued in memory according the order of the heat-array - the data layout is in agreement with the program's memory access pattern.

When compared to the work presented in Jikes, although with the same concerns in mind, the approaches presented in this dissertation rely more on data structure optimization aspects - we do not aim at run-time analysis nor garbage collecting methodologies to improve locality, instead we study data layouts, optimization opportunities in memory access patterns and the problems of core features in object-oriented development.

2.5 Graphs

In this section we focus on graphs, a type of data structures with a seemingly amorphous representation in memory. Graph considerations in computer science can extend from development methodologies for generic purposes and usability [35] (such as Jung and JGraph libraries), to lower level considerations like memory usage and performance [29, 43, 50, 36].

2.5.1 Existing graph libraries and tools

Although serving different purposes, there are several libraries and tools that allow the use of graphs in the most distinct areas like social networking, flight travel map, statistical analysis or general optimization problems. Libraries like JUNG or JGraph implement generic graph data structures and algorithms that consider a wide range of concepts - directionality, weighed or unweighed, multi-graph (one edge connecting more than two nodes), etc. These general-purpose tools and data structure implementations are separated in several layers, distinguishing their formal specification from the implementation decisions, by applying APIs, generic data types and inheritance to represent multiple graphs variations. Graph data structures in this general context

are usually represented has *maps* between nodes and a corresponding list of edges representing the neighbouring nodes (and respective weights if applied).

2.5.2 Development methodologies

When it comes to develop a tool or library capable of efficiency and structural flexilibity there is a wide range of methodologies possible to apply. In this section we will focus on development methodologies applied in the field of software engineering and advanced computing more turned to our scope, graph based irregular applications. The use of generic ideals is the main goal when mentioning development issues, therefore generic programming is widelly cared for.

Product-line methodologies

A *product-line* (PL) is software product that results from using abstraction models and interfaces in order to compose a set of features an application should have - thus making an application - where the idea is to define isolated members and building elements that are able to be articulated and combined with each other, thus creating an application with its own set of features. The purpose of PL methodologies comes from the convenience of building an application based on a group of features, rather than writing a whole application from scratch; PL is more practical and cheaper.

The authors of [35] show that the areas in which one should apply different kinds of *product-lines* are still unclear in the scope of computer science. Different technologies, as well as different design orientations (such as object-oriented) may have applied and may themselves apply different PL methodologies.

Graph product-line

Regarding implementation issues, the studies made in [35] provide a good insight in this subject. Graph product-line (GPL) is a family of classical graph applications where each application is built by combining graph-based features modelled as a common grammar. A central property of PL applications, is that some features may block the appliance of some other features. For instance, in order to implement an MST algorithm the graph must be weighted. In this thematic *GenVoca* is introduced as a step-wise extension modelling view of building an application - the underlying idea is "*programs are values* [35]".

The evolution of implementations presented in [35] is very important since it denotes the fact that the usage of different implementations in different applications may result in very different performance outcomes. The three different implementations presented are: *(i)* adjacency lists, *(ii)* neighbour list and *(iii)* edge-neighbour representation. The differences between each one of the representations came (as the authors imply) from a series of optimization attempts and improvements. One of the problems identified by the authors was conceptual shifts that might

have been occurring when evolving the implementation, and its effects on the final application/algorithmic results; the example given was the folding of conceptual objects into singled physical objects, i.e., premature optimizations can obstruct development.

The generic graph component library (GGCL)

GGCL is a framework developed in C++ that makes use of technologies like the Standard Template Library (*STL*). The goal is to produce a tool that follows object-oriented philosophies, providing functionality, reliability, usability, efficiency, maintainability and portability [37, 32]. The work presented at GGCL shows a generic graph programming framework, where the concepts and ideals of the *generic programming paradigm* are applied.

Algorithms implemented with GGCL don't operate directly with data-structures, where usually graph specific data like *node* and *edge data* is stored, unlike the majority of graph tools available (LEDA, Graph Template Library, etc.). Most existing graph tools are inefficient when it comes to flexibility - properties like the *weight* and *color* of an edge are often explicitly hard-coded on algorithms, which leaves little further usage on other possible contexts or algorithms. The GGCL framework makes use of the *visitor pattern* concept, applied with generic programming concepts in order to implement generic operations (a concept similar to functors). The interaction between data-structures and algorithms is made via abstract interfaces on the graph domain: vertex, edge, visitor and decorator. Generic data-structures may not be usable enough in graph domain problems, therefore, specific interaction interfaces were needed so that the authors define generic enough algorithms (using graph abstractions).

The *graph*, *vertex* and *edge* concepts are fairly easy to understand. On the other hand, *visitor* and *decorator* concepts are more complex, however interesting. A *decorator* is presented as a "*generic mechanism for accessing vertex and edge properties*"; these properties are, for instance, weight or color. *Visitors*, much like functors, define the behaviour of generic operations. This proved to be an easy and efficient way of adding operations to an algorithm without changing the source code of the original implementation. These two features, allow for more flexibility when using graph structures or algorithms. Also, the programmer is able to extend and add new user-defined visitors and decorators. In terms of performance, the GGCL framework proved to be more efficient than other commercial and vendor-tuned libraries like LEDA.

2.5.3 Parallel Irregular applications

Irregular algorithms are mostly based in pointer-based data structures and are called so because one cannot assume or decide on the future operations it will do due to an external factor that is out of control of the programmer or algorithmic control. This factor can be influenced, for instance, by probability or the data itself, i.e., the fact that we do not previously know data values and layouts does not let us apply optimizations such as in matrix multiply with tiling or *for*-loop reordering for better cache locality. In typically irregular algorithms one is not aware

of the sequence order of data accesses.

A specific branch of irregular algorithms are graph related algorithms and a good part of recent research on parallel irregular applications using graphs is done in Galois - a parallel irregular algorithms framework that unveils the concept of parallelism at run-time.

Parallelism in Galois

We will see that the Galois parallel framework is based on the *optimistic* parallelization technique, over the *pessimistic* and *inspector-executor* techniques [29]. A brief overview of the parallelization techniques is followed.

- *Pessimistic parallelization*: The standard way to start parallelizing sequential code is to perform an analysis to identify the code regions that are independent from one another. Once these independent regions are identified they can be run in parallel, therefore, if the iterations on a loop are independent they can all be run in parallel. The reason Galois cannot use a pessimistic parallelization technique is that, for example, for the *Delaunay Mesh Refinement* [50] problem some iterations may be dependent from each other, since the mesh can have overlapped cavities/regions.

- *Inspector-Executor parallelization* [13, 44]: This approach splits the process of parallelizing in two phases: (i) *inspector* phase that determines dependences between work units and (ii) an *executor* phase that uses the schedule to perform the parallel computations. This approach is not suitable when using data-structures that change throughout the algorithm, since the structure inspection is only performed at the beginning and ceases to be accurate whenever there is a change in the original structure.

- *Optimistic parallelization*: The optimistic approach is based on speculatively executing regions of code while relying on some *run-time mechanism* to check for dependencies. If a dependency is detected and one of the threads cannot run in parallel, one of the changes is rolled back. Some work has been made regarding run-time mechanisms that check for loop conflict detection at hardware level and rollback systems are also studied under the subject of *Thread-Level Speculation* [54].

Galois was born due to the need to explore parallelism in irregular algorithms more easily by using the available API to represent irregular structures (*amorphous data-structures*) and by annotating which loops are to be run in parallel. The key idea here is to improve performance not by focusing on the study of detailed low-level pointer-based structure implementations, but by looking at algorithm's patterns and focusing more on data parallelism that may come from it [43].

The work done with the Galois system has proved that one of the main problems in graph usage in many applications is in fact the level of sparsity that a graph may reach, posing unique and difficult problems to memory architectures. Galois has its goal centred on optimizing not

only the irregular programs that are possible to run on such amorphous data structures, but also identify general-purpose optimizations that may be present on several irregular algorithms. Galois has also provided several tools of parallelizing an irregular algorithm operating on irregular data structures, and introduced the concept which the researchers named *amorphous data-parallelism*.

The initial studies of the Galois research team showed that despite of the apparent lack of parallelism it is possible to distinguish a series of parallization options on irregular data-structures. These optimizations consist in properly using abstractions for the data-structures in such programs. This led to an OO development of the Galois system, and proved that *optimistic parallelization* (based on abstractions) is a viable way and a starting point to parallelize irregular algorithms. The conclusions drawn from their experiments assert that exploiting data-types semantically proved to be a good way to allow concurrent accesses and updates to shared objects [30].

The authors of [36] provide the definition of amorphous data-parallelism: *"Amorphous data-parallelism is a generalization of conventional data parallelism in which: (i) concurrent operations may conflict with each other, (ii) activities can be created dynamically and (iii) activities may modify the underlying data-structure"*.

Unveiling parallelism in irregular applications

Unveiling the parallelism in applications has been increasingly critical since the appearance of parallel environments. Many regular applications like matrix factorization and BLAS kernels are well known and the memory access pattern is made well-conducted through a series of optimizations - however, irregular codes do not share this feature. The main difference, stated in [28], between regular and irregular codes is that: *regular* codes depend on the input size as for *irregular* codes depend upon the values of such input. The ParaMeter tool featured in [28], works under the *inspector-executor* technique in order to unveil the possible parallelism in an irregular application by (i) creating and analysing *computation graphs*, and (ii) creating a schedule on such computations by deciding over the *computation graph* (that may be undirected, and made directed[5]). In ParaMeter, *inspection phases* generate the computation graph and identifies dependencies; *execution phases* execute the current computation step and sets up next steps considering data dependence statuses. Both these phases are interleaved in order to generate the conflict graph correctly.

Delaunay Mesh Refinement in Galois

One of the case studies of Galois is the Delaunay Mesh Refinement [50] (DMR), which is a technique to generate: triangular meshes suitable for interpolation, the finite element method and the finite volume method (see Figure 2.2 [55]). The challenge with this technique is to get

[5]An undirected computation graph means the order of actions in the program is not relevant; in a directed graph the order in which actions are taken is considered - a schedule is created.

to a triangulated solution where a good result is fixed on a series of constraints: *(i)* the angles in the triangulations should insert themselves in an acceptable interval; *(ii)* the triangles should not be smaller than necessary nor bigger than desired (solution quality).

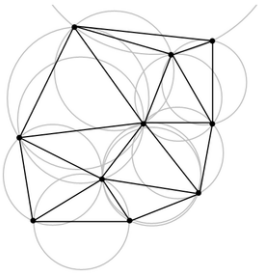


Figure 2.2: A Delaunay triangulation in the plane with circumcircles shown [55]

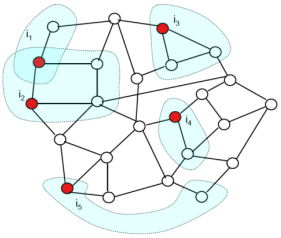


Figure 2.3: Galois view of DMR problem [36]

As we can see in Figure 2.3 [36] , Galois looks at data in a *data-centric* perspective. This data-centric view is based on thinking of an algorithm in terms of the data it is applied to [36]. An algorithm is thought of as the repetition of certain *activities* on nodes or edges of the graph. For this example, it identifies the active (red) nodes in each partition (highlighted blue areas), where there is a case of data dependency between i_1 and i_2. This is solved by implementing a roll-back system and backing up data in order to recover the data that would have been lost in locked decisions. With Figure 2.3, we see more easily the concepts in which Galois based it optimizations: *(i)* the locks necessary to apply are only applied at the *boundary nodes* instead of locking the whole partition and *(ii)* activities are more easily modularized with generic iterators throughout the data.

Chapter 3

Optimizing data structures

Optimizing data layouts in data structures may be challenging because they have specially deep impact in irregular sorting and graph algorithms are a good example of that. In modern high-level languages the problems of memory usage are aggravated. In this chapter, the main subjects are a class of memory straining algorithms, heap sorting problems, and an irregular graph algorithm, Prim's MST, where the underlying data structures (graphs and priority-queues) and possible optimizations are studied. Section 3.1 works as a case study introduction to the problems found when applying data layout optimizations in OO frameworks. Graphs are addressed in a more formal manner in Section 3.3, where graph concepts are exposed (undirected, weighed, adjacency lists, etc.). Data layout optimizations are applied to a regular binary heap in order to improve locality - the van Emde Boas layout is presented in Section 3.2. The compositions applied in cross-structure optimizations raise some problems regarding object-oriented (OO) collections and solutions are proposed at Section 3.4. The benchmarks presented for both case studies are in accordance to the methodologies explained in Section 3.5.

3.1 Data containers in Object-Oriented frameworks

The appearance of class hierarchies in programming has allowed software developers to build, change and maintain software solutions in a relatively easy manner enhancing the importance of abstraction and reuse. This reuse philosophy has enabled developing methodologies where flexibility and modularity are key features. The concept of object-oriented (OO) is based on abstractions in which a programmer looks at parts of a program in a modular way. This abstraction in OO programming is bloated in such a way that even more abstract design methodologies have risen to the level of almost visual programming.

On the one hand, OO abstractions enabled the rising of new design patterns and methodologies like UML (Unified Modelling Language) [6], developed up to the point of visual engineering and aiding programmers by defining helpful abstract program semantics. APIs (Application

Programming Interface) play an important role in this context, these represent the set of generic operations used in a certain domain by multiple entities. The consequence of this is encapsulation which offers security to possibly critical decision implementations. On the other hand, the mechanisms that manage these abstractions may be a source of overhead.

In this section we focus on generic typed mechanisms often used in OO frameworks and how abstract data types (ADT) can influence performance for scientific applications. The focused subjects are related to the problems of auto-boxing types and encapsulating APIs. Let us refer to the specific case of Java OO development.

3.1.1 Type erasure in Java collections

Type erasure in Java is known as the process in which the Java compiler only treats data types at compile time for type safety checking and then removes all type information for classes, methods, arguments and parameters. For instance, if a anyList = new List<Integer>() is declared in Java, the compiled byte codes will only recognize the object anyList as being of type List, being able to contain arbitrary type objects, or memory references to objects. This also happens when using generic data types, where type compatibility checking is only done at compile time. In C++, for instance, when using generic data types mechanisms known as *templates* the compiler does not perform type erasure: template <typename T>.

The template mechanisms in C++ are basically code generation mechanisms called at pre-processing phases whenever a template is instantiated. Despite the absence of type-erasure, using templates in C++ leads to instruction-level code bloating: different instantiations of the same template with different types, generates different codes; for instance, the generated code for myvector<int> is different than that of myvector<float>.

In Java this code bloat does not happen because every generic data type is treated as an Object instance, thus it is accessed via an *untyped* memory reference. In the case of generic-typed collections, this means that all elements are stored in untyped memory and accessed via a collection of memory references. This feature makes code reuse in Java a powerful tool for developers. It is more or less intuitive that abstract data types lead to re-usability and improves software development productivity - abstraction is a primary concern. In Figure 3.1 are outlined the main concepts that led to the main problems found in the scope of this dissertation.

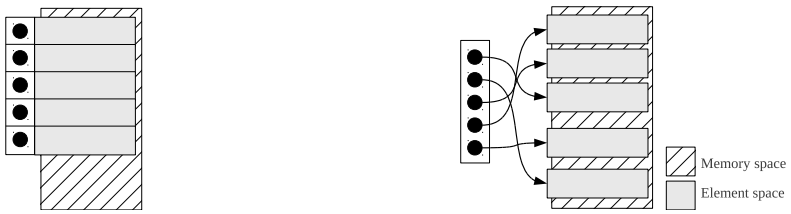
3.1.2 Main reasons for overhead

Boxing is the process of wrapping primitive data type values in corresponding classes which are object memory references, in the case of Java. Auto-boxing occurs in the Java compiler as a way to automatically do this conversion with no need of explicit specification. The reason is: when using, for instance, a primitive type value int in a generic type environment (such as an API using the type T), Java automatically boxes the value into the corresponding type object Integer. For instance, when an array of type T is declared, one normally expects to have all

Abstraction
⇓
Code reuse and usability
concepts
⇓
Type erasure: data types are
not needed to run a program
⇓
All objects are treated as
memory references
⇓
Leading to APIs, auto-boxing,
encapsulation, etc.

Figure 3.1: Concepts that led to problems of object-oriented in HPC.

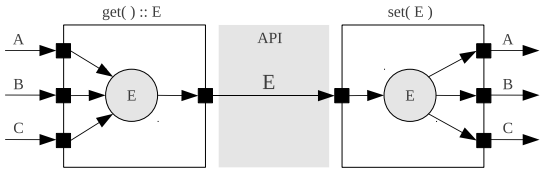
its elements (of type T) to be in contiguous memory regions. The Java generic mechanisms apply however another level of indirection between the collection and the stored value. Also, and due to traditional garbage collection, the JVM manages the allocation and moves objects around according to its policies - this may break the expected element contiguity (and often does). In the themes approached in this dissertation, where an intensive use of collections is applied, automatic processes like *boxing* can result in major overhead. The main problems are: (i) auto-boxing means extra overhead-ish operations and (ii) elements in Object/generic-typed collections may have an unexpected layout in memory. Figure 3.2 depicts this phenomena.



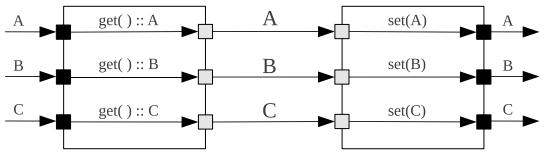
(a) Primitive array (b) Generic collection

Figure 3.2: Primitive and generic type array representations in memory

Another aspect in OO is the usage of generic APIs to modularize the access to certain properties in specific parts of a program. The overhead related to APIs has to do with the data types handled by them: APIs can handle generic data types which can be bottlenecking areas for data layouts because data structures are obligated to obey a certain structure which may originate redundant object instantiations.



(a) Encapsulation



(b) Decapsulation

Figure 3.3: API encapsulation and decapsulation schematic

Encapsulation in OO collections

Optimizing data-structures is not enough on most cases. In OO languages that handle generics, *encapsulation*[1] is usually a good feature because it is a consequence of generic APIs which favours modularity and programming flexibility. However, if we use optimized data-structures in conjunction with APIs, the attempt of simplifying implementations through the use of a common *language* (an API) creates an implementation prone to instantiation overheads. The specific case of combining optimized structure with APIs that handle generic data types (i.e., objects) originates redundant objects, when the goal could simply be *communicating a value*. The communication of values via a generic API is made through objects, and to use an object one must create an object, which means additional computations and memory allocation.

In scientific algorithms and specifically in pointer-chasing problems, encapsulation can introduce redundant object instantiation and deletion operations resulting in instruction overhead and straining of the memory system. In Figure 3.3a we see an example of the encapsulation process: here we illustrate a hypothetical *get* method from a data structure - in order to obey the API, the initial structure must return an object E, therefore the method must create a new object to return at the second structure[2] (which is obeying the same API), where its *set* method *decapsulates* the previously encapsulated attributes.

As expected, encapsulation is an extremely heavy operation for data-intensive algorithms, one way to optimize this process is creating a *decapsulated* API (Figure 3.3b) where each attribute is *returned* (by the first structure) and *received* (by the second structure) independently. This kind of API is now not as modular and flexible than the previous one but the bottleneck is greatly

[1]The concept of encapsulation is also referred to as the process of encapsulating data in an abstract entity, an object. In the context of this dissertation, we consider encapsulation as the liabilities APIs impose in data structures.

[2]In practical terms, the first structure does not return directly to the second structure - there is usually a middle layer.

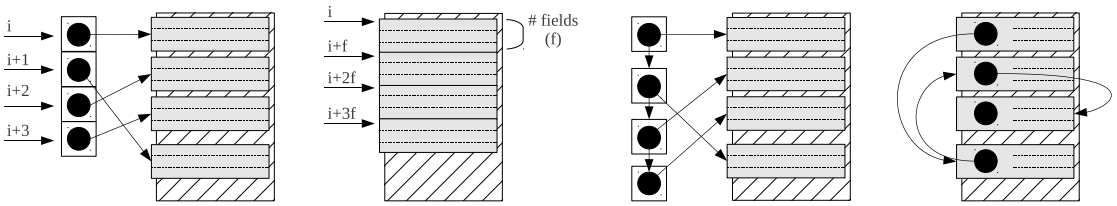
reduced.

Clarifying generic APIs

We consider a generic API as, not only using generic data types, but also as an abstract concept, or a tool to imprint modularity into implementations; they are usually used to form a common language in an application, which different parts, or modules of the program use to communicate - this creates application layers.

Generic collections benchmark: overhead of boxing

This section aims to show the overhead of auto-boxing in collections with generic APIs. We study two kinds of collections - linked-lists and array-lists - in order to show the underlying overhead of auto-boxing. For each collection it was implemented an optimized variant with unboxed elements to remove the level indirection added by generics.



(a) Java AL (b) Opt. AL (c) Java LL (d) Opt. LL

Figure 3.4: OO collections and pointer complexity.

In Figure 3.4a (Java AL), the native implementation in Java of an ArrayList, is noticeable that Java generic mechanisms introduce *unnecessary* pointer-resolving operations: between the list element and the actual element in memory. The other collection is a LinkedList, and as the first one, it creates implicit memory references. In order to measure the instruction and cache behaviour, these implicit pointers are removed as illustrated in Figures 3.4b and 3.4d. In the figures we consider objects with multiple fields, however, for our benchmark test case we perform a sum over a list of integer elements, one field solely is considered. In Table 3.1, we present the benchmarks made to the structures in Figure 3.4, using $10{\times}10^6$ integer elements, and performing a sum over the entire list.

In array-list implementations: Java AL accesses elements by first resolving memory references to objects that were previously boxed. In addition to failing to ensure element contiguity, it needs more operations to resolve an access to an element. The raw view of the problem: an array is expected to store all its elements in contiguous in memory addresses, as it is in Figure 3.4b. The opt AL version is more efficient because it accesses elements directly. Notice that the count of L1 accesses corresponds approximately to the number of int elements added to the array ($10{\times}10^6$) - this means that elements are in contiguous locations in memory and pertinent data is brought

Table 3.1: Benchmarks comparing *Array-Lists*, *Linked-Lists* and its variants

	java AL	opt AL	java LL	opt LL
Instructions ($\times 10^7$)	24.952	1.333	22.989	24.204
Cycles ($\times 10^7$)	19.810	2.360	28.639	20.854
L1 accesses ($\times 10^6$)	80.415	10.136	90.611	101.799
L1 misses ($\times 10^6$)	5.295	0.628	7.571	4.664
L2 accesses ($\times 10^6$)	10.564	0.018	15.766	9.510
L2 misses ($\times 10^6$)	5.327	823×10^{-6}	7.831	4.698
Time (s)	0.100	0.016	0.148	0.107

into L1 cache - however, still incurring in misses because the dataset does not fit in memory (approx. 39 MB).

Comparing the results for *Java LL* and *opt LL*, it is noted an increase in instruction count due to the increase in code complexity for the optimized version: this fact would not be expectable, because a pointer resolving layer was removed, unless we account for JVM management of allocated objects and corresponding memory references, i.e., the JVM needs less operations to load elements due to implicit memory referencing, which results in less explicit loading instructions - this is confirmed by the lower L1 cache accesses count for Java LL. However, cache efficiency is higher for *opt LL* - approximately 39% less L1 cache misses - making the number of cycles decrease and thus, running time. This decrease in L1 cache misses is explained by the lower memory space requirements for the optimized version: the native Java Linked-List stores elements in boxes with meta-data, making each list element take up more memory space - traditional object encapsulation in Java.

Summary

Object-oriented frameworks usually operate with high-level languages and abstractions that may introduce features that go against core functionalities of high performance computing. Features like memory management, type erasure that transform types in memory references, element adjacency in collections and over-layering in cross structure optimizations. We identify object encapsulation (related to type erasure and auto-boxing) and structure encapsulation (related to API usage and generic mechanisms) as problematic issues in the resolution of overheads and refinement of intensive algorithms.

3.2 Sorting

In this section we address to the problem of memory efficient sorting, which is a well known problem in memory architectures due to its memory-bound nature. We address to heap based sorting primitives, specifically a binary heap and an optimized variant based on the Van Emde Boas (VEB) layout [15].

The importance of this matter is mentioned by many authors as being of crucial importance to several cache-oblivious graph algorithms - the main bottleneck in MST graph algorithms is precisely the memory bound complexity created by high connection rates in dense/sparse graphs, where the efficiency of an algorithm depends on efficiently sorting connections between nodes of the graph. Before jumping into more complex sorting primitives a simpler implementation is presented, the *binary-heap*, in order to gradually advance to our VEB implementation.

3.2.1 The heap sort problem

The objective of many existing implementations of *priority-queues* is to quickly find the minimum or maximum element, at the account that it is at the root position of a tree. A heap is a tree based data-structure that follows a specific property (*heap property*), which is defined as follows: the key of a child node must be higher or equal than the key of its father - when applied in a *min-heap* - a *min-heap* (Figure 3.5) is sorted in a decreasing manner. There are no restrictions limiting the number of children each node has in a heap.

The operations often found in a heap are: find minimum/maximum, delete node, insert node and increase/decrease key. There is a wide range of heap variants, each with its peculiarities when maintaining the heap property. For example, the *binary heap* needs sift-up and sift-down operations when inserting and removing, in order to keep the heap property verified. A simpler case of variant is a *d-heap* where each node has d children (d corresponds to the degree of the tree).

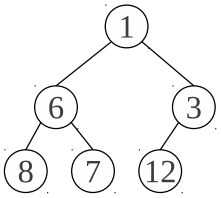


Figure 3.5: A min-heap example

A binary heap is a data-structure based on a binary tree and it is the specific case of a *d-heap* with $d = 2$. In order to maintain the heap property, this heap needs to implement two specific operations commonly known as sift (or percolate) operations: *sift-up* and *sift-down* - these operations maintain the heap property verified and deserve proper focus because they traverse the tree up and down (respectively), until the reaching the *heap-property* or the top/bottom of the tree, representing the memory straining operations in this implementation.

Inserting in a binary heap

When inserting an element in a binary heap, the new element is added at a leaf position and swapped iteratively up through the tree until the heap property is reached - this is called the *sift-up* operation. This operation is performed in $O(log_2(N))$ time (N = number of heap elements).

See schematic in Figure 3.6.

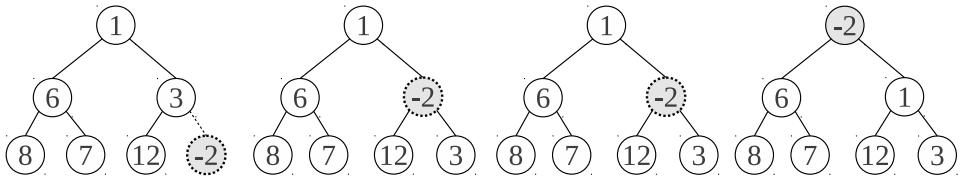


Figure 3.6: Inserting a new element (-2) and sifting up

Removing in a binary heap

As for removing, the underlying operation is *sift-down*. The time it takes to find the lowest key value is $O(1)$ since this element is the root element. After removing the root, the last element stored in the heap (in a leaf position) is made the root and sifted down the tree to assert the heap property. The sift-down operation also runs in asymptotic time of $O(log_2(N))$. See schematic in Figure 3.7.

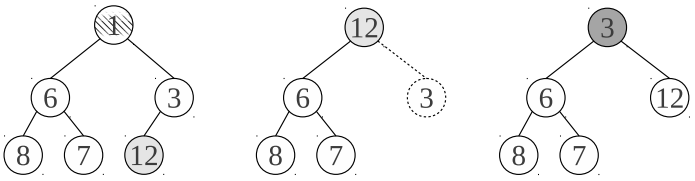


Figure 3.7: Removing smallest element (root) and sifting down

Implementation details

Implementing the insertion, removal and sifting operations is fairly simple. An important detail regarding the data-structure is that it is an array containing all heap elements we want to order. The pointers to children and parents of each node are not explicit, these are computed at run-time. Because the data layout is represented in a breadth-first manner, the parent/children index computations are fairly simple (see Figure 3.8).

- computing children indices: the expressions for left and right child are, respectively, $2i + 1$ and $2i + 2$
- computing parent indices is also simple, $\frac{i-1}{2}$

In order to simplify the computations for navigating in the tree, there is one simple optimization one can perform: "ignore" the first array position (index 0), making it *null*, i.e., perform a conceptual right-shift in order to ignore the 0-index element because it is the null-able multiplier, this way we are able to find the left/right children and parent indices with the following expressions:

$$left(i) = 2i \tag{3.1}$$

$$right(i) = 2i + 1 \tag{3.2}$$

$$parent(i) = \lfloor \tfrac{i}{2} \rfloor \tag{3.3}$$

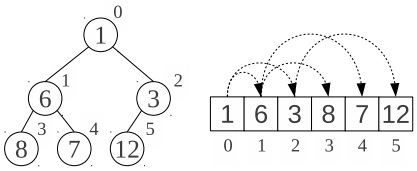


Figure 3.8: A binary heap and its array representation

This heap version is simple to implement and understand, also it behaves rather well in terms of asymptotic times and complexity. However, locality is a main concern, that is why we continued to study this heap type and tried to optimize it and combine it with other important and ingenious ideas and layouts - we focus on data layouts in the next section.

Sorting with auto-boxing

To demonstrate the overhead that auto-boxing adds to data intensive applications we perform some benchmarks with generic data typed priority-queues (with Integer) using a binary heap sort method. In Table 3.2 we present the native priority-queue used in Java and an analogous generic implementation of a binary heap (using Integer elements), and a binary heap variant with primitive data types (int).

Table 3.2: Benchmarks comparing priority-queues with boxed and unboxed types.

	Java PQ	BinHeap Integer	BinHeap int
Instructions ($\times 10^8$)	132.15	151.86	73.33
Cycles ($\times 10^8$)	201.71	195.00	58.14
L1 accesses ($\times 10^8$)	68.91	79.75	28.15
L1 misses ($\times 10^6$)	234.73	231.12	39.51
L2 accesses ($\times 10^6$)	417.60	405.62	93.94
L2 misses ($\times 10^6$)	279.37	261.63	50.82
Time (s)	10.113	9.700	2.868

In the first two columns we compare the analogous implementations of binary heaps (Java's native priority-queue is also a binary heap). The implementations differ in object management related issues and thus the differences in instruction and cycle count. For disambiguation, these differences are in object allocation: in Java PQ the JVM is in charge of allocating all elements, as for BinHeap Integer, the storage space is declared at the initial instantiation; this may take

more instructions but takes less cycles and execution time. When using primitive data types, i.e., removing one level of indirection between the collection and memory addresses, the JVM is no longer in charge of element management in memory (happening in Java PQ and BinHeap Integer). Instruction counts decrease considerably due to reduced pointer-chasing operations. Also, since the data layout in BinHeap int is not handled by JVM, the implementation has tighter control of memory accesses taking benefits from the loads of elements made in the same array. Its storage array is a contiguous chunk of memory, where all its elements are layed out according to the binary tree specifications. Sift operations are the main bottlenecks: to swap elements the implementation does not swap the memory references but the values in memory - in order to reach, get or set a value in BinHeap Integer, several pointer-chasing operations have to be resolved to change the value encapsulated in an object.

The highest overhead causing factor is the fact that elements are not in the same memory regions. Java PQ and BinHeap Integer version are pointer-based structures, only references are made implicitly by the JVM. In summary, boxed data types have an additional cost of *dereferencing* values.

3.2.2 Van Emde Boas data layout

The pattern in which memory is accessed depends, among several architectural details, in the way data is distributed through memory, for instance, the data layout of the previously mentioned binary heap is a *breadth-first* layout - looking at the array distribution in a binary heap (Figure 3.8), parent and children nodes are not in contiguous array positions, unless at the root of the tree.

In order to introduce memory locality in these implementations we studied the Van Emde Boas layout, presented in [15]. The work at [15] introduces a hierarchically decomposed dynamic search tree structure, in our case, the tree is implicitly decomposed by redefining index computations (for children and parent) to achieve the desired access patterns. The data structure is hierarchically decomposed tree that takes advantage of the *depth-first* layout of elements in memory, so it can perform its operations with good asymptotic memory bounds.

The main feature about this layout is the arrangement of elements which allows to minimize accesses to lower and more expensive memory levels. Since children elements in deeper levels of the tree are in the same index/memory regions cache miss rates naturally decrease.

To avoid creating unused memory wholes in the structure the tree is balanced in a quick and efficient way: by using a packed memory structure, an m-bit array that makes use of the mathematical properties of binary trees to represent indexes with operations over bit-values. This heap allows for good asymptotic times with space complexity of $O(m)$[3] and insertion, deletion and search operations with complexity $O(log\ log\ m)$ (see schematic in Figure 3.9).

[3]The original representation [15] uses an auxiliary structure, an associative m-bit array to keep track of key positions and balance.

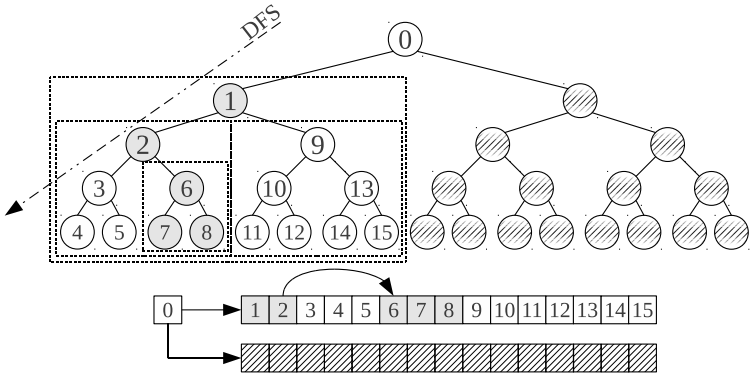


Figure 3.9: Van Emde Boas layout

Note about priority-queues

As an important side note, the originally presented Van Emde Boas heap uses keys to prioritize heap elements. So this heap sorts non repeating values - each key is unique. In our case, we use priority-queues as sorting primitives that retrieve the maximum/minimum element from a list where repeated values are allowed, we do not use prioritizing keys because the main goal is to study the cache behaviour of the presented data layout. For our work, we consider key-prioritizing elements an additional cost. So we move forward in implementing a simpler version of a Van Emde Boas-based heap, or blocked version of a binary heap.

Cache-friendly heap sort - Van Emde Boas Heap

We implement this data layout as a binary tree also, however data will be distributed differently from earlier versions of binary heaps. We did not explicitly implement a VEB data-structure, we simply organized array elements using this concept. It consists in looking at the binary tree data-structure from a blocked point-of-view[4] so that each memory access to the root of each block access may also bring data from adjacent nodes, preferably children nodes. Note that there is a difference between this concept and our first binary heap example: in this version we use the first array position to store the root element, contrary to the earlier binary heap version which maintains the first position *null* to simplify index expressions.

The tree structured layout in Figure 3.10 has a block height of 1, thus, in a binary heap, the number of elements per block is 3. As shown, we now also consider a *block hierarchy* - apart from regular children and parent nodes, we now have the notion of children and parent *blocks*. For VEB-based binary heaps with $height = 1$, each block has 4 children blocks. These specifications vary according to the degree of the tree (binary, ternary, etc.) and the height assigned to each block. Other tree degrees are left aside and we focused only on *binary* trees.

[4]There are not explicit memory blocks, only the data will be layed out in a blocked manner

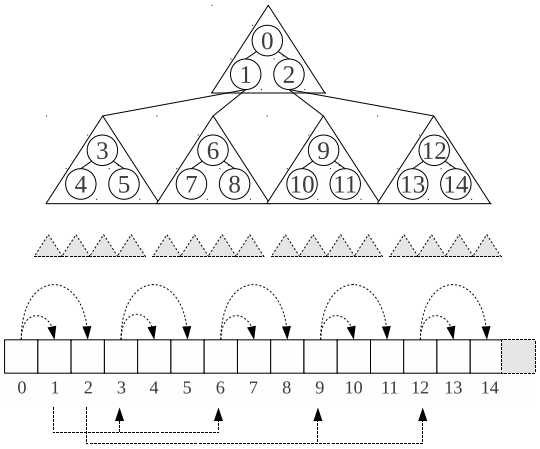


Figure 3.10: The VEB data layout - numbers correspond to array indices

Objective of Van Emde Boas-based layout

The objective with this layout is to increase the cache hits by placing in contiguous memory addresses the children of root elements in each block. In a quick and shallow analysis to the hit rates of a binary heap we perform an example tree traversal on both binary and VEB heap data-structure so we can better figure out how the cache hit rates increase in VEB heap. Let us consider:

- for simplicity and easier explanation, that we are working with a cache line size of 3 elements (not considering type sizes) - assume that each memory access to an array position will bring into cache memory the *two* next adjacent array positions.
- the traversal of a tree in both binary and VEB representations - it is important to mention that: even though the tree might not be the same the only concept we want to apply is the traversal following a determined path down the tree in order to check the cache hit/miss rate in both implementations. The objective here is to traverse both structures in the same way. An arbitrary path down the tree was chosen (L = *left,* R = *right*): $L \rightarrow R \rightarrow R \rightarrow L \rightarrow R \rightarrow L \rightarrow R$.
- the basic cache behaviour: an access to an array brings into cache a chunk of size equal to the cache line size, with the adjacent positions of the position accessed.

Considering the traversal path presented, in Figure 3.11 we present a schematic for the traversal in a binary tree. We can see that in a total of 8 array accesses, 6 of them are misses (dark dots), which means the element is not in cache, originating in a cache miss ratio of $\frac{6}{8} = 0.75$, because there is no previous access to an adjacent interval of 3 positions[5]. Note that each block

[5]Depending on the memory architecture we consider, some architectures may provide backward adjacent access, i.e., not only it loads into cache the next adjacent addresses, but also the backward adjacent accesses relative to

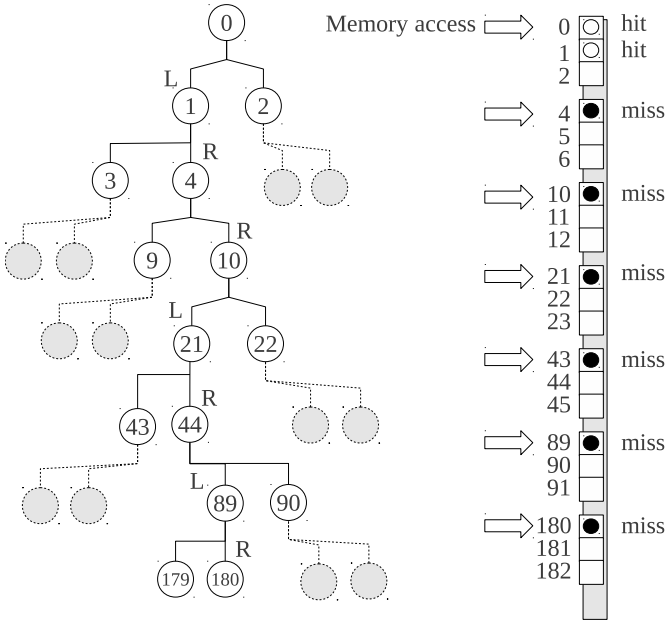


Figure 3.11: Schematic to explain the traversal details in a binary heap

presented in the array corresponds to the amount of elements loaded into cache memory (3 elements).

In Figure 3.12, we present the VEB schematic for the considered traversal path showing a blocked VEB layout. For simplicity we used a VEB block size of 3 elements, which matches the assumed *cache line size*. The hit ratio increases due to the blocked manner in which array elements are organized - at the root of each block we have a miss [6] but the next element in the block is guaranteed in cache. Since the cache line size loads all elements within the block we guarantee that the next element is in cache. Cache hit ratio should increase if we consider a real cache line size with bigger VEB blocks. Cache miss ratio for this example is $\frac{3}{8} = 0.375$.

Index expressions

Finding the index expressions in this layout is more complex than in a regular binary heap, because there are two levels to consider: inner and outer-block indices.

- inner-block indices are computed normally as if they were in a regular binary tree (see equation group 1) - they depend solely on the degree, which for a binary tree is 2 (see Equations 3.4, 3.5, 3.6).

- outer indices, or *block ids*, depend on the inner block height, which defines the number

a memory address. However we will only consider forward accesses to simplify our explanation.

[6] The first block does not miss because usually we maintain a pointer to the root - compulsory misses are ignored in this analysis.

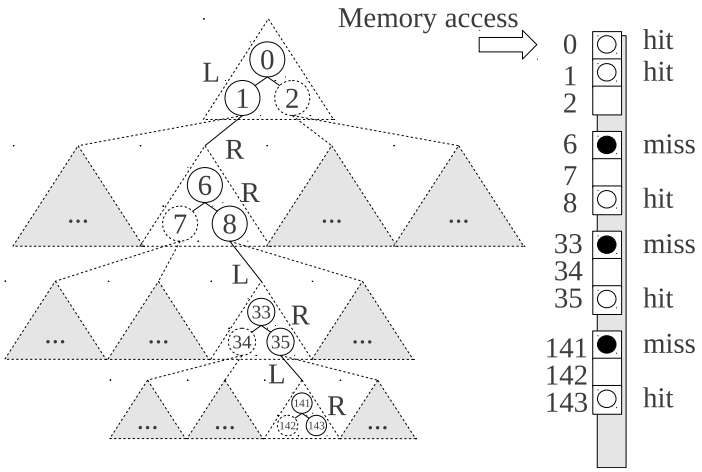


Figure 3.12: Schematic to explain the traversal details in a VEB heap with block size of 3

of leaves per block, which in its turn defines the number of child blocks each block has, $\#ChildBlocks = \#Leaves \cdot degree \cdot 2$ (in Figure 3.10, we have 2 leaves per block, thus 4 child blocks per block).

For this specific case of *inner block height* $= 1$, finding the left and right children and parent indices is simple:

$$inner\ left\ child(i) \quad = i + 1 \tag{3.4}$$

$$inner\ right\ child(i) \quad = i + 2 \tag{3.5}$$

$$inner\ parent\ index(i) \quad = i - (i\ mod\ 3) \tag{3.6}$$

The number 3 in the *inner parent index* expression stands for the number of elements in each block (which we call the *block size*). These expressions only work when navigating inside a block, to navigate outside the blocks we must be aware of

- if a given position i is in a leaf position: the *leaf id(i)* will decide which pair of children blocks to descend to;
- the *block id* of position i, $block\ id(i) = \lfloor \frac{i}{blockSize} \rfloor$.

The expressions to compute the outer indices are somewhat complex resulting in bottlenecks for our VEB data-layout, even for a constant block size of 3 elements, therefore we performed some optimizations to reduce this index computation overhead, presented ahead. The expressions are:

$$outer\ left_{child}(i) = \begin{cases} 4i-1, & \text{if } i \bmod 3 = 1 \\ 2i \cdot (i \bmod 3)+1, & \text{if } i \bmod 3 = 2 \end{cases} \tag{3.7}$$

$$outer\ right_{child}(i) = \begin{cases} i+4+1+((i \bmod 3) \bmod 2), & \text{if } i \bmod 3 = 1 \\ 2i(i \bmod 3)+4, & \text{if } i \bmod 3 = 2 \end{cases} \tag{3.8}$$

$$outer_{parent}(i) = 3\lfloor \frac{BlockId(i)-1}{2\#Leaves} \rfloor + \lfloor \frac{BlockId(i)\ \%\ (2\#Leaves)}{2} \rfloor + 1 \tag{3.9}$$

In the expressions above $BlockId(i) = \lfloor \frac{i}{3} \rfloor$ and $\#Leaves$ corresponds to the number of leaves per block, which in our case is 2. As the reader can perceive these expressions will result in a major bottleneck, since they are massively called inside simple iterations - such complexity in what should be atomic operations is heavy. Gathering all equations, the index expressions are:

$$left_{child}(i) = \begin{cases} i+1, & \text{if } \alpha(i) = 0 \\ i \cdot 4 - 1, & \text{if } \alpha(i) = 1 \\ 2 \cdot i \cdot (i \bmod 3)+1, & \text{if } \alpha(i) = 2 \end{cases} \tag{3.10}$$

$$right_{child}(i) = \begin{cases} i+2, & \text{if } \alpha(i) = 0 \\ i+4+1+((i \bmod 3) \bmod 2), & \text{if } \alpha(i) = 1 \\ 2 \cdot i \cdot (i \bmod 3)+4, & \text{if } \alpha(i) = 2 \end{cases} \tag{3.11}$$

$$parent(i) = \begin{cases} i-(i\ mod\ 3), & \text{if } \alpha(i) = 0 \\ 3\lfloor \frac{BlockId(i)-1}{2\#Leaves} \rfloor + \lfloor \frac{BlockId(i)\ \%\ (2\#Leaves)}{2} \rfloor + 1, & \text{if } \alpha(i) \neq 0 \end{cases} \tag{3.12}$$

$$\alpha(i) = i \bmod 3$$

The main problem with these expressions is that they are based on conditions to check the height a given index i is at - check if i is at a *block-root* position or at a *block-leaf* position (and for leafs we must check the left and right coordinates).

In Table 3.3 we present priority-queue benchmarks made to a binary heap and the VEB-based with a block size of three elements. In both heaps were inserted the same list of `int` values to ensure the same traversal paths; 10×10^6 elements were inserted, to guarantee that all `int` values do not fit in all cache levels (approximately 39 MB).

The VEB 3 heap shows better cache behaviour, about 37% less L1 and L2 cache misses. However, the index expressions for VEB are extremely heavy when compared to binary heap index expressions. The instruction count for VEB 3 is almost two times higher than BinHeap - since VEB 3 index expressions use up several local variables and they do not all fit in the number of registers available, cache memory space is being used, as proved by the higher number of L1 accesses for VEB 3. This increase in instruction complexity also increases the clock cycles the

Table 3.3: Benchmarks comparing a binary heap implementation to VEB-based heap

	BinHeap	VEB 3_{NotOpt}
Instructions ($\times 10^8$)	73.33	141.29
Cycles ($\times 10^8$)	58.14	86.95
L1 accesses ($\times 10^8$)	28.15	49.08
L1 misses ($\times 10^6$)	39.51	24.91
L2 accesses ($\times 10^6$)	93.94	64.24
L2 misses ($\times 10^6$)	50.82	31.66
Time (s)	2.868	5.391

program runs in, which also increases execution time.

Refining index expressions in VEB-based heap

As expected, the overhead caused by index computations in the VEB-based heap is much higher than the one found at regular binary heaps. Children and parent index expressions are now based in conditions which represent a serious stall in performance. We use *if* operations to process the positioning of the current index in the tree: it may be in a parent, child or middle position in a block. With the right amount of loop-unroll applied in sift-down operations one can completely remove *if* operations in index expressions if we assume the sift-down algorithm to start at the root of the tree. For blocks with size of three elements, we applied a loop-unroll of two iterations. In an ideal scenario, the amount of loop unroll is correspondent to the block height, however, this would mean completely unrolling cycles. We applied the same methodology for block sizes of seven elements ($height = 2$).

Another concern to have is the number of accesses to the array; we try to minimize accesses to the storing array by keeping the compared values in local variables (i.e., registers).

Listing 3.1: Traditional (non optimized) sift-down algorithm for heaps

```
1       siftDown(i) {
2         left = leftChild(i);
3         while (left < size) {
4           right = rightChild(i);
5           min = 0;
6           if ( array[left] <= array[right]) {
7             min = left;
8           } else {
9             min = right;
10          }
11          if (array[i] > array[min]) {
12              // swap elements ...
13              i = m;
14              left = leftChild(i);
15          } else {
16              break; // heap property is verified
17          }
18        }
19      }
```

With the VEB layout applied using the algorithm presented in Listing 3.1, *if* conditions in index expressions introduce a serious amount of overhead. In Listing 3.2, we apply loop unroll and make better use of array accesses to ensure that jumping memory accesses that may originate cache misses are not occurring, local variables wleft, wright, wmin, etc., serve this purpose. The optimizations for sift-up operations are analogous, so they will not be presented here (for consulting all optimizations see Appendix A).

Listing 3.2: Optimized sift-down algorithm for VEB-3 heaps

```
1 optSiftDown(i) {
2   left = 1, r = 2, wi = array[i];
3   while (left < size) {
4     wright = array[right]; wleft  = array[left];
5     if ( wleft <= wright) { /* update wmin */  } // ... Loop unroll 1
6     if ( wi > wmin ) { /* swap; left, right = ... */  }
7     wright = array[right];                        // ... Loop unroll 2
8     wleft  = array[left];
9     wi     = array[i];
10    if ( wleft<= wright) {  /* update wmin */   }
11    if (wi > wmin) { /* swap; left, right = ... */  }
12  }
13 }
```

By removing the *if* conditions from index expressions the computational pattern is very similar to the binary heap: instruction counts decrease when applying optimizations in sift-down for VEB 3_d in relation to VEB 3, as well as L1 accesses due to better register usage. Optimizing the sift-up operation also introduces more instructions (about 1.7% more instructions) but the L1 misses decrease - this means that removing the *fat* from the implementation made way for locality benefits inherent from the VEB-based layout that was hidden by index expression overhead. Execution time decreases successively as sift operations are optimized for VEB 3,

Table 3.4: Benchmarks comparing binary heap and VEB 3 and 7 versions: no optimizations, optimized sift-down and optimized sift-down and sift-up operations (d and u in VEB refer to sift-up and sift-down).

	BinHeap	VEB 3	VEB 3_{d+u}	VEB 7_{d+u}
Instructions ($\times 10^8$)	73.33	141.29	74.77	97.70
Cycles ($\times 10^8$)	58.14	86.95	56.93	74.09
L1 accesses ($\times 10^8$)	28.15	49.08	35.89	48.84
L1 misses ($\times 10^6$)	39.51	24.91	28.82	22.76
L2 accesses ($\times 10^6$)	93.94	64.24	71.03	59.66
L2 misses ($\times 10^6$)	50.82	31.66	39.63	30.60
Time (s)	2.868	5.391	2.805	3.661

reaching an execution in the same order of magnitude to the binary heap: from this we draw that the VEB-based heap (VEB 3_{d+u}), with only approximately 2% more instructions, shows better cache behaviour when a more cache-friendly data layout is applied in (approx.) the same execution time, comparing to BinHeap. To see further cache improvements, we applied the same kind of optimizations in sift operations to VEB 7 (block height of 2), making the code more complex (loop unroll now extends to three iterations). Despite the increase in execution time due to code complexity for VEB 7_{d+u}, the cache misses were the lowest.

The average memory access time (AMAT) and instruction counts for different priority-queue implementations are shown in Figure 3.13 for growing element sizes between $[1\times10^6, 10\times10^6]$. As already stated, using generic data types on java pq and gen bin heap we see the most bottlenecking implementations. The AMAT values become more stable as the heap size grows. The instruction count for gen bin heap are higher due to less efficient object in memory management. The implementations (int bin heap, int veb3, and int veb7) use primitive int data types: in these implementations boxing is removed and dereference costs greatly decrease as they all spend around 4.0 clock cycles for each memory access (considering our memory hierarchy, Section 3.5.6). An interesting result is noticed in the instruction counts for int bin heap and int veb3: these follow the same instruction counts - this verifies that the computational pattern between the two heap implementations is similar, however decreasing cache misses for int veb3. The instruction complexity for index expressions in int veb7 is once verified in the graphic, achieving higher instructions counts than int veb3.

Summary

In this section, we discussed the topic of memory efficient sorting, its problems and main bottlenecks, from two distinct points of view:

- memory and computational complexity, analysing traditional binary-heaps and promoting a simplified version of the van Emde Boas heap [15].
- object-oriented related problems; the problem of auto-boxing and its negative impact in

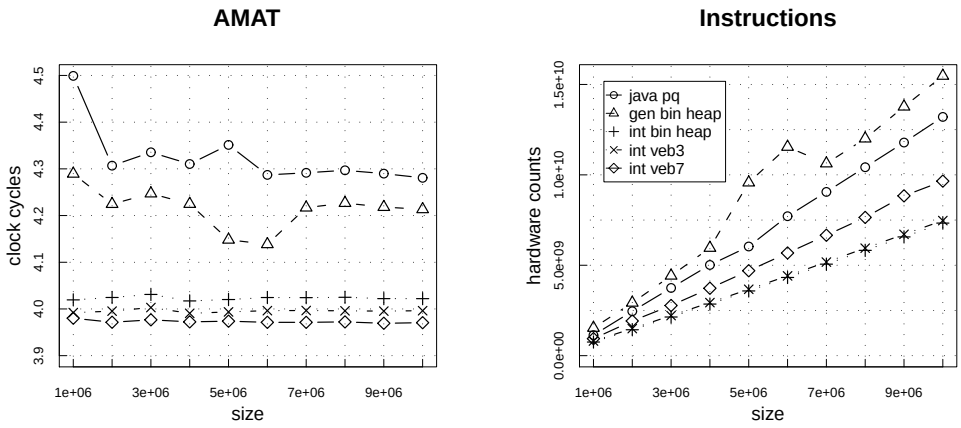


Figure 3.13: AMAT and instruction counts for heaps

maintaining the initially assumed element contiguity in memory, crucial for the VEB data layout optimization.

The execution times obtained for the VEB heap are slightly lower with the addition of lower cache miss counts. Although cache miss behaviour is important, ultimately, the most important metrics is execution time - despite not showing significantly greater improvements in execution time, the VEB heap shows better memory efficiency which can be harnessed into other applications, like graph algorithms.

3.3 Graphs

In computer science, graph representation has been evolving from the use of simple matrix representation to more sophisticated representations, such as compressed row, neighbourhood lists or edge lists. Several graph representations may be more suited than others (depending on their final application) due to details in their implementation, like matrix representations for dense graphs have fast access to neighbours, compressed row for space complexity and locality benefits for sparse graphs. Some representations prime by their simplicity over efficiency, like adjacency matrices. Adjacency lists and compressed row representations do not consider absent edges. Aside from the data-structure implementations possible to implement there is a wide range of support structures we can account in graph problems: sorting primitives, auxiliary data structures like forests, colouring mechanisms for cycle checking, among others.

The case study is Prim's Minimal Spanning Tree (MST) algorithm. The *minimal* spanning tree of a graph is logically, a tree that reaches/touches every node through the smallest weighted edges. In order to apply such algorithms efficiently we consider the implementation frameworks,

pointer resolving complexity, data layouts and cross structure optimizations.

3.3.1 Formal graph description

The main issue to have in mind in this scope is the efficient use of memory regarding space and operational complexity, for instance, the complexity of reaching a neighbouring node in the data structure. In this dissertation we treat weighed undirected graphs.

$$G = (V, E)$$

- V is a non-empty set of vertices,
- E is a set of edges where each edge is a pair of vertices,
- an undirected graph assumes that if there is an edge $e_i = (v_a, v_b) \in E$ we assume the existence of a mirrored edge $e_j = (v_b, v_a) \in E$
- aside from being undirected, graphs are also weighted. In the previous item, e_i and e_j have the same weight.

$$\forall e_i = (a, b) \in E, \exists e_j = (b, a) \in E$$
$$\Downarrow$$
$$w_{e_i} = w_{e_j}, w_{e_i} \in \{W_{e_x}\}$$

  Where w_{e_i} is the weight associated to the edge in question (e_i). It represents the link cost between the vertices a and b. In the case of undirected graphs we assume $w_{e_i} = w_{e_j}$.

Since we treat undirected graphs, in our representation we consider edge derivations to represent each uni-directional side of an edge, we call each side a neighbour, i.e., for each edge $e_i = (a, b)$ there is a pair of neighbours $(n_{a\to b}, n_{b\to a})$.

3.3.2 Graph representations

The main problem in specifying and using a graph data-structure is memory usage. The most known and straight-forward ways of represent a graph using practical ideas are:

(i) *Adjacency Matrices.* An adjacency matrix consists of an $N \times N$ matrix, where $N = \#nodes$. More specifically, if the graph to consider is not weighted, the values in the matrix are boolean typed, if they are weighted, then it stores weight-typed values; if the graph is undirected, the matrix will be symmetric. This is a space consuming representation since it considers that the graph reaches a size of $N \times N$, i.e., a complete graph where every node is connected to itself and every other existing node - space complexity of $O(N^2)$.

(ii) *Adjacency Lists.* Lists where the neighbourhood (other adjacent nodes) of each node is denoted by a list of nodes. If the graph is weighted, another *twin* list may be added with the weights of each edge. Another way to do this is (in a more object oriented approach) to add the weight information to the edge entities. Along with the weight information, information like the source node of each edge may be added. Space complexity of $O(N+E)$, where E is the number of edges.

(iii) *Incidence Lists.* An incidence list basically consists of a list where all the graph edges are stored. The mandatory information to be present in each edge are both the source and target vertices - therefore data-replication occurs, however at a low cost of explicitly represent the connections of the graph. To these edge entities, more data can be added, like its weight. Space complexity of $O(N + E)$.

(iv) *Incidence Matrices.* An incidence matrix, establishes a relation between the nodes of a graph (rows of the matrix) and its edges (columns of the matrix). Altering with the graph structure means matrix resizing operations. Space complexity of $O(N \cdot E)$.

3.3.3 Linked data structures

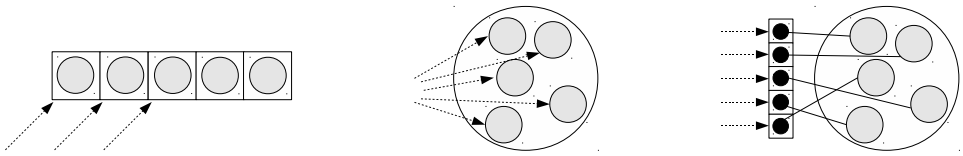
This kind of data structure is know for its referential nature, i.e., it is composed mainly by references (or links) to other data elements in different memory regions, usually offering programming flexibility, but due to its pointer-chasing character, not a good choice for performance issues, specifically memory locality issues. Roth and Sohi [48] studied pre-fetching primitives in pointer-chasing structures, namely *jump-pointer* techniques - consisting in placing explicit memory calls to future called memory addresses. Nevertheless, our main goals are not prefetch-based optimizations but structural optimizations which ultimately benefit locality.

In our scope, linked data structures (LDS) are mixed with graph concepts and the developed structures are pointer-chasing ones, therefore we now explain the constituting variants of the used collections: *sets*, *lists* and *maps*.

Types of data-containers

The basic constituents in our data containers/collections can be combined using *lists*, *sets* and/or *maps*. It is possible to create and derive collections with concepts from other collections, for instance, a *map* can be seen as a set of list-elements. Typically, maps differ from other containers (lists and sets) because they offer another level of *indirectionality*, associating a key to a value or a collection of values, while lists and sets only allow for one level, unless we consider a list do be indexable - an indexable list can be considered a simple map where the index value is the mapping key. The difference is that key-index values in lists are implicit. A *map* is also considered an *associative collection* because an association is made between a collection of unique keys and a collection of values. Each unique key may be associated with a single value or multiple values, in

the latter case the *map* is called a *multi-map*. In irregular graph data structures, *sparse* graphs are usually represented with a multi-map to associate a unique set of vertices to a collection of edges connecting the vertices. In Figure 3.14 we show how these types of collections may be understood by most programmers and programming frameworks.



(a) List (b) Set (c) Map

Figure 3.14: Collections example

3.3.4 Minimal Spanning Tree: Prim

The Minimal Spanning Tree (MST) problem is a graph algorithm, for weighted, and in our case, undirected graphs, that consists in finding the spanning tree of the original graph touching every node through the smallest weighted edges. The algorithm we focus on for our case study is Prim's MST algorithm [46] which uses a priority-queue to find the smallest weighted available edges to add to the resulting MST graph.

Prim's algorithm works by successively adding graph neighbour connections to the PQ, removing the smallest weight connections and marking the touched vertices as visited for cyclic control. The implemented data-structures can take many forms depending on the algorithm and the usage we intend to give to it, for instance: for a different MST algorithm, Kruskal's algorithm [27], one must implement the *Union-Find* data-structure (simply put, a set of trees). After a series of profiling we concluded that the main bottleneck was in using the priority-queue (PQ) to sort-remove the edges in the original graph. We could have opted for a different path to optimization, like implementing a *Fibonacci-Heap* as a PQ (see Section 2.3.1), however, this kind of implementation, using a *decreaseKey* operation (which significantly lowers the total execution time) is not our goal since it changes the algorithm, i.e., we aim for an algorithmic implementation that is simple and as independent as possible from the specific optimizations. In Listing 3.3 the pseudo-code for Prim's MST algorithm is presented - all our implementations follow this pseudo-code.

Listing 3.3: Pseudo-code for Prim's MST algorithm

```
1 Prim MST(G) {
2         CurrentVertex = Random Start Vertex;
3         Neighbours    = Neighbours Of(Current Vertex);
4         Add To PQ(Neighbours);
5         while(Unvisited Vertices Exist) {
6           // lazy removal of previously visited neighbours
7           do {
8                   Min Neighbour  = Get Minimum(PQ);
9                   Current Vertex = Vertex Of(MinNeighbour);
10          } while (is Visited(Current Vertex));
11          visit(Current Vertex);
12          Add To MST(Min Neighbour);
13          Neighbours = Neighbours Of(Current Vertex);
14          // add unvisited neighbours to PQ
15          for each neighbour in Neighbours {
16            if(is Not Visited(Vertex Of(neighbour)))
17              Add To PQ(neighbour);
18          }
19        }
20 }
```

In this specific algorithm there are a few decisions that made the algorithm perform a little better than the initial implementation: visiting interface and cyclic control. *Visiting* vertices is a concept easy enough to implement and allows efficient cyclic control - often implemented with *flag* attributes or auxiliary data structures, like a *visited-vertices* array. *Cyclic control* is important in this algorithm because it prevents processing vertices already visited, therefore infinite loops are prevented. There are several variants of this algorithm that allow cyclic control in various forms and using a wide variety of structures, some more efficient than others. For instance,

(i) according to the chosen graph API, each edge removed from the priority-queue should be checked for cycles, i.e., if the destination-vertex from that edge is already visited we ignore it. The fact that we chose to represent only one vertex per neighbour and not to store both vertices related to an edge in a single structure, greatly simplifies cyclic control because we do not have to check both vertices in a pair of vertices. Therefore, to benefit algorithmic complexity, an *edge* in our graph structure corresponds to *a pair of neighbours*, stored on both ends of an edge, i.e., each neighbour is symmetric to its sibling and is stored at the corresponding source vertex[7]. In the priority-queue we can go for a process of *lazy removal* of the elements we no longer want consider in the heap - it is called lazy because we do not remove the unavailable neighbours right away, instead it is delayed to a check in a *do-while* removal. This allows us to gradually remove elements that are no longer necessary (already visited) in the priority-queue;

(ii) the Fibonacci-Heap (Section 2.3.1) is an efficient priority-queue that implements a *decrease-*

[7]Note that this decision would also be valid for directed graphs, for each edge we would only store one neighbour, a neighbour is an unidirectional edge.

key operation, which in contexts like Prim's MST can be very useful. When we need to decrease an element's priority in the heap, much like what happens when a neighbour gets visited in the graph, the *decrease-key* operation allows for a fast update of that element in amortized constant time $O(1)$. For fast update we must have external access[8] to the element otherwise we would have to search the heap looking for it. This detail changes the algorithm implementation to a slightly more complex one. However, the number of elements added and removed from the heap is substantially decreased, so memory usage is lower. We do not go forward with this implementation because we are aiming at a structure-independent algorithm and the Fibonacci-Heap carries too many implications.

3.3.5 Data layout optimizations

Since graphs are irregular structures, meaning, their representation and traversal path in memory are unknown, it is hard to infer and optimize its data layout because it has an amorphous structure, with no seemingly form. We can however optimize their pointer resolving operations, lowering the pointer complexity often found in graphs, ultimately leading to opportunities in spatial and temporal locality. More specifically changing the data layout of collections of attributes can greatly improve graph implementations.

The optimizations found in this particular kind of data-structure, a graph, are concerned with *locality*. In Figure 3.15 we see a usual representation for graph structures: a map-like structure. The most important optimization opportunity here is related to the neighbours list stored in each vertex object - when it comes to algorithmic challenges, lets consider Prim's MST, neighbours store pointers to other vertices assigning the *pointer-based* nature to the structure. This neighbours list is implemented as an array, however, we are operating with API-compatible types which makes array elements not contiguous in memory - Neighbour objects follow a generic interface - the neighbours list is an array of pointers (AoP).

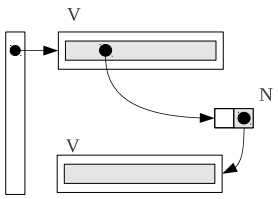


Figure 3.15: Graph-Neighbour-Vertex AoP

After studying the cache-miss behaviour and pointer-resolving complexity from the structure in Figure 3.15, the locality considerations taken are:

(i) Bottlenecking layout with pointer-chasing nature on the neighbours array in each vertex shows low locality - Neighbour objects are not contiguous.

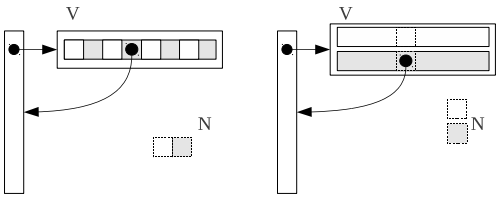
[8]External to the priority-queue, so the algorithm must store a set of pointers to inner heap elements.

(ii) By embodying Neighbour data (*weight*, *neighbour-vertex*, etc.) in the Vertex object, data can be distributed in two possible ways, according to their API usage:

(a) Arrays of Structures (AoS) - several objects (Neighbour) are forced to be adjacent, increasing the overall attributes' spatial locality (Figure 3.16a).

(b) Structure of Arrays (SoA) - several arrays, each holding an attribute for each Neighbour object, increasing the spatial locality for each attribute independently (Figure 3.16b).

(iii) Better chances of increasing *temporal locality* in the initial graph vertex-holding structure by, in each Neighbour, explicitly pointing at the initial array (both sub-figures in Figure 3.16).



(a) Graph-Vertex-Graph AoS (b) Graph-Vertex-Graph SoA

Figure 3.16: Optimized graph representations

The different graph implementations are named as being AoP, AoS or SoA by looking at how the Vertex structure holds the Neighbour array. The *GNV AoP* shows in fact a double pointer-resolving complexity, because it lacks the locality of the neighbour-vertex pointer present, for instance, in *GVG AoS*.

3.3.6 Graph pointer-based complexity analysis

In order to analyse the time complexity of traversing a graph pointer-based[9] structure we consider the sparsity level of the graph so we can choose the most appropriate graph representation and implementation. The graph considered in our implementations is a sparse undirected graph so our preferential graph representation is an adjacency list due to its good space usage by not allocating storage space for absent edges. Regarding implementation details, we implement our structures and algorithm in Java which is an OO language - this has impact as seen on Section 3.1. The objective with this section is to show how we look at pointer complexity in our structures and to provide some basic understanding of how our pointer-based structures are built.

For this we do not only apply the traditional cache-miss concepts - for now we also focus on misses likely to be caused by a *pointer-resolving operation*, for simplicity we refer to this kind of

[9]For coherency, in this section we will also call the term *reference* a *pointer*.

miss as a *pointer-miss*. Our test case is the *is-adjacent* operation: check if two given nodes in a graph are adjacent using the structures presented in Figures 3.15 and 3.16. We present a generic pointer-miss analysis over the mentioned graph structures:

(i) AoP - this structure is composed of an array of pointers to Vertex structures, each Vertex holds an array of pointers to Neighbour structures, each Neighbour holds a pointer to another Vertex. At first glance we identify three pointers that might be causing misses, but we consider an additional (compulsory) pointer-miss which is the access to the first structure holding all vertex objects. Since each Vertex may store many Neighbours the pointer-miss cost is bounded by $O(N)$.

(ii) AoS - in this implementation we implement a hashing mechanism so we can take advantage from the attributes data layout in a primitive-typed array, for the reasons explained in Section 3.1. The neighbours array, instead of storing pointers to structures, now store the attributes themselves eliminating one pointer-resolving operation, one level of indirection is removed. By referring to a Vertex with an *id* field we can directly consider if one exists in another vertex's neighbours list and infer the bounded cost as $O(\frac{N}{B})$., where is the cache block size. In the previous point, the cache block size is not considered because it is an AoP implementation, i.e., adjacency is not guaranteed so loading contiguous element is out of the picture.

(iii) SoA - this analysis is made analogously to the previous point (ii), with the difference that the Neighbour attributes are distributed by several arrays. The pointer-miss costs is also $O(\frac{N}{B})$.

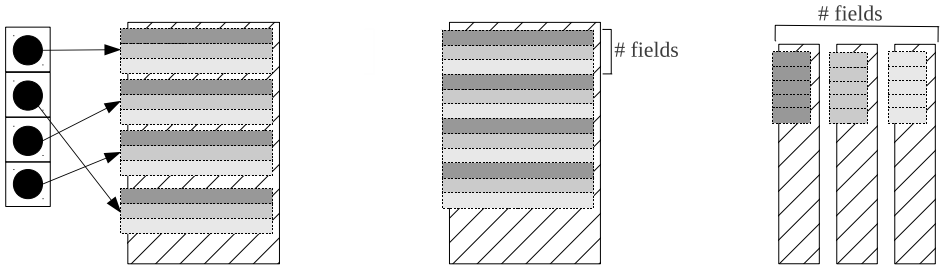
3.4 Composing implementations

The compositions are related to AoS and SoA optimizations applied in graphs and priority-queues. The performance of these layouts is injured by API encapsulating mechanisms and related problems mentioned in sub-Section 3.1.2 - summarizing what is said: APIs are generic mechanisms which offer modularity to software development but may offer performance barriers with cross structure optimizations, i.e., in order to achieve the desired efficiency an API is made compatible with the structures it operates on, in detriment of abstraction and code reuse. In the specific case of graph algorithms, the priority-queue presented in Section 3.2.2 is implemented with AoS and SoA layouts in order to store the attributes used in Neighbour objects (sorted in Prim's MST). One level of indirection between the memory reference to the object and the object attributes was removed. However, in our case this leads to excessive and redundant object creation and discarding, which generates massive overhead.

Data layout for priority-queues

The data layout for priority-queues can also be changed according to AoS and SoA layouts. The native implementation of a priority-queue in Java is represented through an array of pointers (AoP layout).

The AoP layout (Figure 3.17a) is a popular layout due to its abstract features - memory references (or pointers to memory addresses) serve a better purpose of being able to include an object of any data type more easily managed by generic data type mechanisms. Regardless of their ability to easily store abstract objects, this kind of layout is usually adopted by automatic memory management systems like garbage collectors in virtual machines, making them hard to optimize locality-wise. As seen previously this layout works with high dereference costs and element adjacency is not guaranteed.



(a) AoP (b) AoS (c) SoA

Figure 3.17: Priority-queue layout changes.

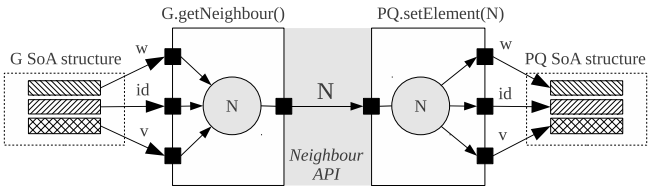
To change the data layout of priority-queue one must consider the fields of the objects intended for storage. In the AoS (Array of Structures) layout, Figure 3.17b, fields are stored continuously in memory, as in SoA (Structure of Arrays), Figure 3.17c, fields are stored into a separate array. Choosing the best alternative depends on how the algorithm accesses data. The SoA layout provides better locality if the algorithm does not require all fields of the original structure in the same time-frame. The AoS layout is the alternative used for problems that require all fields of the structure at once, although this choice is difficult to implement in Java since it is not possible to use explicit pointers to data. It is also more difficult to use if the fields are not of the same type. To improve the spatial locality in Java collections it is necessary to transform an AoP implementation into an AoS or a SoA. In the latter case, the fields of the objects are converted into arrays, which normally evolves removing the encapsulation of data. This provides better performance, but it might enforce significant restructuring of the code. In this work we intend to study the impact on performance of this transformation.

Changing data structures

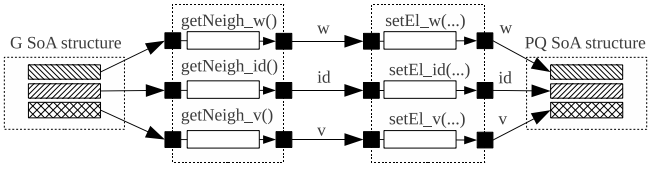
In order to use the optimized heap implementations as priority-queues in conjunction with our Neighbour-API abiding graph data structures, the priority-queue storage structures were specifically changed to store Neighbour attributes like weight or vertex-id. Therefore, the AoS heap implementations (which demand that the attributes are of the same data type; single array) store all attributes of each object contiguously in the array; as for SoA distribute the attributes through several arrays. Thus, for each priority-queue considered in the benchmarks (binary heap and VEB 3 heap) the combinations are: (i) generic binary and (ii) generic VEB 3, (iii) AoS binary and (iv) AoS VEB 3 heap and (v) SoA binary and (vi) SoA VEB 3.

Decapsulating API

Beside altering the storage data structure, decapsulating the API (as depicted in Figure 3.18b), means having having direct access to each attribute via a previously defined method that specifically accesses (get and set) each attribute, instead of creating an object to communicate the values of attributes (API abiding implicit operations). This process enables us to eliminate redundant object instantiation costs, shown in Figure 3.18a. The figure shows the process of querying the graph data structure for an element and storing it it in the priority-queue data structure. The API is also changed for graph data structures, GVG AoS and GVG SoA which initially use the AoS/SoA layouts but still encapsulates data.



(a) Encapsulation



(b) Decapsulation

Figure 3.18: Neighbour API encapsulation and decapsulation schematic

Code bloating

One of the main setbacks in performing transformations in data spaces is that all accesses of a program to an altered data structure must be revised and the code may suffer significant changes if it is not *protected* behind a common interface. As explained earlier, APIs are removed in order

to demean encapsulating overhead. There is a trade-off between modularity and efficiency. In Table 3.5 we measure the non commenting source statements (NCSS) measured with JavaNCSS[10]. For each combination of graph, priority-queue and API we measure the program's NCSS of all pertinent classes pertinent to each combination; then we calculate the average for each API implementation. In Table 3.5, we present the average NCSS of: (i) generic implementations, i.e., AoP for both priority-queues and graphs; (ii) combinations between AoS and SoA for graphs and priority-queues, but still with a generic API; (iii) and combinations between AoS and SoA with decapsulating APIs.

Table 3.5: Non Commenting Source Statements average for each API version and average increase in % for encapsulated and decapsulated APIs for all implementations of graphs and priority-queues.

API	NCSS Average	% NCSS grow
Generic	355	-
Encapsulated	470.25	24.5%
Decapsulated	409.75	13.4%

The NCSS count for Encapsulated is higher than Decapsulated because the attributes of each Neighbour object are passed directly accessible: get and set methods are directly called by external structures. These direct calls to get and set methods, instead of undergoing through several layers, make source code complexity slightly decrease. Nevertheless, NCSS increases in relation to generic implementations.

3.4.1 Benchmarks

The benchmarks for combinations of graphs and priority-queue implementations are shown in Table B.2 (Appendix B), in it the generic, encapsulated and decapsulated APIs are shown. Tests were performed with the following environment:

- The benchmarked algorithm is Prim's MST, with data structure variations to check which ones perform better. The same graph is considered for all runs (note, the same graph shape to guarantee similar access patterns), it contains 3000 nodes with a connectivity rate of 50%, containing a total number of 4,498,870 edges, or 8,997,740 Neighbour objects;
- Generic implementations for graphs and priority-queues were used - using only objects - as a starting point for comparison (the generic API);
- AoS and SoA layout changes are applied in graphs and priority-queues. The underlying API in this case still uses objects (as depicted in Figure 3.18) - the objective is to show the overhead of API encapsulation and its negative impact on instruction count and cache performance;

[10] `http://javancss.codehaus.org/`

  - Each priority-queue is experimented with a different layout (binary and VEB 3 are experimented with AoS and SoA layouts), giving origin to several combinations between graphs and priority-queues;

- Finally, the common Neighbour API is decapsulated for graphs and priority-queues.

Figure 3.19 shows the average values for instructions, cache misses and execution times of all implementations for each API variation. It is easily understood that generic APIs due to the overhead of using abstract layers throughout the program: JVM object management is prone to pointer chasing operations. Encapsulated APIs introduce another level of operational complexity, as seen on instruction counts (the creation and discarding of objects). Since the objective of using the same graph is to lead to similar access patterns, we can conclude that the slightly higher overhead of *generics* in L1 cache is in JVM management and implicit pointer resolving nature of generic collections. In accordance to this premiss, analysing execution times (which are congruent with clock cycles), we see that the encapsulated API average is approximately half of the generic API - this means that despite having more instructions, the encapsulated API is still more efficient than JVM native object management due to locality benefits of the applied AoS/SoA layouts.

Applying API decapsulating optimizations, we get better results - let us remind that the *encapsulated* structures already use primitive data types (to demean auto-boxing overheads) only under an object-based API. The objective with decapsulated API is to show that in order to attain the desired efficiency in optimized structures, abstract development mechanisms may not leverage performance.

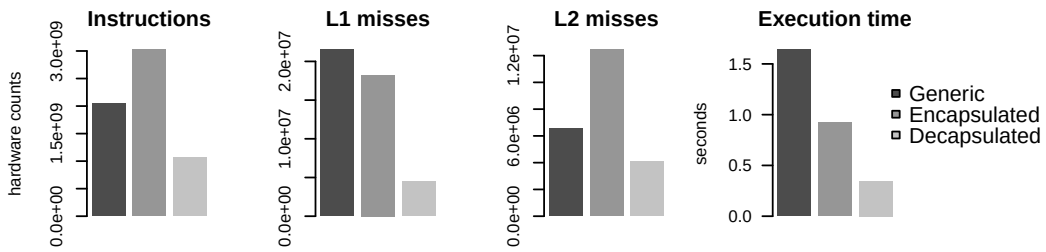


Figure 3.19: Summarized performance hardware counts of instructions, L1 and L2 accesses/misses and execution time (the average values for each API implementation are shown).

Benchmark analysis and discussion

So that the reader can understand the graph represented in Figure 3.20, the three APIs are represented through color groups - generic, encapsulated and decapsulated API: (i) first two columns refer to generic, (ii) the following eight columns refer to encapsulated and (iii) last eight to decapsulated.

It is quickly noticeable that the graph bars in Figure 3.20 for instructions represents the three groups in first graph of Figure 3.19. In the generic API, the one that shows the higher instruction count is the variation using the VEB 3 heap, because its code is more complex as it is stated in Section 3.2.2 - this happens for all VEB 3 heap implementations. The most important evidence is the second block of columns where we notice a high overhead created by encapsulated APIs, surpassing the code instruction complexity even for fully generic implementations. By eliminating such redundancy the instructions necessary to complete execution greatly decrease.

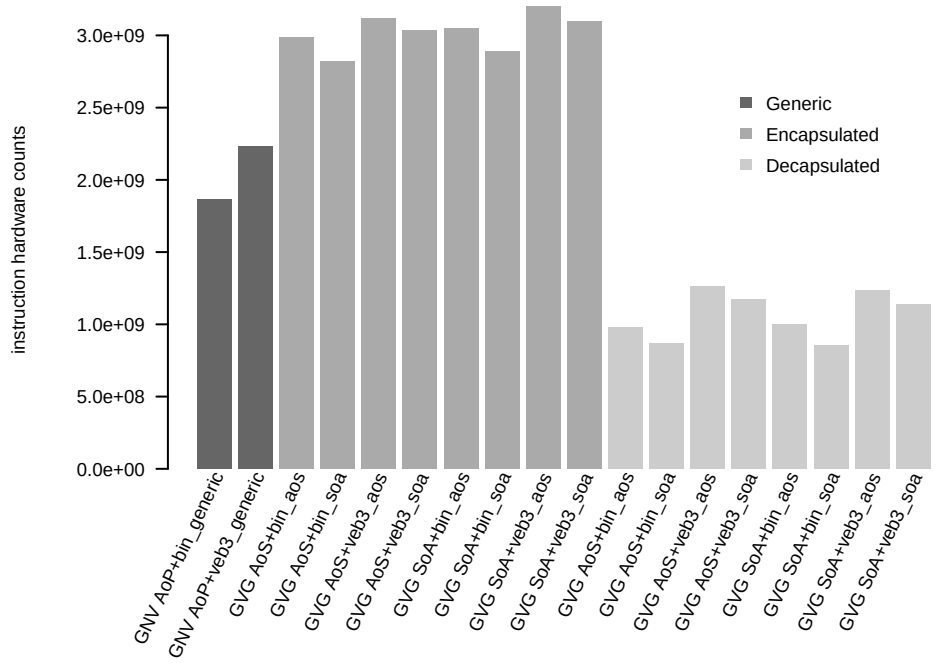
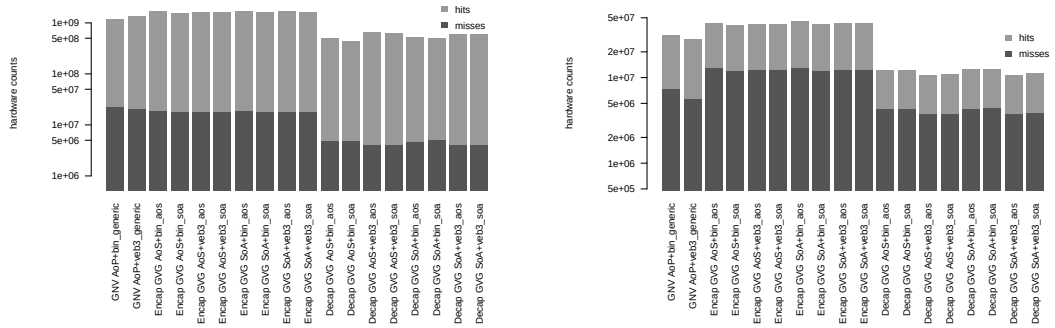


Figure 3.20: Discriminated performance hardware counts for instructions

Cache miss analysis

Figure 3.21 shows the discriminated values for cache accesses and misses. Although miss ratios can be misleading they are presented in Figure 3.22: unoptimized versions of APIs or generic implementations that do not show the desired cache efficiency can present low cache miss ratios, despite having low performance, due to garbage-ish operations that might be occurring in JVM object management.

As seen in the graphs in Figure 3.21, the main differences in cache efficiency comes from changing the underlying API the structures use. When comparing cache miss average values of generic to encapsulated and decapsulated values (for L1 and L2: <L1, L2>) we get the



(a) L1 cache misses and hits

(b) L2 cache misses and hits

Figure 3.21: L1 and L2 cache miss and hit counts (note: logarithmic scale o y-axis).

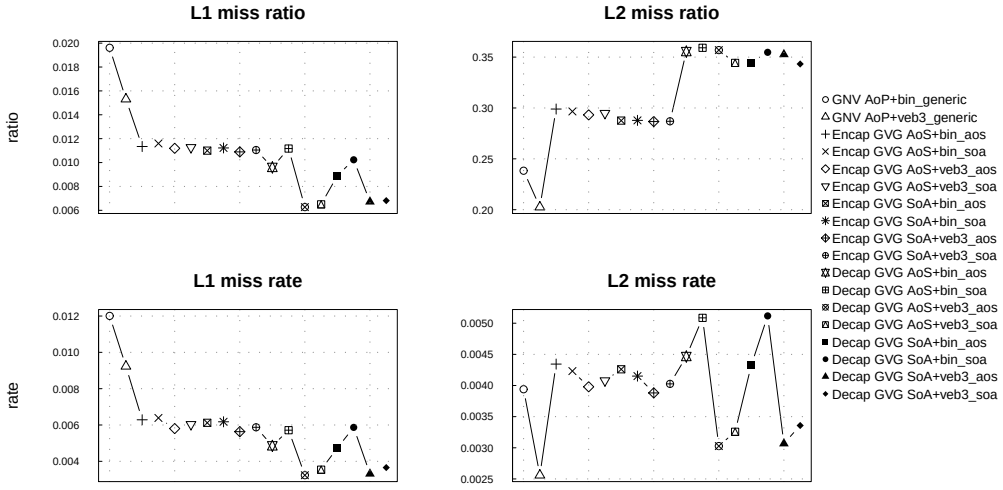


Figure 3.22: L1 and L2 miss ratios and rates.

approximate percentages of (i) <89%, 190%> comparing generic to encapsulated, and (ii) <20%, 62%> comparing generic to decapsulated. The total cache miss counts greatly decrease for decapsulated. The values for L1 in encapsulated decrease due to AoS/SoA layout changes, however, there is an increase in L2 encapsulated due to inefficient memory management.

Although miss ratios can be misleading they are useful to show that VEB 3 heap layout optimizations are mostly noticed in decapsulated API versions - L1 and L2 miss ratios for decapsulated are lower than when using the traditional binary priority-queue layout. This VEB 3 improvement is also visible when relating cache misses and instructions, in Figure 3.22 showing the L1 and L2 miss per instruction, also known as cache miss *rate* (not *ratio*).

TLB miss analysis

The translation look-aside buffer (TLB) is a content addressable memory used to store the mappings between virtual and physical addresses. In order to translate virtual to physical addresses, the system has to perform a page table walk to search for the right page-id, which ultimately results in four memory accesses [38]. TLB memory caches the recently used page table entries (PTEs). As for TLB miss handling, they can be handled by hardware or software depending on the architecture:

- Hardware TLB miss handling: the CPU searches for the correct PTE with a page table walk, if it finds it the PTE is marked as *present* and stored in the TLB. Otherwise, if the PTE is not found the process is handled by the operating system (OS).
- Software TLB miss handling: a *page fault* is returned by the CPU and the OS intercepts it. A page table walk is made by the software and if the PTE is found it is equally marked as *present* and stored in TLB; if not found the page fault handler takes control.

The costs of a TLB miss, whether in hardware or software, are heavy: they require a page table walk. Hardware solutions for this problem are usually faster and less flexible than in software TLB miss handling. Our results for TLB data and instruction misses were captured via performance hardware counters, so the measurements are related to hardware-handled TLB misses.

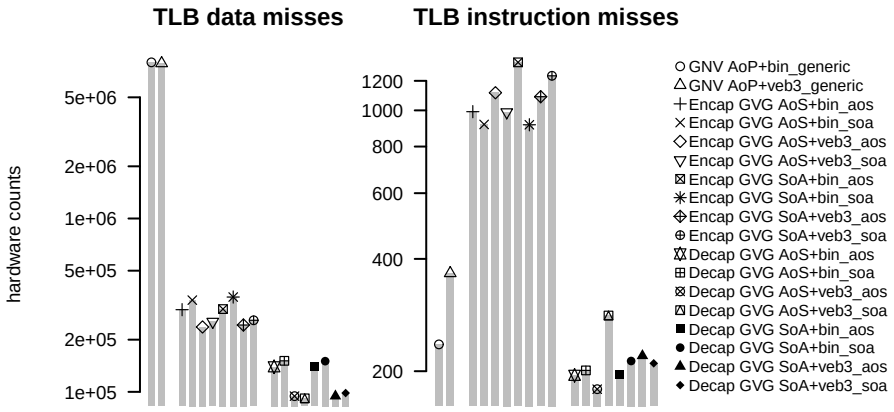


Figure 3.23: Performance hardware counts of TLB data and instructions misses for all implementations (note: logarithmic scale on y-axis).

In Figure 3.23, we present the TLB data misses to show the cache improvements from changing the data layout, and TLB instructions misses to denote the differences in using different kinds of APIs. In the first graphic, the counts for each API version successively decrease, being the peek at the generic implementations showing a lot of TLB data misses probably due to inefficient

JVM object and address management (excessive pointer resolving operations to non contiguous memory addresses); and it is interesting to see that all VEB 3 versions show less TLB data misses than in binary heap, specially in decapsulated API - this means the VEB 3 layout arrangements show better memory address usage. When comparing AoS and SoA layout arrangements in graphs and priority-queues the differences are not too noticeable.

For TLB instruction misses we can confirm the overhead caused by encapsulating APIs - constructing and destroying objects just to communicate values. However we can see that the generic mechanisms in Java perform relatively good, Java's generic code complexity is low, almost to the degree of decapsulated versions, with the exception of generic VEB 3 which has higher code complexity due to more complex index computations - this higher TLB instruction misses applies in all VEB 3 implementations. Finally, when comparing AoS and SoA arrangements in decapsulated API we can see that graphs with AoS layout show a little less misses with one exception: GVG AoS + VEB 3 SoA

Average memory access time (AMAT)

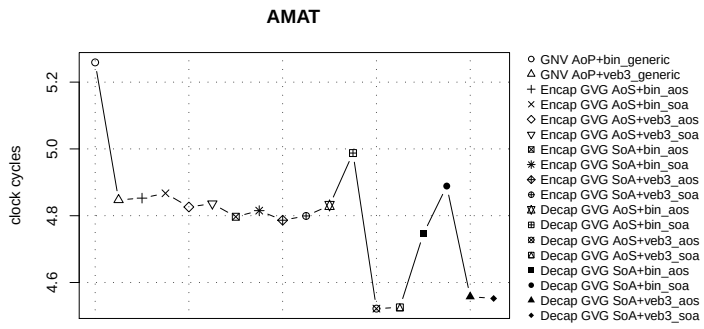


Figure 3.24: Average memory access time metrics for all implementations.

The AMAT performance metrics shows the average number of time units each memory access spends in the total execution time (time unit for this case is *clock cycles*) - as the previous graphics, the graphic in Figure 3.24 is divided by vertical lines in three groups: generic, encapsulated and decapsulated API. We are able to see that *decapsulated* versions with VEB heap spend less cycles accessing elements than binary heap, due to its cache-friendly layout - as for binary heaps uses a näive element layout. Regarding AoS and SoA layouts we do not see much differences in using one or another, unless we look at *decapsulated binary* heap with AoS (GVG AoS+bin_AoS, GVG SoA+bin_AoS) which shows less access time than the concurrent heap SoA implementations (GVG AoS+bin_SoA, GVG SoA+bin_SoA). This is due to the fact of all fields being in the same memory region, and because we have small and little number of fields, their usage is not too critical in our case: it is best to access all fields of each object in one memory access.

Why do encapsulated and decapsulated AMATs have the same order of grandeur?

Although the absolute values presented in previous graphics show big differences between API version (generic, encapsulated and decapsulated), the graphic in Figure 3.24 shows a line that does not varies much across all implementations. This is because within the AMAT metrics, we consider a relation between cache misses and instruction counts, which are absolute measures. In AMAT we consider the cache miss rate

$$Miss\ Rate = Misses\ per\ instruction = \frac{Misses}{Instructions},$$

which is a relative measure, relating total number of cache misses to the instruction count.

Summary

To summarize the results, when comparing API versions we see that generic and encapsulated APIs introduce a high amount of overhead causing excessive instructions to be run. Code complexity greatly increases for encapsulated versions. We removed one level of indirection (generic collection API) between collection and data; and simplified the API specification to eliminate API abiding instruction operations (encapsulated collection API), leading us to decapsulated API. It was in decapsulation experiments that we began to see the locality improvements from cache-friendly heap sort and graph data layout changes (AoS/SoA). Sorting proved to be a good aspect to optimize since the overall results are better for VEB heap versions.

3.5 Benchmark methodology

This section explains the methodologies used for measuring the graph and PQ tests ran considering a number of details such as profiling tools used, structure considerations like level of graph sparsity and randomized but constant values list for sorting.

3.5.1 Benchmark environment

The hardware and software specifications of all tests ran for this dissertation are presented in Table 3.6[11].

3.5.2 Hardware performance counters

One way to perform thorough analysis on modern architectures is recurring to hardware performance counters, a special set of registers that store information related to hardware events (e.g.., number of instructions executed, number of accesses to L1 or L2 caches, etc.). These registers are embedded in most modern microprocessors and provide a powerful analysis tool for developers enabling them to find and tune bottlenecks.

[11] CPU information taken also from `http://www.cpu-world.com/`

Table 3.6: Benchmark environment settings.

Processor	Name: Intel(R) Core(TM)2 CPU T7200 @ 2.00GHz Frequency: 2.0 GHz Cores: 2
Cache	L1 Data: 32 KB, 8-way set associative, cache line size 64 bytes L1 Instruction: 32 KB, 8-way set associative, cache line size 64 bytes L2: 4 MB, 16-way set associative, cache line size 64 bytes
TLB	Data TLB: 4 KB Pages, 4-way set associative, 256 entries Data TLB: 4 MB Pages, 4-way set associative, 32 entries Instruction TLB: 2 MB pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries Instruction TLB: 4 KB Pages, 4-way set associative, 128 entries L1 Data TLB: 4 KB pages, 4-way set associative, 16 entries L1 Data TLB: 4 MB pages, 4-way set associative, 16 entries
Main memory	2 GB
Java	Version: 1.6.0_26 JVM: Java HotSpot^{TM}Server VM (build 20.1-b02, mixed mode)
PAPI version	4.1.0.0
OS	Kernel Linux 2.6.35-30-generic (Ubuntu 10.10)

Although providing in-depth perspective to behavioural issues in an architecture, in profiling tools that use hardware performance counters, correlating low-level overhead and source code, forces the analyser (programmer) to have good knowledge of the measured application. Also, the programmer may be limited by the number of hardware performance counting registers and may have to perform the same program run multiple times in order to gather all the desired information.

3.5.3 Profiling with PAPI

PAPI[12] stands for Performance Application Programming Interface, it is a programming interface to call hardware counter related routines with the use of code instrumentation. It provides an architectural abstraction between the available register set and programming level by considering a set of pre-set events that are easily perceptible through the use of the caller API. The events considered for measurement in the architecture described in Section 3.5.1 are shown in Table 3.7 (the list of the total PAPI pre-set event names are listed in Appendix B).

Since there are not enough performance hardware counters in the hardware specification (Section 3.5.1) several runs are executed in order to measure the events mentioned in Table 3.7:

- first run measures PAPI_TOT_INS and PAPI_TOT_CYC events, the total number of instructions and cycles executed. Cycles per instruction (CPI) metrics can be extracted from this.
- second run measures L1 data accesses and misses (PAPI_L1_DCA and PAPI_L1_DCM) from where we can extract L1 miss rates.
- third run measures only L2 data accesses (PAPI_L2_DCA) due to the lack of sufficient hardware counters.

[12] `http://icl.cs.utk.edu/papi/`

Table 3.7: Considered hardware performance counters pre-set measurement events in PAPI

Event name	Description
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L2_DCM	Level 2 data cache misses
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_TOT_INS	Instructions completed
PAPI_TOT_CYC	Total cycles
PAPI_L1_DCH	Level 1 data cache hits
PAPI_L1_DCA	Level 1 data cache accesses
PAPI_L2_DCA	Level 2 data cache accesses
PAPI_L1_TCA	Level 1 total cache accesses
PAPI_L2_TCA	Level 2 total cache accesses

- fourth run measures L2 data misses (PAPI_L2_DCM).
- consequent runs may vary depending on the context but the methodology is analogous - measuring mainly TLB (translation lookaside buffer) behaviour in independent runs.

3.5.4 Benchmark decisions

For the data structures addressed in this dissertation (priority-queues and graphs) we mention decisions taken into account for fairness of comparison.

In priority-queues the benchmarks are performed with the same values list so that the memory access pattern is the same for analogous implementations and in order to confirm improvements when applying different data layouts. At instantiation time the priority-queue allocates the full size of used number elements so that possibly distracting operations like re-dimensioning the storage data structure are avoided.

For graph data structures the method is the same - the same size, connectivity rate and edge weights (i.e., in general the same graph shape) is used for all competing implementations for fair comparison in data layout analysis.

3.5.5 Measuring relevant parts of the program

The presented implementations are run in Java, and because (i) measurements use hardware performance counters, (ii) the PAPI profiling tool does not have any Java based integration module and (iii) since we do not want to consider any aspects unrelated to the memory access patterns of our algorithms we used an auxiliary tool to help us isolate the computational relevant parts of the execution for measurement - this tool works by using AspectJ [26] and its use of point-cuts for isolation, and PAPI calls integrated with Java through the use of JNI (Java Native Interfaces). We perform two kinds of measurements regarding priority-queues and graph structures, which are each stored in file to avoid having different layouts caused by random

generation.

Priority-queue regions relevant for measuring

In priority-queues we load from file a previously generated values list to avoid random generation costs - I/O phases are not considered in measurements. For isolated benchmarks (i.e., not considering graph applications) to priority-queues we consider the structure's initializing time. The main goal in measuring priority-queues isolated from graph contexts is to analyse the memory straining properties of the presented heaps (mainly swapping elements in memory), by inserting a full bulk of elements and then removing them.

Priority-queues use integer values (Integer objects or int primitive values) and in order to measure cache-miss improvement we promote a high count of cache-miss by completely filling all cache levels mentioned in Section 3.5.1. Therefore, and considering that *integers* (or *int*) have a size of 4 bytes, $10{\times}10^6$ int/Integer elements are inserted and removed from the priority-queue, occupying a total of (approx.) 38.14 MB in memory - filling all cache levels.

Graph regions relevant for measuring

Graph I/O phases are also not considered as relevant for measuring; Prim's MST algorithm is the initial point of measurement, with the graph already allocated. Unlike graph data structures the priority-queue initialization is considered as stated in Prim's Minimal Spanning Tree (MST) algorithm (graph initialization is considered an I/O phase). Also, the resulting MST is initialized and considered for run-time.

3.5.6 Performance measurement metrics model

In this sub-section we present the metrics used in this dissertation: the Average Memory Access Time (AMAT), proposed in [41], to show the average time spent in memory accesses for each implementation. The formulae of this metric may depend on the number of cache levels in the memory hierarchy, and since the current architecture where all tests are run has up to L2 cache, AMAT values consider hit times and miss penalties for L1, L2 and main memory. The AMAT expression for a 2-level cache environment is has follows:

$$\text{AMAT} = \textbf{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\textbf{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \textbf{Miss Penalty}_{L2})$$

In order to find the suitable values of hit-time and miss-penalty for each cache level an auxiliary tool is used to retrieve useful information about memory latency - with this we can infer the intended information. We used the *Hound* tool integrated with the PerfExpert[13] tool-kit. PerfExpert is used to automatically analyse performance opportunities in applications. An example output for the *Hound* tool in the mentioned benchmark environment is followed. All values presented in Listing 3.4 use the clock cycles (CC) unit to measure latency.

[13] `http://www.tacc.utexas.edu/perfexpert`

Listing 3.4: Ecample output for *hound* tool in PerfExpert.

```
1       CPI_threshold = 0.5
2       L1_dlat = 3.65
3       L1_ilat = 3.65
4       L2_lat = 13.40
5       mem_lat = 218.65
6       CPU_freq = 996000000
7       FP_lat = 3.00
8       FP_slow_lat = 52.87
9       BR_lat = 1.00
10      BR_miss_lat = 12.00
11      TLB_lat = 15.63
```

The costs of cache misses and hits between two consecutive memory levels are inferred through the values of `L1_dlat`, `L2_lat` and `mem_lat`. In Figure 3.25 is represented how we infer the miss penalties and how they are related to hit latencies. When CPU accesses the L1 cache we consider the hit latency returned by *hound* output (`L1_dlat`); when data is not in L1 cache an L1 cache miss occurs and the L2 cache is accessed, this is the L1 cache miss penalty and simultaneously the L2 hit latency (`L2_lat`); finally, when data is not in L2 cache, the process is followed to main memory and consequent deeper levels - L2 miss penalty equals the main memory hit latency (`mem_lat`). These assumptions are confirmed in [41].

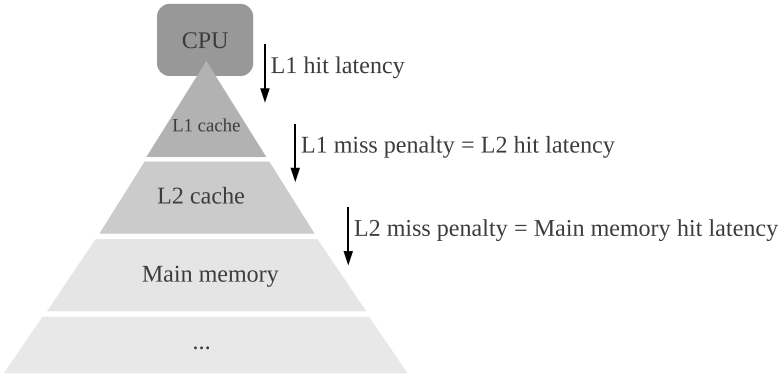


Figure 3.25: Schematic for costs between levels in memory hierarchy.

The *hound* tool was ran multiple times and the average was made for all values - only the pertinent values for the presented AMAT expression are considered (L1 hit latency, L1 miss penalty, L2 hit latency and L2 miss penalty) using clock cycles (CC) as the time unit, see Table 3.8.

Table 3.8: Average values for memory levels latency and miss penalties.

Consideration	Hound designation	AVG CC values
L1 hit latency	`L1_dlat`	3.9
L1 miss penalty, L2 hit latency	`L2_lat`	14.8
L2 miss penalty, main memory latency	`mem_lat`	221.3

Chapter 4

Conclusions and Future Work

In this final chapter we end this dissertation with a summary of the topics mentioned, some concluding remarks and possible future research directions.

4.1 Summary

Modern memory architectures are increasingly strained as data sets grow larger. Many of these applications are irregular, meaning the memory usage pattern is unknown and most of the times unpredictable, taking optimization decisions is hard. For data intensive irregular applications the data layout in memory is unknown and the leeway for memory optimizations is low. In this dissertation we mention the topics of locality of reference optimizations in typically irregular applications. Locality optimizations took place when altering data space layouts in memory for heap-sort and graph related algorithms. There are many studies made around memory efficient sorting, namely through several authors that have contributed with their own implementations of recursive data layouts in memory of heap elements in order to increase. A recursive layout is presented by van Emde Boas [15] where all descending elements in a tree are stored in the same memory regions than each of their parents thus decreasing the number of load operations made from non-contiguous memory regions. We presented a simplified version of van Emde Boas heap based on a binary heap, where its elements are arranged in blocks in order to increase cache hit ratio within a heap block.

One of the main problems addressed in this dissertation has to do with the core nature of object-oriented programming - its abstract mechanisms although allowing for easy, secure and modular software development may not combine well with high performance computing (HPC) aspects. A basic aspect of HPC in object-oriented languages, memory management, is usually left out of programming scope for virtual machines to control like the Java Virtual Machine (JVM), not designed for scientific nor HPC purposes. The main problems are: (i) lack of element adjacency due to type erasure and auto-boxing in abstract object management in Java; (ii) the

creation of encapsulating APIs which are good for modular OO development and *hiding* possibly critical implementation details but representing natural bottlenecks when applying data layout changes originating in redundant object instantiation.

Proposed solutions

Locality optimizations have been achieved in the case of heap sorting methods, by altering with the data layout of the collection, thus increasing cache-friendly patterns - a simplified version of VEB layout is introduced, where gains (compared to a binary heap) are approximately 28% less L1 and L2 cache misses (for a test case of 10×10^6 `int` elements).

For the graph algorithm test case, two data layout changes are proposed, going against Java collections' native layout of Array of Pointers (AoP): array of structures (AoS) and structure of arrays (SoA). The level of indirection between structure and data is removed in order to directly operate with the structure rather than with pointers (or memory references). The objective is to improve field/attribute memory usage by reordering memory accesses to data structures.

Another crucial goal in this subject is to dramatically reduce the pointer resolving complexity, specially aggravated in irregular data structures. One of the main problems found in optimizing data structures is the encapsulation occurring due to Java data type handling mechanisms - encapsulating APIs - the overhead of creating/discarding objects (consuming precious resources) with the sole purpose of communicating attribute values inside an object. In order to take advantage of memory locality optimizations, abstraction is broken, giving origin to the *decapsulated* API which is less generic and built to specifically interoperate with other structures by passing the intended values of attributes. With this, we greatly reduce the overall pointer-chasing complexity implicitly added by generic mechanisms: instruction complexity greatly decreases for *decapsulated* versions; the locality optimizations and performance investments in cache-friendly sorting (VEB) finally gain ground when used in conjunction with other concepts - L1/L2 miss counts, ratios and rates for VEB versions show better behaviour and less bottlenecking stalls; when combining AoS and SoA implementations between graphs and priority-queue layouts we do not notice many changes because all fields/attributes are accessed with similar patterns, i.e., the differences between using AoS and SoA in our test cases are little, but more specific cases of graph algorithms might benefit from such layout combinations.

4.2 Future work

We intent to develop further these optimizations by combining AoS and SoA layouts (hybrid layouts), also not breaking abstract concepts. Because Jikes Research Virtual Machine is an open-source project, implementing these kinds of optimizations to automatically adjust the memory layout of data structures without breaking abstraction concepts would be interesting - more specifically there is the idea of optimizing generic collection data layouts and object management through the use of source code *pragma* primitives. Another advantage of interfering with low-

level JVM specifications would be: harnessing the abstract character caused by type-erasure in all objects, to improve AoS layouts - one of the problems with AoS layouts is that different typed values may not be present in the same array. With JVM byte encoding occurring in all objects, one can optimize a collection's data distribution in memory, by packing all desired (possibly different typed) attributes/fields/elements to store them in contiguous memory regions, thus creating an implicit AoS layout.

Bibliography

[1] USA Advanced Micro Devices, Inc., Sunnyvale, CA. AMD Opteron Processor Product Data Sheet. (March), 2007.

[2] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPOPP '95*, volume 30, pages 166–178, New York, New York, USA, August 1995. ACM Press.

[3] Lars Arge, Michael a. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing - STOC '02*, page 268, 2002.

[4] Abdel-hameed Badawy. The Efficacy of Software Prefetching and Locality Optimizations on Future Memory Systems. *Computer Engineering*, 1, 2004.

[5] Michael a. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-Oblivious B-Trees. *SIAM Journal on Computing*, 35(2):341, 2005.

[6] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*, volume 3 of *The Addison-Wesley object technology series*. Addison-Wesley, 1998.

[7] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. *Proceedings of the thirty-fifth ACM symposium on Theory of computing - STOC '03*, page 307, 2003.

[8] Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. *Engineering a cache-oblivious sorting algorithm*. PhD thesis, June 2008.

[9] A. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee. Enhancing spatial locality via data layout optimizations. In *In Proceedings of Euro-Par'98, number 1470 in LNCS*, pages 422–434. Springer Verlag, 1998.

[10] R.A. Chowdhury. *Algorithms and data structures for cache-efficient computation: Theory and experimental evaluation*. PhD thesis, 2007.

[11] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms, Second Edition*, volume 7. The MIT Press, 2001.

[12] B.T. Davis and M. Jordan. Performance evaluation of exclusive cache hierarchies. *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004*, pages 89–96.

[13] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. *ACM SIGPLAN Notices*, 34(5):229–241, May 1999.

[14] Amr Elmasry. Violation heaps: A better substitute for fibonacci heaps. *CoRR*, abs/0812.2851, 2008.

[15] P. Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, December 1976.

[16] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: High-level Low-level Programming. *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments - VEE '09*, (c):81, 2009.

[17] Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM*, 46(4):473–501, July 1999.

[18] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, November 1986.

[19] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[20] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 285–297.

[21] Hwansoo Han and Chau-wen Tseng. Locality Optimizations For Adaptive Irregular Scientific Codes. Technical report, 2000.

[22] M.D. Hill and A.J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[23] Xianglong Huang, SM Blackburn, KS McKinley, J.E.B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *ACM SIGPLAN Notices*, volume 39, pages 69–80. ACM, 2004.

[24] Intel and Jeff Casazza. First the Tick , Now the Tock : Next Generation Intel ® Microarchitecture (Nehalem). *Power*, pages 1–9, 2009.

[25] Mahmut Kandemir, Alok Choudhary, J. Ramanujam, and Prith Banerjee. A Graph Based Framework to Detect Optimal Memory Layouts for Improving Data Locality. In *Proc. IPPS 99*, 1999.

[26] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An Overview of AspectJ. *Main*, 2072(4):327–353, 2001.

[27] J B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[28] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? *ACM SIGPLAN Notices*, 44(4):3, February 2009.

[29] Milind Kulkarni and Keshav Pingali. Scheduling Issues in Optimistic Parallelization. In *2007 IEEE International Parallel and Distributed Processing Symposium*, page 301. IEEE, 2007.

[30] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*, page 211, 2007.

[31] Monica D. S Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGPLAN Notices*, 26(4):63–74, April 1991.

[32] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. *ACM SIGPLAN Notices*, 34(10):399–414, October 1999.

[33] Shun-tak Leung and John Zahorjan. Optimizing Data Locality by Array Restructuring. Technical Report September, 1995.

[34] Vincent Loechner, Benoit Meister, and Philippe Clauss. Precise Data Locality Optimization of Nested Loops. *J. SUPERCOMPUT*, 21:37–76, 2002.

[35] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Generative and Component Based Software Engineering*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24. Springer, 2001.

[36] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven optimizations for amorphous data-parallel programs. *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '10*, page 3, 2010.

[37] Bertrand Meyer. *Object-Oriented Software Construction*. Computer Science. Prentice Hall PTR, 1988.

[38] David Mosberger and Stephane Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall, 2002.

[39] Jesper Holm Olsen and Søren Christian Skov. *Cache-Oblivious Algorithms in Practice*. PhD thesis, 2002.

[40] N. Park, D. Kang, K. Bondalapati, and V.K. Prasanna. Dynamic data layouts for cache-conscious factorization of DFT. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 693–701. IEEE Comput. Soc.

[41] D A Patterson and J L Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 4th edition, 2008.

[42] S. Pettie. Towards a Final Analysis of Pairing Heaps. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 174–183. IEEE, October 2005.

[43] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, M. Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular algorithms. 2009.

[44] R. Ponnusamy, J. Saltz, and a. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. *Proceedings of the 1993 ACM/IEEE conference on Supercomputing - Supercomputing '93*, pages 361–370, 1993.

[45] V.K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, July 2003.

[46] R C Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[47] Frederik Rønn. *Cache Oblivious Searching and Sorting*. PhD thesis, 2003.

[48] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. *ACM SIGARCH Computer Architecture News*, 27(2):111–121, May 1999.

[49] R Schaffer. The Analysis of Heapsort. *Journal of Algorithms*, 15(1):76–100, July 1993.

[50] J Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry*, 22(1-3):21–74, May 2002.

[51] S S Skiena. *The Algorithm Design Manual*, volume 40. Springer, 1998.

[52] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[53] John T. Stasko and Jeffrey Scott Vitter. Pairing heaps: experiments and analysis. *Communications of the ACM*, 30(3):234–249, March 1987.

[54] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. *ACM SIGARCH Computer Architecture News*, 28(2):1–12, May 2000.

[55] Wikipedia. *Delaunay triangulation*, image (Accessed: 25-Oct-2011). `http://en.wikipedia.org/wiki/File:Delaunay_circumcircles.png`.

[56] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation - PLDI '91*, volume 26, pages 30–44, New York, New York, USA, May 1991. ACM Press.

[57] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing - Supercomputing '89*, pages 655–664, New York, New York, USA, August 1989. ACM Press.

Appendix A

Sift optimizations in Van Emde Boas-based heap

Listing A.1: Traditional (non optimized) sift-down algorithm for heaps

```
siftDown(i) {
  left = leftChild(i);
  while (left < size) {
    right = rightChild(i);
    min = 0;
    if ( array[left] <= array[right]) {
      min = left;
    } else {
      min = right;
    }
    if (array[i] > array[min]) {
        // swap elements ...
        i = m;
        left = leftChild(i);
    } else {
        break; // heap property is verified
    }
  }
}
```

Listing A.2: Traditional (non optimized) sift-up algorithm for heaps

```
siftUp(i) {
  while (i > 0) {
    parentIndex = parent(i);
    if ( array[parentIndex] <= array[current] )
      break;
    // swap parentIndex and i
    i = parentIndex;
  }
}
```

Listing A.3: Optimized sift-down algorithm for VEB-3 heaps

```
1 optSiftDown(i) {
2   nmod3 = 1;
3   left  = 1, r = 2;
4   wi    = array[i];
5   while (left < size) {
6     // local vars store array values for lower misses
7     wright = array[right];
8     wleft  = array[left];
9     // When we are at the first level
10    if (right >= size OR wleft<= wright) {
11      min   = left;
12      wmin  = wleft;
13      nmod3 = -1;
14    } else {
15      min   = right;
16      wmin  = wright;
17      nmod3 = 1;
18    }
19    if (wi > wm) {
20      // swap elements ...
21      i = min;
22      left  = i * 4 + nmod3;
23      right = left + 3; // position assumption
24    } else {
25      break;
26    }
27    // Loop unroll - leaf level
28    if (left < size) {
29      wright = array[right];
30      wleft  = array[left];
31      wi     = array[i];
32      if (right >= size OR wleft<= wright) {
33        min  = left;
34        wmin = wleft;
35      } else {
36        min  = right;
37        wmin = wright;
38      }
39      if (wi > wmin) {
40        // swap elements ...
41        i     = m;
42        left  = i + 1;
43        right = i + 2; // position assumption
44      } else {
45        break;
46      }
47    }
48  }
49 }
```

Listing A.4: Optimized sift-up algorithm for VEB-3 heaps

```
1 optSiftUp(i) {
2   mod = i % 3;
3   if (mod == 0 AND i > 0) {
4     // root level, compute parent index ...
5     if (array[parentIndex] > array[i]) {
6       swap(parentIndex, i);
7       i = parentIndex;
8      }
9   }
10  while (i > 0) {
11    // leaf level - loop unroll 1
12    mod = i % 3;
13    parentIndex = i - mod;
14    if (array[parentIndex] <= array[i])
15      break;
16    // swap parent index and i ...
17    i = parentIndex;
18    if (i <= 0)
19     break;
20
21    // root level - loop unroll 2
22    // compute parent index ...
23    if (array[parentIndex] <= array[i])
24      break;
25    // swap parentIndex and i ...
26    i = parentIndex;
27  }
28 }
```

Appendix B

Tables Appendix

Event name	Description
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L2_DCM	Level 2 data cache misses
PAPI_L2_ICM	Level 2 instruction cache misses
PAPI_L3_DCM	Level 3 data cache misses
PAPI_L3_ICM	Level 3 instruction cache misses
PAPI_L1_TCM	Level 1 cache misses
PAPI_L2_TCM	Level 2 cache misses
PAPI_L3_TCM	Level 3 cache misses
PAPI_CA_SNP	Requests for a snoop
PAPI_CA_SHR	Requests for exclusive access to shared cache line
PAPI_CA_CLN	Requests for exclusive access to clean cache line
PAPI_CA_INV	Requests for cache line invalidation
PAPI_CA_ITV	Requests for cache line intervention
PAPI_L3_LDM	Level 3 load misses
PAPI_L3_STM	Level 3 store misses
PAPI_BRU_IDL	Cycles branch units are idle
PAPI_FXU_IDL	Cycles integer units are idle
PAPI_FPU_IDL	Cycles floating point units are idle
PAPI_LSU_IDL	Cycles load/store units are idle
PAPI_TLB_DM	Data translation lookaside buffer misses

continued on next page

Event name	Description
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_TLB_TL	Total translation lookaside buffer misses
PAPI_L1_LDM	Level 1 load misses
PAPI_L1_STM	Level 1 store misses
PAPI_L2_LDM	Level 2 load misses
PAPI_L2_STM	Level 2 store misses
PAPI_BTAC_M	Branch target address cache misses
PAPI_PRF_DM	Data prefetch cache misses
PAPI_L3_DCH	Level 3 data cache hits
PAPI_TLB_SD	Translation lookaside buffer shootdowns
PAPI_CSR_FAL	Failed store conditional instructions
PAPI_CSR_SUC	Successful store conditional instructions
PAPI_CSR_TOT	Total store conditional instructions
PAPI_MEM_SCY	Cycles Stalled Waiting for memory accesses
PAPI_MEM_RCY	Cycles Stalled Waiting for memory Reads
PAPI_MEM_WCY	Cycles Stalled Waiting for memory writes
PAPI_STL_ICY	Cycles with no instruction issue
PAPI_FUL_ICY	Cycles with maximum instruction issue
PAPI_STL_CCY	Cycles with no instructions completed
PAPI_FUL_CCY	Cycles with maximum instructions completed
PAPI_HW_INT	Hardware interrupts
PAPI_BR_UCN	Unconditional branch instructions
PAPI_BR_CN	Conditional branch instructions
PAPI_BR_TKN	Conditional branch instructions taken
PAPI_BR_NTK	Conditional branch instructions not taken
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_BR_PRC	Conditional branch instructions correctly predicted
PAPI_FMA_INS	FMA instructions completed
PAPI_TOT_IIS	Instructions issued
PAPI_TOT_INS	Instructions completed
PAPI_INT_INS	Integer instructions
PAPI_FP_INS	Floating point instructions
PAPI_LD_INS	Load instructions
PAPI_SR_INS	Store instructions

continued on next page

Event name	Description
PAPI_BR_INS	Branch instructions
PAPI_VEC_INS	Vector/SIMD instructions (could include integer)
PAPI_RES_STL	Cycles stalled on any resource
PAPI_FP_STAL	Cycles the FP unit(s) are stalled
PAPI_TOT_CYC	Total cycles
PAPI_LST_INS	Load/store instructions completed
PAPI_SYC_INS	Synchronization instructions completed
PAPI_L1_DCH	Level 1 data cache hits
PAPI_L2_DCH	Level 2 data cache hits
PAPI_L1_DCA	Level 1 data cache accesses
PAPI_L2_DCA	Level 2 data cache accesses
PAPI_L3_DCA	Level 3 data cache accesses
PAPI_L1_DCR	Level 1 data cache reads
PAPI_L2_DCR	Level 2 data cache reads
PAPI_L3_DCR	Level 3 data cache reads
PAPI_L1_DCW	Level 1 data cache writes
PAPI_L2_DCW	Level 2 data cache writes
PAPI_L3_DCW	Level 3 data cache writes
PAPI_L1_ICH	Level 1 instruction cache hits
PAPI_L2_ICH	Level 2 instruction cache hits
PAPI_L3_ICH	Level 3 instruction cache hits
PAPI_L1_ICA	Level 1 instruction cache accesses
PAPI_L2_ICA	Level 2 instruction cache accesses
PAPI_L3_ICA	Level 3 instruction cache accesses
PAPI_L1_ICR	Level 1 instruction cache reads
PAPI_L2_ICR	Level 2 instruction cache reads
PAPI_L3_ICR	Level 3 instruction cache reads
PAPI_L1_ICW	Level 1 instruction cache writes
PAPI_L2_ICW	Level 2 instruction cache writes
PAPI_L3_ICW	Level 3 instruction cache writes
PAPI_L1_TCH	Level 1 total cache hits
PAPI_L2_TCH	Level 2 total cache hits
PAPI_L3_TCH	Level 3 total cache hits
PAPI_L1_TCA	Level 1 total cache accesses

continued on next page

Event name	Description
PAPI_L2_TCA	Level 2 total cache accesses
PAPI_L3_TCA	Level 3 total cache accesses
PAPI_L1_TCR	Level 1 total cache reads
PAPI_L2_TCR	Level 2 total cache reads
PAPI_L3_TCR	Level 3 total cache reads
PAPI_L1_TCW	Level 1 total cache writes
PAPI_L2_TCW	Level 2 total cache writes
PAPI_L3_TCW	Level 3 total cache writes
PAPI_FML_INS	Floating point multiply instructions
PAPI_FAD_INS	Floating point add instructions
PAPI_FDV_INS	Floating point divide instructions
PAPI_FSQ_INS	Floating point square root instructions
PAPI_FNV_INS	Floating point inverse instructions
PAPI_FP_OPS	Floating point operations
PAPI_SP_OPS	Floating point operations; single precision
PAPI_DP_OPS	Floating point operations; double precision
PAPI_VEC_SP	Single precision vector/SIMD instructions
PAPI_VEC_DP	Double precision vector/SIMD instructions

Table B.1: Hardware performance counters pre-set measurement events in PAPI

Table B.2: Benchmarks combining graph and priority-queue versions (a = $\times 10^8$; b = $\times 10^6$)

API	graph+pq	inst (a)	cyc (a)	L1acc (a)	L1miss (b)	L2acc (b)	L2miss (b)	time (s)
Generic	GNV AoP+bin_gen	18.681	32.413	11.438	22.431	30.886	7.361	1.535
	GNV AoP+veb3_gen	22.317	37.329	13.456	20.604	28.200	5.712	1.745
Encapsulated	GVG AoS+bin_aos	29.862	17.930	16.550	18.775	43.390	12.971	0.888
	GVG AoS+bin_soa	28.186	16.867	15.525	18.006	40.215	11.927	0.812
	GVG AoS+veb3_aos	31.204	19.825	16.179	18.108	42.348	12.417	0.989
	GVG AoS+veb3_soa	30.320	19.522	16.238	18.297	41.944	12.366	0.966
	GVG SoA+bin_aos	30.491	18.237	16.972	18.665	45.168	12.990	0.888
	GVG SoA+bin_soa	28.914	17.382	15.911	17.858	41.705	12.003	0.857
	GVG SoA+veb3_aos	31.981	20.393	16.536	18.014	43.300	12.412	1.005
	GVG SoA+veb3_soa	30.975	19.881	16.467	18.194	43.465	12.471	0.978
Decapsulated	GVG AoS+bin_aos	9.792	7.066	4.953	4.751	12.302	4.373	0.294
	GVG AoS+bin_soa	8.657	6.227	4.428	4.947	12.260	4.403	0.259
	GVG AoS+veb3_aos	12.618	10.294	6.512	4.081	10.705	3.820	0.420
	GVG AoS+veb3_soa	11.716	9.667	6.381	4.148	11.084	3.816	0.391
	GVG SoA+bin_aos	10.027	7.434	5.336	4.754	12.603	4.342	0.293
	GVG SoA+bin_soa	8.568	6.636	4.914	5.028	12.362	4.385	0.282
	GVG SoA+veb3_aos	12.389	9.974	6.109	4.104	10.777	3.800	0.430
	GVG SoA+veb3_soa	11.395	9.640	6.118	4.167	11.148	3.826	0.400