Parallel query evaluation on multicore architectures

Ulf Lilleengen

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Lasse Natvig, IDI
Co-supervisor: Cyril Banino Rokkones, Yahoo! Technologies Norway A/S

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The introduction of multicore architectures brings the promise of great performance at low cost. However, achieving high performance on multicores requires an adequate application model, i.e. with a level of parallelism in line with the underlying hardware architecture. Tuning for multicore requires insights into complex resource sharing (e.g., caches and bandwidth) and thread interaction (e.g., synchronization, coherence and false sharing) and the application workload.

This project is an experimental redesign of a search engine threading model for multicore processors. The main goal of this project is to implement an alternative thread parallelisation based on a per-core index partitioning scheme for the search core in Vespa, and evaluate its performance compared to today's model.

The project is in cooperation with Yahoo! Technologies Norway AS (YTN), who wants to optimize performance of its search engine (Vespa) on multicore architectures.

Assignment given: 15. January 2010
Supervisor: Lasse Natvig, IDI

Abstract

Multicore processors are common in server systems sold today. Writing application software that takes advantage of such systems, not to mention adopting existing software to the parallel domain, is complex. Workloads such as web servers, database servers and search engines are easy to parallelize, because each incoming client may be handled in a separate thread of execution. However, as as cache coherence schemes on multicore processors do not scale with the number of cores, new ways of scaling existing applications may be needed to make better use of the cache hierarchy.

This study evalutes an alternative method of running search engine queries in a search engine core developed by Yahoo! Technologies Norway. The method seeks to lower query latencies and average memory access times of the search core by making better use of multicore processor caches. Through the study of Vespa, the search engine platform used at Yahoo!, and techniques for using processor caches as good as possible, an alternative design based on parallel query evaluation is proposed. The design is evaluated in a simulator of the search engine core and tested in different configurations. The performance of the alternative design depends highly on the workload. However, the alternative design can be configured to act as the existing design, which makes it possible to get the best of both worlds.

Preface

This thesis was written at the Department of Computer and Information Science, Norwegian University of Science and Technology. I would like to thank my supervisors, Professor Lasse Natvig at IDI and Cyril Banino-Rokkonnes at Yahoo! Technologies Norway for their guidance and support. Thanks to Henning Baldersheim for sharing his technical expertise and experience with the Vespa search engine. I would also like to thank the employees at Yahoo! Technologies Norway and friends in the Computer Architecture group at IDI for our interesting discussions and for your support.

Contents

List of Figures

List of Tables

List of abbreviations

AMD	Advanced Micro Devices
API	Application Programming Interface
CPU	Central Processing Unit
CMP	Chip Multi Processor
HTTP	Hyper Text Transfer Protocol L
OS	operating system
POSIX	Portable Operating System Interface for Unix
QRS	Query Result Server
QPI	Quick Path Interconnect
TLP	Thread Level Parallelism
TLD	Top Level Dispatch
YTN	Yahoo! Technologies Norway AS
FS	Full System
QPI	QuickPath Interconnect
SE	System-call Emulation
ISA	Instruction Set Architecture
IDI	Department of Computer and Information Science
PID	Process Identifier
SLA	Service Level Agreement
TLB	Translation Lookaside Buffer

Chapter 1

Introduction

It is almost 10 years ago since IBM introduced the POWER4, one of the first multicore processors [SKT+05]. The later years have seen the introduction of multicore processors in commodity server and desktop systems, where dual and quad core Chip Multi Processors have become standard hardware components. With the recent introduction of the 8 core Xeon processor and the 80 core tiled-based processor from Intel [Neh09, Pol], as well as the 100 cores on chip packed by Tilera [TIL10] (sometimes referred to as *manycore* processors), software designers must design their applications with focus on Thread Level Parallelism. This report investigates an alternative design of a search engine in order to better utilize multicore processors.

Search engines need to scale in data size and query traffic volume dimensions. This is not an easy task because of the increasing data volume, but also because of changing trends as to which data is interesting. Key aspects of a successful search engine today is its ability to provide quick indexing as well as search latencies. Quick indexing latencies relates to *real time* search, which aims at making content available for search within a few seconds after the content is initially pushed into the search engine, even with thousands of such updates per second.

Processors with 100 cores are built for parallel workloads such as network processing, video processing as well as web and database applications, which are similar to that of search engines. Making sure that the search engine is ready to handle such processors is important in order to handle an increasing traffic volume, and to reduce the number of search nodes in a search cluster.

This study would not be possible without the cooperation of Yahoo! Technologies Norway AS (YTN). YTN is a small engineering team located in Trondheim working on the Vertical search platform at Yahoo!. The team has its origins in FAST Search and Transfer, which developed a much used web search engine at http://www.alltheweb.com .

1.1 Assignment interpretation

Redesigning the existing search core completely is a complex task, and the effects of a new design are not known. Spending a lot of time on an alternative design can be wasteful if one is not certain that the new design will be an improvement over the old design. Moreover, measuring the effects of such a redesign is a complex affair as there are many active components that may interfere with the measurement tools. By implementing a search core simulator that supports both designs using a realistic workload, the alternative design can be evaluated using simulator data and other measurement tools.

The assignment can be split into the following tasks:

T1 Design an alternative search engine threading model. This requires knowledge of the current search engine design and threading model. As the new threading model should be tuned for multicore processors, several factors of multicore architectures must be taken into consideration such as the target architecture's *organization* (how the cores are structured), and *cache parameters* (how many levels, access times, coherence schemes, etc). Further, the specific parts of the query evaluation that may benefit from parallelisation must be identified in cooperation with the experienced software engineers at Yahoo! Technologies Norway.

T2 Design and implement a search core simulator capable of running the workload of the current search core threading model as well as the alternative design. This task involves designing and implementing the software for a search engine simulator. The design employs techniques and principles to fully utilize multicore processors discussed in Chapter 2 and is the largest task in terms of the work required.

T3 Compare the new model with the current threading model in the search core simulator and evaluate whether or not the alternative threading model is useful or not. This task is important because it gives guidelines for a real implementation. Ideally, the simulator should be compared to the original search engine. The approach taken in this thesis is to discuss the design with employees at Yahoo! and present intermediate results in order to gradually increase the confidence in the simulator. The most important result of this task is a comparison of both designs in the search engine simulator that will give valuable experience when implementing an alternative design in the real search engine. The biggest challenge in this assignment is to understand the current search engine design and to write the simulator with that design in mind.

1.2 Main contributions

The main contributions of this report can be summarized as follows:

- C1 A study of techniques and challenges facing application developers interested in fully utilizing multicore processors.
- C2 A study of the Vespa search core and its threading model. The parts of the search core eligible for parallelisation are identified.
- C3 A search core simulator capable of simulating parts of the Vespa search core. The simulator can be extended to support other parts of query evaluation for other projects.
- C4 An algorithm for dynamically adjust partition boundaries.
- C5 A comparison of the original and an alternative threading model using parallel query evaluation.
- C6 An analysis of the behavior of parallel query evaluation in terms of scalability, its impact on the processor cache and its behavior during various workloads.

The first two contributions are useful for other employees at YTN, as it further documents the design of Vespa, and describes how parallel query evaluation can be implemented in Vespa. The third contribution is useful for further projects involving the Vespa search core, as it can be modified to evaluate experimental algorithms. The fourth contribution comes as a result of the alternative design, and can be usable in other applications. The fifth and sixth contribution answer the main research question in this assignment: how does the parallel query evaluation perform and is it worth implementing it in the real search engine?

1.3 Report outline

The rest of the report is laid out as follows: The first part of the report introduces the theoretical basis for the work performed in the second part of the study. Chapter 2 gives an overview of search engines, both in general and the search engine used in this study, a presentation of typical cache coherence schemes in today's CPUs, a discussion on possible tools that can be used to measure application impact on the hardware, and an overview of principles and techniques important for developing software on multicore processors. Chapter 3 provides an in-depth explanation of the core of the search engine used in this study, together with a design proposal for parallel query evaluation. Chapter 4 describes the search engine simulator developed in this study. Chapter 5 presents relevant metrics for evaluating the performance and the impact on the hardware. Chapter 6 presents experiments performed on the search engine simulator and compares the alternative design to the original design.

Chapter 2

Background and state of the art

Understanding the software architecture of the search engine as well as the hardware on which it is running is necessary in order to suggest any improvements to the software. Furthermore, as this study involves developing a search engine simulator, techniques for efficiently using multicore processors must be studied, and strategies and tools for evaluating its performance must be decided. This chapter consists of the following parts: Section 2.1 describes the basic structure of a typical search engine while Section 2.2 describes Vespa, the search engine used in this study. Section 2.3 describes multiprocessor cache coherence and techniques for making the software reduce the coherence traffic. Section 2.4 describes what tools can be used for measuring the impact on the processor cache hierarchy, while Section 2.5 gives an introduction to *profiling* tools used in this study. Section 2.6 discusses two well known laws in the parallel computing domain. Section 2.7 describes software libraries that are commonly used in applications that take advantage of multicore processors.

2.1 Search engines

Search engines play an important role in the modern Internet. The popularity comes from the need to make information easily and quickly available. Search engines are continuously enhanced to handle the ever growing amounts of data. A search engine operates on *queries*, which are automatically generated strings containing *terms*. Today, these queries are usually generated from a set of user actions such as clicking on an image or a hypertext link. The queries are analyzed by the search engine to find all relevant *documents*. In the end, the documents best matching these terms will be returned to the user.

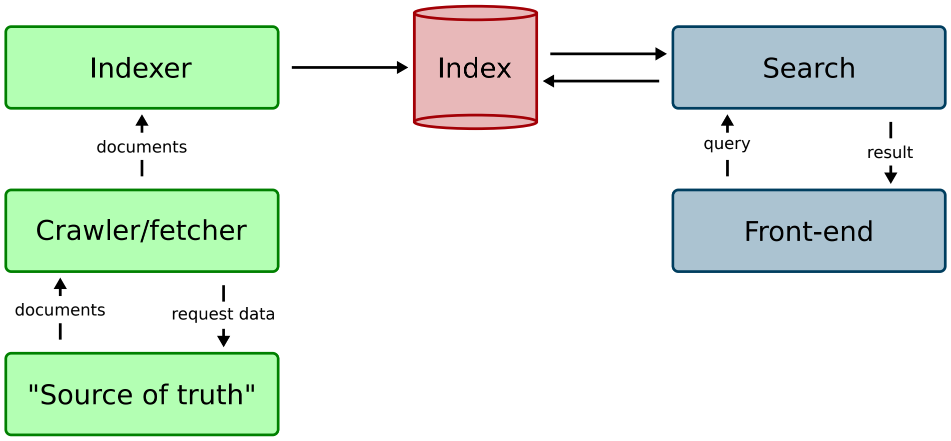


Figure 2.1: A generalized search engine architecture

Search engines are a relatively new field within the software industry, and new ways of calculating the *relevancy* of documents as well as scaling information retrieval on thousands of machines are developed to keep up with the vast amount of information as well as accuracy requirements from the user. Search engines can usually be described in three parts: *information retrieval*, *indexing* and *searching*. Figure 2.1 shows how these components interact with each other. Following is a quick description of these components. Risvik covers the different components in more detail [Ris04].

2.1.1 Retrieval

A web search engine will usually retrieve information by *crawling*[1] information from web pages. Another way is to fetch information from a "source of truth". The "source of truth" is the origin of a document that should be made searchable. The crawler/fetcher then *feed* or *push* the document to the search engine indexer. This method is typically used in search engines configured to search in a specific type of documents, such as e-mail or images, like Vespa.

2.1.2 Indexing

As search should be fast, the information needs to be categorized or *indexed*. One common way to organize information is to use an *inverted index* (shown in Figure 2.2). This type of index stores a mapping between a term and documents in which that term occurs in *posting lists entries*. Given a term, the position of all the documents matching this can easily be found by traversing the *posting lists*.

[1]crawling is a form of processing information and following any outgoing hypertext links

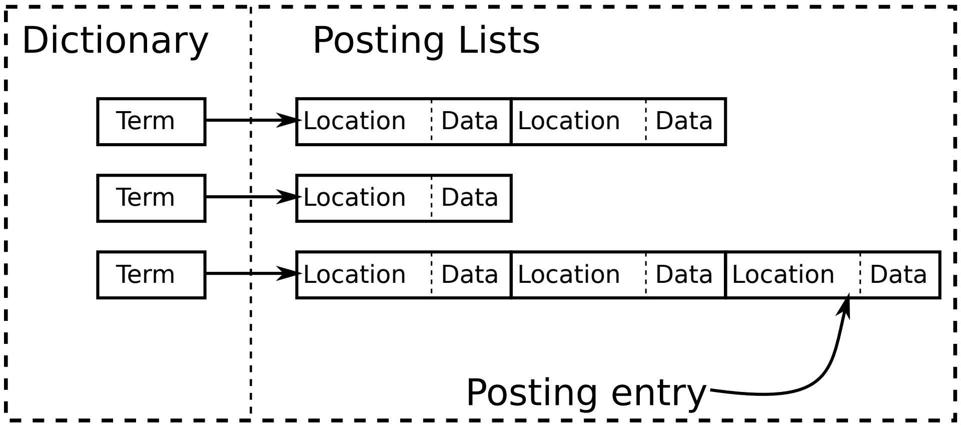


Figure 2.2: An inverted index (inspired from [FLQZ06])

2.1.3 Searching

Query evaluation and fetching the relevant documents are the main functionalities of the search engine. The engine must also parse the query, search the index for documents, and calculate a *rank* or *score* for each document. The rank describes how relevant the search engine thinks a particular document is. The most relevant documents are then returned.

2.2 Vespa

This section gives an overview of Vespa. All subsections except Section 2.2.2, 2.2.6, 2.2.7 and Figure 2.4, 2.5, 2.6 are copied and modified from earlier project work in course TDT4592 - "Computer Design and Architecture, Specialization Project" [Lil09].

The search engine used as a case study in this project is the Vespa search engine developed by YTN. Vespa is a scalable search engine, intended to be used by verticals (a service or application) within Yahoo! to provide *search functionality*. Vespa is able to handle data of various types and size. Figure 2.3 depicts the components within Vespa and how they interact.

Examples of applications using Vespa are Delicious [DEL10], Mail [YMA10] and Flickr [FLI10]. Delicious provides online *bookmarking*, browsing and *tagging*[2] of bookmarks. Delicious use Vespa to search for bookmarks and bookmarks with a specific tag, etc. Yahoo! Mail is the worlds largest e-mail service, and Vespa provides their users with the ability to search for information in their e-mails.

[2]A tag is similar to a keyword

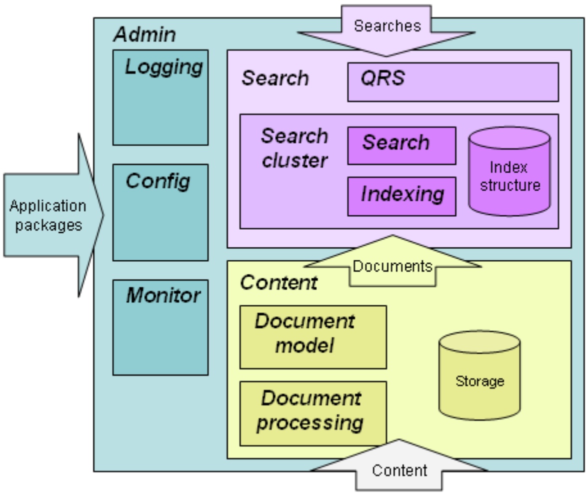


Figure 2.3: An overview of Vespa [VES10]

A complete Vespa application setup is called a *deployment*. The term *node* is used to describe a physical server in a Vespa deployment. Further, a collection of Vespa nodes is called a *cluster*, and a collection of nodes running Vespa search services is called a *search cluster*. Vespa is a collection of services, both the search core and supporting services. The services in Vespa may be configured to run on the same node, on completely separate nodes, or on a node together with only some of the other services. The configuration service takes care of managing all services on all nodes in a deployment. The configuration can be altered and re-deployed on the configuration service node, which will push it to all other nodes.

The administration services provide logging, configuration management, monitoring, and controls the other services in Vespa. The configuration service contains service specifications, which include node names and their assigned services. Also specified in the configuration is the *search definition*, which describes the content to be processed and served by Vespa as a *document*. Documents are *fed* into Vespa in a specific format matching that of the search definition. Each document is given a unique *document identifier*, commonly referred to as the *doc id*. During the *feed* operation, the content can be preprocessed by a *document processor*. Document processors can transform docs, filter docs and forward docs to their destination.

The services involved with indexing and query processing are the *indexer*, the QRS, the *Top Level Dispatch (TLD)*, and the *search* service. The indexer usually runs on the same node as the search service, while the QRS and TLD services run on separate nodes. Multiple *search nodes* running the indexer and search service may be specified in order to scale with the number of documents in an index and to get lower search latencies. A collection of search nodes form a *search cluster*, which is controlled by the TLD. Multiple search clusters may be specified, which duplicates data, but provides reliability and *load balancing*.

2.2.1 Indexing

After pre-processing, the document is sent to the *search cluster*, which indexes the document and makes it available for search. There are currently three types of different indexing modes, with different characteristics: *batch*, *incremental* and *realtime*.

Batch mode builds the whole index from scratch. One of the drawbacks with batch indexing is that it is not possible to add new documents without re-adding all documents currently in the index, but also that it typically requires two nodes: one for indexing and one for search, since this mode does not allow for the index to be searched while it is being built.

Incremental mode allows adding documents without having to re-add existing documents. Incremental mode treats each update as a new increment to the index. As documents are fed, a small index is created from these documents. This index is then later merged with the original index.

Realtime mode aims to make documents available as early as possible after indexing. Realtime indexing uses timing constraints set by the application in order to provide reliable indexing times.

2.2.2 Disk index and attributes

The index is stored on hard disk drives (together with the data), as the index size is usually much larger than what can be stored in memory. However, a high number of I/O operations to the disks will significantly increase the search latencies. Therefore, Vespa also supports keeping parts of a document in memory, called *attributes*, to provide lower search latencies and the ability to use *aggregate functions* (such as *max* or *min*) on query results. Attributes may be of fixed or variable size. The doc id, a few keywords or the creation date of a document are usually used as attributes. In case the query asks for the newest documents, having the creation date as an attribute can lower the query latency to find the correct documents significantly.

Figure 2.4 shows how the attributes are stored in memory. A vector is allocated for each attribute type, which contains an entry for each document. The doc id may be used as an array index to find the attribute value of that particular document.

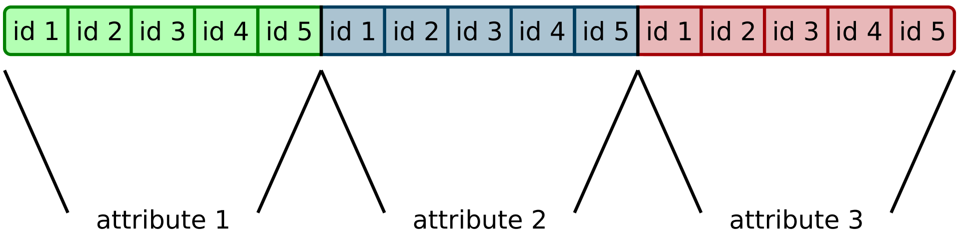


Figure 2.4: Attributes stored in memory. Each attribute is represented with a vector with entries for all doc ids

```
10.0.0.1 - - [24/Jan/2008:05:29:04 -0800] "GET /?query=yahoo
    HTTP/1.1" 200 270 "" "Mozilla/5.0 (Windows; U; Windows NT 5.1;
    en-US; rv:1.7.12) Gecko/20050915 Firefox/1.0.7" 2.792 1 0 1
```

Listing 2.1: QRS log file entry

2.2.3 Query Result Server

The initial query is sent to the QRS using Hyper Text Transfer Protocol (HTTP). The QRS parses the incoming queries and forwards them to the TLD of each search cluster, via internal protocols. The results from the TLDs are *blended*[3] and presented to the user, as shown in Figure 2.5. The QRS stores requests in a log file, using the format shown in Table 2.1. Some of the fields are left empty for future use. All fields are separated by a white space character. Listing 2.1 gives an example of a QRS log entry.

2.2.4 Top Level Dispatch

Risvik and Michelsen defines two dimensions where a search engine must scale [RM02]. The first dimension is the data size, where a search engine must be able to deal with an increasing data volume. Vespa scales in this dimension by using multiple search nodes, forming a *search cluster*. The second dimension is the traffic volume. Vespa handles this dimension by forming multiple search clusters, each controlled by a TLD. The TLD receives queries from the QRS and dispatches them to all search nodes. The results returned from the nodes are *merged* and returned to the QRS.

[3] *blending* is a form of merging and duplication removal

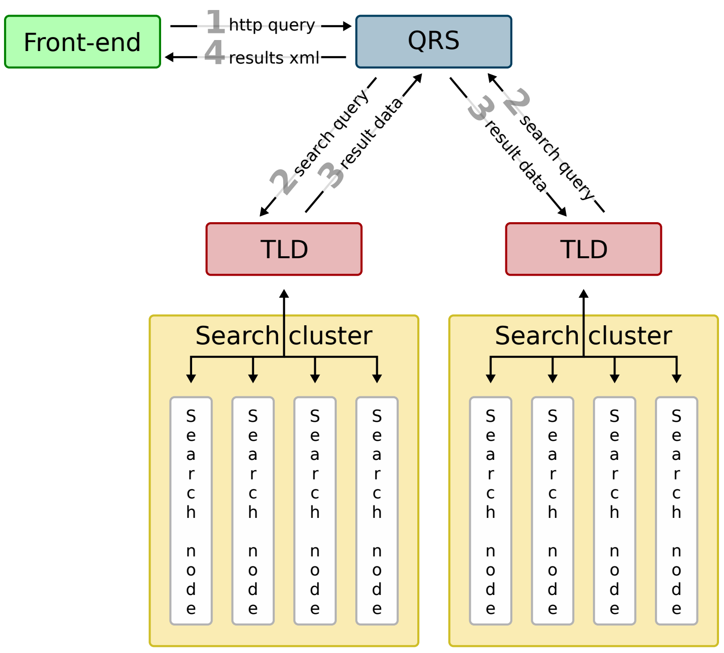


Figure 2.5: Data flow in Vespa

Table 2.1: The QRS log format [VES10]

Log Field	**Explanation**
SourceIP	IP address of client
-	Nothing
-	Nothing
Date	Timestamp (localtime) when query was executed with offset from GMT
"HTTPRequest"	HTTP request string
HTTPReturnCode	HTTP return code for the query
ByteCount	Number of bytes sent to the client
"Referrer"	HTTP referrer string
"UserAgent"	Client user agent
ProcessingTime	Time used to process the query in seconds. Including query parsing, back end execution. Does by design not include query reception from client and result set transfer time to client
TotalHitCount	The total number of hits for the result set
0	Reserved for later expansion - always 0
HitCount	Number of hits in the query result page. Note that the number of aggregation groups (if there are any) is included in this number.

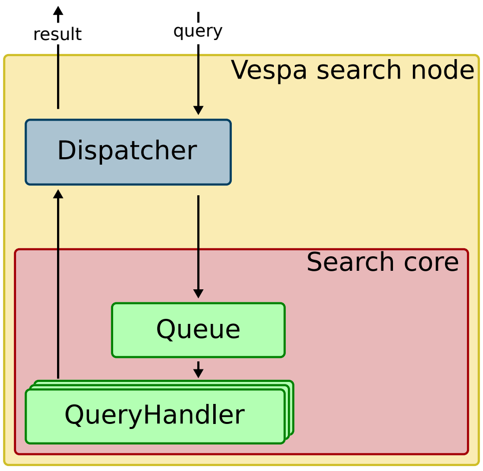


Figure 2.6: An overview of a Vespa search node

2.2.5 The Vespa search node

The performance of each search node is critical for the overall performance of Vespa, and is the Vespa component of focus in this study. The search service in a search node consists of a dispatcher and the *search core*. The number of search cores running on search node depends on the number of *slots*[4] the indexing mode uses. Realtime mode uses four slots in sizes from small to the complete index, which requires four search core instances. This allows new documents to be merged with the smallest slot quickly, and later be merged down with larger slots. Incremental mode uses only one slot, which in turn requires only one search core running on each node.

The search core is responsible for finding all documents in its index matching a query and thereafter compute a relevancy score for each hit using the method specified in the query. Results are then aggregated and sorted, and a subset of the results is returned.

In a search cluster with more than one search node, each search node handles only a piece of the *global* index structure. This way, a search query can be performed in parallel on multiple search nodes, thereby lowering response times for large data volumes as the number of nodes in the cluster is increased.

Figure 2.6 shows how the components of a search node work. The dispatcher receives queries from the TLD and forwards them to the search core. The search process puts the query in a *queue*, and the query is assigned to an available *query handler*, which also reports back the results.

If realtime mode is used, the results will have to be merged at the dispatcher

[4]a slot is an internal term used for an index that can be merged with other indices

before being returned to the TLD. The dispatcher uses a two-phase protocol when communicating with the search core instances. In the first phase, the hits from the different search cores are merged. In the second phase, the document summaries for each hit is fetched.

The search core caches frequent results in memory and uses other optimization techniques (discussed in Section 2.3.3) to reduce the latency of frequent hits and the number of disk accesses. The search core has undergone much development and refactoring over the course of the last 10 years, and has for a long time handled multiple queries by using threading [Bal10]. Therefore, the search core is already tuned for multicore by handling each query in a new thread. There is a big interest in improving performance by using multicore processors more efficiently. A detailed explanation of the search core design and the threading model is given in Chapter 3.

A typical search node running Vespa is a server with one or more Intel Xeon multicore CPUs, with a minimum of 8 GB of RAM and two or more disks (usually striped[5]).

2.2.6 Vespamalloc

Vespa uses a scalable memory allocated developed at YTN instead of the allocator coming with the GNU C library. Vespamalloc is designed to be fast and scalable, and ignores any extra memory usage necessary to achieve this goal. For instance, all internal data structures are pre-allocated by heavy use of C++ templates. This means that the allocator parameters are specified at compile time, instead of run time, which sacrifices flexibility to achieve higher speeds [VES10].

2.2.7 Radix sort

Radix sort is a sorting algorithm much used within Vespa, because of its ability to quickly sort an array of fixed-width numbers even if the array contains many of them. Radix sort has a time complexity of $O(kN)$, where k is the number of keys (for a 64 bit integer, the value of k is 8), and N is the number of entries in the array. Knuth gives a detailed analysis of the radix sort algorithm [Knu68].

2.3 Multicore processors and cache coherency

With multicore processors it is even more important to understand the processor architecture on which the application is running compared to single core processors, because the memory buses on such processors are vulnerable to contention. With multiple cores using the same memory resources, having a high hit ratio in the

[5]Striped organization is sometimes referred to as RAID-0 organization [PGK87]

different cache levels is necessary for scalability on multicore architectures. The cache parameters of the architecture used by the application should be collected in order to optimally structure the data.

2.3.1 Cache coherence

To understand the cache bottleneck of multicore systems, one has to look at the cache coherence mechanisms and protocols used in a modern processor. In a typical Chip Multi Processor (CMP), each core has its own level-1 cache, but the second or third level is shared with other cores, which requires a mechanism to make sure each cache is aware of other caches containing the same data. A common mechanism found in today's CMPs is snooping [HP07]. Snooping works by keeping the state of each cache line distributed between all caches and having each cache snoop (or listen) on the system bus. On a cache miss, a request is broadcast on the system bus, and if any other cache does have this cache line, it will respond. Clearly, the broadcast of cache misses can cause a lot of traffic on the system bus, and applications should strive to minimize this traffic. Usually, a cache coherence protocol like MESI, which defines a set of states and rules for transitions between them, is used to maintain coherence. This means a cache line can be in a modified, exclusive, shared or invalid state.

The importance of "playing nice" with the cache coherence mechanism and protocol is illustrated in Figure 2.7 with a dual core processor having one level-1 cache per core and a shared level-2 cache. The data structure is an entry into a cache containing a doc id, a reference count for the entry and the some data for that document.

By keeping as little shared data as possible between threads in the search core, the cache coherence traffic can be minimized. However, it is not always possible to share no data at all when thread communication is based on a shared memory model. For instance, the search core uses a result cache for the most frequent query results. Although the data in such a cache is not modified, the meta data used to keep reference counts etc. may be updated frequently. If several threads work on the same data set in the cache, the coherence traffic may increase significantly, affecting performance.

2.3.2 Cache affinity

Because of the limited bandwidth of the processor memory systems today, it is important to reuse as much of the data in the cache as possible. Suppose a thread running on a processor core makes good use of the cache and sustains a high hit rate. If this thread is moved to another core, the contents of the cache may not be the data that the thread is using, and the thread will suffer many cache misses until the cache gets filled up again. The relationship between thread data and

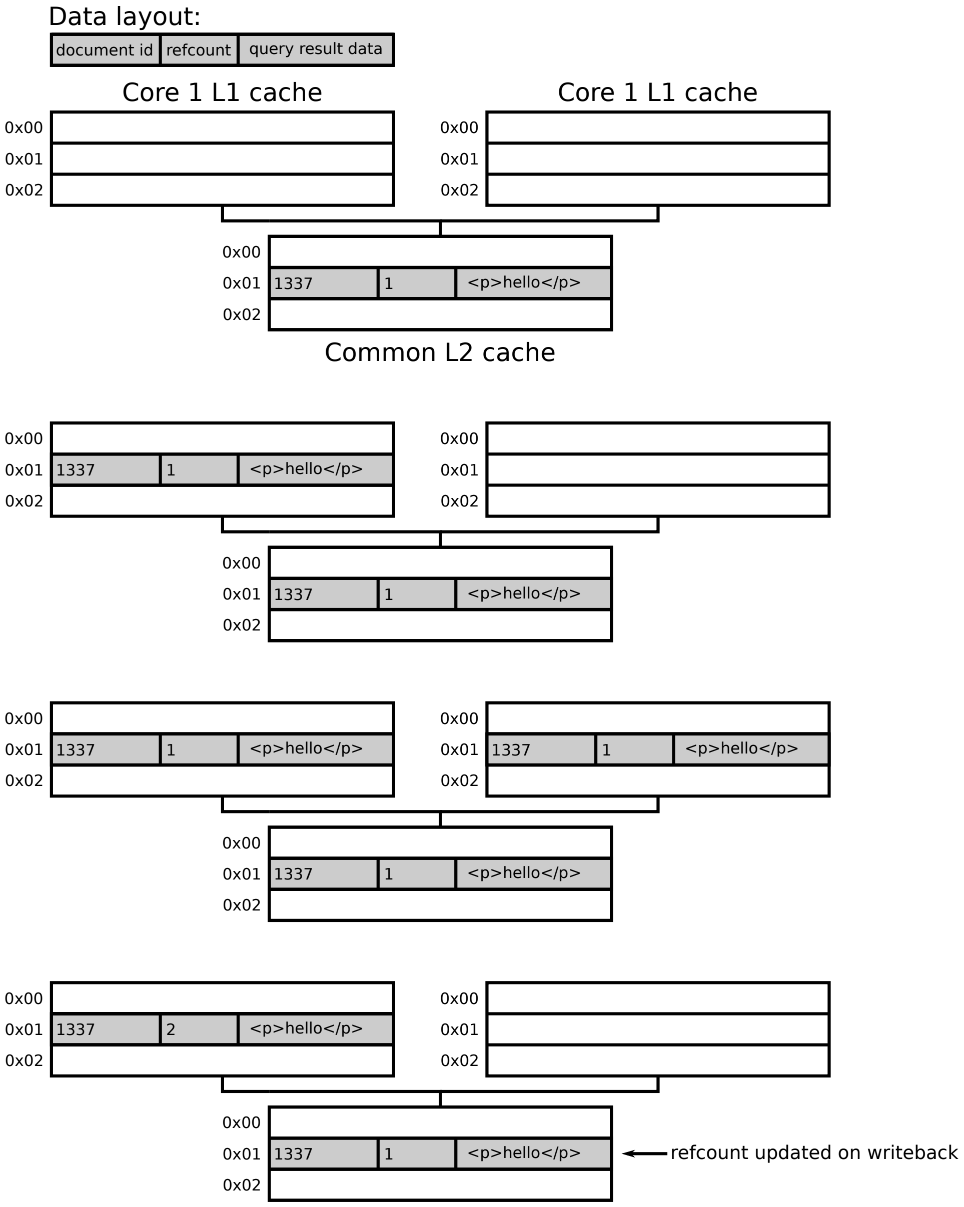


Figure 2.7: Two threads try to access the same data. a) The shared L2 cache contains result data. b) Thread running on core 1 loads data into its L1 cache. c) Thread running on core 2 loads the same data into its L1 cache. d) Thread on core 1 increases reference count and invalidates cache line in the L1 cache of core 2. Eventually, the second core has to reload the cache line from the level-2 cache to get the correct copy.

cache placement is usually referred to as *cache affinity* or *thread affinity.*

The overhead of reloading data into another cache on another core is called *thread migration.* Teng et. al. analyzed the cost of migrating threads across cores, both on-chip and off-chip, and lists several important factors of thread migration. The most important factor is the migration frequency. If there is a low number of migrations, there is no big impact on the performance. Furthermore, there are different types of migration, where a migration between separate physical chips are more costly then a migration to another core on the same physical chip. The working set size also matters a great deal, as applications with a small or very large working set do not get a notable performance impact.

Today's operating systems do take cache affinity into account when deciding on which core to schedule threads, but it is not always possible to use a busy core. The ability for an application to decide on which core a thread should run can decrease the number of thread migrations. Foong et. al. investigates the use of thread affinity to potentially increase the throughput of network processing, and a throughput gain of up to 25% is achieved by forcing threads to run on a specific core. Although network processing is a different workload from that of a search engine, the use of tools and libraries for setting affinity is explained, and contributes a valuable experience of using such tools.

2.3.3 Software techniques for improving multicore performance

In software design, there are several techniques that the designer may use to improve performance on multicore processors. There are many books on this subject, and for the C++ programming language, Bulka et. al provides an in-depth study of how the designer can employ these techniques both generally and in C++ [BM99]. Although written before the multicore era, the techniques discussed for multiprocessor systems are still highly relevant for multicore processors. Maurice et. al. provides an in-depth explanation of synchronization primitives, as well as data structures that performs well on multicore processors [HS08]. Following is a description of some of these techniques and how they are applied in this study. Techniques not directly related to Vespa or the simulator written in this study, can be found in Section A.3.

Minimize synchronization overhead

For shared memory systems, using shared data structures protected by *locks* are a common way for applications to ensure data consistency. However, as the number of threads trying to access the same data increases, the locks will experience *contention.* A simple way of achieving better scalability is to use *finer grained locking* by performing *lock decomposition.* However, there is ultimately an end to how well

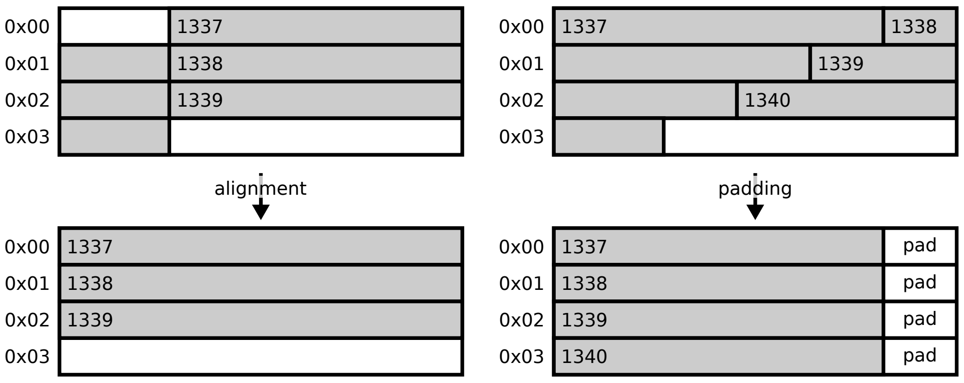


Figure 2.8: Alignment (to the left) and padding (to the right) of data

an application can scale using this method. Most CPUs come with *atomic instructions*, which are mostly used to build locks. These primitives can also be used to implement atomic types, which can replace locks in some situations. Further, more advanced data structures such as lock-less queues can be built using atomic instructions.

Vespa and the simulator created in this study both tries to minimize synchronization overhead by making each component as self contained as possible. Instead of using a global lock to protect the result cache, a lock per entry in the cache can be used to allow concurrent access to other entries.

Eliminate data sharing

The best way to reduce conflicts is to share as little data as possible between threads. Instead of sharing data, each thread may keep a local copy of the data. This approach leads to a higher memory usage, as the same data is stored at multiple addresses, and is not applicable for large data volumes on machines without huge amounts of memory. From the software design point of view, this is major design decision, as the data model of applications may have to be design with cache coherence in mind.

In Vespa, each query is executed within its own thread, and does not share any data structures related to the query evaluation with other threads. As mentioned above, though, the result cache and other caches needs to be locked, which can become a problem if the number of threads using the same cache is large. In the simulator developed in this study, even less data is shared, as each thread operates in a completely separate memory range than other threads.

Alignment and padding of data structures

Careful design of the data structures used in a program can save the application from extra cache misses and extra delay when accessing main memory. Figure 2.8 a illustrates the importance of keeping data aligned. If a thread reads misaligned data, the data occupies two cache lines, when it could have occupied only one. Figure 2.8 b illustrates the use of padding. By using padding, all first accesses to the data structures will cause only 1 cache miss instead of 2.

It is worth mentioning that newer CPU architectures such as Nehalem supports unaligned access, making these optimizations less important. By using the GCC compiler specific operations such as `__aligned()`, one can align static data structures to cache line boundaries if desired. For systems supporting POSIX, `posix_memalign()` provides a way to dynamically allocate aligned memory. The POSIX alignment functions are used in the simulator to make sure that attribute data is aligned. In Vespa, a set of wrappers are used to make sure that I/O requests are aligned. Moreover, data structures are carefully planned to make sure that 64-bit boundaries are ensured.

Cache line utilization

If data is not packed or if only parts of the data in a data structure is used during each access, one can observe poor *cache line utilization*. By packing data better, fewer cache lines are needed to access the data, and more data can be placed in the cache. However, this must be seen in combination with padding, as crossing cache line boundaries can have bad effects when data is shared [AB10b]. A previous study revealed that, in some parts of the code, Vespa does not properly utilize each cache line [Lil09]. It is, however, not always possible to pack data optimally. A *heap* data structure, used in Vespa, will not get proper cache line utilization, because of a random access pattern when using it and inserting new entries.

2.4 Measuring effects on the cache hierarchy

There are several ways to measure effects on the processor cache hierarchy. One way would be to run it in a CPU *simulator*. Another is to measure the application on the target platform hardware by using the *performance counters* of the CPU.

2.4.1 CPU simulators

CPU simulators are often used by hardware architects to measure the effects of modifications to a hardware design. However, it may also be used to measure the impact of the software on the hardware. As these simulators are written as software, any signal or component may be inspected for relevant statistics. Most simulators also allow the user to specify parameters such as *cache size*, *cache organization*, the *number of processor cores*, the *interconnect* and so on. As this study tries to improve parallel query evaluation on multicore CPUs, being able to adjust the number of processor cores is an important property of a CPU simulator. Lande evaluates a set of CPU simulators for the Computer Architecture Group at Department of Computer and Information Science (IDI) [Lan06].

The Simics simulator

According to Lande, the Simics [SIM10] simulator system can simulate many different CPU architectures[6]. Simics aims to run complete and *realistic workloads* and is a *full system* simulator, which means that it runs a complete operating system on top such as Windows or Linux. However, it is not free of cost, and requires setting up appropriate model libraries to make sure the simulator behaves as the target platform.

The M5 simulator

Another simulator evaluated by Lande is the M5 simulator. According to [M5S10], M5 is a "modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture.". The M5 simulator can run in two different modes. A Full System (FS) mode, which behaves like an ordinary computer system or Simics, and in System-call Emulation (SE) mode, which emulates some of the system calls of the Linux operating system. In FS mode, only the Alpha architecture model supports more than one CPU core. Unfortunately, the Alpha architecture is different from the Vespa target architecture.

In SE mode, there are no theoretical limit on the number of processor cores, and the CPU architecture can be configured easily with different Instruction Set Architecture (ISA)s. However, as there are no actively developed *threading library*, *thread scheduler* and other important operating system mechanisms available in SE mode, a lot of time would have to be spent on creating these facilities. And more importantly, having them behave as a real system can be difficult.

2.4.2 Performance counters

Most modern processors provide one or more *counters* that can be configured by the software to count *events*. An example of an event is a retired instruction, or a cache miss in the level-1 cache. The counter can be configured to set an interrupt signal once the counter has increased beyond a certain value (overflowed).

[6]UltraSPARC, Alpha, x86, x86-64, PowerPC, Itanium, MIPS and ARM

Performance counters are used as a building block in advanced profiling tools. The Nehalem architecture has four counters which may be configured to sample over 100 different events [Int10]. Clearly, using these counters requires careful study of their meaning and how to derive relevant numbers for a particular application. The Intel optimization reference manual [Int09] and Drepper [Dre07] discuss how some counters may be combined to give cache hit and miss rates.

2.4.3 Comparing against the target platform

The main arguments against using a CPU simulator in this study are that they may significantly divert from the target platform of the real world application, and that they do not offer big enough advantages in terms of simplicity and accuracy compared to CPU performance counters and cache simulators on the target system.

The performance counters of modern CPUs can measure cache effects on all cache levels, and profiling tools can create in-depth profiles of where an application spends most of its time. In this study, performance counters are used for application profiling, because it allows the application to run on the target platform.

2.5 Software profiling tools

A useful and systematic way to analyze an application is to gather various statistics, such as CPU time or cache miss count, in order to analyze application performance. From these statistics one can generate a *profile* of the application. With a profile, one can get insights into the application behavior and possibly compare it to another profile for an earlier version of the application. However, as application complexity grows, gathering and interpreting statistics gets increasingly harder. Fortunately, there are so called *profiling tools* that help in this task, both commercial and open source.

2.5.1 Profiling tools using performance counters

The *OProfile* tool (available for Linux platforms) is an open source tool bundled with most Linux distributions today [OPR02]. Each time the operating system is interrupted by an overflowed performance counter, an OProfile kernel module records the program address of each thread in the operating system together with the event type [Lev03]. After the sampling is finished, one can combine the program address and symbols of the running program to find where the event occurred. It is important to note that OProfile is a *statistical* profiling tool, which means that a sampled event may not have occurred at the exact program address, and that confidence of the correct address is achieved through gathering many samples

[Lev04]. This property is common for all tools using performance counters as a basis.

Zaparanuks et. al. [ZJH09] provides a detailed study of hardware performance counters and of the accuracy of these counters when profiling small segments of code. The measurement error for counters when running user mode instructions is very low. However, for kernel mode instructions, the error is larger. This is because more registers are available in kernel mode, which all requires to to be stored somewhere before accessing the counters, and because interrupt handlers may cause events that can be attributed to the currently running thread. As the application in this study will be running in user mode, it is expected that the measurements are reliable. Further, by using long and multiple benchmark runs together with statistical analysis, the confidence in the measurements is increased.

The Acumem ThreadSpotter tool from Acumem [AB10a] takes the profiling a step further. By doing post analysis of the sampled data, this tool can generate suggestions as to what the application designer can do to use the hardware more efficiently. ThreadSpotter was used on Vespa in an earlier project study [Lil09], and was able to detect low cache line utilization in some parts of the code. However, the issues detected were not fixed, because fixing them usually meant sacrificing flexibility and good programming practice. An advantage of ThreadSpotter is that it does a lot of the calculation of the various performance counters for you, which does save the performance analyst some time as opposed to OProfile. It was however unable to identify fundamental software design inefficiencies in any greater details than OProfile. This and the fact that OProfile is a free, open source tool made OProfile the sample based profiler of choice in this study.

The VTune profiling tool from Intel is a widely used commercial tool capable of producing profiles on cache usage and thread interaction in multithreaded applications. Much like Acumem ThreadSpotter, VTune is able to give advice on how to restructure the program code to better utilize the hardware. In addition, VTune generates call graphs and is integrated with common development environments [VTU10]. VTune has not been used, since OProfile covers the same functionalities necessary for this study.

2.5.2 Valgrind

Valgrind is a collection of tools such as *callgrind* used to debug and analyze programs. The callgrind tool is a *callgraph* generator, capable of profiling an application in more details and greater accuracy than OProfile. Valgrind runs the application within its own control, by translating program instructions at run-time into its internal instruction representation that may be transformed in the way the user specifies, and then converted back to instructions to be run on the processor [VAL09]. As callgrind is able to get a very accurate profile of an application, it can be used to verify that the OProfile results are somewhat correct.

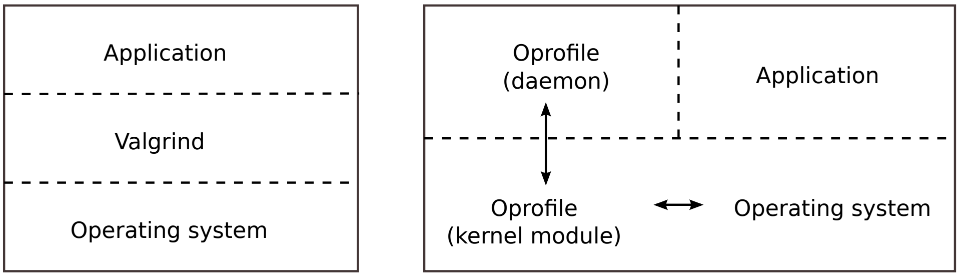


Figure 2.9: Comparing OProfile and Valgrind(from [Lil09])

2.5.3 What kind of profiling tool to use?

The big advantage of sampling based tools such as OProfile is the low performance impact on the profiled application. According to initial test measurements done in this study, application performance is decreased by only 1-18% when using OProfile, depending on the parameters given. For comparison, an application running in Valgrind may run 4-5 times slower [VAL00]. Figure 2.9 shows the difference in how OProfile and Valgrind operates. The overhead clearly comes from the extra processing done per instruction. OProfile on the other hand, does not directly intervene in the application execution. In this study, OProfile is the main profiler for cache, but Valgrind is used during the development of a search core simulator.

2.6 Amdahl's and Gustafson's law

One of the most famous laws in the parallel computing domain is undoubtedly *Amdahl's law*, as a result of Amdahl's considerations outlined in [Amd67]. The law states that the speedup of an application with some parts running in parallel is inherently limited by the serial fraction of the execution time. This means that although the fraction running in parallel achieves *perfect scaling* (that the time spent is halved when the number of parallel jobs are doubled) but that fraction does not constitute 100% of the run time when run in serial, the speedup will in the end be limited by the serial fraction of the application. The speedup achieved according to Amdahl's law, S_{amdahl}, can be defined as follows [Nat91]:

$$S_{amdahl} = \frac{1}{s + \frac{(1-s)}{N}}$$

Where s is the *serial fraction* of the execution time. For instance, if the serial fraction is 10% of the execution time and the other fraction is parallelized using 8 processors, the maximum theoretical speedup achieved according to Amdahl's law

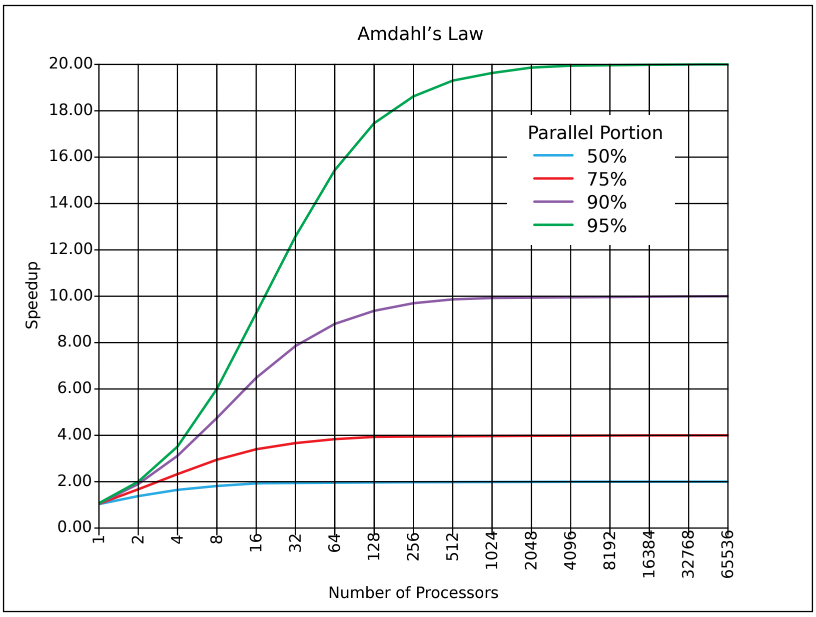


Figure 2.10: The speedup of an application with different portions of parallelism according to Amdahl's law [Dan09]

is

$$\frac{1}{0.1 + \frac{(1-0.1)}{8}} = 4.7$$

Figure 2.10 shows the plots of speedups for different portions of parallelism. There has been much debate around the implication of Amdahl's law, as it displays a very pessimistic view of parallel computing. It is therefore important to take into account that Amdahl's law can be seen as solving a *problem of the same size* while increasing the number of processors. Another law describing a case where the *problem size increases* together with the number of processors is often referred to as *Gustafson's law* after an article by John L. Gustafson [Gus88], where the assumptions of Amdahl's law did not seem to be valid in practice. Gustafson and others working at Sandia proposed an alternative formulation of speedup:

$$S_{gustafson} = N + (1 - N)s'$$

where N is the number of processors and s' is the serial fraction *when running on N processors*. Using this formula, the speedup of the previous example (given that the serial fraction when running on 8 processors are the same) becomes:

$$8 + (1 - 8) * 0.1 = 7.3$$

This assumes that the problem size increased with the same factor as the number of processors. However, both laws are actually proved equal, only having different

Table 2.2: Library calls used to set processor affinity in Linux

API call	**Description**
`CPU_ZERO(set)`	Clear all CPUs from *set*
`CPU_SET(cpu, set)`	Enable CPU *cpu* in the CPU set *set*
`sched_setaffinity(pid, set)`	Set affinity of *pid* to *set*
`sched_setaffinity(pid, set)`	Store affinity of *pid* in *set*

parameters, according to Shi [Shi96]. This means eventually that both laws may be used to describe speedup, as seen in Chapter 3.

2.7 Software libraries

Several libraries that helps the programmer in writing scalable parallel programs are available. These libraries usually employ techniques discussed in Section 2.3.3, are optimized for each particular architecture, portable, and saves the programmer the time of creating his or her own version.

2.7.1 POSIX Threading library

The Portable Operating System Interface for Unix (POSIX) threading library (called *pthreads*), implements thread creation and scheduling on UNIX systems as well as a set of synchronization primitives. Using multiple threads within an application allows tasks to run in parallel and share data. The synchronization primitives offered by POSIX can be used to ensure data consistency in case the data is read and written in different threads.

Pthreads provides synchronization mechanisms like *spin locks*, *mutexes*, *conditional variables* and *barriers*. Spin locks are the quickest synchronization mechanism available, as a thread will busy wait when acquiring the lock. However, if the lock will be held over a longer period of time, a mutex may be a better choice, because it allows the operating system to put the thread to sleep and allow another thread to run. Condition variables are often used by producers to notify consumers that a queue is no longer empty and by consumers to notify producers that the queue is no longer full. Barriers allows many threads to be synchronized by incrementing a counter up to a specified value. When the counter has reached the specified value, all threads are allowed to continue execution. A thorough analysis of the different primitives in the light of multicore systems can be found in [HS08].

2.7.2 Affinity APIs

Linux provides an Application Programming Interface (API) for an application thread to set the affinity of a process or individual threads via the GNU C Library. This API is not yet standardized, which means that is not yet portable, although portable libraries exists [oB09]. The function calls in Table 2.2 show a simplification of the most crucial library calls needed to set the affinity. The CPU_-macros operate on a *set data structure* where CPU *identifiers* can be added or removed. The sched_-functions are system calls that interact with the internal data structures of the Linux kernel, setting or retrieving the currently active CPU set for a thread or process, identified with a Process Identifier (PID).

Chapter 3

Vespa search core design

The Vespa search core is in charge of evaluating queries and returning relevant documents from the index. To implement parallel evaluation on queries, the existing design is analyzed to see which parts of the query evaluation that may benefit from the increased parallelism. Section 3.1 discusses a previous attempt at parallel query evaluation. Section 3.2 describes query evaluation in the original search core design, while Section 3.3 discuss what parts of the query evaluation that can be parallelized. Section 3.4 describes an alternative search core design based on parallel query evaluation, while Section 3.6 discusses the implications of such a design.

3.1 Query evaluation strategies

Bonacic et. al. present strategies for taking advantage of CMPs in search engines on a Sun Niagara T1 CPU [BGM+07]. Two different strategies for query evaluation are proposed: a *synchronous* approach, where each query is evaluated in its own thread, using the same code to evaluate different queries. This approach is simple, and requires little modification to the serial version of the query evaluation, but may cause contention for resources, as threads operate on a large data set while using the same cache. Strategies to minimize this cost involve running queries that matches the same terms on the same core. The synchronous approach is used in the Vespa search core, and is explained in detail in Section 3.2.

Another approach, similar to the parallel query evaluation approach presented in this study, is the *asynchronous* approach, where query evaluation itself is parallelized. This approach splits the inverted index into one partition per CPU core, which is traversed in parallel. This approach potentially benefits from local CPU core caches, and can be advantageous during low loads, where some cores may stay idle in the synchronous approach.

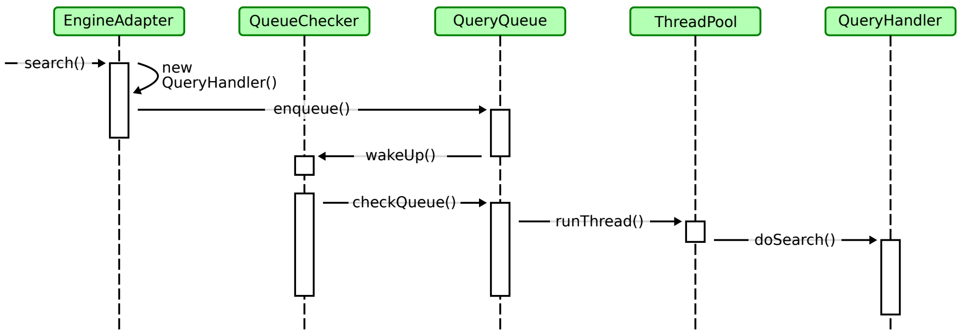


Figure 3.1: A sequence diagram of query dispatching

The synchronous approach gives a high throughput as the number of threads increases. The conclusion from [BGM+07] is that the asynchronous approach gives little gain because the amount of data to be processes is too small, causing a large overhead of dispatching and merging.

Why explore the asynchronous approach of parallel query evaluation then? First, the study does not go into detail of the workload used when evaluating the performance of the two strategies. Moreover, the Niagara architecture used in their experiments is different from the Nehalem architecture used in the current Vespa search nodes. Furthermore, study does not give an analysis of the impact on the different cache levels, or any analysis of the serial and parallel fraction of the query evaluation. Future processors may lack much of the cache coherence mechanisms that we take for granted today. The synchronous approach will then loose any benefits it has from sharing data among threads. Finding the reason for any limitations of parallel query evaluation is important in order to suggest improvements and to provide rationale for any future redesigns of the search core.

3.2 Query evaluation in the Vespa search core

The search core handles incoming queries and performs *matching* (finding documents that matches the query terms), *ranking* (calculating how relevant each document is), and *fetching* (retrieving the documents from disk). An overview of how the search core handles an incoming query is shown with a simplified sequence diagram in Figure 3.1. The search core operates with a *thread pool* of pre configured size. Each new query is evaluated by a *query handler*, which runs within its own thread. Each new query generates a *query handler*, which is placed on a *query queue*. A specialized *queue checker* thread checks this queue for new entries. When a query handler is dequeued from the query queue, the query handler is started.

The query handler itself is the start point of query evaluation, which includes running a *query processor* to perform matching and perform ranking. A simplified

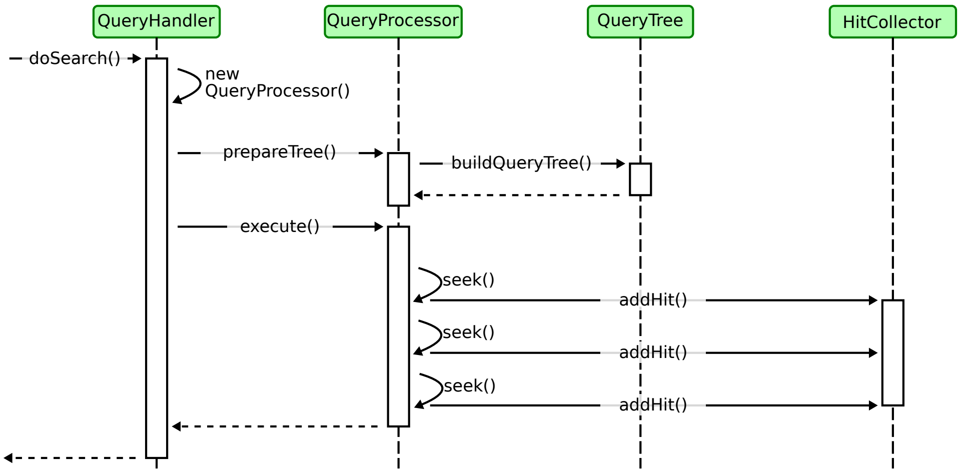


Figure 3.2: A sequence diagram of query evaluation

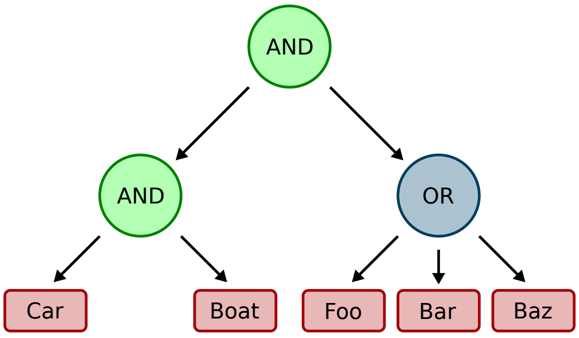


Figure 3.3: An example of a query tree from the following query: ("Foo" OR "Bar" OR "Baz") AND ("Car" AND "Boat")

sequence diagram of the query evaluation is shown in Figure 3.2. A query processor is used to find documents matching a query by generating a *query tree*. An example tree is shown in Figure 3.3. A query tree is tree representation of the terms and the operators used in a query. The *leaf nodes* of such a tree is the terms or *phrases* in the query, while the intermediate nodes are operators that describe how these are combined. An inverted index is used to locate the doc ids of the documents matching the terms.

The ranking phase uses a *hit collector*, which works as follows: For each matching doc id, a *rank score* is calculated from the *attributes* of a document, and a *hit*, consisting of a doc id and the rank, is inserted into an array. When the number of hits in the array reaches the maximum limit, it is transformed into a min-heap, putting the lowest ranking hit at the top. Subsequent hits are now checked against the lowest ranking hit in the heap. If the rank score of the new hit is higher, it is swapped with the lowest ranking hit. See the description of the search core simulator in Section 4.5.8 for a detailed description of how hit calculation works.

As this design is similar to the synchronous approach taken in [BGM$^+$07], it scales with the increasing number of queries, as each new query is handled by a separate thread. The design is simple, as there is little overhead in dispatching a query to a thread. A higher query load can be handled by adding more processors.

3.3 Parts eligible for parallelisation

The time, T, spent on query evaluation can be split into T_m, the time spent on query parsing and matching, and T_f, the time spent on fetching documents and calculating the rank. The total query evaluation time T is composed of these two:

$$T = T_m + T_f$$

The first part of query evaluation is not easily parallelizeable. Parsing a query is quick, and generating a query tree in parallel would involve a lot of locking of the tree data structure, which is not believed to give any notable gain. Most importantly, the time T_m is believed to be only a small fraction of T [Bal10].

The second part of query evaluation involves calculating the rank of documents, and constitutes the largest fraction of the query evaluation time. Furthermore, this part can easily be parallelized by splitting the doc id range into *partitions* and handling the documents within each partition in a separate thread.

It is important to note that the ranking only involves the attributes residing in memory, and not on disk. This means that no disk I/O is required during the first phase of ranking, which probably would have become a bottle neck of parallel query evaluation. Therefore, the alternative design assumes that only the attributes residing in memory is used to calculate the rank of a document.

3.4 Alternative query evaluation design

Splitting the doc id range into partitions makes it possible to perform ranking in parallel. These partitions may be variable in size depending on the number of matches occurring within each partition over time. For each partition range, a *fetcher* calculates a rank and stores a hit using a hit collector. As soon as the hits within a partition is collected, they are sorted and merged with the hits from other partitions. Using only 1 partition corresponds to the original design.

Figure 3.4 shows how the doc id range is mapped in the original design as well as the alternative design. Ideally, the fetchers in charge of the same partition will run on the same core and share little data with the other cores. In practice, however, the

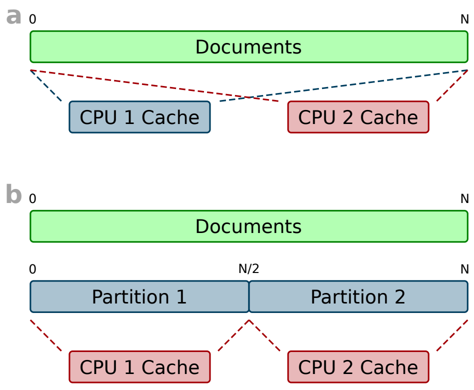


Figure 3.4: The mapping of the doc id range to partitions for a) The original design, where each CPU cache will contain data from the whole document range. b) The alternative design, where the range is split into partitions that potentially map to a CPU cache

time spent fetching documents in a partition will not be equal across the fetchers. Allowing threads to migrate to other cores is necessary to avoid under-utilizing CPU cores. However, today's operating system thread schedulers takes the *CPU topology* (the layout of cores and on which physical chip they reside) into account. which should prevent unnecessary thread migration.

3.4.1 Cache effects

A possible effect of partitioning the data is a higher hit rate in the per core on-chip caches. Since the number of hits within each partition decreases as the number of partitions increase, the probability of having to reuse data in the cache is higher. Moreover, as each core operates on separate data, the impact on the cache-coherence protocols should be reduced. However, the benefits could vary, depending on the cache hierarchy. If the cores share a common cache, the there will be little advantage when using the last level cache, because the threads operate on separate data sets.

Assume a quad core processor, where each core has its own L1 cache of 32kB, and the four cores share an L2 cache of 1MB. Further assume an index containing 2 million documents and that each query requires 1000 documents to be evaluated on average, while the 100 best ranked hits should be returned for each query. Given that a document contains 120 bytes of attributes to evaluate its rank, and that a hit is stored as a tuple consisting of the doc id and a rank of 8 bytes each, each query requires 120×1000 bytes to be imported into the cache, while the heap requires 16×100 bytes.

First, consider the original search core design running four queries at the same time. The L1 cache can fit the attributes of 273 documents. This gives a probability of $\frac{273}{2000000} = 0.00014$ for finding the attributes in the cache, since the whole document range is searched. The heap requires only 1600 bytes, which easily fits in the L1 cache. Since the heap is used frequently, it can be assumed to be contained within

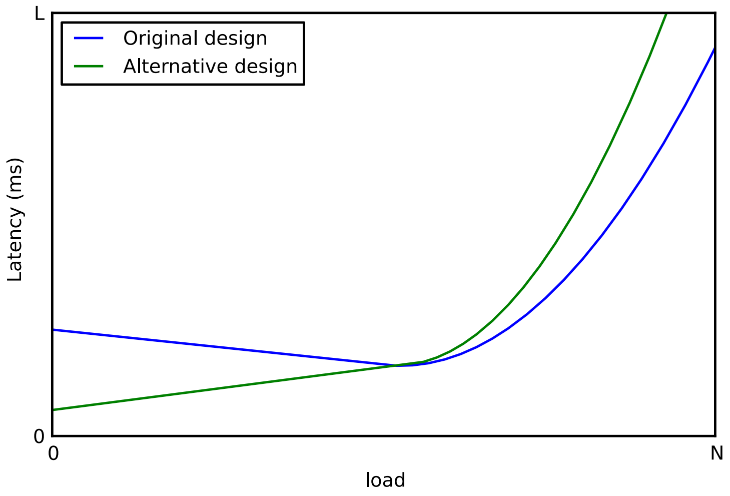


Figure 3.5: A theoretical comparison showing the "hockey stick" shape

the L1 cache at all times. A shared L2 cache is advantageous for this design. Since all queries operate on the same data range, attributes used to calculate the rank of a document may be reused by another query. The L2 cache can fit the attributes of 8738 documents, which gives a probability of $\frac{8738}{2000000} = 0.0044$ of finding the attributes of a document in the L2 cache. All these rates are fairly low, which indicates that the cache hit rate is not very high.

If one considers the alternative design, each query is assumed to be evaluated in parallel using four fetchers, each running on a separate core. The documents are assumed to be uniformly distributed among the partitions. Each partition covers 500000 documents. The probability of finding a document in the L1 cache is now $\frac{273}{500000} = 0.00055$, which is 4 times the probability of the original design. Unfortunately, the shared L2 cache is not used to its full potential in this design. The cache is assumed to be split into 4 pieces, one for each fetcher. Thus, the attributes from 2184 documents may reside in the cache, giving the same probability of finding the attributes as in the original design.

Thus, the alternative design should perform better on architectures when the caches are not shared between processor cores. Newer processor architectures, such as the Tilera TileGX [TIL10], uses different cache coherence schemes, and sharing as little data as possible may be even more advantageous in the future.

3.4.2 Latency effects

Another result of the new design may be lower latencies during low load. The original design of using one large partition could let processors stay idle when there is not enough queries to handle during low load periods. Figure 3.5 shows the typical graph of the search latency when increasing the system load [VES10], showing a behavior typically observed by performance analysts at YTN in the original search core design. The latency improves a little as the load increases, because the cache hit rates increase. However, when the system is saturated, the latency increases exponentially in terms of the load. In terms of clients, the latency will increase linearly with the query queue length.

A plot of the hypothetical behavior of the alternate design is also shown. Using more than one partition to perform parts of the query evaluation in parallel should decrease search latency for each query during low load. During high load, the query latency should go up to the same level as if there were only one partition, but would have a lower break point due to dispatch and merge overhead. The overhead of dispatching and merging query results for each partition may lower the total system throughput. This design should get an increased hit rate in the processor caches, which can make up for some of the dispatch and merge overhead.

3.4.3 Parallel query evaluation in the Vespa search core

Figure 3.6 shows the sequence of operations in the alternative design, if implemented in the Vespa search core. The process of building the query tree is the same, as it constitutes only a small fraction of the time. The query tree is evaluated in parallel (Internally, specialized iterators are used, that takes the doc id range you want to traverse as arguments). A query handle containing a reference to the query tree is put on a set of *partition queues* (one for each partition).

Each partition queue is handled by a set of fetchers. Each fetcher listens on its partition queue for incoming queries. Having dequeued the query tree, each fetcher will traverse it, and rank all documents matched within that particular partition. When a fetcher is finished, it notifies the query processor (which runs within the context of the query handler). The query processor will immediately start to merge the results with the results from other fetchers.

Separating the time spent on query execution separate steps helps to better anticipate the impact of parallel query execution:

$$T_f = T_d + (\max_{0<i<k} T_i) + T_m$$

T_d is the dispatch overhead (the time spent on putting the query on all the partition queues). T_m is the merge overhead (the time spent merging the results from all partitions). T_i is the time spent on evaluating the query for partition i, and k is

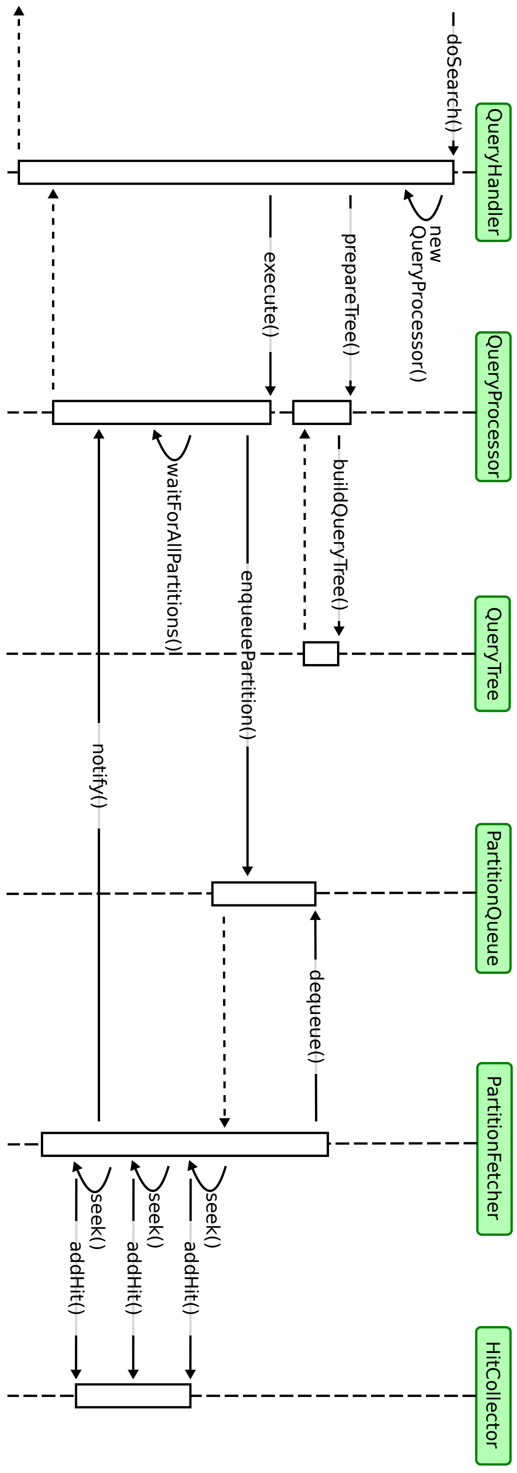


Figure 3.6: A sequence diagram of query evaluation in the alternative design

the number of partitions. For any query, the critical path goes through the slowest fetcher. The fraction of query evaluation performed in serial is then $\frac{T_d+T_m}{T_f}$.

3.5 Using Amdahl's and Gustafson's law

Both Amdahl's and Gustafson's law can be used to evaluate the performance of search core design. First, however, one must define a problem size.

The problem size can be defined as the *number of hits per query*, which is usually a fraction of the *number of documents*. When the number of hits double, the amount of work per partition doubles. If the number of partitions increases with the same factors as the number of hits, the time spent evaluating the query should be the same given enough processors. This definition of the problem size applies to both designs. The *number of attributes used for ranking* per document is an alternative definition of the problem size. If the number of attributes involved in the rank calculation for each hit doubles, so will the time spent evaluating the query.

Amdahl's law may be used to predict the speedup if the problem size is kept constant when the number of partitions increase. But if the number of hits per query increases with the same factor as the number of partitions, Gustafson's law should be more accurate.

To compare the alternative design with the original design, the speedups of each design should be calculated. But a problem arises when one wants to compare them. Whereas the original design is easily able to use all cores all the time, parallel query evaluation raises an interesting issue when the fetchers complete at different times. If that happens, one or more fetchers may stay idle because the partition queues are empty, waiting for the next query to arrive. To handle this case, the number of query handlers should be increased as well to provide a sufficiently high load for the fetchers. Thus, the designs are not mutually exclusive: the number of query handlers should be tuned together with the number of partitions to find the optimal configuration.

3.6 Implications of parallel query evaluation

Splitting the document range into more than one partition has an important consequence: The most relevant documents may be unevenly distributed across the partitions, which means that one partition could get more hits than the others, causing the query evaluation time to become worse as the fetcher for that partition performs more work than the fetchers for the other partitions. In the real world, Vespa deployments such as Yahoo! News may get an uneven distribution in its index, as newer articles added to the index may not be evenly distributed. And for a news site, popular topics such as recent events are more likely to be search for than 30 year old news. This means in turn that some partitions will experience a higher load than the others.

To solve this problem, the partitions should be able to *grow* or *shrink* according to their fraction of the total hits. Algorithm 1 describes a simplified version of the

```
$numpart \leftarrow$ number of partitions
$numhits_{numpart} \leftarrow$ number of hits experienced per partition
$numdocs_{numpart} \leftarrow$ number of documents per partition
for $i = 1$ to $numpart$ do
  if $numhits_i < idealhit$ then
    $request_i \leftarrow idealhit - numhits_i$
    for $j = numpart$ to $i$ do
      $available_j \leftarrow \min(numdocs_j, request_i)$
      $numdocs_{j-1} \leftarrow numdocs_{j-1} + available_j$
      $numdocs_j \leftarrow numdocs_j - available_j$
      $j \leftarrow j - 1$
    end for
  end if
  if $numhits_i > idealhits$ then
    $leftover_i \leftarrow numhits_i - idealhit$
    for $j = i$ to $numpart$ do
      $available_j \leftarrow \min(numdocs_j, leftover_i)$
      $numdocs_{j+1} \leftarrow numdocs_{j+1} + available_j$
      $numdocs_j \leftarrow numdocs_j - available_j$
      $j \leftarrow j - 1$
    end for
  end if
end for
```

Algorithm 1: A simple autopartition algorithm for adjusting the number of docs available to a partition using the hit statistics

autopartitioning algorithm where the partition range is adjusted according to the number of hits. An implementation of this algorithm for growing and shrinking the size of partitions is presented in Section 4.5.9.

Chapter 4

Vsim - Vespa search core simulator

The Vespa search core simulator (hereby called Vsim) can ease the task of evaluating an alternative search core design suggested in Chapter 3. A new design can thus be accepted or rejected in a shorter time. By using Vsim, one can easily measure factors such as cache effects and dispatch/merge overhead, because the effects of surrounding applications and parts of the search core are not there to interfere with the measurement tools. In the event that the positive effects described in Chapter 3 do not occur, or unforeseen negative effects do occur, Vsim can indicate what needs to be changed in order for it to work better. In the future, Vsim may be used to experiment with alternative algorithms and data structures.

Section 4.1 describes some libraries used when developing Vsim. Section 4.2 lists the requirements for Vsim, while Section 4.3 discusses the development methodology. Section 4.5 describes the software design of Vsim in detail. Section 4.8 discusses how well Vsim models the search core.

4.1 Considerations regarding language and tools

Vsim is written in C++ in order to mimic the memory footprint of Vespa. A language such as Java or Python may be quicker in terms of development speed, but the familiarity with C++ and the fact that the Vespa search core is written in C++ made it the logical choice. Vsim is designed to run on Linux 64-bit platforms, as this is the target platform for Vespa. Unfortunately, some not yet standardized library calls prevents it to run on *any* POSIX compliant platform. Further, the implementation of *atomic instructions* requires the target processor to support the 64 bit ISA from Intel/AMD. Atomic instructions for other target platforms may be implemented if needed.

4.1.1 Boost

Boost is a collection of *peer-reviewed* C++ libraries which are not yet part of the official C++ standard. However, a large number of boost libraries are going to be included in the next C++ standard, *C++0X*. Boost can be considered as a test bed for new C++ standard libraries. Boost is used in both commercial and non-commercial software, and is in fact used in some parts of Vespa.

The reason boost is used in some parts of this project is that it provides easy to use and fast implementations of libraries that are not yet standardized in C++. The boost libraries provide more sophisticated and native C++ math libraries, with a greater variety of tuning. Further, some libraries, such as smart pointers, can ease programming tasks such as memory handling. In Vsim, there are three boost libraries in use: `boost::random`, `boost::math` and `boost::smart_ptr`.

`boost::random` provides *pseudo-random* number generators with different properties such as *speed* and degree of "randomness". Each of these generators may be further used together with a distribution mapping, which can be used to generate numbers from a probability distribution such as the *normal distribution*. In Vsim, `boost::random` is used together with the uniform distribution to generate random document identifiers.

`boost::math` provides many convenience classes in fields such as trigonometry, complex number theory and statistics. In Vsim, different *probability distribution* implementations in `boost::math` are used to generate *skewed* workloads.

`boost::smart_ptr` are objects which store pointers to dynamically allocated objects and handles deallocation as soon as the object is no longer referenced. These objects can save the programmer from a lot of work, and bugs, as one does not have to worry about memory deallocation in detail. It is however, as always, important to consider the implications when such data structures are shared between threads, as there is a cost of using atomic operations in the smart pointer implementations.

4.2 Simulator requirements

Vsim is designed with the following requirements in mind:

- Support two configurations:
  - The original threading model (using one thread to evaluate one query).
  - The alternative threading model using parallel query evaluation.

- Be able to timestamp different parts of query evaluation in order to measure where a query spends most of its time.
- Be able to use a QRS log file as input.
- Be able to generate *unbalanced* workload across the partitions using different statistical distributions.
- Store the configuration and the result in a sample file on disk.
- Be able to process the sample file for further processing in scripts.
- Be able to be profiled by OProfile or Valgrind.

Vsim should try to give the same workload as the rank phase of the Vespa search core. Furthermore, supporting parallel query evaluation is the main requirement, and the reason for creating Vsim in the first place. Measuring the time of query evaluation is important both in order to improve the simulator during development and for doing performance analysis after wards. Moreover, QRS logs can provide a more varying and more realistic number of hits per query than a static number. According to Table 2.1 in Chapter 2, the TotalHitCount field in the log entry counts the number of hits found in a query. This field can be used as input to the simulator when generating queries.

As mentioned in Section 3.6, the documents matching a query may not necessarily be uniformly distributed. This means that, once multiple partitions are used, one partition may get more hits compared to another. For testing purposes, Vsim needs to support generating such workloads.

Moreover, post-processing of output is needed in order to analyze Vsim. By storing the results from Vsim in a predefined format on disk, scripts written in other programming languages can be developed to parse and transform the simulator data. OProfile and Valgrind are useful tools (see Section 2.5), and being able to use these with Vsim makes it possible to analyze impact on the cache hierarchy.

4.3 Development methodology

Vsim was developed in an *iterative* fashion, where new ideas and improved implementations appeared after analyzing the results from Vsim, Valgrind and OProfile. Most of the time of this study has been spent designing and implementing Vsim as well as evaluating and testing different implementations.

4.4 User interface

The user interface of Vsim is a simple command line interface. Vsim supports different command line options to control its behavior. All parameters listed below have a default value. Vsim has one non optional parameter, which is a number that tells how long the simulation should run, in seconds. In addition to the command line parameters, several compile time options may be given as parameters to the `Makefile`. To compile Vsim with OProfile support, the `WITH_OPROFILE` knob may be set to "yes". To compile Vsim with the autopartitioning algorithm, the `WITH_AUTOPART` knob should be set to "yes". To enable affinity, the `WITH_AFFINITY` knob should be set to "yes". To use affinity, the number of CPU cores in the system should also be specified by setting the `NUMCPU` knob to the number of processor cores in the system. Following is a description of the run time Vsim command line parameters.

-a <number of attributes> As mentioned in Section 2.2.2, each document has in-memory attributes. This parameters controls the number of attributes to generate for each document. A large number will increase memory usage for each document.

-c <number of connections> This parameter can be used to configure Vsim in a similar way to Vespa. The number of query handlers will be the same as this parameter, thereby controlling the maximum number of clients handled at the same time.

-d <number of documents> This parameter describes the number of documents available in Vsim, i.e. how many documents there should be created attributes for. A large number of documents will increase memory usage, and can be used to create large data volumes, thereby decreasing locality.

-h <number of best hits to collect> If a query matches a large number of document, a huge amount of memory must be used to store all of them. Instead, a finite buffer containing only the best hits is used. This parameter controls the size of this buffer, which again controls the number of hits that should be returned for each query.

-i <probability distribution> One of the requirements is that Vsim should be able to generate unbalanced loads, where some documents are more relevant than others. This parameters takes the name of the probability distribution to be used to calculate which parts of the document range should be most relevant to each query. Valid distributions are *binomial*, *gauss*, *poisson* and *geometric*.

-l <number of slots per partition> This parameter specifies the granularity of the probability distribution specified with the `-i` parameter. Each *slot* is a range of doc ids together with a probability of a query to find matching documents within that slot. A partition initially contains a number of these slots, but they may be moved to another partition during simulation if the autopartitioning mechanism is enabled.

-n <number of hits per query> This parameter controls the number of hits generated for each query. A high number requires more processing power and can be used to increase the problem size. The hits are distributed to the slots according to the probability distribution selected with the **-i** parameter. This parameter is ignored if the **-q** parameter is specified.

-o <filename> This parameter specifies where to save a simulation configuration and the result data. The data is stored in a binary format, and multiple simulation runs may be stored in the same file.

-p <number of partitions> This parameter controls how many partitions to create. A partition may be searched independently of the others, and more than one will evaluate a query in parallel. A high number of partitions increases parallelism for each query, but also increases the overhead of dispatching and merging.

-q <filename> This parameter specifies a QRS log file, which is used to calculate the number of hits for each query. If the end of the QRS log file is reached before the simulator is finished, Vsim continues at the beginning of the file. The file format is a bit simpler than the standard QRS log format: each line contains the number of hits to collect for each query. This reduces the parsing speed.

-r <first attribute-last attribute> Sometimes, only a few of the attributes in a document is used for rank calculation. This parameter describes a range or a set of attributes to be used for ranking. For instance, given that the number of attributes is set to 8, a valid input for this parameter is `-r 3-6`, which will use the attributes 3, 4 and 5 (not inclusive 6) to calculate the rank of each document.

-s <number of rounds> This parameter determines how many times the experiment will be run, using the same configuration parameters for each run. Before each run, all attribute values are seeded with different values, to give each document a different rank.

```
#partitions:            2
#handlers:              1
#hits:                  8000
#clients:               1
heapsize:               1000
numdocs:                100000
num attributes:         8
attributes used:        3 4 5
#slots/part:            1
distribution:           binomial
#queries:               9845
simtime:                10
cputime:                18

Results:
Queries/sec Weighted_Q/s time_queue time_execution time_fetcher
        984          546         10            980          920

time_partitionqueue time_handler %handler
                 20           40      4.1

Per partition stats:
0 H: 39380000 P: 12 F: 853
1 H: 39380000 P: 27 F: 913
```

Listing 4.1: Example output from Vsim

-t <number of clients> This parameter is used to set the number of clients to run. Each client is run within its own thread, and generates queries for the search engine to evaluate.

4.4.1 Vsim output

Listing 4.1 shows an example output from Vsim. The first part of the output describes the configuration of Vsim, as well as the simulation time in seconds and the actual CPU time spent, in seconds.

The second part give a set of post-calculated metrics such as the number of queries executed per second, and the time spent on query evaluation (in microseconds). The output also includes statistics for each partition such as the number of hits received or the average time spent waiting in a partition queue or the time spent in the fetcher.

The output is stored in a binary format, which may be read by scripts or other applications wishing do display the data in another fashion, or do further calculations.

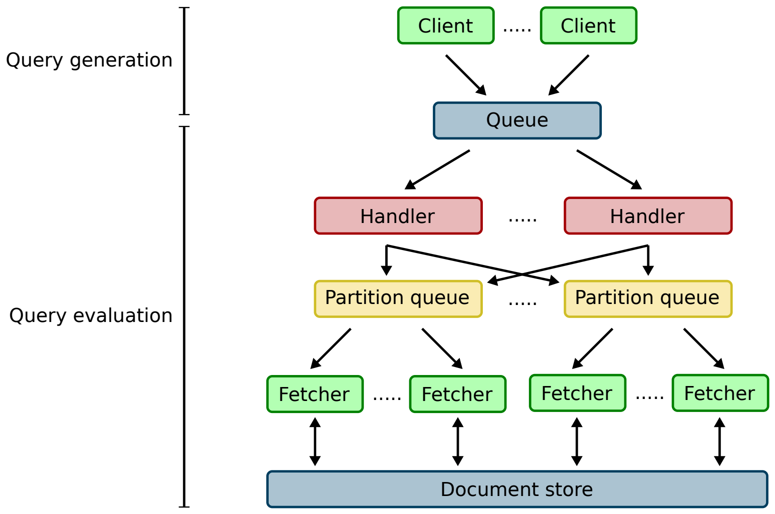


Figure 4.1: An overview of Vsim design

4.5 Software design

This section describes Vsim in detail. As the only relevant part of parallel query evaluation happens within the query processor (as discussed in Section 3.2), Vsim only needs to support the ranking phase, and not the query parsing and matching phase. Each query contains the number of hits, which determines how many doc ids to generate and *rank*. The number of hits may either be specified statically (this way all queries will cause the same amount of hits), or according to a QRS log file. The number of hits are distributed according to a statistical distribution.

The rest of this section gives an overview of the software design, followed by a thorough description of the different components of Vsim.

4.5.1 Vsim overview

Figure 4.1 shows an overview of Vsim program structure. Vsim can be divided into two logical parts, *query generation* and *query evaluation*. The query generation part consists of a number of clients set by the `-t` parameter, that generate queries which are put on a global input queue. The client waits until the query is finished before generating a new.

The query evaluation part consist of evaluating the queries that are put in the input queue. A set of *query handlers* listens on this queue for incoming queries. The number of *query handlers* equals the argument given with the `-c` parameter. The query is dequeued from the queue by one of the query handlers. The query is then put on a set of *partition queues* (one for each partition specified with the

`-p` parameter). Each partition queue has a number of *fetchers* listening to it for incoming queries. The number of fetchers listening to each partition queue equals the number of query handlers.

A fetcher dequeues a query from its designated partition queue, and generates doc ids (the amount of doc ids to generate for each slot within a partition is specified in the query), calculates the *rank* for each of them using the document store, and uses a *hit collector*, to collect the top ranking hits. When a fetcher is finished, it notifies the query handler. The query handler immediately starts to merge the hits from each partition into a *result buffer*. The result buffer will contain the top ranking hits from all partitions when the results from all partitions are merged. When the merging is finished, the client is notified, and the process starts over again with the next query.

4.5.2 Partitions, slots and hit distributions

In Vsim, a partition is a logical segmentation of the doc id range. A partition is further divided into a number of *slots*. A partition may contain one or more slots. This means that a slot may either span the whole partition, or only a fraction of the partition. The number of slots per partition is specified with the `-l` parameter. The number of slots for each partition is fixed at start up, but a slot may be *moved* to another partition if the autopartitioning algorithm chooses to do so.

The partition ranges and their slots are generated at start up. Each partition contains an array of slots, while each slot contains the first and last doc id within its range together with a *probability*. This probability is used to specify how many hits this slot is likely to get of the total hits. A *probability distribution* is used to generate the probability for each slot, which may be any of the standard *binominal*, *geometric*, *poisson* or *gauss*.

4.5.3 General framework code

Some parts of Vsim consists of abstractions on top of operating system interfaces to get advantages of object oriented software design. Moreover, some of these abstractions are similar to those used in Vespa.

Thread abstractions and interfaces

Figure 4.2 shows the classes involved in thread dispatching and execution. The `ThreadPool` class is used to start and dispatch multiple threads using a *thread pool*, without worrying about details of starting and running threads using the POSIX thread library. Typically, only one instance of this class exists. The `Thread` class is a wrapper on top of `POSIX` threads data structures, and *encapsulates* knowledge

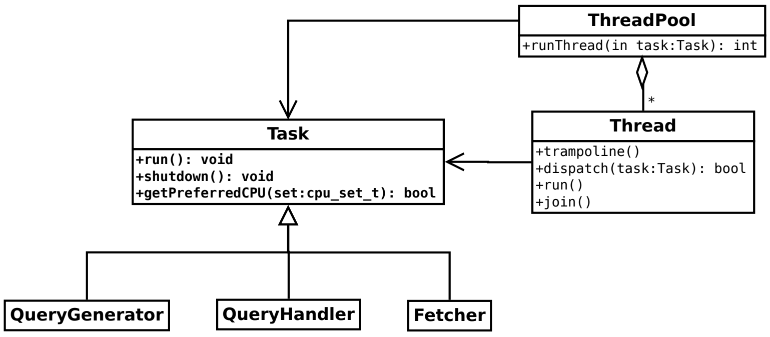


Figure 4.2: The classes involved in thread execution

about these data structures.

A `ThreadPool` contains an array of `Thread` objects. Each `Thread` may be set to run a `Task`, which is similar to how the Vespa search core manages threads.

The `Task` interface specifies the methods `run()` and `shutdown()` which are used to control task execution. An interesting interface method is `getPreferredCPU()`, which a `Task` implementation may use to control affinity. A class wishing to be dispatched and run by the `ThreadPool` class should implement the `Task` interface.

If a `ThreadPool` is asked to dispatch a task, but does not have any available threads to run the task, the caller must check the return value in order to see if the task was actually dispatched or not.

Synchronization primitives

The `Mutex`, `Cond`, `Barrier` and `SpinLock` classes implements abstractions of POSIX synchronization primitives with the same semantics. The most notable difference is the conditional variable, `Cond`, which inherits the `Mutex` class, thereby carrying an implicit lock used for the conditional variable. This is different from the POSIX conditional variable, which is explicitly associated with a lock when used.

4.5.4 Query generators and query data

`QueryGenerator`

A client in Vsim is represented by the `QueryGenerator` class. A `QueryGenerator` implements the `Task` interface, and runs within its own thread. Moreover, there are two subclasses of `QueryGenerator`. The `StaticQueryGenerator` class statically specifies how many hits a query is supposed to give, specified with the `-n` parameter. The `QRSLogQueryGenerator` uses the hit counts in a QRS log file to specify the

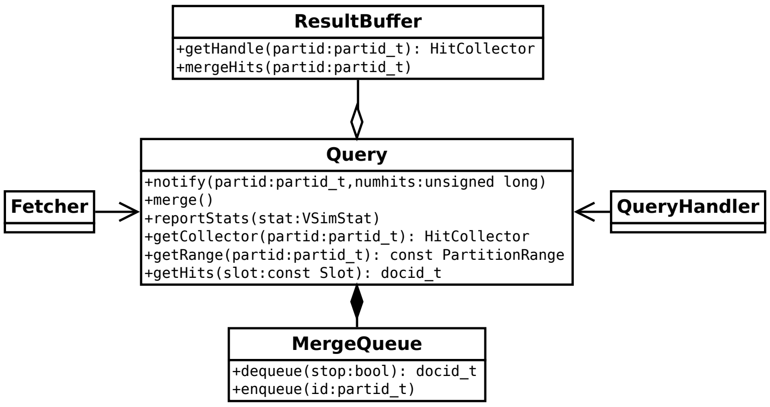

ResultBuffer
+getHandle(partid:partid_t): HitCollector
+mergeHits(partid:partid_t)
Query
+notify(partid:partid_t,numhits:unsigned long)
+merge()
+reportStats(stat:VSimStat)
+getCollector(partid:partid_t): HitCollector
+getRange(partid:partid_t): const PartitionRange
+getHits(slot:const Slot): docid_t
Fetcher
QueryHandler
MergeQueue
+dequeue(stop:bool): docid_t
+enqueue(id:partid_t)

Figure 4.3: Classes dealing with a query

number of hits for each query.

All query generators use the same query queue as input, and enqueues the queries they generate in this queue. The query generator sleeps until it is notified of a query's completion.

`Query`

Figure 4.3 shows the `Query` class and related classes. The query contains all information related to the *evaluation* of a query, such as the number of hits that should be generated for each slot, and a *result buffer*.

After the query is evaluated, the result buffer contains all the highest ranked hits. A merge queue is used to keep track of fetchers that are finished and have their results ready for merging, and in which order they may be merged.

A query also contains various time stamps gathered during evaluation, and reports these to the simulator statistics module on completion. For each query, the following is recorded:

T_q - The time spent waiting in the query queue.

T_{pi} - The time spent waiting in partition queue i

T_{fi} - The time spent on ranking in fetcher i

T_{exec} - The time spent on query evaluation from when its dequeued from the query queue until completion

From these numbers, one can easily get the time spent within the query handler that is not overlapping with the fetchers:

$$T_{qh} = T_{exec} - (\max_{0<i<\#partitions} T_{pi} + T_{fi})$$

It is important to use the partition queue and the fetcher with the largest wait and evaluation time combined, because the total evaluation time of a query depends on the slowest fetcher. The serial fraction of time spent in the handler can then be specified:

$$F_{qh} = \frac{T_{qh}}{T_{exec}}$$

This fraction is the serial fraction of query evaluation, which enables further reasoning on the potential speedup of the design given a set of parameters. It may also be used to compare the actual speedup to the predicted speedups of Amdahl's and Gustafson's law.

4.5.5 Document storage and ranking

Documents in Vsim are only contained in memory, which corresponds to attributes in Vespa. A simple abstraction is used to contain attributes and to provide operations such calculating the rank of a document.

`VSimStore`

The `VSimStore` class manages the document store. The number of doc ids in the store is determined by the `-d` parameter. The attributes are stored as an array for each type, where each entry contains a random integer value determined when running the `seed()` method.

The store provides another method, `getRank()`, to calculate the rank of a document. The rank of a document is determined by adding the values of the attributes of a particular document. The attributes used in this calculation is specified with the `-r` parameter. Although this is a very naive way of calculating the rank, is is run more than one time (the number of rounds is specified with a compile time constant) so that the ranking phase burns more CPU cycles. This way, a workload similar to that of the Vespa search core can be achieved. The Vespa search core performs this calculation in a more complex fashion because it supports features such as using variable length strings as attributes. In the final version of Vsim, the ranking is run 20 times, as it gives a realistic query evaluation time on the target architecture. This also enables QRS log files to be used without pre processing, as the hit counts does not have to be adjusted to get enough work to do for the fetchers.

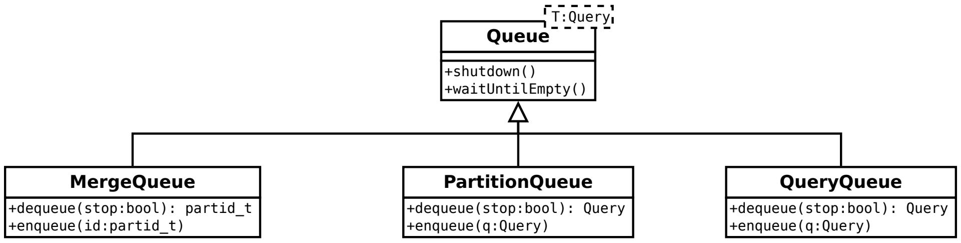


Figure 4.4: The `Queue` template interface and the classes implementing it

`VSimSeeder`

The `VSimSeeder` implements the aforementioned `Task` interface, and is in charge of generating random attribute values between each simulation run. Multiple seeder threads can be run, each pinned to its own CPU core. The number of seeder threads to run is set to the number of processors by default, in order to make sure that the operating system allocates memory at different physical locations in the system. Storing attributes in this fashion ensures that one gets the memory layout as in Figure 2.4.

4.5.6 Queue generalizations

There are three types of queues: query queues, partition queues and merge queues. All of these use the same template class as base, `Queue`, which implements methods for interrupting any listeners and for waiting until the queue is empty. Figure 4.4 shows how the classes are related.

`QueryQueue`

Only one instance of the query queue exists, as it acts as a global input queue used by all clients. The queue implements methods for enqueueing and dequeueing queries. The length of this queue is set to the number of query handlers. If the queue is full, the caller is blocked until an element is removed.

`PartitionQueue`

The `PartitionQueue` class is almost identical to the query queue, but it does not have a maximum length. One partition queue is created for each partition, and it may be used by one or more fetchers.

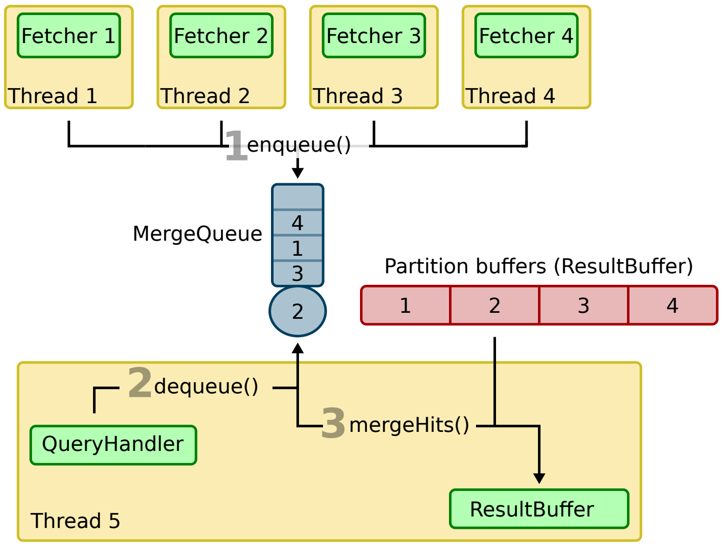


Figure 4.5: Pipelined merge. Fetcher 4 copies its results to partition buffer 4, and inserts its id into the queue. The query handler picks id 2 from the queue, and merges the results from partition buffer 2 (contained within the hit collectors inside the `ResultBuffer` class) into the result buffer

`MergeQueue`

The merge queue object is used to serialize the merging of the query results. Figure 4.5 describes the merge process logically. Each fetcher enqueues its *partition number* and wakes up any objects listing on the queue (which in this case, is the query handler). As soon as the query handler is woken up, it dequeues the partition id. The partition id is used as an index into an array containing the hit collectors for all partitions. When the appropriate hit collector is picked, the internal buffer of that hit collector is merged into the result buffer. When the results from all partitions have been processed, the merging is finished.

4.5.7 Vsim threading model

After a query has been generated and put on the input queue, the query is processed by several other classes in vsim. Each of these classes run within their own threads, and uses queues to communicate.

`QueryHandler`

The `QueryHandler` class controls the whole query evaluation process. The query handler implements the interfaceTask interface, and runs in its own thread. The query handler contains references to all of the partition queues in the system, on which it places a query as soon as it is dequeued from the query queue. Having enqueued the query in the partition queues, the query handler starts to merge any available results. Since the results are not available until the fetchers have evaluated the query, the query handler sleeps until it is notified.

`Fetcher`

The `Fetcher` class implements the `Task` interface, and performs the query evaluation within a partition. Having dequeued a query from its designated partition queue, the fetcher iterates through all slots within a partition. For each slot, the amount of hits encountered is specified within the query. The fetcher then generates that many doc ids. For each doc id generated, the rank of that document is retrieved from the document store. The doc id and its rank is then given to the hit collector, which decides if the hit is relevant enough.

When the fetcher has iterated over all slots, the contents of the hit collector is sorted. Then, the query handler is notified, so that the results may be merged into the result buffer.

4.5.8 Collecting hits and storing the result

Figure 4.6 shows the classes involved when a query is evaluated within the fetcher. The hit collector is used to store the best ranking hits for one partition, while the result buffer is used to store the overall best ranked hits. The hit vector is used as the container for hits.

`HitCollector`

The `HitCollector` class is responsible for collecting the best ranked hits. The hits are stored in a buffer of which size is specified with the `-h` parameter. The hit collector provides methods for adding hits to the internal buffer.

Figure 4.7 shows how the implementation works. In *a*, new hits are added to the buffer until its full. In *b*, when the internal buffer is full, it is transformed into a *min-heap* in order to quickly throw away low ranking hits while having a low memory usage. In *c*, for each hit encountered after the buffer is full, the rank of that hit is compared to the lowest ranking hit in the heap. If the hit rank is higher, it replaces the hit and is *floated* down the heap to maintain the *heap property*. In

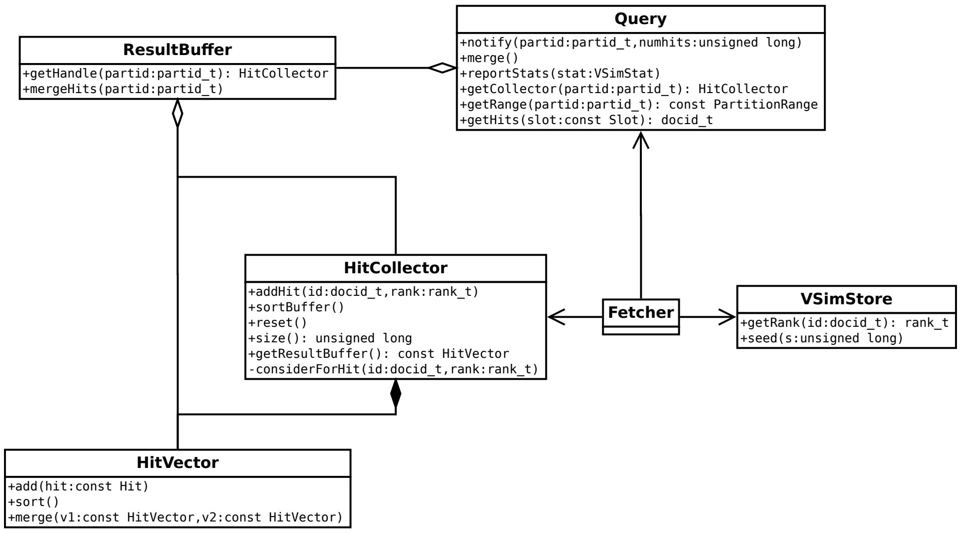

ResultBuffer
+getHandle(partid:partid_t): HitCollector
+mergeHits(partid:partid_t)
Query
+notify(partid:partid_t,numhits:unsigned long)
+merge()
+reportStats(stat:VSimStat)
+getCollector(partid:partid_t): HitCollector
+getRange(partid:partid_t): const PartitionRange
+getHits(slot:const Slot): docid_t
HitCollector
+addHit(id:docid_t,rank:rank_t)
+sortBuffer()
+reset()
+size(): unsigned long
+getResultBuffer(): const HitVector
-considerForHit(id:docid_t,rank:rank_t)
Fetcher
VSimStore
+getRank(id:docid_t): rank_t
+seed(s:unsigned long)
HitVector
+add(hit:const Hit)
+sort()
+merge(v1:const HitVector,v2:const HitVector)

Figure 4.6: Classes involved in query evaluation within the fetcher

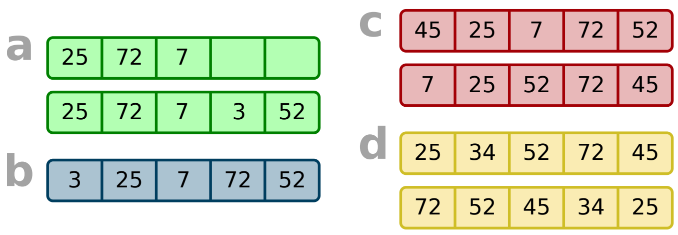


Figure 4.7: The `HitCollector` algorithm implementation used in Vsim

d, the hit collection is finished, and the results are sorted.

`HitVector`

The `HitVector` class is used to represent the internal buffer in the hit collector and the result buffer. The class is implemented as a wrapper on top of a pre-allocated array. It keeps an internal pointer to the first unused element in the internal array, where new hits are inserted. It also provides a method to reset this pointer without having to deallocate any elements in the array. This makes the hit vector reusable without any overhead.

`ResultBuffer`

The ResultBuffer class contains a buffer with the most relevant hits after a query has been evaluated by all fetchers. The size of this buffer is determined by the `-h` parameter. The result buffer implements a method for merging a hit vector into one of the result buffer. This method is used by the query handler when merging the results from each partition. The merging is performed using two buffers of the same size. For each merge, one of the buffers together with the internal buffer of a hit collector is used as operands to the merge algorithm. The result is stored in the second buffer. The second buffer is then used as an operand during the next merge. After merging the results from all partitions, the final result is contained within one of these buffers (marked with a pointer).

The `ResultBuffer` also contains the hit collectors that are used by each fetcher. Before the fetcher starts to iterate through the slots in its partition, it retrieves a handle to one of these hit collectors via the query.

4.5.9 Autopartitioning

The `VSimStat` class provides adjustment of partition boundaries with the `updatePartitionScheme()` method. This kind of runtime *autopartitioning* can be used when experiencing skewed distribution of the doc ids, as explained in Section 3.6. This feature may be necessary if the Vespa search core is to support parallel query evaluation, because an unbalanced load will decrease the effectiveness of this scheme without it. In Vsim, the autopartitioning algorithm operates on slots as the basic unit, as it is otherwise hard to generate a workload and measure the results appropriately. Figure 4.8 shows how the autopartitioning algorithm is used in Vsim with a few modifications. In *a*, the first partition is evaluated. Clearly, the number of hits experienced by the first partition is much larger than that of the other partitions. As the ideal ratio of $\frac{\#hits}{\#totalhits} = \frac{4}{16}$, and the first partition got $\frac{7}{16}$ of the hits, $\frac{3}{16}$ of the slots are removed and pushed back, which will expand the last partition with the same number of slots.

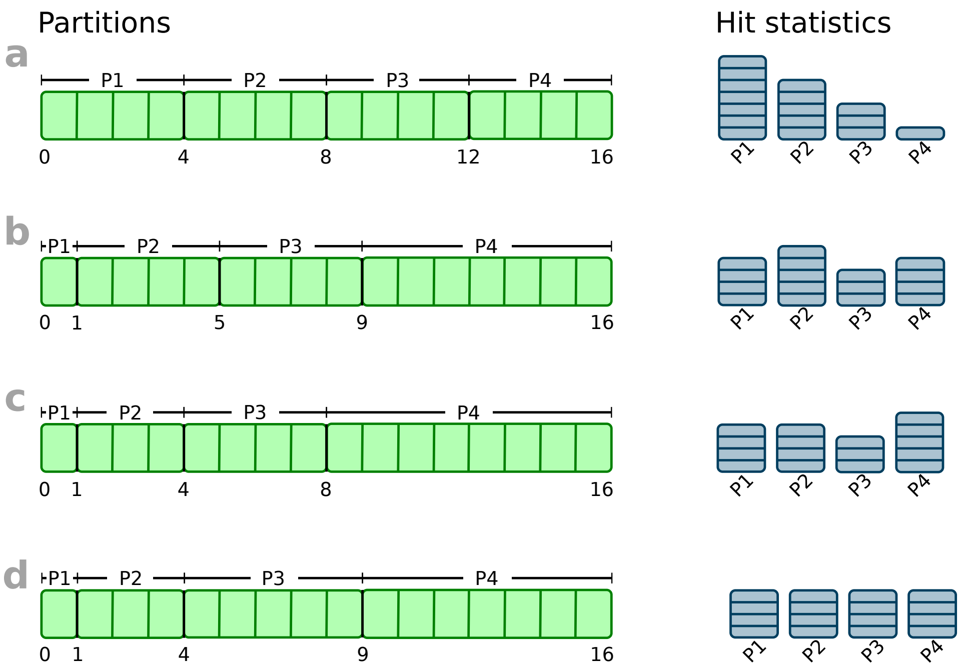


Figure 4.8: Autopartitioning of a system with four partitions and four slots per partition initially. The left side shows the partitions and their range. The right side show the distribution of hits as experienced by the simulator itself.

In *b*, the second partition has $\frac{1}{16}$ slots too many, and one of them will be removed and pushed through to the back.

In *c*, the third partition has only gets $\frac{3}{16}$ of the hits, which means that it should get an additional slot. The slot is taken from the last partition, and the resulting partition range after running the algorithm is shown in *d*.

4.5.10 Simulation setup and configuration management

The initialization part of Vsim parses all arguments provided by the user into a `VSimConfig` object, which contains all configuration parameters in Vsim.

Next, all initial data structures such as the `QueryQueue`, `QueryGenerators`, `ThreadPool`, `VSimStore`, `PartitionQueues` and all `QueryHandler` and `Fetcher` objects are allocated and initialized. After all threads are started, the main thread sleeps until the simulation is finished and tells all other threads to shut down.

4.5.11 Collecting statistics and generating a simulation report

The `VSimStat` class is used to collect interesting numbers regarding a query, and generates various statistics which are stored in the `VSimReport` class. The `VSimReport` class is able to write and read a sample file which includes configuration data for the simulation as well as the results.

4.6 Alternative designs and ideas

During development of Vsim, many ideas were tried out. Some of them were successful and exists in the current version, and some of them failed and were dropped. This section describes some of these alternative design choices.

4.6.1 Parallel merge

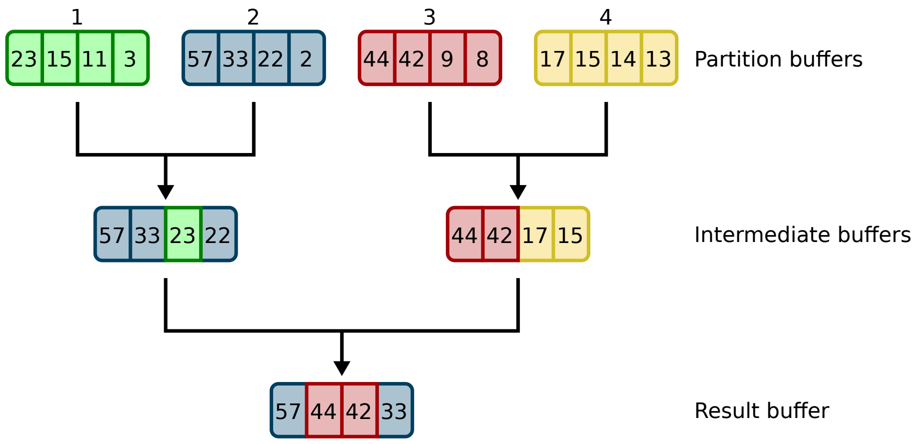


Figure 4.9: Parallel merge: each partition is merged into a temporary buffer, which is again merged with other temporary buffer until only one buffer remains

Merging the results in pipeline is a simple and effective way of merging the results. However, an even more effective scheme is a *parallel merge* scheme. Figure 4.9 shows how four partitions can be merged.

Unfortunately, implementing parallel merge in Vsim complicated the design, and since the overhead of the `QueryHandler` using 8 cores was only around 2%, very little gain would be achieved, and the cost would be increased complexity.

4.6.2 Experimenting with alternative `HitCollector` implementations

As mentioned in Section 4.5.8, a min-heap is used to efficiently store and compare the best hits with new hits. However, a heap is not necessarily the best choice when the results should be sorted on finish. Different algorithms for collecting hits have been suggested and implemented during the course of this study. These algorithms are not directly related to the study and not described here. A full description of the algorithms can be found in Appendix A.1.1.

4.7 Initial flaws

Initially, Vsim contained a few flaws based on unrealistic assumptions or bad performance. Appendix A.2 describes these flaws for future reference.

4.8 Accuracy of Vsim model

Vsim is not guaranteed to behave exactly like the Vespa search core. One way to verify that Vsim is correct would be to compare it directly against Vespa. But, since one of the reasons for writing Vsim is that it is hard to measure the performance of the Vespa search core in detail, it is also hard to compare Vsim and Vespa. However, during Vsim development, discussions with people at Yahoo! with experience with development of the Vespa search core helps ensure that the design is somewhat similar. It is more important that Vsim is able to compare the two threading model designs, than Vsim being identical to Vespa, and it is more important that Vsim provides a somewhat correct workload of these threading models, than running the exact same algorithms as the Vespa search core. Although in some cases, approximating the algorithms may be necessary to get the same workload, such as the data structures and sorting used in the hit collector.

Chapter 5

Evaluation methodology

A set of metrics and tools as well as knowledge about the target platform is required in order to evaluate Vsim. Section 5.1 describes the target platform for the experiments, while Section 5.2 explains which performance counters to use for measuring the effects on the cache hierarchy. Section 5.4 describes the metrics used to evaluate Vsim.

5.1 Target platform

The target platform for the experiments uses state of the art hardware for Vespa search nodes. Table 5.1 shows the hardware components of such a node. The primary interest in this study is the interaction between the CPU and the memory hierarchy.

Table 5.1: Hardware components of the target platform

Component	**Description**
CPU	2 x Xeon E5530 2.40GHz 5860MHz FSB (HT missing, 8 cores, 8/16 threads) - Gainestown D0, 64-bit, quad-core, 45nm, L3: 8MB
RAM	47.3 GB
Hard drive	RAID-0 == 2 x 300GB 10K SAS/6 2.5" Seagate Savvio 10K.3 16MB
Operating system	Red Hat Enterprise Linux 4 U8, Linux 2.6.30.9 x86_64, 64-bit

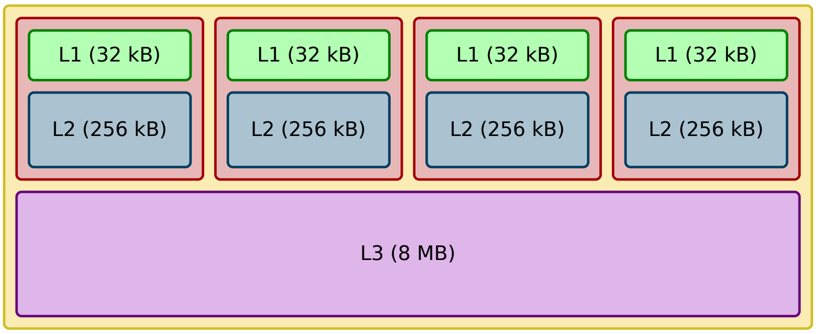


Figure 5.1: An overview of the Xeon E5530 cache hierarchy

5.1.1 Intel Xeon E5530

The CPUs used for benchmarking are based on the Nehalem architecture [neh10], more specifically the Xeon E5530 model. It is important to understand the cache hierarchy in order to decide what experiments to perform as well as for evaluating the results. Figure 5.1 shows the E5530 cache hierarchy. Each core contains a 64kB cache split into a 32kB data cache (shown in figure) and a 32kB instruction cache. Each core has its own L2 cache of 256kB, containing both data and instructions. The last level cache (L3 8MB) is shared by all four cores. The L3 cache is an *inclusive* cache, meaning that all entries in each cores L2 cache is contained in the L3 cache as well. This reduces traffic between the L2 caches, as each core only needs to look in the L3 cache to know if it is contained within other core's caches or not.

The CPU also supports *hyperthreading*, making it possible to run two threads of execution of the same core. It is disabled for these experiments, as it makes it harder to evaluate the performance when two threads are using the same cache. By passing the `maxcpus=8` parameter to the Linux kernel at boot up, the hyperthreading capability of each core is disabled.

5.2 Performance counters

Although Vsim is able to output statistics related to its own execution, the effects on processor architecture are measured from outside. Performance counters are used to get insight into cache behavior. The counters used for the target architecture, Nehalem, are shown in Table 5.2 using a simplified naming scheme. For the real mapping of these simple names to the real counter names, see Appendix B.1.

Each performance counter is used in combination with an argument mask to specify which data it should collect. The counters alone do not give any interesting numbers, but they can be combined to find the cache miss ratio at each cache level.

Table 5.2: Performance counters used

Name	**Description**
L1 MISSES	The number of L1 cache read misses
L1 LOADS	The number of L1 cache reads
L2 MISSES	The number of loads that cache
L2 LOADS	The number of L2 cache references
L3 MISSES	The number of loads that missed the L3 cache
CYCLES	The number of cycles when CPU is not idle. Used for tuning

Table 5.3: Cache parameters of the Nehalem architecture

Level	**Capacity**	**Access latency (cycles)**	**Access latency Xeon E5530 (nanoseconds)**
L1 Data	32kB	4	1.7
L2	256kB	10	4.2
L3	8MB	35-40+	14.6 - 16.7+

The miss ratio of a cache can be found by dividing the number of misses by the number of loads for that cache level. For instance, the following formula gives the L1 miss ratio: $\frac{\text{L1 MISSES}}{\text{L1 LOADS}}$.

For the hit and miss ratios on the different cache levels to be of any use, the parameters of the particular test architecture must be fetched. Table 5.3 shows the cache parameters for the Nehalem architecture [Int09]. The L2 latencies seen from software can vary depending on access pattern and other factors. The L3 latencies depends on the frequency of the core requesting the cache line compared to that of the *uncore*[1]. For the Vsim target architecture, the latency, L, can be calculated using the number of cycles, C, to access the different cache levels, and using the clock frequency, F:

$$L = \frac{C}{F}$$

which is used to calculate the access latencies in nanoseconds. These numbers are used in Chapter 7 to compare the different search core designs.

To not blindly rely on numbers from Intel, cache latency numbers from outside Intel are used to validate them. Molka et. al. from TU-Dresden analyze the

[1]*uncore* referes to everything that is not *on* core

Table 5.4: Cache read latencies from [MHSM09]

Source	**Exclusive cache lines**			**Modified cache lines**			**Shared cache lines**			**RAM**
	L1	L2	L3	L1	L2	L3	L1	L2	L3	
Local	1.3 (4)	3.4 (10)	13.0 (38)	1.3 (4)	3.4 (10)	13.0 (38)	1.3 (4)	3.4 (10)	13.0 (38)	65.1
Core1 (on die)	22.2 (65)			28.3 (83)	25.5 (75)		13.0 (38)			
Core4 (QPI)	64.4 (186)			102 - 109			58.0 (170)			106.0

memory performance and cache coherency effects of the Nehalem microarchitecture [MHSM09], using two Xeon X5570 CPUs. The Xeon X5570 is similar to the E5530 model used in this study. The main difference between these models seems to be the clock frequency (2.933 GHz vs. 2.40 GHz) and QuickPath Interconnect (QPI) bandwidth (6.4 GT/s vs. 5.86 GT/s[2]). The numbers from Molka are shown in Table 5.4. The latencies satisfied by local cache is almost the same as given by Intel. However, the latencies when accessing non-local data are worse, so the worst case numbers are used when calculating the average memory latency. Memory is assumed to have an average latency of 100 ns.

5.3 OProfile

OProfile is used to collect the performance counter values used to get the cache miss ratios. OProfile is controlled by two commands: `opcontrol` and `opreport`. `opcontrol` controls an OProfile *server process*, which interacts with the OProfile *kernel module* and collects the values of the different performance counters. The `opreport` command is used to generate profiles using sampled data and program symbol tables. This can be used to get the time spent in different functions. Listing 5.2 shows the output from `opreport` after profiling the application shown in Listing 5.1. The program itself is meaningless, and is only made to show how OProfile works. The `foo` function does all the work in the test application, while the main function generates very few samples. The amount of samples for each program symbol varies, depending on how long the application is run, as well as how often the functions are used. More interesting for the user, is the relative time spent in each function, shown in a percentage of total samples.

```
int foo(int val)
{
    int ret = 0;
    for (int i = 0; i < val; i++)
        ret += i * val;
    return (ret);
}

void main(void)
{
    int other = 0;
    for (int i = 0; i < 100000; i++) {
        int res = foo(10000);
        other +=  res / (i + 1);
        int a = other * i;
    }
}
```

Listing 5.1: Example application for OProfile

[2]GT/s = Giga Transfers/second

```
CPU: Core 2, speed 1600 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a
    unit mask of 0x00 (Unhalted core cycles) count 100000
samples  %        symbol name
90414    99.9558  foo
40        0.0442  main
```

Listing 5.2: Example OProfile output

5.4 Basic metrics used to evaluate Vsim

The experiments performed in this study are used to see the effects of parallel query evaluation. There are several key metrics that can be used to evaluate the Vsim workload. Some of them are given by Vsim itself, while others are provided via profiling tools.

5.4.1 Throughput

Throughput describes how many elements that is processed per time unit. In Vsim, the throughput can be determined from the *number of queries* processed *per second.* The throughput can also be described as *number of queries* processed *per CPU second*, which remove the time spent while waiting for threads to run.

5.4.2 Latency

The time spent in evaluating the query is useful to see the maximum throughput achievable while keeping the latency low. The throughput and latency are related, but can behave differently when going past the load that the system is designed to handle. The latency includes all time spent from the moment the query is added to the query queue, until it is evaluated.

5.4.3 Speedup and efficiency

The efficiency, E, shows how well the CPU resources are being used, and is dependent on several other variables such as the speedup S and the latency L or the throughput T. The speedup can be calculated in two ways. Either as a function of the throughput:

$$S = \frac{T_n}{T_1}$$

where T_n is the throughput when running with n number of concurrently executing queries (or threads), or as a function of the latency:

$$S = \frac{L_1}{L_n}$$

where L_n is the latency when running with *n partitions* (or threads).

The reason for using two speedup metrics is simple: When increasing the number of concurrently evaluated queries, the latency remains fairly identical while the throughput increases. When increasing the number of partitions, the latency goes down while the throughput goes up. Thus, for the second case, the speedup should be equal when using both formulas.

The efficiency, E, is defined as:

$$E = \frac{S}{n}$$

where n is the number of threads. The efficiency shows how good the speedup is compared to ideal speedup. A value below 1 means that the speedup is not ideal, and that the application fails to fully utilize the processing power. A value above 1 is the consequence of a phenomenon called *super linear* speedup. The reason for super linear speedup is that the baseline configuration is not optimal. One common problem reason for super linear speedup is the cache. Some times, elements in the cache are reused between multiple concurrent threads, and results in each thread running faster than it would if it would run alone on the processor. Grama and Kumar gives further details on scalability of parallel programs [RR07], and proposes several other metrics for evaluating parallel programs.

For the experiments in this report, the speedup is used instead of the efficiency. The reason is that the speedup gives a more intuitive way of comparing the designs to the speedup predicted by Amdahl's and Gustafson's law.

5.4.4 Average memory access time

The average memory access time indicates the expected memory latency over time. To calculate it, the access ratios together with the access times for each level in the memory hierarchy are used. Hennessy and Patterson describes the following formula to calculate the average memory access time [HP07]:

$$\begin{aligned} \text{Average memory access time} = {} & \text{Hit time}_{\text{L1}} + \text{Miss ratio}_{\text{L1}} \\ & \times(\text{Hit time}_{\text{L2}} + \text{Miss ratio}_{\text{L2}} \\ & \times(\text{Hit time}_{\text{L3}} + \text{Miss ratio}_{\text{L3}} \\ & \times \text{Miss penalty}_{\text{L3}})) \end{aligned}$$

All the variables in this equation are provided by the performance counters and values found in Section 5.2.

Chapter 6

Experiments

To decide if parallel query evaluation is worth the effort or not, a few experiments are run on Vsim and analyzed. Section 6.1 lists the default parameters used in the experiments. Section 6.2 presents the scalability of Vsim and parallel query evaluation. Moreover, the theories of potential performance improvements are discussed in Chapter 3 are evaluated in Section 6.3 and 6.4. The *autopartitioning* algorithm presented in Section 4.5.9 is evaluated in Section 6.5, using various probability distributions as input to Vsim. Each section contains a description of experiments, followed by an analysis and a discussion of the results. Section 6.6 concludes the experiments with a general discussion and conclusion.

6.1 Vsim parameters

Not all of the parameters in Vsim are adjusted for the experiments presented in this chapter. Table 6.1 shows a list of default parameters and their abbreviations used when displaying experiment parameters. The parameters are set to be as realistic as possible compared to settings used in a typical Vespa search node. Vsim will use these parameters *unless otherwise specified* in each experiment. Most experiments are run 10 times for 60 seconds each, while the scalability experiments are run 20 times for 60 seconds each. The numbers presented are the average of all runs. In some cases, the standard deviations are shown in the graphs.

6.2 Scalability

Vsim needs to be scalable in terms of the number of clients running on the system. Table 6.2 show the experiments and their parameters. All experiments run with

Table 6.1: Vsim default parameters

Parameter	**Value**	**Abbreviation**
# Attributes	32	-
Max number of connections	1	CONN
# Documents	40,000,000	DOC
Heap size	1000	-
Hit distribution	binomial	DIST
slots per partition	1	SLT
Hits per query	10000	HITS
Number of partitions	1	PART
Attribute indices used to calculate the rank	0, 31, 4, 13, 5, 27, 18, 29	-
Number of clients	the same as the max number of connections	CL

Table 6.2: Vsim parameters and values for evaluating scalability

Experiment	CONN	CL	PART	HITS	DOCS
1	1 - 12	24	1	CONN $\times$ 2000	CONN $\times$ 3000000
2	1	24	1 - 12	PART $\times$ 2000	PART $\times$ 3000000
3	1	24	1 - 10	10000	30000000

the same number of clients in order to give an equal load. The first experiment tests the speedup as a function of the number of connections running at the same time. It is important that Vsim scales with the number of connections in order to be a realistic model of the Vespa search core. The second experiment tests the speedup as a function of the number of partitions. This tests how the alternative threading model using parallel query evaluation scales. Both experiments uses a scaled problem size, by increasing the number of documents as well as the number of hits per query together with the number of connections/number of partitions. These experiments are run with affinity disabled and enabled in order to compare the scheduler impact. In contrast to the first two, the third experiment investigates the scalability when using the same problem size.

6.2.1 Scalability of query handlers

Figure 6.1 shows the speedup as a function of the maximum number of connections from the first experiment with affinity disabled and enabled. The predictions of Gustafson's law are also shown.

Super linear speedup is achieved when affinity is disabled. When the number of connections goes past the number of processor cores, the speedup flattens. The super linear speedup comes from the inefficiencies of the 1 connection case, which

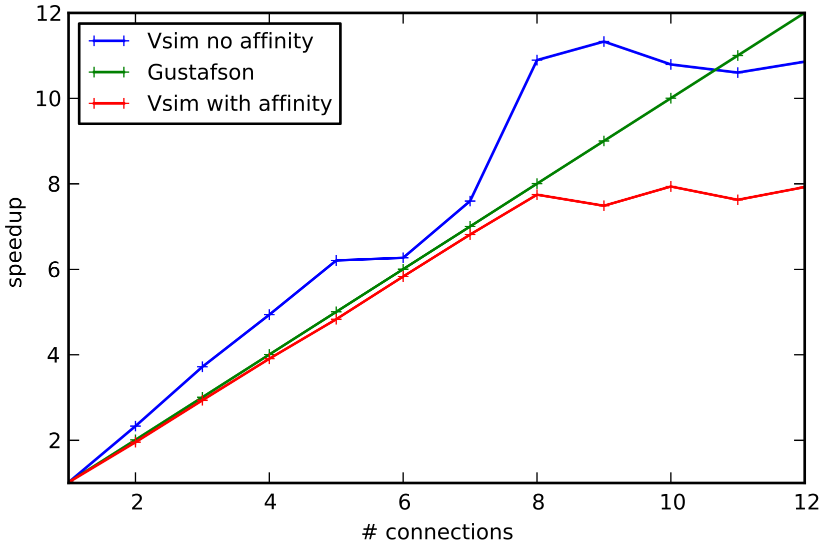


Figure 6.1: Speedup as a function of the number of connections

is used as the baseline when calculating the speedup. Such inefficiencies can be attributed to an increased L3 cache reuse, because multiple threads use the same cache. The inefficiencies can also be caused by bad scheduler decisions, if threads are not optimally scheduled in the baseline configuration.

When the experiment is run with affinity enabled, the fetcher for each connection is pinned to a specific processor core ("connection number" mod "# processor cores"). With affinity, the speedup curve closely follows the Gustafson's curve up to the number of processor cores in the system. The curve flattens out when the number of connections goes past 8. This curve confirms that the super linear speedups can be attributed to bad scheduling decisions in the baseline configuration.

6.2.2 Scalability of parallel query evaluation

The second experiment measures the performance when varying the *number of partitions*. The goal is to evaluate the scalability of parallel query evaluation. Figure 6.2 shows the speedup as a function of the number of partitions with affinity disabled and enabled. The predictions by Gustafsons's law when using the serial fraction for 1 partition as a basis are also shown.

Without affinity, the speedup follows the linear curve up to 5 partitions. At 6 and 7 partitions, there are minor deviations from the linear speedup. At 8 partitions, however, the speedup increases to 10. As fetchers do not share data at all, there

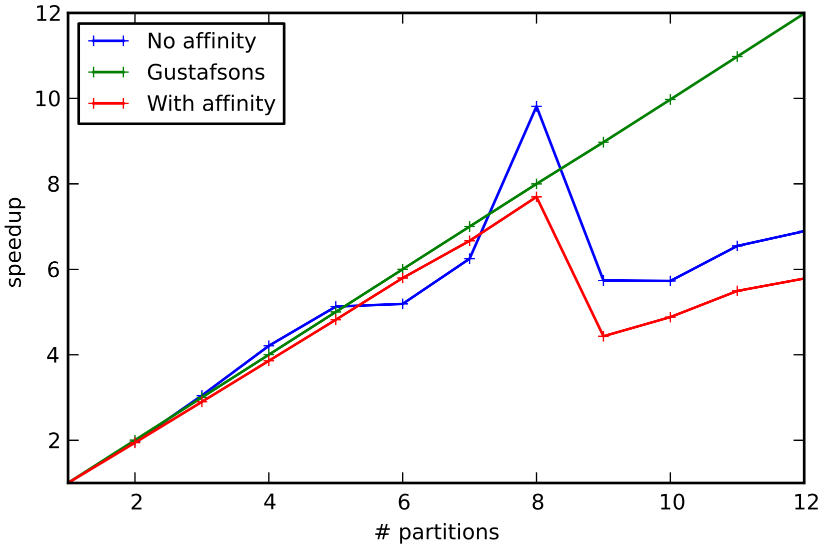


Figure 6.2: Scalability as a function of the number of partitions

should be no additional gain from cache reuse. Therefore, the behavior most likely comes from scheduler inefficiencies when using fewer partitions. The experiment with affinity enabled supports this explanation. When affinity is enabled, the speedup is bounded by the linear curve, as the scheduler takes no part in deciding where a thread should run.

When the number of partitions is larger than the number of cores, the speedup goes down, because $\mathtt{p} - 8$ fetchers will have to wait for other fetchers to finish before running, which could result in a doubling of the worst case query evaluation time. Gustafson's law predicts the speedup somewhat accurately up to 8 cores.

Serial fraction of fixed problem sizes, and Amdahl's law

Results from the third experiment, are shown in Figure 6.3. Each query gives 10000 hits, spread uniformly across all partitions. Since the problem size is fixed, the amount of work performed in parallel remains the same. The serial fraction (spent in the query handler) increases as a function of the number of partitions. Once the number of partitions exceed the number of processor cores, the fraction jumps up to above 30%. The reason is that the time spent on the merge within the query handler not only increases because of more partitions to merge, but also because it has to wait for these partitions to finish before merging.

Figure 6.4 shows the same figure zoomed in to the lower 10% fraction. The time

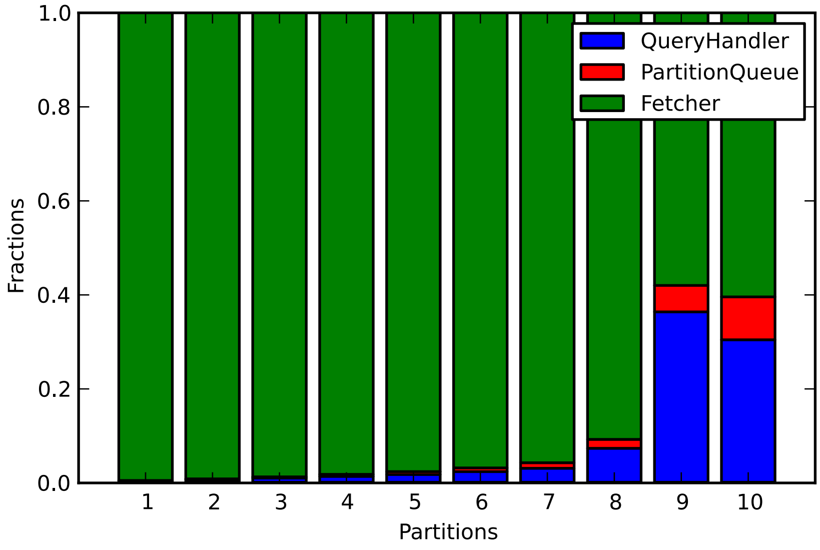


Figure 6.3: Fraction of the total query evaluation time spent in various stages of query evaluation

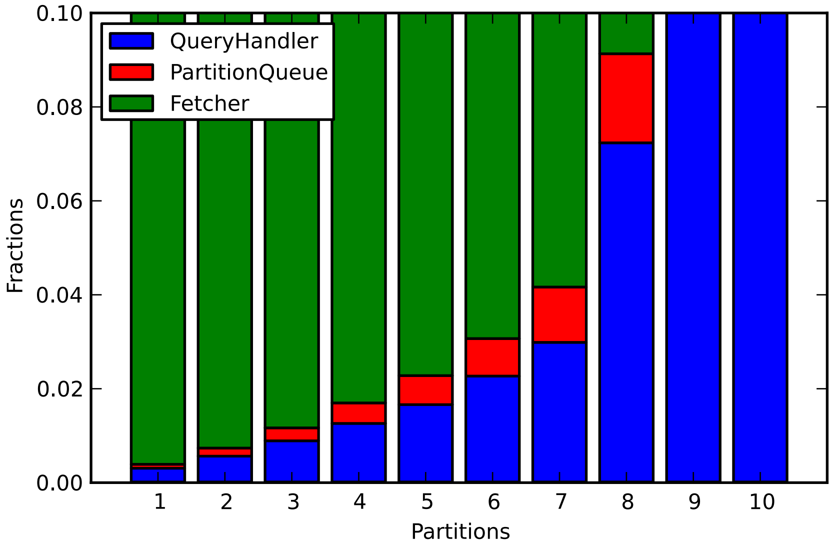


Figure 6.4: Fraction of the total query evaluation time spent in various stages of query evaluation, lower 10%

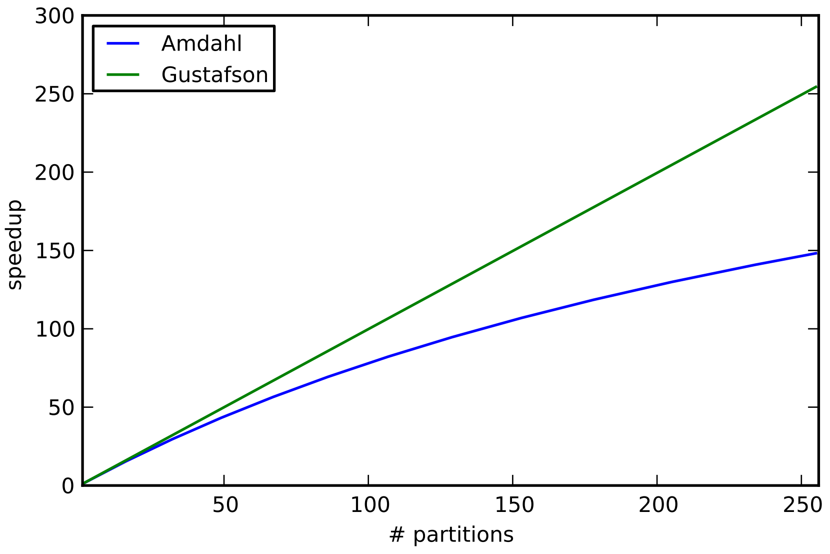


Figure 6.5: Speedup calculated using Amdahl's law

spent in the fetchers constitutes the largest fraction of query evaluation time. As the number of partitions increases, the fractions of the time spent in the partition queues as well as the query handler increases.

Figure 6.5 shows the speedup predicted by Amdahl's law and Gustafson's law as a function of the number of processor cores. The predictions are calculated using the baseline serial fraction of the 1 partition configuration. Amdahl's law does not seem very pessimistic in this case, as the speedup is reasonable even past 64 partitions. The predictions from Gustafson's law follows the linear curve and cannot be seen in the plot, because of the low serial fraction.

In Vsim, the amount of data that needs to be processed in the serial part increases linearly with the number of partitions because of the pipelined merge scheme. To increase scalability, the number of documents returned for each partition can be reduced. If the N hits are required in total, only $\frac{k \times N}{P}$ documents is returned per partition, where P is the number of partitions and k is a factor that can be adjusted to get the desired constraint. If $k = P$, the memory requirements will double when the number of partitions are doubled, which is the current behavior. If $k = 1$, the data size is kept constant even though the number of partitions is increased. The problem with this approach, is that the guarantee of returning the best hits are sacrificed. To improve the scalability without sacrificing precision, one can use parallel merge. Parallel merge would increase the serial fraction with $\log N$ instead of N. As mentioned in Section 4.6.1, the implementation of such a scheme is complex. The parallel merge scheme can be implemented when it

Table 6.3: Vsim parameters and values for checking behavior during low load

Experiment	DOCS	CONN	HITS	PART	CL	QRS log source
1	20000000	16	16000	1 - 12	1 - 20	-
2	20000000	16	-	1 - 12	1 - 20	Yahoo! News Search
3	20000000	16	-	1 - 12	1 - 20	Yahoo! Image Search

becomes necessary.

6.3 Performance of parallel query evaluation

In Chapter 3, a hypothesis was stated saying that using more than one partition would lower the query latencies during lower load periods. Ideally, query latencies should behave similar to the "hockey stick" shown in Figure 3.5 in Chapter 3. Table 6.3 shows the experiments and their parameters. The maximum number of connections is set to 16, while the load is varied by changing the number of clients. The number of partitions is varied between 1 and 12 to get an idea how the latency varies for different configurations. The first experiment uses a static query generator for generating hits. The second and third experiment uses the qrs log file query generator. The second experiment uses QRS log files from Yahoo! News Search, while the third uses QRS log files from Yahoo! Image search. An important observation when evaluating these experiments is to see where the latency cross-over for the different configurations is. This cross-over indicates where one configuration becomes better or worse than the other.

6.3.1 Using the same number of hits per query

Figure 6.6 shows the results from the first experiment. For 1 partition, the latency behaves as expected. As the number of clients increases, the latency decreases a little until the number of clients is equal or greater to the number of processor cores. After that point, the latency goes up linearly with the queue size.

The cross-over between the configuration with one partition and configurations with multiple partitions moves to the right as the number of partitions increases. Moreover, the latencies for a configuration with multiple partitions are lower at the left side of the cross-over. For eight partitions, the crossover happens between 8 and 9 clients, from where the latencies become roughly the same. There is one exception to the cross-over pattern, which happens for the two partition configuration. Possible reasons are how the threads are placed by the scheduler. Since fetchers operate on the same data range, there is no guarantee that the operating

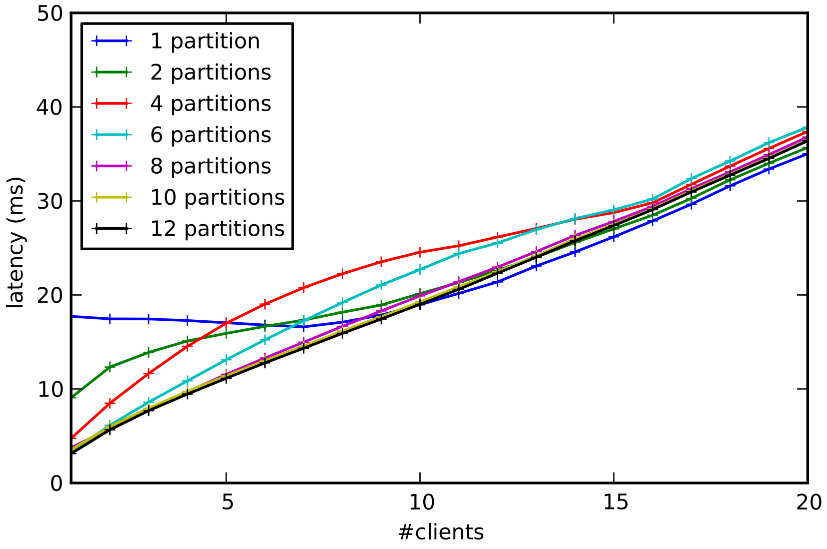


Figure 6.6: Latency as a function of the number of clients

system will put them to run on separate cores, since it takes affinity into account as well. As seen in the scalability experiments, the scheduler can make non-optimal decisions. However, going further into detail of the operating system scheduler workings is outside the scope of this study.

There is another unexpected behavior happening for the 2, 4 and 6 partition configurations, where the latency curve is simply bent. When the number of clients are lower than the number of processor cores, the latencies go up quickly. A possible explanation is that may compete with other fetchers for the same processor cores. As the number of clients exceed the number of cores, the latency becomes the same across all configurations.

For 8 partitions, the curve is close to linear. The latency is low from 1 to 8 partitions, and the cross point happens when stepping up from 8 to 9 clients. The curve then follows the 1 partition curve linearly, but at a higher offset due to the extra overhead. These numbers show that for low loads, the query latency benefits from using more than one partition. Further, the latency does not increase faster for 8 partitions than 1 partition, as they are both bounded by the number of query handlers running on the 8 processor cores. Setting the number of partitions to a higher value than the number of cores does not result in lower, nor higher, search latencies because of the limited number of cores.

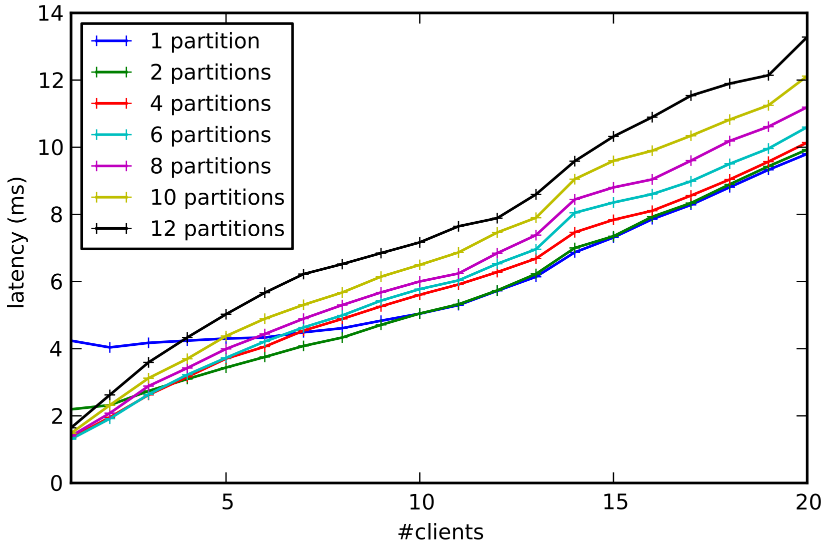


Figure 6.7: Latency as a function of the number of clients using QRS log files from Yahoo! News

6.3.2 Using QRS logs from Yahoo! News

Figure 6.7 shows the results from the second experiment where QRS log files from Yahoo! News are used as input. Characteristics of the input queries are that they generate few hits on average. This is a disadvantage for parallel query evaluation. The pattern is almost opposite to that of the first experiment. If the number of partitions is increased, the latency curve gets steeper. This most likely comes from the overhead of dispatch and merge, which gets to big compared to the savings of performing the ranking in parallel. The only exception, however, is the configuration with 2 partitions, which seems to be the ideal configuration for this workload.

6.3.3 Using QRS logs from Yahoo! Image Search

Figure 6.8 shows the results from same experiment as the previous one when using QRS log files from Yahoo! Image Search as input instead. In contrast to the log files from Yahoo! News, these log files generates a higher average number of hits per query, which becomes an advantage for parallel query evaluation. All configurations supporting more than one partition gives a lower search latency all over. For configurations from 6 to 12 partitions, there is almost no difference in performance.

From both QRS log based experiments, it is clear that the performance of parallel

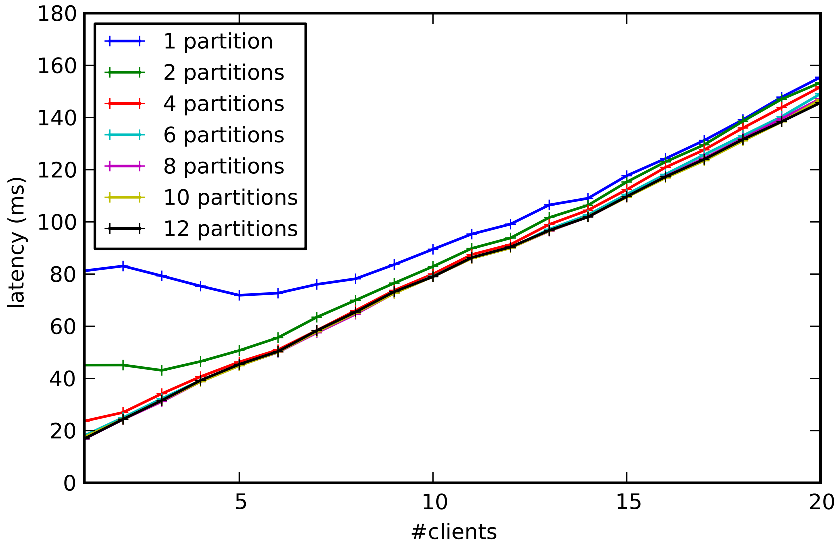


Figure 6.8: Latency as a function of the number of clients using QRS log files from Yahoo! Image Search

query evaluation depends on the workload. When the amount of work per query is low, evaluating each query in parallel quickly becomes a disadvantage as the number of clients increases. It is therefore necessary to make the number of partitions a configuration parameter that can be tuned for different workloads.

6.4 Behavior of the cache hierarchy

Another hypothesis of the new design, is that it should achieve a higher *hit ratio* in the per core caches. To check this, OProfile is used to gather statistics from

Table 6.4: Vsim parameters and values for checking cache behavior

Experiment	CONN	CL	PART	DOCS	HITS	Using thread affinity
1	1	1	1	$2^n, n = 12...25$	DOCS $\times$ 0.002	NO
2	8	8	1	$2^n, n = 12...25$	DOCS $\times$ 0.002	NO
3	1	1	8	$2^n, n = 12...25$	DOCS $\times$ 0.002	NO
4	1	1	8	$2^n, n = 12...25$	DOCS $\times$ 0.002	YES

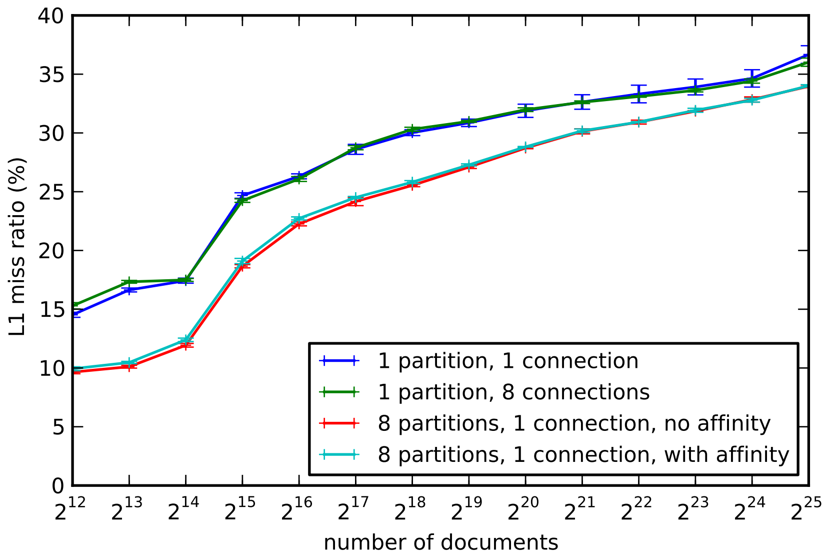


Figure 6.9: L1 cache miss ratio as a function of the number of documents

the *performance counters*. The *average memory access time* metric (discussed in Chapter 5) is used to compare the overall memory performance. Table 6.4 lists the parameters used for these experiments. The first experiment uses only one partition and one connection. The second experiment uses only one partition, but runs 8 connections at the same time. The third experiment evaluates the query in parallel using only one connection at a time in order to reduce interference. The fourth experiment is the same as the third, but with affinity enabled. This means that each fetcher is pinned to a specific core ("fetcher number" modulo # processor cores). For all experiments, the number of documents is increased in power of two's, and the number of hits per query is set to a fixed fraction of the number of documents. The fraction is derived from a realistic number, where an index of 4000000 documents may experience 8000 hits on a query. All graphs are presented with a logarithmic x axis.

6.4.1 L1 cache miss ratio

Figure 6.9 shows the miss ratio in the L1 cache as a function of the number of documents. The graph shows the miss ratios for all four experiments. The predictions were, that by splitting the documents into smaller ranges, the miss ratio would be lower because the threads for each partition would run on a separate processor core. This should in turn reduce cache interference.

As predicted, parallel query evaluation experiences a lower miss ratio in the L1 cache. The miss ratio increases monotonically as the number of documents increases. There is, however a notable bump between 2^{14} and 2^{15} number of documents. To investigate this behavior, consider the parameters of the L1 data cache. The L1 data cache size is at 32kB with a block size of 64 bytes. This means that the L1 cache can hold a maximum of 512 cache lines, or 4096 attributes in the optimal case, without taking associativity into account. Given that the data access is random, and that 8 attributes are read for each document, this means 8 cache lines have to be fetched for each hit in the worst case. The number of hits per query is 33 and 66 at the bump points, which corresponds to 264 and 528 cache lines. Whether or not this is the cause of the bump is hard to tell. According to OProfile, the number of L1 cache references do not increase more than 0.3%, while the number of misses increases by 41 %. Other factors, such as Translation Lookaside Buffer (TLB) misses may be involved. Another factor could be that the number of page tables required may have increased and caused indirections for the operating system when multi level page directories are used. Further investigation into this matter is outside the scope of this thesis, but interesting for future study.

The curves for the 8 partition configurations are similar to the curves for the 1 partition configurations, but at a lower vertical offset. The difference between these configurations decreases a little as the number of documents increases. This is likely due to the low probability of finding an attribute in the L1 cache.

6.4.2 L2 cache miss ratio

Figure 6.10 shows the miss ratios for the L2 cache. The curves have the same form in all cases, but the curves for 8 partition configurations are at a higher offset on the x axis. This offset is in the order of 4 - 8 times the document size. For instance, at 2^{20} documents, the miss ratio for the 1 partition configurations is around 45%. At 2^{23}, the miss ratio for the 8 partition configurations is around 46%. The document range covered per partition at 2^{23} documents for the 8 partition configurations is the same as the range covered at 2^{20} documents in the 1 partition configurations. As with the L1 miss ratios, there appears to be no difference in the miss ratios experienced whether only one or eight query handlers are used, as each core has its own L2 cache.

There is an overall lower miss ratio for 8 partition configurations, for the same reasons as the lower miss ratio in the L1 cache. For 8 partitions, the L2 cache gets a lower miss ratio within the range of the experiment. The the gap between 1 partition and 8 partitions closes for large document sizes. This is only natural, as the cache looses its effectiveness in the same way as the L1 cache when the data volume becomes huge.

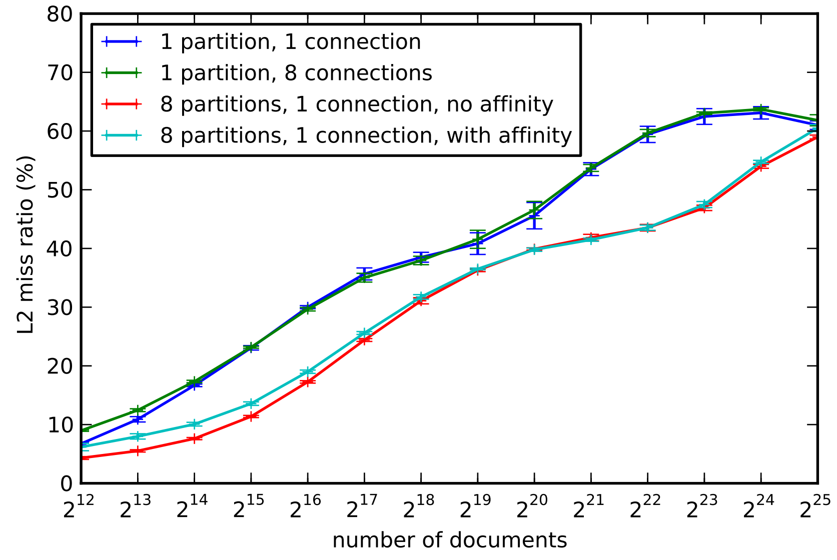


Figure 6.10: L2 cache miss ratio as a function of the number of documents

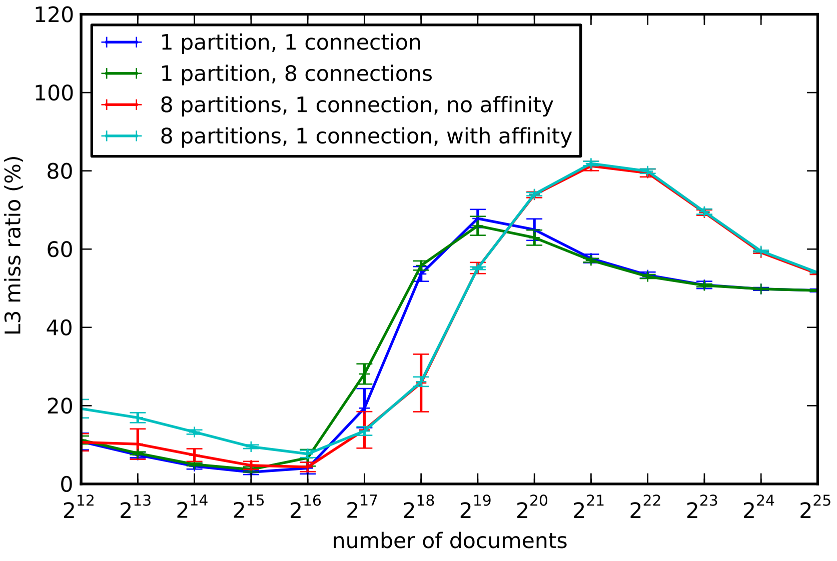


Figure 6.11: L3 cache miss ratio as a function of the number of documents

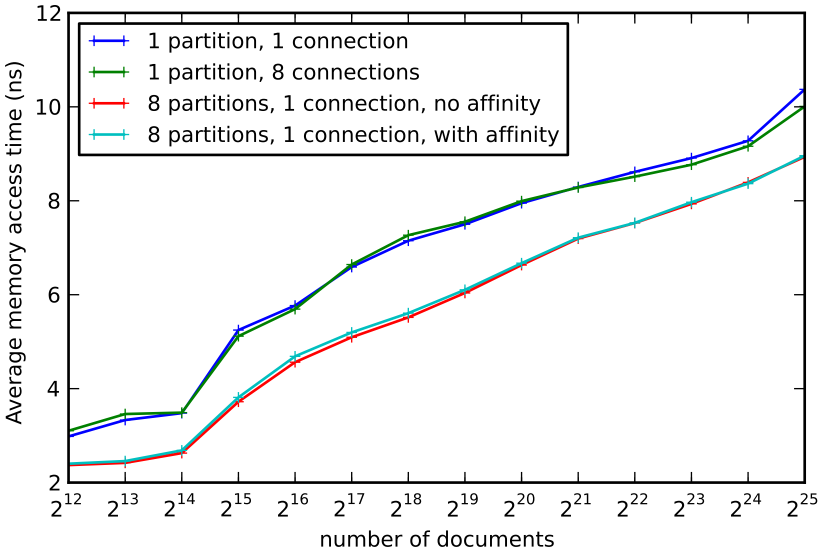


Figure 6.12: Average memory access time as a function of the number of documents

6.4.3 L3 cache miss ratio

Figure 6.11 shows the miss ratios for the L3 cache. The simplest case using only one partition and one query handler performs as expected, as the L3 will act as a regular cache. The miss ratio decreases as a larger portion of the cache gets used. In the range between 2^{16} and 2^{19} documents, the L3 cache can no longer contain all the documents. The L3 cache has space for 8MB of attributes. At 2^{16} documents, 4MB of attribute data can be accessed in total. This means that everything fits in the L3 cache. However, the attribute data size doubles at 2^{17}, and the documents can only fit in the L3 cache in the best case. Moreover, at 2^{18} documents, only half the attributes may fit in the cache. From 2^{19} documents and out, the miss ratio decreases slowly as it reuses entries in the cache.

As with the L2 cache, there is an offset as to when the cache fills up when comparing 8 to 1 partition. For 8 partitions, however, the reuse effect is not as big when using 1 partition. As stated earlier, the L3 cache is shared by four cores. When using 8 partitions, the fetchers are not able to share their data with other fetchers in this cache. This results in a miss ratio of 80% in the worst case. The miss ratio decreases as the number of documents approaches 2^{25} though, indicating cache reuse.

Table 6.5: Vsim parameters and values for evaluating autopartitioning

Experiment	PART	SLT	DIST
1	2, 4, 8	1000	binomial, geometric, gauss
2	2, 4, 8	1, 10, 100, 1000	geometric

6.4.4 Average memory access time

As previously explained, and as argued by Hennessy and Patterson [HP07], the miss ratios at the individual cache levels can be misleading, and the average memory access time should be calculated to get an overall impression of the performance. Figure 6.12 shows the average memory access times for memory requests using the miss ratios shown above. Using 8 partitions gives an overall lower access time than with one partition, even though using one partition with many query handlers should increase L3 cache reuse.

From these experiments, it can be concluded that parallel query evaluation uses the per core caches more efficiently. Parallel query evaluation does not, however, use the shared L3 cache as efficiently as the original design.

6.5 The impact of slots and autopartitioning

The *autopartitioning* algorithm presented in Section 4.5.9 is believed to improve throughput and latency when using a *skewed* hit distribution. Table 6.5 lists the parameters for the experiments related to autopartitioning. The first experiment uses different hit distributions, to see how Vsim performs with and without the autopartitioning algorithm. Figure 6.13 shows how the number of slots per partition is distributed after running the autopartitioning algorithm with the normal distribution as input (a coarse plot of the input distribution is shown in blue). The slots are distributed as an inverse to the input, which is expected. If a partition gets few hits, it will acquire slots from the other partitions. The middle partitions, partition 4 and 5 contains only a small fraction of the slots, because they get a lot of hits.

Figure 6.14 shows how autopartitioning performs in terms of latency, for the binomial and geometric probability distributions when using 2, 4 or 8 partitions. As expected, the latency is almost the same for autopartitioning on and off when using the binomial distribution. The small differences are of no statistical significance. For the geometric distribution, the latency decreases significantly when enabling the autopartitioning algorithm. The largest gain is at 8 partitions, where there is a speedup of over 4 when using autopartitioning. The autopartitioning algorithm is clearly necessary when having a skewed hit distribution. The algorithm can also be tuned by varying the number of slots/slot size, as well as the increments for each iteration of the algorithm.

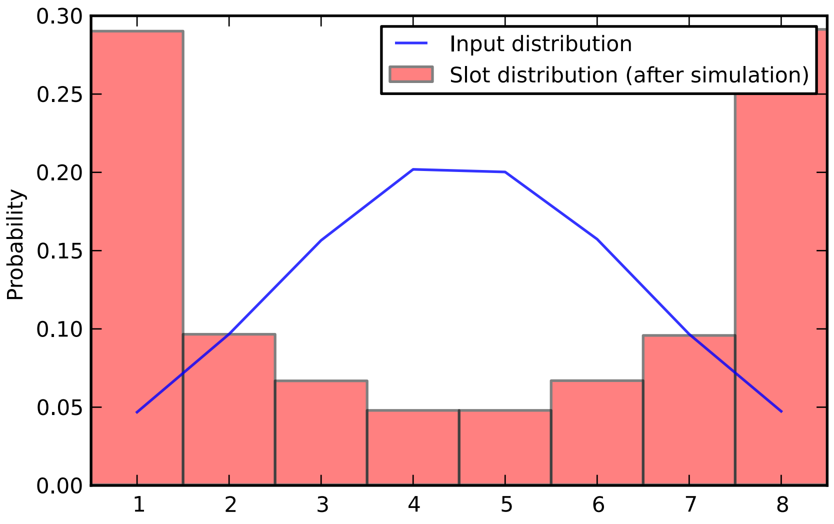


Figure 6.13: Autopartitioning using the normal distribution as input

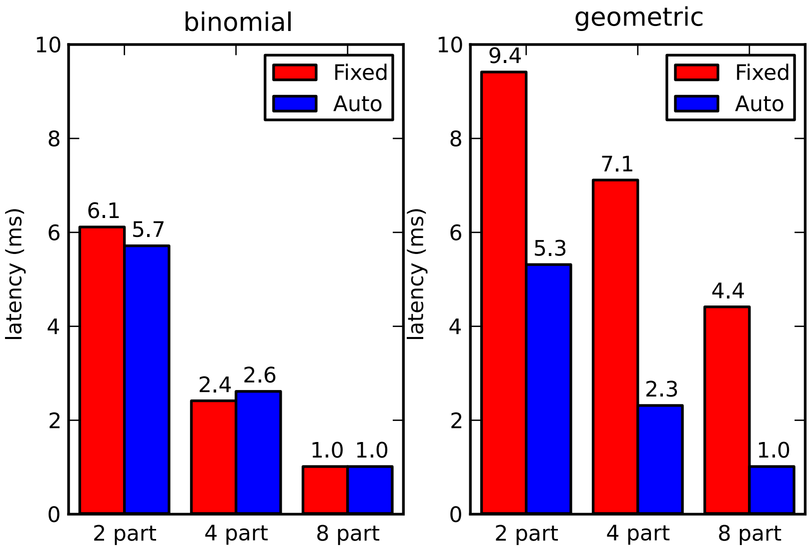


Figure 6.14: Autopartitioning using the binomial and geometric distributions as input

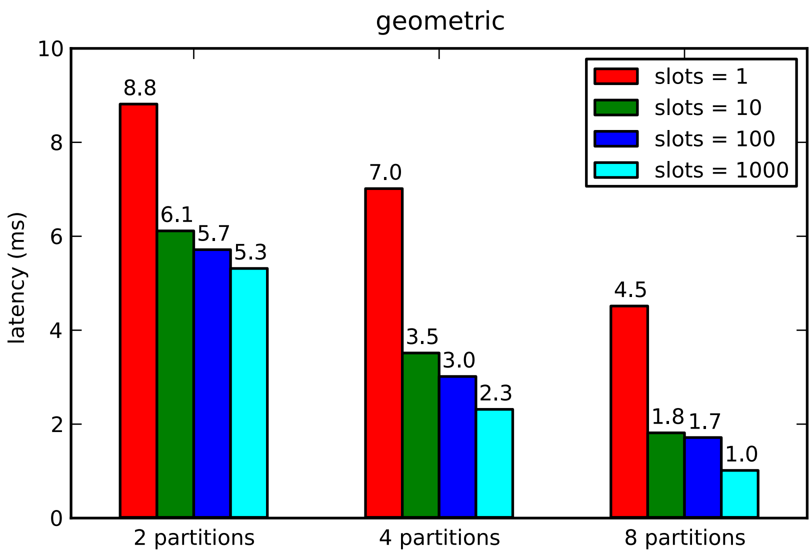


Figure 6.15: The impact of the number of slots for the geometric distribution

The second experiment varies the number of slots and the number of partitions. The goal of this experiment is to see the impact of using more than one slot per partition. Figure 6.15 shows the results when testing four different slot sizes of 1, 10, 100 and 1000 with the geometric distribution as input. The biggest reduction in latency comes from increasing the number of slots per partition to 10. Going any further does not seem to have a big effect on the latency. The larger number of slots should, however, be able to deal with fine grained "fluctuations" in the hit distribution, which are not specified in the distributions used in Vsim. To investigate the impact of slots even more, one can change the parameters used to generate the input distribution in the source code itself. This is, however, outside the scope of evaluating parallel query evaluation.

6.6 Discussion

The alternative design gets a linear speedup as the number of cores increase, which is the same as the original search core design. Inefficiencies in the speedup baselines are the primary reason for super linear speedup in the scalability experiments. It is, however, reason to believe that parallel query evaluation will not be able to scale perfectly due to several factors. The memory requirements for the heaps increase linearly with the number of partitions. This, in turn, increases the computation required in the merge phase, which increases the serial overhead. One way to overcome this limitation, is to decrease the number of hits collected per partition while increasing the number of partitions. Another way to overcome such a limitation would be to perform the merge in parallel.

The primary benefit of parallel query evaluation is a lower latency when the number of clients are fewer than the number of processor cores in the system. However, this performance depends greatly on the workload, as the results of running the simulation with QRS logs show. With data from Yahoo! News, the amount of work in the fetchers is so low that it gives a negative latency effect when increasing the number of partitions beyond 2. When using data from Yahoo! Image search, however, the work required per query is greater. This results in a gain when using parallel query evaluation, even when the number of partitions is larger than the number of processor cores in the system. There is, however, nothing against combining the best of both worlds. Since the alternative design can be configured as the original design (using only 1 partition), there will be an "escape route" in case parallel query evaluation performs worse than expected. Another way to improve the workload independence would be to dynamically change the number of partitions as the load changes.

Even though splitting the document range into partitions increases the locality for the per core L1 and L2 caches, the shared L3 cache does not give any extra benefit other than being yet another cache level for the partitioned scheme. In contrast, the original search core design gets a higher miss ratio in the L1 and L2 caches, but regains some of the loss by taking advantage of the shared L3 cache. The average memory latency helps putting everything into perspective, and shows that the alternative search core design gets an overall lower latency. But, it is still not enough to nullify the overhead of dispatching and merging and improve throughput performance during high loads. Further, it is hard to tell how the design will continue to scale on manycore processors due to the potential scalability problems discussed above. But, as the shared cache play no important role in parallel query evaluation, it is reason to believe that it could scale well on manycore processors from the perspective of the cache hierarchy.

The reasons as to why the asynchronous approach performs badly in [BGM$^+$07] becomes clearer when taking the results from the cache hierarchy experiments into account: The Niagara T1 processor used in their study contains a small L1 cache of 8kB, and an L2 cache of 3MB shared between all 32 threads [KAO05]. Most likely, the scheme proposed in this study would not be able to use the shared L2 cache efficiently because of the shared L2 cache. The original search core design, however, would be able to take advantage of the shared L2 by reusing data from the other threads. Conversely, their asynchronous strategy would perhaps perform different, if there were less or no shared cache in their system.

The need for an autopartitioning algorithm is a good example of the increased complexity of an alternative design, as it is not necessary if the data is contained within only one partition. However, the need for such an algorithm can be questioned when looking at a large scale Vespa deployment, where the index is partitioned across hundreds of search nodes without any sort of run-time partitioning algorithm. Rather than re-balancing, the data is hashed to a random node. Ideally, one could to the same hashing of documents as in a search cluster within each search node, to make sure the load gets evenly distributed.

Another negative effect of parallel query evaluation is the increase in complexity within the ranking phase itself. Since the original design puts the thread handling "outside" of the query evaluation, the evaluation of a query can be treated as a sequential program independent of other queries. If algorithms needs to be rewritten, the threading will be the same. By introducing more parallelism in the search core, data structures may have to be altered, and the overhead related to dispatching and merging of results can outweigh the advantages, depending on the load and the CPU architecture. However, if properly isolated, this change is arguably not very intrusive as there are ideally no shared data structures across the partitions.

Thus, the gain of parallel query evaluation must be weighted against the increased complexity and the uncertainty of its scalability on manycore systems. However, the risk of implementing the alternative design is minimal, as it can be configured to behave as the original design, if implemented properly. This allows different users of Vespa to configure the number of partitions to their needs. If parallel query evaluation is implemented, each deployment of Vespa should consider their workload and configure the number of partitions thereafter.

Chapter 7

Conclusion and further work

7.1 Conclusion

The main goal of this study is to design and implement an alternative threading model for the Vespa search engine that would evaluate queries in parallel. The first part of the study examined the Vertical search platform (Vespa) developed at YTN. Studying the architecture of Vespa was necessary to understand the implications of any new design choices to the search core and for developing the search engine simulator, Vsim.

Today's multicore processors use cache coherence schemes that fail to scale with a large number of processors cores. Studying the behavior of cache coherence protocols and techniques for playing nice with the cache helped understand design choices in Vespa, as well as making design choices in Vsim. Moreover, learning different tools for measuring application performance helped during the development phase of Vsim, but also when evaluating Vsim and parallel query evaluation. Classic laws in the field of parallelism and scalability such as Amdahl's and Gustafson's law are used when evaluating the alternative search core design, and have given useful insights into scalability past eight cores.

The primary development effort in this study, Vsim, makes it possible to validate any further work on parallel query evaluation. The results obtained in Chapter 6 show that parallel query evaluation scales with the data size, provides lower search latencies than the original design during low query loads, performs on par with the original design during high query loads, and does indeed experience a higher cache hit rate in the per core processor caches. As discussed in Chapter 3, a similar approach to parallel query evaluation was tried with unexpectedly poor results [BGM+07]. This study presents different results, where parallel query evaluation may actually be advantageous for low loads. Unfortunately, their study does not go into details about the reasons for these performance issues.

The last level cache on today's CMPs ensures that the original design works well, but with the introduction of multicore chips with a large number of cores, the architectures will most probably have to change. An important advantage of the alternative design is that it can be configured as the original design. This brings the best of both worlds together, and reduces the risk of implementing the alternative design in the real search core.

7.2 Further work

Following is a list of further work and directions to take from here.

- An implementation of parallel query evaluation in Vespa is a logical next step from this study. Currently, the Vespa search core is under heavy refactoring to support quicker indexing latencies. The new search core design is simpler, which means that supporting parallel query evaluation can be done in different ways than proposed in this study. This study only investigates any potential benefit of parallel query evaluation. An important part of further work would be to adopt parallel query evaluation to the new search core.
- The basis of the work performed in this study is the Vsim simulator. Another way to evaluate parallel query evaluation would be to instrument the search core itself to get detailed performance numbers. Further, profiling support could be built into the search core in order to easily evaluate performance when code is changed. This requires much development effort, but it is probably worth it in the long term, especially with the rapid change of computer hardware.
- As this study evaluates parallel query evaluation on an 8 core system. As hardware with more cores becomes available, evaluating Vsim on such systems would give an even better indication of the usefulness of parallel query evaluation. Moreover, seeing how the design behaves on different cache architectures could help choosing what hardware would be appropriate to use with Vespa.
- In order to test a design on manycore processors, the simulators discussed in Chapter 2 could be improved to support such processors. Implementing support for shared memory multithreaded programs in the M5 simulator would help evaluate Vsim on manycore processors with minimal cost.
- The hit collector algorithms proposed in Section A.1.1 are alternatives that could improve search performance. Although few experiments testing these implementations were performed, they are not considered relevant for the research questions in this study, and as such, the results are not discussed.
- The autopartitioning algorithm that is proposed in Appendix 3.6 is implemented in Vsim and evaluated in this study. There are, however, other

areas of Vespa that could benefit from this algorithm. At the search cluster level, the index is split across many search nodes to meet a Service Level Agreement (SLA). If the distribution of the documents are skewed, the same algorithm may be used to balance the index between the search nodes in a cluster.

References

[AB10a] Acumem AB. Acumem - Multicore Performance. `http://www.acumem.com/`, 2010. [Online; accessed February-2010].

[AB10b] Acumem AB. Acumem ThreadSpotter User manual, 2010. [Online; accessed February-2010].

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[Bal10] Henning Baldersheim. Personal communication, February 2010.

[BGM$^+$07] C. Bonacic, C. Garcia, M. Marin, M. Prieto, F. Tirado, and C. Vicente. Improving search engines performance on multithreading processors. Technical report, Depto. Arquitectura de Computadores y Automática a Universidad Complutense de Madrid, 2007.

[BM99] Dov Bulka and David Mayhew. *Efficient C++: Performance Programming Techniques.* Addison-Wesley, first edition, 1999.

[Dan09] Daniel. Amdahl's law visualized. `http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg`, 2009. [Online; accessed February-2010].

[DEL10] Delicious - Social bookmarking. `http://delicious.com/`, 2003-2010. [Online; accessed January-2010].

[Dre07] Ulrich Drepper. What Every Programmer Should Know About Memory. Technical report, Red Hat Inc., November 2007.

[FLI10] Flickr - Photo sharing. `http://www.flickr.com/`, 2004-2010. [Online; accessed January-2010].

[FLQZ06] Marcus Fontoura, Ronny Lempel, Runping Qi, and Jason Zien. Inverted Index Support for Numeric Search. *Internet Mathematics*, 3(2), 2006.

[Gus88] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, fourth edition, 2007.

[HS08] Maurice Herily and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, first edition, 2008.

[Int09] Intel, editor. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, 1997-2009. [Online; accessed April-2010].

[Int10] Intel, editor. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3B: System programming guide, part 2. Intel, 2007-2010. [Online; accessed April-2010].

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[Knu68] Donald Ervin Knuth. *The art of computer programming / Donald E. Knuth*. Addison-Wiley, Reading, Mass., :, 1968.

[Lan06] A. J. Lande. Evaluering av chip multiprosessor simulatorer. Master Thesis, NTNU, Juli 2006.

[Lev03] John Levon. Oprofile Internals. `http://oprofile.sourceforge.net/doc/internals/index.html`, 2003. [Online; accessed January-2010].

[Lev04] John Levon. Oprofile Manual. `http://oprofile.sourceforge.net/doc/index.html`, 2004. [Online; accessed January-2010].

[Lil09] Ulf Lilleengen. Impact of multicore architectures on search engines. Technical report, NTNU, 2009.

[M5S10] M5 Simulator. `http://m5sim.org/wiki/index.php/Main_Page`, 2010. [Online; accessed January-2010].

[MHSM09] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 261 –270, sept. 2009.

[Nat91] Lasse Natvig. *Evaluation Parallel Algorithms - Theoretical and practical aspects*. PhD thesis, The Norwegian Institute of Technology, 1991.

[Neh09] Intel Previews Intel Xeon 'Nehalem-EX' Processor. `http://www.intel.com/pressroom/archive/releases/2009/20090526comp.htm`, May 2009. [Online; accessed June-2010].

[neh10] Intel Nehalem microarchitecture overview. `http://www.intel.com/technology/architecture-silicon/next-gen/index.htm`, 2010. [Online; accessed May-2010].

[oB09] University of Bordeux. Portable Hardware Locality (hwloc). `http://www.open-mpi.org/projects/hwloc/doc/v1.0.1/`, 2009. [Online; accessed June-2010].

[OPR02] Oprofile - A system profiler for Linux. `http://oprofile.sourceforge.net/`, 2002. [Online; accessed January-2010].

[PGK87] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). Technical report, Berkeley, CA, USA, 1987.

[Pol] Teraflops research chip. `http://techresearch.intel.com/articles/Tera-Scale/1449.htm`. [Online; accessed June-2010].

[Ris04] Knut Magne Risvik. *Scaling Internet Search Engines: Methods and Analysis*. PhD thesis, NTNU, May 2004.

[RM02] Knut Magne Risvik and Rolf Michelsen. Search engines and web dynamics. *Computer Networks*, 39(3):289 – 302, 2002.

[RR07] Sanguthevar Rajasekaran and John Reif, editors. *Handbook of Parallel Computing*. Chapman and Hall Publishers, 2007.

[Shi96] Yuan Shi. Reevaluating Amdahl's law and Gustafson's law. November 1996. [Online; accessed May-2010].

[SIM10] SIMICS. `http://www.virtutech.com/`, 2010. [Online; accessed January-2010].

[SKT$^+$05] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.

[TIL10] Tilera corporation. `http://www.tilera.com/products/TILE-Gx.php`, 2004-2010. [Online; accessed May-2010].

[VAL00] Valgrind Instrumentation Framework. `http://www.valgrind.org`, 2000. [Online; accessed March-2010].

[VAL09] Valgrind User Manual. `http://valgrind.org/docs/manual/manual.html`, 2009. [Online; accessed March-2010].

[VES10] Vespa documentation. `http://vespa.corp.yahoo.com/documentation/`, 2005-2010. [Internal company documentation. Online; accessed January-2010].

[VTU10] Intel VTune. http://software.intel.com/en-us/intel-vtune/, 2010. [Online; accessed February-2010].

[YMA10] Yahoo! Mail. http://www.ymail.com/, 1997-2010. [Online; accessed January-2010].

[ZJH09] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. Technical report, Faculty of Informatics, University of Lugano, 2009.

Appendix A

Additional notes

A.1 Vsim alternative designs and ideas

A.1.1 Alternative HitCollector implementations

Figure A.1 describes a different hit collector implementation based on radix sort and merging. The heap based hit collector is changed to keep three buffers for collecting hits. A buffer may have three roles, either as a *collection buffer*, where new hits are collected, or as a *operand buffer*, which contains a sorted array of the best hits encountered so far, or as a *result buffer* to where the collection and operand buffers should be merged.

Henning [Bal10] suggested to try approach, based on radix sort of a buffer with a size of twice the size specified with the `-h` parameter. This buffer is then split in half, where the first half is used to store the heap size number of hits, while the other half is used to store new hits that are collected. Figure A.2 illustrates this scheme. After the first half of the buffer fills up, it is sorted. The last element of the first half is now the lowest ranking element collected. All new hits collected are compared against this element. If the rank of the new hit is larger than the lowest ranking element in the first half of the buffer, the new hit is added to the second half of the buffer. When the second half of the buffer is full, heap size elements of the whole buffer is sorted, and the second half is set to be empty. The buffer now contains the best hits so far, and new hits are collected in the second half. When all hits are collected, the algorithm is complete.

Experiments involving the alternative `HitCollector` implementations are outside the scope of this study, and is therefore only included as a reference for further work.

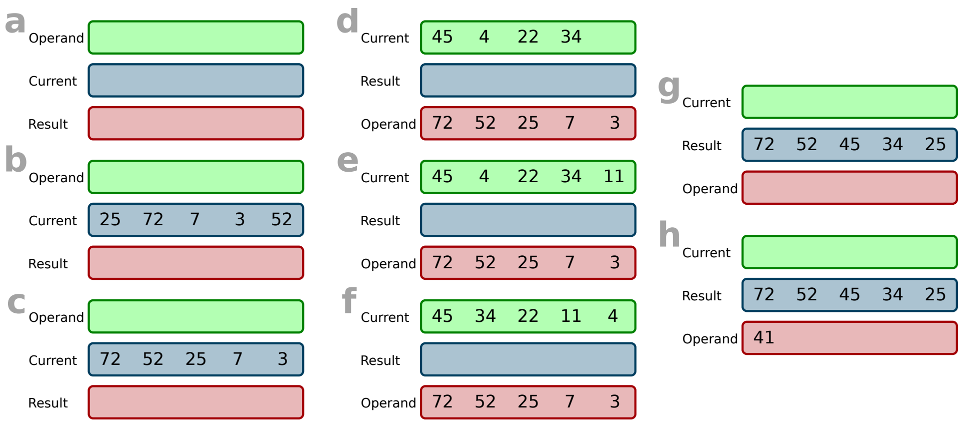
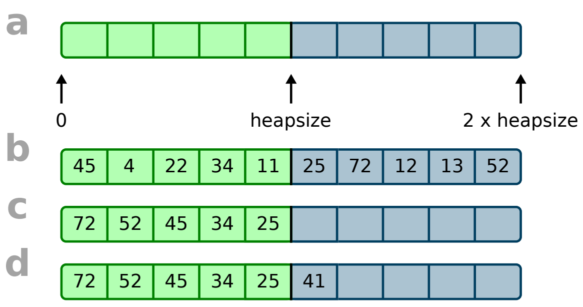


Figure A.1: Alternative algorithm for collecting hits using radix sort and merging: a) Buffers start out empty. b) The current buffer is filled up with the newest hits. c) When the buffer is full, the hits are sorted. d) The buffer pointers are incremented, and the previous operand is now the current buffer. New hits are added to the new current buffer. e) The current buffer fills up. f) The current buffer is now sorted before g) being merged together with the operand into the result buffer. h) The buffer pointers are again incremented, and the new current buffer is used for collecting new hits.

Figure A.2: Another algorithm for collecting hits using radix sort on a large buffer: a) The initial buffer is twice the size of the result buffer. b) Hits are added to the buffer until c) The buffer is full, and sorted. Only half of the buffer is kept. d) New hits are added to the second half of the buffer until it is full and step c is repeated.



A.1.2 Allocating a `QueryHandler` and `Fetchers` for each `Query`

The first version of Vsim allocated a new `QueryHandler` for each new `Query` object and used the `QueryQueue` for enqueueing and dispatching `QueryHandlers`, much like the Vespa search core. Each `QueryQueue` had a `QueueChecker`, which ran as a separate thread which checked the `QueryQueue` for new entries and dispatched them. This approach seamed to work, but it made it hard to control how many `QueryHandler` threads that was executing at the same time. An important note in this scheme is that since each new `Query` results in the allocation of a new `QueryHandler` and each new `QueryHandler` allocates a `Fetcher` for each partition. This means that the memory allocator needs to be scalable when multiple threads are allocating memory at the same time.

The biggest problem with this approach is that by allocating a new handler for each `Query` object, the control of the simulation is lost in the sense that one does not know how to limit the number of `QueryHandlers` to the number given by the `-c` parameter without keeping track of how many handlers are dispatched and completed. One can keep track by counting the number of dispatched `QueryHandlers` and demanding that all of them reports their exit to the `VSimStat` object, but this resulted in extra locking overhead in the `VSimStat` object.

The designed was changed into the current "queue based" design, because it allows detailed monitoring of each query and how long time it spends in all stages of query evaluation.

A.2 Initial flaws in Vsim

A.2.1 Performing result sorting instead of merging in the `QueryHandler`

In the first version of Vsim, the results for each `Fetcher` were not sorted, but *concatenated* onto a result buffer. This decreased the time spent in each `Fetcher` sorting, but resulted in a large fraction of the time being spent in the `QueryHandler`, because the `QueryHandler` had to sort the results from all partitions. As the number of partitions increased, the fraction of time spent in the `QueryHandler` increased even more.

The solution was to move the sorting process within each `Fetcher` and perform only a merge step in the `QueryHandler`. This is intuitive: the first version has a $O(kN)$ complexity for the QueryHandler while the second version has a $O(N)$ complexity, where N equals the number of hits encountered in total by all partitions, and k equals the number of digits for each number in the sort (in Vsim, $k = 8$).

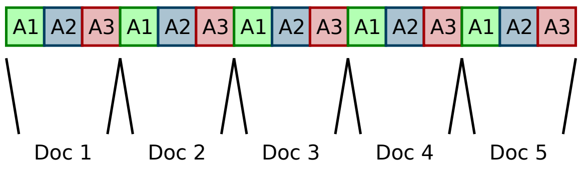


Figure A.3: The layout of attributes in the first version of Vsim

A.2.2 Initial implementation of the `VSimStore`

The initial implementation of the `VSimStore` class used a different memory layout than the current implementation, shown in Figure A.3. This memory layout of the initial implementation is not the same as in Vespa, as all attributes of each document are stored together in an array, which is clearly different from that in Figure 2.4. The layout made the access pattern very predictable, as a cache line may contain many if not all attributes of a document if they are small enough.

A.3 Techniques to promote effective cache reuse

Bypass cache

Each intermediate level in the memory hierarchy eventually causes extra delays for memory requests that do not hit the cache at all. If the software is aware of such data not being reused, it is possible to *bypass* the cache with special instructions [Dre07]. Such instruction can be useful for *streaming* workloads, such as live video or music, which do not want to re-use any data in the cache.

Prefetching

Although the hardware prefetcher may easily detect patterns and fetch the appropriate cache lines before they are used, the software has most knowledge of what data should be prefetched. By accessing data before it is used, the software is able to get quick access to the data when it will actually be used. Implementing software prefetching may be done by using helper threads [Dre07], which are threads instructed to access data in order to bring it into the cache. For this to work, the one has to make sure the helper thread runs on the same core as the processing thread, and care must be taken to stop the helper thread from negatively affecting performance of the worker thread.

Appendix B

Clarifications

B.1 Performance counter mappings

Table B.1 shows the mapping of performance counter names, their mask used to get the actual counter, and the short name used in Section 5.2.

Table B.1: Performance counters used

Name	**Mask**	**Short name**
L1D_CACHE_LD.I_STATE	0x01	L1 MISSES
L1D_CACHE_LD.MESI	0x0f	L1 LOADS
L2_DATA_RQSTS.I_STATE	0x01	L2 MISSES
L2_DATA_RQSTS.MESI	0xff	L2 LOADS
MEM_LOAD_RETIRED.LLC_MISS	L3 MISSES	0x10
CPU_CLK_UNHALTED	No mask	CYCLES

Appendix C

Code

C.1 Vsim source code

Here follows the source code of vsim. All source files are included, except some detailed files regarding the radix sort implementation used in Vespa. The source code for atomic instructions is also from Vespa, and not included.

C.1.1 barrier.h

```
#ifndef _VSIM_BARRIER_H_
#define _VSIM_BARRIER_H_
#include <pthread.h>

namespace vsim {

/**
 * A wrapper for POSIX barriers.
 */
class Barrier {
public:
    Barrier(unsigned);
    virtual ~Barrier();
    void wait(void);
protected:
    pthread_barrier_t _barrier;
};

} // end vsim
#endif
```

C.1.2 barrier.cpp

```
#include "barrier.h"

namespace vsim {

Barrier::Barrier(unsigned count)
{
    pthread_barrier_init(&_barrier, NULL, count);
}

Barrier::~Barrier()
{
    pthread_barrier_destroy(&_barrier);
}

void
Barrier::wait(void)
{
    pthread_barrier_wait(&_barrier);
}

} // end vsim
```

C.1.3 cond.h

```
#ifndef _VSIM_COND_H_
#define _VSIM_COND_H_
#include <pthread.h>
#include "mutex.h"

namespace vsim {
/**
 * A wrapper for POSIX conditional variables, which carries an
     implicit mutex.
 */
class Cond : public Mutex
{
public:
    Cond();
    ~Cond();
    void wait(void);
    void timedWait(unsigned long);
    void notify(void);
    void notifyAll(void);
private:
    pthread_cond_t _cond;
};

} // end vsim
#endif
```

C.1.4 cond.cpp

```
#include <pthread.h>
#include <time.h>
#include "cond.h"
```

```
namespace vsim {

Cond::Cond() : Mutex()
{
    pthread_cond_init(&_cond, NULL);
}

Cond::~Cond()
{
    pthread_cond_destroy(&_cond);
}

void
Cond::wait(void)
{
    pthread_cond_wait(&_cond, &_lock);
}

void
Cond::timedWait(unsigned long seconds)
{
    struct timespec tv;
    tv.tv_sec = seconds;
    tv.tv_nsec = 0;
    pthread_cond_timedwait(&_cond, &_lock, &tv);
}

void
Cond::notify(void)
{
    pthread_cond_signal(&_cond);
}

void
Cond::notifyAll(void)
{
    pthread_cond_broadcast(&_cond);
}

} // end vsim
```

C.1.5 fetcher.h

```
#ifndef _VSIM_FETCHER_H_
#define _VSIM_FETCHER_H_
#include <boost/shared_ptr.hpp>
#include <boost/random/linear_congruential.hpp>
#include <boost/random/uniform_int.hpp>
#include <boost/random/variate_generator.hpp>
#include "task.h"
#include "query.h"
#include "vsimstore.h"
#include "cond.h"
#include "barrier.h"
#include "partitionqueue.h"
```

```
namespace vsim {

typedef boost::minstd_rand base_generator_type;

/**
 * The Fetcher class is in charge of generating document ids according
     to the
 * number of hits given and retrieve the document ids and their rank
    from the
 * document store. Finally, the query is notified when the fetcher is
    finished.
 */
class Fetcher : public Task {
public:
    typedef boost::shared_ptr<Fetcher> SP;
    /**
     * Construct a Fetcher, initialize the range in which valid
        document ids can be
     * constructed.
     */
    Fetcher(const VSimStore &, PartitionQueue &, partid_t, unsigned
        long, unsigned int, docid_t);
    virtual ~Fetcher();

    /**
     * Generate docids to be fetched from the document store. Retrieve
         these
     * documents and merge them with documents from the other threads.
     */
    void run(void);

    /**
     * Shutdown Fetcher. Signal the PartitionQueue to wake up all
        other fetchers
     * and exit.
     */
    void shutdown(void);

    /**
     * Return our affinity preference.
     */
    bool getPreferredCPU(cpu_set_t &);

private:
    const VSimStore &_ds;
    PartitionQueue &_queue;
    partid_t _id;
    unsigned int _handlerid;
    base_generator_type _generator;
    boost::uniform_int <> _dist;
    boost::variate_generator<base_generator_type&, boost::uniform_int
        <> > _vgen;
    Barrier _done;
};

} // end vsim
```

```
#endif
```

C.1.6 fetcher.cpp

```
#include <iostream>
#include <sys/time.h>
#include "task.h"
#include "fetcher.h"
#include "queryhandler.h"
#include "misc.h"

namespace vsim {

Fetcher::Fetcher(const VSimStore &ds, PartitionQueue &queue,
    docid_t id, unsigned long seed, unsigned int hid, docid_t maxdocid
        ) :
    _ds(ds),
    _queue(queue),
    _id(id),
    _handlerid(hid),
    _generator(seed),
    _dist(0, maxdocid - 1),
    _vgen(_generator, _dist),
    _done(2)
{
}

Fetcher::~Fetcher()
{
}

void
Fetcher::run(void)
{
    bool stop = false;
    while (1) {
        Query::SP qsp = _queue.dequeue(stop);
        if (stop)
            break;
        HitCollector &hits(qsp->getCollector(_id));
        unsigned long totalhits = 0;

        const PartitionRange &slots(qsp->getRange(_id));
        // Generate a hit for each slot we have in our partition
        for (docid_t s = 0; s < slots.size(); s++) {
            const Slot &slot(slots[s]);
            unsigned long numhits = qsp->getHits(slot);
            totalhits += numhits;
            for (unsigned long i = 0; i < numhits; i++) {
                docid_t doc = (_vgen() % (slot.second - slot.first)) +
                          slot.first;
                hits.addHit(doc, _ds.getRank(doc));
            }
        }
        hits.sortBuffer();
        qsp->notify(_id, totalhits);
    }
    _done.wait();
}

void
Fetcher::shutdown(void)
{
    _queue.shutdown();
    _done.wait();
}

bool
Fetcher::getPreferredCPU(cpu_set_t &set)
{
#ifdef AFFINITY
    CPU_ZERO(&set);
    CPU_SET(_id % NUMCPU, &set);
    return (true);
#else
    return false;
#endif
}

} // end vsim
```

C.1.7 hitcollector.h

```
#ifndef _VSIM_HITCOLLECTOR_H_
#define _VSIM_HITCOLLECTOR_H_
#include <vector>
#include <stdint.h>
#include "hitvector.h"
#include "types.h"

namespace vsim {

enum HitState { COLLECTHIT, REPLACEHIT };

/**
 * The HitCollector class can be instructed to store hits in a fashion
      similar
 * to the same class in Vespa.
 */
class HitCollector
{
public:
    /**
     * Create a HitCollector with a internal buffer of the size of the
          argument.
     */
    HitCollector(docid_t);

    /**
     * Add a new hit to the hitcollector heap using id and rank from
         document.
     */
    void addHit(docid_t, rank_t);

    inline unsigned long size(void) const { return _hitvector.size();
        }
    inline const HitVector &getResultBuffer(void) const { return
        _hitvector; }

    /**
     * Reset everything to the initial state.
     */
    void reset(void);

    /**
     * Sort the internal buffer.
     */
    void sortBuffer(void);

    /**
     * Organize after rank and document id.
     */
    class ScoreComparator {
    public:
        ScoreComparator() {}
        bool operator() (const Hit & lhs, const Hit & rhs) const {
            if (lhs.second == rhs.second) {
                return lhs.first < rhs.first;
            }
            return lhs.second >= rhs.second; // oparator for min-heap
        }
    };
    void printHits(void) const;

private:
    /**
     * Exchange hit on the heap if the provided hit is higher ranked
         than the
     * lowest ranked in the heap.
     */
    void considerForHit(docid_t, rank_t);

    HitVector _hitvector;
    docid_t _maxhits;
    HitState _state;
};

} // end vsim

#endif
```

C.1.8 hitcollector.cpp

```
#include <algorithm>
#include <iostream>
#include "hitcollector.h"
#include "hitvector.h"
```

```
namespace vsim {

HitCollector::HitCollector(docid_t maxhits) :
    _hitvector(maxhits),
    _maxhits(maxhits),
    _state(COLLECTHIT)
{
}

void
HitCollector::considerForHit(docid_t docId, rank_t rank)
{
    if (_hitvector.empty()) return;

    if (rank > _hitvector[0].second) {
        std::pop_heap(_hitvector.begin(), _hitvector.end(),
            ScoreComparator());
        _hitvector.back().first = docId;
        _hitvector.back().second = rank;
        std::push_heap(_hitvector.begin(), _hitvector.end(),
            ScoreComparator());
    }
}

void
HitCollector::addHit(docid_t docid, rank_t rank)
{
    switch (_state) {
    case COLLECTHIT:
        // Add new hits until we reach the limit
        if (_hitvector.size() < _maxhits) {
            _hitvector.add(Hit(docid, rank));
        } else {
            _state = REPLACEHIT;
            std::make_heap(_hitvector.begin(), _hitvector.end(),
                ScoreComparator());
            considerForHit(docid, rank);
        }
        break;
    case REPLACEHIT:
        considerForHit(docid, rank);
        break;
    }
}

void
HitCollector::sortBuffer(void)
{
    _hitvector.sort();
}

void
HitCollector::printHits(void) const
{
    std::cout << "Hits: \n";
    for (unsigned int i = 0; i < _hitvector.size(); i++) {
        std::cout << "\t" << _hitvector[i].first << " " << _hitvector[
            i].second << std::endl;
    }
}

} // end vsim
```

C.1.9 hitvector.h

```
#ifndef _VSIM_HITVECTOR_H_
#define _VSIM_HITVECTOR_H_
#include <vector>
#include <functional>
#include "types.h"

namespace vsim {
/**
 * The HitVector class implements an array interface towards a
     preallocated
 * buffer of known size, which may quickly be erased and reused.
 */
class HitVector {
public:
    /**
     * Construct a HitVector with an internal buffer of size maxlen.
     */
    HitVector(docid_t maxlen) :
        _buffer(maxlen),
        _len(0),
        _best(0)
    { }
    const Hit &operator[] (unsigned long a) const { return _buffer[a];
         }
    /**
     * Add a new hit to the buffer.
     */
    inline void add(const Hit &h) { _buffer[_len++] = h; }

    /**
     * Sort the array according to the comparators defined in this
         class.
     */
    void sort(void);

    /**
     * Merge two vectors provided as arguments into this vector
         starting at the
     * current offset pointer.
     */
    void merge(const HitVector &, const HitVector &);

    /**
     * Used to compared values for radix sort.
     */
    class ValueCompare : public std::binary_function<Hit, Hit, bool> {
    public:
        bool operator() (const Hit &lhs, const Hit &rhs) const {
            return (lhs.second > rhs.second);
        }
    };

    /**
     * Used to retrieve sort value
     */
    class ValueRadix {
    public:
        uint64_t operator() (const Hit & h) const {
            return (uint64_t)(~h.second);
        }
    };


    /**
     * Various methods defined to be able to be used as an STL
         container.
     */
    inline Hit &back(void) { return (_buffer[_len - 1]); }
    inline std::vector<Hit>::iterator begin(void) { return (_buffer.
        begin()); }
    inline std::vector<Hit>::iterator end(void) { return (_buffer.
        begin() + _len); }
    inline unsigned long size(void) const { return (_len); }
    inline bool empty(void) { return (_len == 0); }
    inline void reset(void) { _len = 0; }
    void printHits(void) const;
private:
    std::vector<Hit> _buffer;
    docid_t _maxlen;
    docid_t _len;
    docid_t _best;
};

} // end vsim
#endif
```

C.1.10 hitvector.cpp

```
#include <iostream>
#include "hitvector.h"
#include "searchlib/common/sort.h"

namespace vsim {

void
HitVector::sort(void)
{
    search::ShiftBasedRadixSorter<Hit, ValueRadix, ValueCompare, 56>::
        radix_sort(ValueRadix(), ValueCompare(), &_buffer[0], _len);
}

void
HitVector::merge(const HitVector &op1, const HitVector &op2)
{
    docid_t i_op1 = 0;
    docid_t i_op2 = 0;

    while (i_op1 < op1.size() &&
           i_op2 < op2.size() &&
           _len < _buffer.size()) {
        const Hit &h1(op1[i_op1]);
        const Hit &h2(op2[i_op2]);
        // If left operand is smaller than right
        if (h1.second > h2.second) {
            add(h1);
            i_op1++;
        } else if (h1.second < h2.second) {
            add(h2);
            i_op2++;
        } else {
            add(h1);
            i_op1++;
            if (_len < _buffer.size()) {
                add(h2);
                i_op2++;
            }
        }
    }
    if (_len < _buffer.size()) {
        // According to the condition above, only one of these while
            loops will
        // get executed
        while (i_op1 < op1.size() &&
               _len < _buffer.size()) {
            add(op1[i_op1++]);
        }
        while (i_op2 < op2.size() &&
               _len < _buffer.size()) {
            add(op2[i_op2++]);
        }
    }
}

void
HitVector::printHits(void) const
{
    for (unsigned int i = 0; i < _len; i++) {
        std::cout << "\t" << _buffer[i].first << " " << _buffer[i].
            second << std::endl;
    }
}

} // end vsim
```

C.1.11 main.cpp

```
#include <iostream>
#include <cstring>
```

```
#include "vsim.h"
#include "vsimconfig.h"
#include "vsimreport.h"
#include "types.h"


void
usage(const char *prog)
{
    std::cerr << "Usage: " << prog << " [OPTION]... simlength" << std
        ::endl;
    std::cerr << "OPTION may be one of these:" << std::endl;

    struct option {
        const char *name;
        const char *description;
    };
    struct option options[] = {
        {"a", "The number of attributes per document"},
        {"c", "Maximum number of queryhandlers"},
        {"d", "Maximum document id"},
        {"h", "Maximum number of hits returned per query {heap size)"
            },
        {"i", "The probability distribution to apply to the slots (
            binomial, poisson, geometric or gauss)"},
        {"l", "The number of slots to initially allocate per partition
            "},
        {"n", "A static number of hits used per query (unless using
            QRS log files)"},
        {"o", "Sample file to store simulation data"},
        {"p", "Number of partitions"},
        {"q", "Prefix of QRS log files used to generate hit counts"},
        {"r", "A comma-separated list of attribute vector indices (or
            a range of indices) used for rank calculation"},
        {"s", "The number of rounds to run the simulation"},
        {"t", "Number of clients to spawn"},
        {NULL, NULL}
    };

    unsigned int i = 0;
    while (options[i].name != NULL && options[i].description != NULL)
        {
        std::cerr << "\t-" << options[i].name << "\t\t" << options[i].
            description << std::endl;
        i++;
    }
}


int
main(int argc, char **argv)
{
    int opt;
    bool set_attr_range = false;
    vsim::docid_t numpart, numslots, maxdocid, hitheapsize, numhits,
        maxdocid_tmp;
    unsigned long numattr, maxconn, clients, numrounds;
```

```
std::string attrrange, qrlog, samplefile, distribution;
vsim::AttributeIndexVector aiv;

// Default values
numpart = numslots = maxconn = 1;
clients = 1;
maxdocid = hitheapsize = numhits = maxdocid_tmp = 1000;
numattr = numrounds = 1;
aiv.push_back(0);

while ((opt = getopt(argc, argv, "a:c:d:h:i:l:m:n:o:p:q:r:s:t:"))
    != -1) {
    switch (opt) {
    case 'a':
        numattr = strtol(optarg, NULL, 10);
        break;
    case 'c':
        maxconn = strtol(optarg, NULL, 10);
        break;
    case 'd':
        maxdocid_tmp = strtol(optarg, NULL, 10);
        break;
    case 'h':
        hitheapsize = strtol(optarg, NULL, 10);
        break;
    case 'i':
        distribution = std::string(optarg);
        break;
    case 'l':
        numslots = strtol(optarg, NULL, 10);
        break;
    case 'n':
        numhits = strtol(optarg, NULL, 10);
        break;
    case 'o':
        samplefile = std::string(optarg);
        break;
    case 'p':
        numpart = strtol(optarg, NULL, 10);
        break;
    case 'q':
        qrlog = std::string(optarg);
        break;
    case 'r':
        attrrange = std::string(optarg);
        set_attr_range = true;
        break;
    case 's':
        numrounds = strtol(optarg, NULL, 10);
        break;
    case 't':
        clients = strtol(optarg, NULL, 10);
        break;
    default:
        usage(argv[0]);
        exit(-1);
        break;
    }
}
if (optind >= argc) {
    std::cerr << "Expected argument after options" << std::endl;
    usage(argv[0]);
    exit(-1);
}

unsigned long simtime = 0;
simtime = strtol(argv[optind], NULL, 10);

if (set_attr_range) {
    aiv.clear();
    // Trim whitespace
    for (std::string::iterator it = attrrange.begin();
         it != attrrange.end();
         it++) {

        if ((*it) == ' ' || (*it) == '\t')
            attrrange.erase(it);
    }
    size_t first = 0;
    size_t last = std::string::npos;
    do {
        last = attrrange.find(',', first);
        std::string attr(attrrange.substr(first, (last - first)));
        // Find the '-' delimiter;
        size_t delimpos = attr.find('-');
        if (delimpos != std::string::npos) {
            std::string tmp(attr.substr(0, delimpos));
            unsigned long startattr = strtol(tmp.c_str(), NULL,
                10);
            tmp = attr.substr(delimpos + 1, (attr.length() -
                delimpos));
            unsigned long lastattr = strtol(tmp.c_str(), NULL, 10)
                ;
            if (startattr >= 0 && lastattr <= numattr) {
                for (unsigned int i = startattr; i < lastattr; i
                    ++) {
                    aiv.push_back(i);
                }
            }
        } else {
            unsigned long attrnum = strtol(attr.c_str(), NULL, 10)
                ;
            if (attrnum >= 0 && attrnum < numattr)
                aiv.push_back(attrnum);
        }

        first = last + 1;
    } while (last != std::string::npos);
}
maxdocid = ROUND(maxdocid_tmp, numpart);

vsim::VSimConfig cfg(numpart, numslots, maxconn, clients, maxdocid
    , numhits,
    hitheapsize, numattr, aiv, vsim::slotDistTextToEnum(
        distribution),
    simtime, numrounds, qrlog);
vsim::VSim sim(cfg);
sim.run(samplefile);
return 0;
}
```

C.1.12 mergequeue.h

```
#ifndef _VSIM_MERGEQUEUE_H_
#define _VSIM_MERGEQUEUE_H_
#include "queue.h"
#include "types.h"

namespace vsim {

/**
 * A merge queue is a queue of partition ids that are ready for
     merging into the
 * result buffer.
 */
class MergeQueue : public Queue<partid_t>
{
public:
    /**
     * Enqueue a partition identifier ready for merging
     */
    void enqueue(partid_t id) {
        _qlock.lock();
        _queue.push_back(id);
        _qlock.notify();
        _qlock.unlock();
    }
    /**
     * Dequeue a partition identifier ready for merging. If the queue
         is empty,
     * block the caller until it is not empty.
     */
    partid_t dequeue(void) {
        _qlock.lock();
        while (_queue.size() == 0) {
            _qlock.wait();
        }
        partid_t ret = _queue.front();
        _queue.pop_front();
        _qlock.unlock();
        return ret;
    }
};

} // end vsim

#endif
```

C.1.13 misc.h

```
#ifndef _VSIM_MISC_H_
#define _VSIM_MISC_H_
#include "types.h"

namespace vsim {

#define ROUND(var, size) (((var) + ((size) - 1)) & ~((size) - 1))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
SlotDistribution slotDistTextToEnum(const std::string &);
std::string slotDistEnumToText(SlotDistribution);

} // end vsim

#endif
```

C.1.14 misc.cpp

```
#include <string>
#include "misc.h"

namespace vsim {

SlotDistribution
slotDistTextToEnum(const std::string &distname)
{
    if (distname.compare("poisson") == 0)
        return (POISSON);
    if (distname.compare("gauss") == 0)
        return (GAUSS);
    if (distname.compare("geometric") == 0)
        return (GEOMETRIC);
    return (BINOMIAL);
}

std::string
slotDistEnumToText(SlotDistribution dist)
{
    if (dist == POISSON)
        return ("poisson");
    if (dist == GAUSS)
        return ("gauss");
    if (dist == GEOMETRIC)
        return ("geometric");
    return ("binomial");
}

} // end vsim
```

C.1.15 mutex.h

```
#ifndef _VSIM_MUTEX_H_
#define _VSIM_MUTEX_H_
#include <pthread.h>
```

```
namespace vsim {
/**
 * The Mutex class is a wrapper around the POSIX mutex synchronization
 * primitive.
 */
class Mutex {
public:
    Mutex();
    virtual ~Mutex();
    void lock(void);
    void unlock(void);
protected:
    pthread_mutex_t _lock;
};

} // end vsim

#endif
```

C.1.16 mutex.cpp

```
#include <pthread.h>
#include "mutex.h"

namespace vsim {

Mutex::Mutex()
{
    pthread_mutex_init(&_lock, NULL);
}

Mutex::~Mutex()
{
    pthread_mutex_destroy(&_lock);
}

void
Mutex::lock(void)
{
    pthread_mutex_lock(&_lock);
}

void
Mutex::unlock(void)
{
    pthread_mutex_unlock(&_lock);
}

} // end vsim
```

C.1.17 partitionqueue.h

```
#ifndef _VSIM_PARTITIONQUEUE_H_
#define _VSIM_PARTITIONQUEUE_H_
```

```
#include "queue.h"
#include "types.h"

namespace vsim {

/**
 * A partition queue implements a queue of queries as well as the
     possibility of
 * timestamping insertion and removal from the queue.
 */
class PartitionQueue : public Queue<Query::SP>
{
public:
    PartitionQueue(partid_t id) : _id(id) {}

    /**
     * Enqueue a query into the partition queue, and record its
         insertion point
     * for the particular partition.
     */
    void enqueue(const Query::SP &qsp) {
        _qlock.lock();
        _queue.push_back(qsp);
        qsp->insertPartitionQueue(_id);
        _qlock.notify();
        _qlock.unlock();
    }

    /**
     * Dequeue a query from the partition queue, and record it removal
          point for
     * the partition.
     */
    Query::SP dequeue(bool &stop) {
        _qlock.lock();
        while (_queue.size() == 0) {
            if (_shutdown) {
                stop = true;
                _qlock.unlock();
                return Query::SP();
            }
            _qlock.wait();
        }
        Query::SP ret = _queue.front();
        _queue.pop_front();
        ret->removePartitionQueue(_id);
        _qlock.notifyAll();
        _qlock.unlock();
        return ret;
    }
private:
    partid_t _id;
};

} // end vsim

#endif
```

C.1.18 query.h

```
#ifndef _VSIM_QUERY_H_
#define _VSIM_QUERY_H_
#include <boost/shared_ptr.hpp>
#include "types.h"
#include "resultbuffer.h"
#include "misc.h"
#include "vsimstat.h"
#include "vsimtime.h"
#include "barrier.h"
#include "mergequeue.h"

namespace vsim {

class QueryHandler;

/**
 * A query contains the information needed to generate document id
    hits and
 * buffers to store the hits.
 */
class Query
{
public:
    typedef boost::shared_ptr<Query> SP;

    Query(const PartitionSpecSP &, docid_t, partid_t, docid_t, docid_t
        );
    /**
     * Get the number of hits calculated from the slots probability
        and the
     * number of hits the query should get.
     */
    docid_t getHits(const Slot &);

    /**
     * Get the specific partition range for the partition given as
        argument.
     */
    const PartitionRange &getRange(partid_t);

    inline HitCollector &getCollector(partid_t part) { return _result.
        getHandle(part); }

    /**
     * Notify that a partition is finished collecting hits and ready
        for
     * merging.
     */
    void notify(partid_t, unsigned long);

    /**
     * Perform the merge of the results from each partition. Use the
        merge queue
     * to serialize merging.
     */
    void merge(void);

    /**
     * For the client, to wait until a query is done.
     */
    void wait(void);

    /**
     * Report all statistics collected for this query to the VSimStat
         object.
     */
    void reportStats(VSimStat &);

    /**
     * Methods used to record time stamps for query events.
     */
    void insertQueryQueue(void) { _insertQueryQueue.recordTime(); }
    void insertPartitionQueue(partid_t partid) { _insertPartitionQueue
        [partid].recordTime(); }
    void removePartitionQueue(partid_t partid) { _removePartitionQueue
        [partid].recordTime(); }
    void stopFetcher(partid_t partid) { _stopFetcher[partid].
        recordTime(); }
    void removeQueryQueue(void) { _removeQueryQueue.recordTime(); }
    void stopHandler(void) { _stopHandler.recordTime(); }

private:
    PartitionSpecSP _spec;
    docid_t _maxdocid;
    partid_t _numpart;
    docid_t _numhits;
    ResultBuffer _result;
    HitStat _hs;
    MergeQueue _done;
    Barrier _finished;

    // Statistics kept for us
    VSimTime _insertQueryQueue;
    std::vector<VSimTime> _insertPartitionQueue;
    std::vector<VSimTime> _removePartitionQueue;
    std::vector<VSimTime> _stopFetcher;
    VSimTime _removeQueryQueue;
    VSimTime _stopHandler;
};

} // end vsim

#endif
```

C.1.19 query.cpp

```
#include <iostream>
#include <algorithm>
#include "atomic.h"
#include "query.h"
#include "types.h"
```

```
#include "misc.h"
#include "vsimtime.h"
#include "hitcollector.h"

namespace vsim {

Query::Query(const PartitionSpecSP &ps, docid_t maxdocid, partid_t
    numpart, docid_t heapsize, docid_t numhits) :
    _spec(ps),
    _maxdocid(maxdocid),
    _numpart(numpart),
    _numhits(numhits),
    _result(heapsize, numpart),
    _hs(numpart, 0),
    _finished(2),
    _insertPartitionQueue(numpart),
    _removePartitionQueue(numpart),
    _stopFetcher(numpart)
{
}

docid_t
Query::getHits(const Slot &slot)
{
    return (docid_t)((double)_numhits * slot.prob);
}

const PartitionRange &
Query::getRange(partid_t partid)
{
    return (*_spec)[partid];
}

void
Query::notify(partid_t partid, unsigned long numhits)
{
    _hs[partid] = numhits;
    stopFetcher(partid);
    _done.enqueue(partid);
}

void
Query::merge(void)
{
    for (partid_t nummerged = 0; nummerged < _numpart; nummerged++) {
        partid_t part = _done.dequeue();
        _result.mergeHits(part);
    }
    _finished.wait();
}

void
Query::wait(void)
{
    _finished.wait();
}
void
Query::reportStats(VSimStat &st)
{
    std::vector<uint64_t> fetchertimes(_numpart);
    std::vector<uint64_t> partqueuetimes(_numpart);
    uint64_t maxfetchertime = 0;
    for (partid_t i = 0; i < _numpart; i++) {
        fetchertimes[i] = _removePartitionQueue[i].getDiffMicroSeconds
            (_stopFetcher[i]);
        if (fetchertimes[i] > maxfetchertime)
            maxfetchertime = fetchertimes[i];
        partqueuetimes[i] = _insertPartitionQueue[i].
            getDiffMicroSeconds(_removePartitionQueue[i]);
    }
    unsigned long handlertime = _removeQueryQueue.getDiffMicroSeconds(
        _stopHandler);
    unsigned long queryqueuetime = _insertQueryQueue.
        getDiffMicroSeconds(_removeQueryQueue);
    st.report(fetchertimes, partqueuetimes, maxfetchertime,
        queryqueuetime, handlertime);
    st.reportHits(_hs);
}

} // end vsim
```

C.1.20 querygenerator.h

```
#ifndef _VSIM_QUERYGENERATOR_H_
#define _VSIM_QUERYGENERATOR_H_
#include <boost/shared_ptr.hpp>
#include <boost/random/linear_congruential.hpp>
#include <boost/random/uniform_int.hpp>
#include <boost/random/normal_distribution.hpp>
#include <boost/random/exponential_distribution.hpp>
#include <boost/random/variate_generator.hpp>
#include <fstream>
#include <string>
#include "vsimtime.h"
#include "query.h"
#include "types.h"
#include "task.h"
#include "queryqueue.h"

namespace vsim {

/**
 * A Query Generator interface, capable of generating queries for use
     by the
 * Vespa simulator.
 */
class QueryGenerator : public Task
{
public:
    typedef boost::shared_ptr<QueryGenerator> SP;
    QueryGenerator(QueryQueue &, const PartitionSpecSP &, docid_t,
        unsigned long);
    virtual ~QueryGenerator();
    Query::SP createQuery(docid_t);
    void setPartitionSpec(const PartitionSpecSP &);
    void wait(void);

    void run(void);
    void shutdown(void);
    bool getPreferredCPU(cpu_set_t &);

    /**
     * Method that should be used by users of query generators, which
         must be
     * implemented by sub-classes.
     */
    virtual Query::SP genQuery(void) = 0;

private:
    QueryQueue &_queue;
    Mutex _lock;
    Barrier _done;
    bool _shutdown;


protected:
    PartitionSpecSP _spec;
    docid_t _maxdocid;
    unsigned long _heapsize;


};

/**
 * The static query generator generates a fixed number of hits for
     each
 * partition, and is mostly used to verify the models in the ideal
     cases.
 */
class StaticQueryGenerator : public QueryGenerator
{
public:
    StaticQueryGenerator(QueryQueue &, const PartitionSpecSP &,
        docid_t,
        unsigned long, docid_t);
    Query::SP genQuery(void);
private:
    unsigned long _numhits;
};

/**
 * The query log query generator is used when generating queries from
     a query
 * log file given as input.
 */
class QRSLogQueryGenerator : public QueryGenerator
{
public:
    QRSLogQueryGenerator(QueryQueue &, const PartitionSpecSP &,
        docid_t,
        unsigned long, const std::string &);
    ~QRSLogQueryGenerator();
    Query::SP genQuery(void);
private:
    FILE *_input;
    static const unsigned int buflen = 128;
    char _buffer[buflen];
};

} // end vsim

#endif
```

C.1.21 querygenerator.cpp

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "query.h"
#include "querygenerator.h"

namespace vsim {

QueryGenerator::QueryGenerator(QueryQueue &queue, const
    PartitionSpecSP &spec,
    docid_t maxdocid, unsigned long heapsize) :
    _queue(queue),
    _done(2),
    _shutdown(false),
    _spec(spec),
    _maxdocid(maxdocid),
    _heapsize(heapsize)
{}

QueryGenerator::~QueryGenerator()
{}

void
QueryGenerator::setPartitionSpec(const PartitionSpecSP &spec)
{
    _spec = spec;
}

void
QueryGenerator::run(void)
{
    _lock.lock();
    while (!_shutdown) {
        _lock.unlock();
        Query::SP qsp(genQuery());
        _queue.enqueue(qsp);
        qsp->wait();
        _lock.lock();
    }
    _lock.unlock();
    _done.wait();
}

void
QueryGenerator::shutdown(void)
{
    _lock.lock();
    _shutdown = true;
    _lock.unlock();
}

void
QueryGenerator::wait(void)
{
    _done.wait();
}

bool
QueryGenerator::getPreferredCPU(cpu_set_t &set)
{
    return false;
}

Query::SP
QueryGenerator::createQuery(docid_t numhits)
{
    return (Query::SP(new Query(_spec, _maxdocid, (*_spec).size(),
        _heapsize, numhits)));
}

StaticQueryGenerator::StaticQueryGenerator(QueryQueue &queue, const
    PartitionSpecSP &spec,
    docid_t maxdocid, unsigned long heapsize, docid_t numhits) :
    QueryGenerator(queue, spec, maxdocid, heapsize),
    _numhits(numhits)
{
}

Query::SP
StaticQueryGenerator::genQuery(void)
{
    return (createQuery(_numhits));
}

QRSLogQueryGenerator::QRSLogQueryGenerator(QueryQueue &queue,
    const PartitionSpecSP &spec, docid_t maxdocid, unsigned long
        heapsize,
    const std::string &qrlog) :

    QueryGenerator(queue, spec, maxdocid, heapsize)
{
    _input = fopen(qrlog.c_str(), "r");
}

QRSLogQueryGenerator::~QRSLogQueryGenerator()
{
    fclose(_input);
}

Query::SP
QRSLogQueryGenerator::genQuery(void)
{
    unsigned long numhits = 0;
    do {
        if (fgets(_buffer, buflen, _input) == NULL) {
            rewind(_input);
        }
        numhits = strtol(_buffer, NULL, 10);
    } while (numhits == 0);

#define MAXHITS 1000000
    if (numhits > MAXHITS)
        numhits = MAXHITS;
    return (createQuery(numhits));
}

} // end vsim
```

C.1.22 queryhandler.h

```
#ifndef _VSIM_QUERYHANDLER_H_
#define _VSIM_QUERYHANDLER_H_
#include <boost/shared_ptr.hpp>
#include <sched.h>
#include <stdint.h>
#include "task.h"
#include "query.h"
#include "fetcher.h"
#include "misc.h"
#include "vsimstat.h"
#include "barrier.h"
#include "queryqueue.h"
#include "partitionqueue.h"

namespace vsim {

/**
 * A queryhandler is responsible for synchronizing document ranking
     and sorting
 * of the result.
 */
class QueryHandler : public Task
{
public:
    QueryHandler(std::vector<PartitionQueue> &, VSimStat &, QueryQueue
         &);
    ~QueryHandler();

    /**
     * Shutdown QueryHandler. Wake up all handlers waiting on the same
     * queue and exit.
     */
    void shutdown(void);

    /**
     * The main loop of a QueryHandler thread. Listens on the query
         queue for
     * new queries and dispatches these queries to all fetchers.
         Immediately
     * starts merging of completed results.
     */
    void run(void);

    /**
     * The QueryHandler should be able to run on any CPU.
     */
    bool getPreferredCPU(cpu_set_t &set) { return false; }

private:
    std::vector<PartitionQueue> &_partitions;
    QueryQueue &_queue;
    VSimStat &_st;
    Barrier _done;
};

} // end vsim

#endif
```

C.1.23 queryhandler.cpp

```
#include <iostream>

#include "atomic.h"
#include "queryhandler.h"
#include "misc.h"
#include "types.h"

namespace vsim {

QueryHandler::QueryHandler(std::vector<PartitionQueue> &partitions,
    VSimStat &st, QueryQueue &queue) :
    _partitions(partitions),
    _queue(queue),
    _st(st),
    _done(2)
{
}

QueryHandler::~QueryHandler() { }

void
QueryHandler::run(void)
{
    bool stop = false;
    while (1) {
        Query::SP qsp = _queue.dequeue(stop);
        if (stop) {
            break;
        }
        for (partid_t i = 0; i < _partitions.size(); i++) {
            _partitions[i].enqueue(qsp);
        }
        qsp->merge();
        qsp->stopHandler();
        qsp->reportStats(_st);
    }
    _done.wait();
}

void QueryHandler::shutdown(void)
{
    _queue.shutdown();
    _done.wait();
}

} // end vsim
```

C.1.24 queryqueue.h

```
#ifndef _VSIM_QUERYQUEUE_H_
#define _VSIM_QUERYQUEUE_H_
#include "queue.h"
#include "query.h"

namespace vsim {

/**
 * A QueryQueue implements a queue of queries with the ability to
     timestamp
 * insertion and removal from the queue. Only one instance of this
     queue should
 * exist.
 */
class QueryQueue : public Queue<Query::SP>
{
public:
    QueryQueue(unsigned int maxsize) : Queue<Query::SP>(maxsize) {}

    /**
     * Enqueue a query and record the time of insertion. If the queue
         is full,
     * block the caller until it is not.
     */
    void enqueue(const Query::SP &qsp) {
        _qlock.lock();
        while (_queue.size() >= _maxsize) {
            _qlock.wait();
        }
        _queue.push_back(qsp);
        qsp->insertQueryQueue();
        _qlock.notify();
        _qlock.unlock();
    }
    /**
     * Dequeue the next query from the queue. If the queue is empty,
         block the
     * caller until it is not.
     */
    Query::SP dequeue(bool &stop) {
        _qlock.lock();
        while (_queue.size() == 0) {
            if (_shutdown) {
                stop = true;
                _qlock.unlock();
                return Query::SP();
            }
            _qlock.wait();
        }
        Query::SP ret = _queue.front();
        _queue.pop_front();
        ret->removeQueryQueue();
        _qlock.notifyAll();
        _qlock.unlock();
        return ret;
    }
};

} // end vsim

#endif
```

C.1.25 queue.h

```
#ifndef _VSIM_QUEUE_H_
#define _VSIM_QUEUE_H_
#include <deque>
#include "cond.h"

namespace vsim {

/**
 * The Queue template supports a queue of various size and implements
 * wait and notification mechanism on shutdown.
 */
template <typename T>
class Queue {
public:
    Queue(unsigned int maxsize) : _maxsize(maxsize), _shutdown(false)
        {}
    Queue() : _maxsize(0), _shutdown(false) {}
    virtual ~Queue() {}
    /**
     * Block the caller until all elements in the queue are dequeued.
     */
    void waitUntilEmpty(void) {
        _qlock.lock();
        while (_queue.size() > 0) {
            _qlock.wait();
        }
        _qlock.unlock();
    }

    /**
     * Notify all listeners that the queue will be destroyed.
     */
    void shutdown(void) {
        _qlock.lock();
        _shutdown = true;
        _qlock.notifyAll();
        _qlock.unlock();
    }
protected:
    std::deque<T> _queue;
    Cond _qlock;
    unsigned int _maxsize;
    bool _shutdown;
};

} // end vsim

#endif
```

C.1.26 resultbuffer.h

```
#ifndef _VSIM_RESULTBUFFER_H_
#define _VSIM_RESULTBUFFER_H_
#include <vector>
#include <boost/shared_ptr.hpp>
#include <stdint.h>
#include "hitcollector.h"
#include "types.h"

namespace vsim {

/**
 * A ResultBuffer contains result buffers for all partitions, and a
     result
 * partition to which the buffers of all partitions may be merged.
 */
class ResultBuffer
{
public:
    ResultBuffer(docid_t numhits, partid_t numpart) :
        _collectors(numpart, numhits),
        _result(2, numhits),
        _resultptr(0)
    { }

    inline HitCollector &getHandle(partid_t partid) { return
        _collectors[partid]; }
    /**
     * Merge the hits from partition with identifier specified as
         argument.
     */
    void mergeHits(partid_t);
    void printHits(void) const;

private:
    std::vector<HitCollector> _collectors;
    std::vector<HitVector> _result;
    unsigned int _resultptr;
};

} // end vsim

#endif
```

C.1.27 resultbuffer.cpp

```
#include <iostream>
#include "resultbuffer.h"
#include "hitvector.h"

namespace vsim {

void
ResultBuffer::mergeHits(partid_t part)
{
    HitVector &op(_result[_resultptr]);
    HitVector &res(_result[((_resultptr + 1) % 2)]);

    res.merge(op, _collectors[part].getResultBuffer());
    op.reset();
    _resultptr = ((_resultptr + 1) % 2);
}

void
ResultBuffer::printHits(void) const
{
    std::cout << "Complete hits: \n";
    const HitVector &result(_result[_resultptr]);
    result.printHits();
}

} // end vsim
```

C.1.28 simendian.h

```
/*-
 * Copyright (c) 2002 Thomas Moestl <tmm@FreeBSD.org>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above
     copyright
 *    notice, this list of conditions and the following disclaimer in
    the
 *    documentation and/or other materials provided with the
    distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS''
    AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
    THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE
    LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
    CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
    GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
    INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
     STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
    ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
     OF
 * SUCH DAMAGE.
 *
 * $FreeBSD$
 */

#ifndef _SYS_ENDIAN_H_
#define _SYS_ENDIAN_H_

#include <stdint.h>
#include <ctype.h>

#define  byteswap64(x) (((x) >> 56) | (((x) >> 40) & 0xff00) | (((x)
    >> 24) & 0xff0000) | (((x) >> 8) & 0xff000000) | (((x) << 8) & ((
    uint64_t)0xff << 32)) | (((x) << 24) & ((uint64_t)0xff << 40)) |
    (((x) << 40) & ((uint64_t)0xff << 48)) | (((x) << 56)));

/*
 * Host to big endian, host to little endian, big endian to host, and
    little
 * endian to host byte order functions as detailed in byteorder(9).
 */
#if __BYTE_ORDER == __LITTLE_ENDIAN
#define  vsimhtobe64(x) byteswap64((x))
#define  vsimbe64toh(x) byteswap64((x))
#else /* _BYTE_ORDER != _LITTLE_ENDIAN */
#define  vsimhtobe64(x) ((uint64_t)(x))
#define  vsimbe64toh(x) ((uint64_t)(x))
#endif /* _BYTE_ORDER == _LITTLE_ENDIAN */
#endif   /* _SYS_ENDIAN_H_ */
```

C.1.29 spinlock.h

```
#ifndef _VSIM_SPINLOCK_H_
#define _VSIM_SPINLOCK_H_
#include <pthread.h>

namespace vsim {

/**
 * A SpinLock is a wrapper around POSIX spinlocks. Synchronization
     primitive
 * that can be used if locks should be hold for a short period.
 */
class SpinLock {
public:
    SpinLock();
    virtual ~SpinLock();
    void lock(void);
    void unlock(void);
protected:
    pthread_spinlock_t _lock;
};

} // end vsim

#endif
```

C.1.30 spinlock.cpp

```
#include <pthread.h>
#include "spinlock.h"

namespace vsim {

SpinLock::SpinLock()
{
    pthread_spin_init(&_lock, 0);
}

SpinLock::~SpinLock()
{
    pthread_spin_destroy(&_lock);
}

void
SpinLock::lock(void)
{
    pthread_spin_lock(&_lock);
}

void
SpinLock::unlock(void)
{
    pthread_spin_unlock(&_lock);
}

} // end vsim
```

C.1.31 task.h

```
#ifndef _VSIM_TASK_H_
#define _VSIM_TASK_H_
#include <boost/shared_ptr.hpp>
#include "query.h"

namespace vsim {

/**
 * The Task class is an interface towards a runnable entity in the
     Vespa
 * simulator. A class implementing this interface may be scheduled by
     the
 * threadpool class.
 */
class Task {
public:
    typedef boost::shared_ptr<Task> SP;
    virtual ~Task() {}

    virtual void run(void) = 0;
    virtual void shutdown(void) = 0;
    virtual bool getPreferredCPU(cpu_set_t &) = 0;
};

} // end vsim

#endif
```

C.1.32 thread.h

```
#ifndef _VSIM_THREAD_H_
#define _VSIM_THREAD_H_
#include <pthread.h>
#include "task.h"
#include "cond.h"

namespace vsim {

class ThreadPool;

/**
 * A Thread is the internal wrappers around POSIX threads used by the
     threadpool
 * to dispatch the runnable entities implementing the Task interface.
 */
class Thread {
public:
    typedef boost::shared_ptr<Thread> SP;

    Thread();
    ~Thread();
    void join(void);

    /**
```

```
     * Dispatch a task to this thread. If the thread is busy with
         another task and
     * the task not started, return false.
     */
    bool dispatch(Task::SP);
    void run(void);
    static void *trampoline(void *);
    friend class ThreadPool;
private:
    pthread_t _handle;
    Cond _dispatched;
    bool _finished;
    bool _busy;
    bool _started;
    pid_t _tid;
    cpu_set_t _set;
    Task::SP _task;
};

} // end vsim

#endif
```

C.1.33 thread.cpp

```
#include <iostream>
#include <errno.h>
#include <string.h>
#include "thread.h"
#include "threadpool.h"
#include "task.h"

namespace vsim {

Thread::Thread() :
    _finished(false),
    _busy(false),
    _started(false)
{
}

Thread::~Thread()
{
}

void
Thread::join(void)
{
    _dispatched.lock();
    _finished = true;
    _dispatched.notify();
    _dispatched.unlock();
    if (_started)
        pthread_join(_handle, NULL);
}
```

```
bool
Thread::dispatch(Task::SP t)
{
    _dispatched.lock();
    if (_busy) {
        _dispatched.unlock();
        return (false);
    }
    // First time this thread was picked
    if (!_started) {
        _started = true;
        pthread_create(&_handle, NULL, Thread::trampoline, this);
    }
    _task = t;
    _busy = true;
    _dispatched.notify();
    _dispatched.unlock();
    return (true);
}

void
Thread::run(void)
{
    _tid = (pid_t)syscall(186); // gettid x86-64
    if (sched_getaffinity(_tid, sizeof(_set), &_set) != 0) {
        std::cout << "Error getting initial affinity of thread " <<
            _tid << ": "
            << strerror(errno) << std::endl;
    }
    _dispatched.lock();
    while (!_finished) {
        while (!_busy) {
            if (_finished) {
                break;
            }
            _dispatched.wait();
        }
        _dispatched.unlock();
        if (_finished)
            break;
        cpu_set_t tset;
        if (_task->getPreferredCPU(tset)) {
            // Set preferred affinity of the running task
            if (sched_setaffinity(_tid, sizeof(tset), &tset) != 0)
                std::cout << "Error setting affinity of thread " <<
                    _tid << std::endl;
        } else {
            // Reset affinity settings of this thread
            if (sched_setaffinity(_tid, sizeof(_set), &_set) != 0)
                std::cout << "Error setting affinity of thread " <<
                    _tid << std::endl;
        }
        _task->run();
        _dispatched.lock();
        _busy = false;
        _task.reset();
    }
}

void *
Thread::trampoline(void *arg)
{
    Thread *t = (Thread *)arg;
    t->run();
    return (NULL);
}

} // end vsim
```

C.1.34 threadpool.h

```
#ifndef _VSIM_THREADPOOL_H_
#define _VSIM_THREADPOOL_H_
#include <vector>
#include "task.h"
#include "thread.h"

namespace vsim {

/**
 * A ThreadPool implements a set of threads that may be assigned a
     task and
 * run.
 */
class ThreadPool {
public:
    ThreadPool(unsigned int);
    ~ThreadPool();
    int runThread(Task::SP);
private:
    std::vector<Thread::SP> _pool;
    unsigned int _poolsize;
};

} // end vsim

#endif
```

C.1.35 threadpool.cpp

```
#include <iostream>
#include <vector>
#include <pthread.h>
#include "threadpool.h"
#include "thread.h"
#include "task.h"

namespace vsim {

ThreadPool::ThreadPool(unsigned int size) : _poolsize(size)
{
    for (unsigned int i = 0; i < _poolsize; i++) {
        Thread *t = new Thread();
        _pool.push_back(Thread::SP(t));
    }
}

ThreadPool::~ThreadPool()
{
    for (unsigned int i = 0; i < _poolsize; i++) {
        Thread &t = (*_pool.at(i));
        t.join();
    }
}

int
ThreadPool::runThread(Task::SP task)
{
    while (1) {
            for (std::vector<Thread::SP>::iterator iter = _pool.begin
                ();
                 iter != _pool.end(); ++iter) {
                if ((*iter)->dispatch(task)) {
                    return 0;
            }
        }
    }
    std::cout << "Found no thread to dispatch!\n";
    return -1;
}

} // end vsim
```

C.1.36 types.h

```
#ifndef _VSIM_TYPES_H_
#define _VSIM_TYPES_H_
#include <boost/shared_ptr.hpp>
#include <vector>
#include <list>
#include <stdint.h>

namespace vsim {

typedef unsigned long docid_t;
typedef unsigned long partid_t;
typedef uint64_t rank_t;
typedef std::vector<docid_t> DocidVector;
typedef std::list<docid_t> DocidList;
enum VSimMode { TIMEMODE, QUERYMODE };
enum SlotDistribution { BINOMIAL, GAUSS, POISSON, GEOMETRIC };
class Slot {
public:
    Slot(docid_t f, docid_t s, double p) : first(f), second(s), prob(p
        ) {}
    docid_t first;
    docid_t second;
    double prob;
};
typedef std::vector<Slot> PartitionRange;
typedef std::vector<PartitionRange> PartitionSpec;
typedef boost::shared_ptr<PartitionSpec> PartitionSpecSP;

typedef std::pair<docid_t, unsigned long> Hit;
typedef docid_t PartitionHit;
typedef std::vector<PartitionHit> HitStat;

typedef unsigned long attr_t;
typedef attr_t * attrptr_t;
typedef std::vector<attrptr_t> AttributeVector;
typedef std::vector<unsigned long> AttributeIndexVector;

} // end vsim

#endif
```

C.1.37 vsimconfig.h

```
#ifndef _VSIM_VSIMCONFIG_H_
#define _VSIM_VSIMCONFIG_H_
#include <string>
#include "types.h"

namespace vsim {

/**
 * The VSimConfig class handles configuration for the simulator run
     time. It can
 * be used to inspect the configuration status from the system.
 */
class VSimConfig
{
public:
    VSimConfig(docid_t numpart = 1,
               docid_t numslots = 1,
               unsigned long maxconn = 1,
               unsigned long clients = 1,
               docid_t maxdocid = 10000,
               docid_t numhits = 1000,
               docid_t hitheapsize = 100,
               unsigned long numattr = 0,
               AttributeIndexVector attrrange = AttributeIndexVector
                   (0),
               SlotDistribution slotdist = BINOMIAL,
               unsigned long simtime = 10,
               unsigned long numrounds = 1,
               const std::string &qrlog = "") :
        _numpart(numpart),
        _numslots(numslots),
        _maxconn(maxconn),
        _clients(clients),
        _maxdocid(maxdocid),
        _numhits(numhits),
        _hitheapsize(hitheapsize),
        _numattr(numattr),
        _attrindices(attrrange),
        _slotdist(slotdist),
        _simtime(simtime),
        _numrounds(numrounds),
        _qrlog(qrlog)
    {
    }

    docid_t getNumPartitions(void) const { return _numpart; }
    docid_t getNumSlots(void) const { return _numslots; }
    unsigned int getMaxConnections(void) const { return _maxconn; }
    unsigned long getNumClients(void) const { return _clients; }
    docid_t getMaxDocId(void) const { return _maxdocid; }
    docid_t getNumHits(void) const { return _numhits; }
    docid_t getHitHeapSize(void) const { return _hitheapsize; }
    SlotDistribution getSlotDistribution(void) const { return
        _slotdist; }
    const std::string &getQRSLogFile(void) const { return _qrlog; }
    PartitionSpecSP createPartitionSpec(void);
    void exportConfig(const std::string &);
    unsigned long getSimTime(void) const { return _simtime; }
    unsigned long getNumberOfRounds(void) const { return _numrounds; }
    unsigned int getNumAttributes(void) const { return _numattr; }
    const AttributeIndexVector &getAttributeIndices(void) const {
        return _attrindices; }

private:
    docid_t _numpart;
    docid_t _numslots;
    unsigned int _maxconn;
    unsigned long _clients;
    docid_t _maxdocid;
    docid_t _numhits;
    docid_t _hitheapsize;
    unsigned int _numattr;
    AttributeIndexVector _attrindices;
    SlotDistribution _slotdist;
    unsigned long _simtime;
    unsigned long _numrounds;
    std::string _qrlog;
};

} // end vsim

#endif
```

C.1.38 vsimconfig.cpp

```
#include <iostream>
#include <unistd.h>
#include "vsimconfig.h"
#include "vsimstore.h"
#include "querygenerator.h"
#include "misc.h"
```

```
#include <boost/math/distributions/normal.hpp>
#include <boost/math/distributions/poisson.hpp>
#include <boost/math/distributions/negative_binomial.hpp>

namespace vsim {

using boost::math::pdf;
using boost::math::cdf;

#define calcProb(s, start, end) (cdf((s), (end)) - cdf((s), (start)))


PartitionSpecSP
VSimConfig::createPartitionSpec(void)
{
    PartitionSpecSP ps(new PartitionSpec);

    docid_t maxdocid = getMaxDocId();
    partid_t numpart = getNumPartitions();
    docid_t numslots = getNumSlots();
    docid_t num_per_part = (maxdocid / numpart);
    docid_t num_per_slot = (num_per_part / numslots);

    double totalprob = 0.0;
    partid_t i = 0;
    while (maxdocid > 0) {
        docid_t start = i * num_per_part;
        docid_t range = (num_per_part >= maxdocid) ? maxdocid :
            num_per_part;
        if (i == (numpart - 1))
            range = maxdocid;
        PartitionRange pr;
        docid_t slotid = range;
        docid_t slotnum = 0;
        while (slotid > 0) {
            docid_t slotstart = (slotnum * num_per_slot) + start;
            docid_t slotrange = (num_per_slot >= slotid) ? slotid :
                num_per_slot;
            if (slotnum == (numslots - 1))
                slotrange = slotid;
            docid_t slotend = slotstart + slotrange;
            // TODO: Support different
            double prob = 0.0;
            switch (_slotdist) {
            default: // Fallthrough, default is BINOMIAL
            case BINOMIAL:
                prob = (double)slotrange/(double)getMaxDocId();
                break;
            case GAUSS:
                {
                    boost::math::normal gauss(getMaxDocId() / 2,
                        getMaxDocId() / 4);
                    prob = calcProb(gauss, slotstart, slotend);
                }
                break;
            case POISSON:
                {
                    boost::math::poisson poisdist(getMaxDocId() / 2);
                    prob = calcProb(poisdist, slotstart, slotend);
                }
                break;
            case GEOMETRIC:
                {
                    // First parameter gives average for geometric,
                        and second
                    // gives steepness of the curve
                    boost::math::negative_binomial negbinom(1, 0.99);
                    prob = calcProb(negbinom, ((double)slotstart / (
                        double)getMaxDocId()), ((double)slotend / (
                        double)getMaxDocId()));
                }
            }
            totalprob += prob;
            pr.push_back(Slot(slotstart, slotend, prob));
            slotid -= slotrange;
            slotnum++;
        }
        (*ps).push_back(pr);
        maxdocid -= range;
        i++;
    }
    // Spread the rest of the probabilities on the rest of the slots
        according
    // to the ration compared to totalprob!
    double remaining = 1.0 - totalprob;
    double newtotal = 0.0;
    for (docid_t i = 0; i < (*ps).size(); i++) {
        for (docid_t j = 0; j < (*ps)[i].size(); j++) {
            double prob = (*ps)[i][j].prob;
            double ratio = prob / totalprob;
            (*ps)[i][j].prob += (remaining * ratio);
            newtotal += (*ps)[i][j].prob;
        }
    }
    return (ps);
}

} // end vsim
```

C.1.39 vsim.h

```
#ifndef _VSIM_VSIM_H_
#define _VSIM_VSIM_H_
#include <string>
#include "vsimreport.h"
#include "vsimstat.h"
#include "vsimconfig.h"
#include "vsimstore.h"
#include "querygenerator.h"

namespace vsim {

/**
```

```
 * The VSim class implements the simulator run-time. The class may
     generate a
 * report of the simulator run.
 */
class VSim {
public:
    VSim(VSimConfig &vcfg) : _vcfg(vcfg) { }
    ~VSim() { }

    /**
     * Dispatch a set of threads (equal to the number of partitions)
         to seed the
     * data storage in order to get memory optimally allocated to each
          CPUs DRAM
     * controller.
     */
    void seed(VSimStore &);
    QueryGenerator::SP createQueryGenerator(QueryQueue &,
        const PartitionSpecSP &, unsigned int);

    void run(const std::string &);
    VSimReport doRun(VSimStore &);
private:
    VSimConfig &_vcfg;
};

} // end vsim

#endif
```

C.1.40 vsim.cpp

```
#include <iostream>
#include <time.h>

#include "vsim.h"
#include "vsimstat.h"
#include "query.h"
#include "querygenerator.h"
#include "queryhandler.h"
#include "fetcher.h"
#include "threadpool.h"
#include "vsimstore.h"
#include "vsimconfig.h"
#include "vsimprofiler.h"
#include "vsimseeder.h"

namespace vsim {

void
VSim::run(const std::string &samplefile)
{
    VSimStore ds(_vcfg);
    VSimProfiler::stop();
    for (unsigned int i = 0; i < _vcfg.getNumberOfRounds(); i++) {
        seed(ds);
        ds.memlock();
        VSimProfiler::start();
        VSimReport report = doRun(ds);
        VSimProfiler::stop();
        ds.memunlock();
        report.printReport();
        // Output samples
        if (samplefile.size() > 0) {
            report.exportReport(samplefile.c_str());
        }
        sleep(1);
    }
}

void
VSim::seed(VSimStore &ds)
{
    std::vector<VSimSeeder::SP> seeders;
    PartitionSpecSP ps = _vcfg.createPartitionSpec();
    ThreadPool tp(ps->size() + 1);

    for (partid_t i = 0; i < (*ps).size(); i++) {
        PartitionRange &pr((*ps)[i]);
        docid_t start = pr[0].first;
        docid_t end = pr[pr.size() - 1].second;
        VSimSeeder::SP vssp(new VSimSeeder(ds, start, end, i));
        seeders.push_back(vssp);
        tp.runThread(vssp);
    }
    for (partid_t i = 0; i < seeders.size(); i++) {
        seeders[i]->shutdown();
    }
}

QueryGenerator::SP
VSim::createQueryGenerator(QueryQueue &queue, const PartitionSpecSP &
    spec,
    unsigned int id)
{
    std::string qrlog(_vcfg.getQRSLogFile());
    if (qrlog.size() > 0) {
        char buf[128];
        if (id < 10)
            snprintf(buf, 128, "0%u", id);
        else
            snprintf(buf, 128, "%u", id);
        qrlog += buf;
        std::cout << qrlog << std::endl;
        return (QueryGenerator::SP(new QRSLogQueryGenerator(queue,
            spec,
            _vcfg.getMaxDocId(), _vcfg.getHitHeapSize(), qrlog)));
    }
    return (QueryGenerator::SP(new StaticQueryGenerator(queue, spec,
        _vcfg.getMaxDocId(), _vcfg.getHitHeapSize(), _vcfg.getNumHits
            ())));
}
```

```
VSimReport
VSim::doRun(VSimStore &ds)
{
    VSimStat st(_vcfg);
    PartitionSpecSP ps = _vcfg.createPartitionSpec();
    ThreadPool tp(300);

    std::cout << "Setting up simulation\n";
    std::vector<PartitionQueue> partitions;
    std::vector<Fetcher::SP> fetchers;
    std::vector<QueryGenerator::SP> clients;
    QueryQueue qq(_vcfg.getMaxConnections());
    std::vector<QueryHandler::SP> handlers;

    // Create partitions
    for (partid_t i = 0; i < _vcfg.getNumPartitions(); i++) {
        partitions.push_back(PartitionQueue(i));
    }

    // Create clients
    for (unsigned int i = 0; i < _vcfg.getNumClients(); i++) {
        QueryGenerator::SP qgsp(createQueryGenerator(qq, ps, i));
        clients.push_back(qgsp);
    }

    // The number of fetchers per partition depends on the number of
        partitions.
    unsigned int fetchers_per_part = _vcfg.getMaxConnections();
    for (partid_t i = 0; i < partitions.size(); i++) {
        for (unsigned int j = 0; j < fetchers_per_part; j++) {
            VSimTime vst;
            vst.recordTime();
            Fetcher::SP fsp(new Fetcher(ds, partitions[i], i,
                (vst.getTimeNanoSeconds() + j + i), j, _vcfg.
                    getMaxDocId()));
            tp.runThread(fsp);
            fetchers.push_back(fsp);
        }
    }

    // Create query queue and queryhandlers
    for (unsigned int i = 0; i < _vcfg.getMaxConnections(); i++) {
        QueryHandler::SP qhsp(new QueryHandler(partitions, st, qq));
        tp.runThread(qhsp);
        handlers.push_back(qhsp);
    }

    std::cout << "Let threads settle down\n";
    sleep(2);
    std::cout << "Starting simulation\n";
    st.startSimulation();

#ifdef AUTOPARTITIONING
#define PARTCHECKNUM   100
    unsigned int partcheck = PARTCHECKNUM;
#endif
```

```
    // Start all clients
    for (unsigned int i = 0; i < clients.size(); i++) {
        tp.runThread(clients[i]);
    }
    st.sleepUntilFinished();
#ifdef AUTOPARTITIONING
            if (partitions.size() > 1) {
                partcheck--;
                if (partcheck <= 0) {
                    ps = st.updatePartitionSpec(ps);
                    partcheck = PARTCHECKNUM;
                }
            }
#endif
    std::cout << "Stopping simulation\n";
    for (unsigned int i = 0; i < clients.size(); i++) {
        clients[i]->shutdown();
    }
    for (unsigned int i = 0; i < clients.size(); i++) {
        clients[i]->wait();
    }
    qq.waitUntilEmpty();
    qq.waitUntilEmpty();
    qq.shutdown();
    for (unsigned int i = 0; i < handlers.size(); i++) {
        handlers[i]->shutdown();
    }
    for (partid_t i = 0; i < partitions.size(); i++) {
        partitions[i].waitUntilEmpty();
        partitions[i].shutdown();
    }
    for (partid_t i = 0; i < fetchers.size(); i++) {
        fetchers[i]->shutdown();
    }
    st.stopSimulation();
    std::cout << "Simulation finished\n";
    return st.generateReport(*ps);
}

} // end vsim
```

C.1.41 vsimprofiler.h

```
#ifndef _VSIMPROFILER_H_
#define _VSIMPROFILER_H_

namespace vsim {

/**
 * The VSimProfiler class is a simple interface towards oprofile by
    wrapping
 * system() commands.
 */
class VSimProfiler {
public:
    static void start(void);
    static void stop(void);
};

} // end vsim

#endif
```

C.1.42 vsimprofiler.cpp

```
#include <cstdlib>
#include <iostream>
#include "vsimprofiler.h"

namespace vsim {

void
VSimProfiler::start(void)
{
#ifdef OPROFILE
    if (system("opcontrol --start") == -1)
        std::cout << "Error starting OProfile" << std::endl;
#endif
}

void
VSimProfiler::stop(void)
{
#ifdef OPROFILE
    if (system("opcontrol --stop") == -1)
        std::cout << "Error stopping OProfile" << std::endl;
#endif
}

} // namespace vsim
```

C.1.43 vsimreport.h

```
#ifndef _VSIM_VSIMREPORT_H_
#define _VSIM_VSIMREPORT_H_
#include <vector>
#include <string>
#include <fstream>
#include <stdint.h>
#include "types.h"

namespace vsim {

/**
 * The report class describes a simulation run. The report may be
 * exported to file, or imported from a file.
 */
class VSimReport
{
public:
    void exportReport(const char *);
    void importReport(const char *);
    int importReportData(std::ifstream &);
    void printReport(void);
    static SlotDistribution distIntToName(uint64_t);
    static uint64_t distNameToInt(SlotDistribution);

    uint64_t numpart;                // Number of partitions
    uint64_t numhandlers;            // Number of query handlers
    uint64_t numhits;                // Number of hits per query (if
        not using qrlog)
    uint64_t numclients;             // Number of clients
    uint64_t heapsize;               // Number of hits returned
    uint64_t numdocs;                // Number of documents stored
    uint64_t numattr;                // Number of attributes per
        document
    uint64_t numslots;               // Number of slots per partition
    uint64_t distribution;           // Which probability distribution
        used
    uint64_t numqueries;             // Number of queries finished
    uint64_t simtime;                // Simulation time, in seconds
    uint64_t cputime;                // CPU time, in seconds
    uint64_t qps;                    // Queries per simulation second
    uint64_t qps_weighted;           // Queries per CPU second
    uint64_t time_queue;             // Time spent in query queue
    uint64_t time_exec;              // Time spent on query execution
    uint64_t time_fetcher;           // Time spent in fetcher
    uint64_t time_partitionqueue;    // Time spent in partition queue
    uint64_t time_handler;           // Time spent in query handler
    double fraction_handler;         // Fraction of execution time
        spent in handler
    HitStat hitstats;                // Per partition hit counts
    std::vector<uint64_t> partitionqueuetimes; // Per partition queue
        times
    std::vector<uint64_t> fetchertimes;        // Per partition
        fetcher execution times
    uint64_t numindices;                       // Number of attribute
        indices
    AttributeIndexVector attributeindices;     // Attributes used for
        calculating rank
    std::vector<uint64_t> partitionscheme;     // Number of slots per
        partition at the end
};

std::vector<VSimReport> importReportBatch(const char *);

} // end vsim

#endif
```

C.1.44 vsimreport.cpp

```
#include <string>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstring>
```

```
#include "vsimreport.h"
#include "simendian.h"
#include "misc.h"

namespace vsim {

#define WRITE64(fd, var) do {                   \
    uint64_t tmp = vsimhtobe64((var));          \
    (fd).write((char *)&tmp, sizeof(tmp));      \
} while (0)

void
VSimReport::exportReport(const char *outfile)
{
    std::ofstream fd(outfile, std::ios::app | std::ios::binary);

    // Write configuration options first
    WRITE64(fd, numpart);
    WRITE64(fd, numhandlers);
    WRITE64(fd, numhits);
    WRITE64(fd, numclients);
    WRITE64(fd, heapsize);
    WRITE64(fd, numdocs);
    WRITE64(fd, numattr);
    WRITE64(fd, numslots);
    WRITE64(fd, distribution);
    WRITE64(fd, numqueries);
    WRITE64(fd, simtime);
    WRITE64(fd, cputime);
    WRITE64(fd, qps);
    WRITE64(fd, qps_weighted);
    WRITE64(fd, time_queue);
    WRITE64(fd, time_exec);
    WRITE64(fd, time_fetcher);
    WRITE64(fd, time_partitionqueue);
    WRITE64(fd, time_handler);
    fd.write((char *)&fraction_handler, sizeof(fraction_handler));
    for (docid_t i = 0; i < numpart; i++) {
        uint64_t val = hitstats[i];
        WRITE64(fd, val);
        WRITE64(fd, partitionqueuetimes[i]);
        WRITE64(fd, fetchertimes[i]);
    }
    WRITE64(fd, numindices);
    for (unsigned long i = 0; i < attributeindices.size(); i++) {
        uint64_t val = attributeindices[i];
        WRITE64(fd, val);
    }
    for (unsigned long i = 0; i < partitionscheme.size(); i++) {
        uint64_t val = partitionscheme[i];
        WRITE64(fd, val);
    }
    fd.close();
}

#define READ64(fd, var) do {                    \
    uint64_t tmp = 0;                           \
    (fd).read((char *)&tmp, sizeof(tmp));   \
    if ((fd).eof())                         \
        return -1;                          \
    (var) = vsimbe64toh(tmp);               \
} while (0)

int
VSimReport::importReportData(std::ifstream &fd)
{
    READ64(fd, numpart);
    READ64(fd, numhandlers);
    READ64(fd, numhits);
    READ64(fd, numclients);
    READ64(fd, heapsize);
    READ64(fd, numdocs);
    READ64(fd, numattr);
    READ64(fd, numslots);
    READ64(fd, distribution);
    READ64(fd, numqueries);
    READ64(fd, simtime);
    READ64(fd, cputime);
    READ64(fd, qps);
    READ64(fd, qps_weighted);
    READ64(fd, time_queue);
    READ64(fd, time_exec);
    READ64(fd, time_fetcher);
    READ64(fd, time_partitionqueue);
    READ64(fd, time_handler);
    fd.read((char *)&fraction_handler, sizeof(fraction_handler));
    for (docid_t i = 0; i < numpart; i++) {
        uint64_t hit;
        READ64(fd, hit);
        uint64_t pqt;
        READ64(fd, pqt);
        uint64_t ft;
        READ64(fd, ft);
        hitstats.push_back(hit);
        partitionqueuetimes.push_back(pqt);
        fetchertimes.push_back(ft);
    }
    READ64(fd, numindices);
    for (unsigned long i = 0; i < numindices; i++) {
        uint64_t val = 0;
        READ64(fd, val);
        attributeindices.push_back(val);
    }
    for (docid_t i = 0; i < numpart; i++) {
        uint64_t val;
        READ64(fd, val);
        partitionscheme.push_back(val);
    }
    return 0;
}

void
VSimReport::importReport(const char *infile)
{
    std::ifstream fd(infile, std::ios::in | std::ios::binary);
    importReportData(fd);
    fd.close();
}

std::vector<VSimReport>
importReportBatch(const char *infile)
{
    std::vector<VSimReport> reports;
    std::ifstream fd(infile, std::ios::in | std::ios::binary);
    while (!fd.eof()) {
        VSimReport report;
        if (report.importReportData(fd) != 0)
            break;
        reports.push_back(report);
    }
    fd.close();
    return (reports);
}

SlotDistribution
VSimReport::distIntToName(uint64_t val)
{
    return ((SlotDistribution)val);
}

uint64_t
VSimReport::distNameToInt(SlotDistribution sd)
{
    return ((uint64_t)sd);
}

void
VSimReport::printReport(void)
{
    std::cout << "Configuration:\t\t" << std::endl;
    std::cout << "#partitions:\t\t" << numpart << std::endl;
    std::cout << "#handlers:\t\t" << numhandlers << std::endl;
    std::cout << "#hits:\t\t\t" << numhits << std::endl;
    std::cout << "#clients:\t\t" << numclients << std::endl;
    std::cout << "heapsize:\t\t" << heapsize << std::endl;
    std::cout << "numdocs:\t\t" << numdocs << std::endl;
    std::cout << "num attributes:\t\t" << numattr << std::endl;
    std::cout << "attributes used:\t";
    for (unsigned int i = 0; i < attributeindices.size(); i++) {
        std::cout << attributeindices[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "#slots/part:\t\t" << numslots << std::endl;
    std::cout << "distribution:\t\t" << slotDistEnumToText(
        distIntToName(distribution)) << std::endl;
    std::cout << "#queries:\t\t" << numqueries << std::endl;
    std::cout << "simtime:\t\t" << simtime << std::endl;
    std::cout << "cputime:\t\t" << cputime << std::endl;
    std::cout << std::endl;

    const unsigned int NUMCOLUMNS = 8;
    std::cout << "Results:" << std::endl;
    const char menu[NUMCOLUMNS][128] = { "Queries/sec", "Weighted_Q/s"
        , "time_queue", "time_execution", "time_fetcher", "
        time_partitionqueue", "time_handler", "\%handler" };
    std::vector<int> widths(NUMCOLUMNS);
    for (unsigned int i = 0; i < NUMCOLUMNS; i++) {
        widths[i] = strlen(menu[i]);
        if (i > 0)
            widths[i]++;
        std::cout << std::setw(widths[i]) << menu[i];
    }
    std::cout << std::endl;
    std::cout << std::setw(widths[0]) << qps;
    std::cout << std::setw(widths[1]) << qps_weighted;
    std::cout << std::setw(widths[2]) << time_queue;
    std::cout << std::setw(widths[3]) << time_exec;
    std::cout << std::setw(widths[4]) << time_fetcher;
    std::cout << std::setw(widths[5]) << time_partitionqueue;
    std::cout << std::setw(widths[6]) << time_handler;
    std::cout << std::setw(widths[7]) << std::setprecision(2) <<
        fraction_handler;
    std::cout << std::endl;
    std::cout << std::endl;
    std::cout << "Per partition stats: " << std::endl;
    for (docid_t i = 0; i < numpart; i++) {
        std::cout << i << " H: " << hitstats[i] << " P: " <<
            partitionqueuetimes[i] << " F: " << fetchertimes[i] << std
            ::endl;
    }
}

} // end vsim
```

C.1.45 vsimseeder.h

```
#ifndef _VSIMSEEDER_H_
#define _VSIMSEEDER_H_

#include "types.h"
#include "vsimstore.h"
#include "task.h"
#include "barrier.h"

namespace vsim {

/**
 * The VSimSeeder task seeds a range of the document store attributes
     in order
 * to place data close to a local DRAM controller.
 */
class VSimSeeder : public Task {
public:
    typedef boost::shared_ptr<VSimSeeder> SP;
    VSimSeeder(VSimStore &, docid_t, docid_t, int);
    void run(void);
    void shutdown(void);
    bool getPreferredCPU(cpu_set_t &);

private:
    VSimStore &_store;
    docid_t _start;
    docid_t _end;
    int _whichCPU;
    Barrier _done;
};

} // end vsim

#endif
```

C.1.46 vsimseeder.cpp

```
#include <sys/time.h>
#include <time.h>
#include "task.h"
#include "vsimstore.h"
#include "vsimseeder.h"

namespace vsim {

VSimSeeder::VSimSeeder(VSimStore &store, docid_t start, docid_t end,
    int which)
    : _store(store),
      _start(start),
      _end(end),
      _whichCPU(which),
      _done(2)
{ }

void
VSimSeeder::run(void)
{
    _store.seedRange(time(0), _start, _end);
    _done.wait();
}

void
VSimSeeder::shutdown(void)
{
    _done.wait();
}

bool
VSimSeeder::getPreferredCPU(cpu_set_t &set)
{
    CPU_ZERO(&set);
    CPU_SET((_whichCPU % NUMCPU), &set);
    return (true);
}

} // end vsim
```

C.1.47 vsimstat.h

```
#ifndef _VSIM_VSIMSTAT_H_
#define _VSIM_VSIMSTAT_H_
#include <sys/time.h>
#include <sys/resource.h>
#include <stdint.h>
#include "vsimconfig.h"
#include "vsimreport.h"
#include "vsimtime.h"
#include "mutex.h"
#include "cond.h"
#include "types.h"
#include "misc.h"

namespace vsim {

/**
 * The VSimStat class keeps statistics. The statistics object can be
    used by the
 * internal query classes to report relevant information. These
    results can be
 * used to create a report which can be stored to disk.
 *
 * The class may also generate a new partition specification from the
    hit
 * statistics of each partition in order to handle dynamic workloads.
 */
class VSimStat
{
public:
    VSimStat(VSimConfig &);
    void report(std::vector<uint64_t> &, std::vector<uint64_t> &,
        uint64_t, uint64_t, uint64_t);
    void startSimulation(void);
    void stopSimulation(void);
    void sleepUntilFinished(void);
    void reportHits(HitStat);
    VSimReport generateReport(const PartitionSpec &);
    PartitionSpecSP updatePartitionSpec(const PartitionSpecSP &);

private:
    VSimConfig &_vcfg;
    Mutex _hslock;
    HitStat _hs;
    HitStat _parths;
    uint64_t _numfinished;
    std::vector<uint64_t> _fetchertimes;
    std::vector<uint64_t> _partitionqueuetimes;
    uint64_t _totalmaxfetchertime;
    uint64_t _totalqueryqueuetime;
    uint64_t _totalhandlertime;
    VSimTime _startSim;
    VSimTime _stopSim;
    struct rusage _rusage_start;
    struct rusage _rusage_end;
};
} // end vsim

#endif
```

C.1.48 vsimstat.cpp

```
#include <iostream>
#include <iomanip>
#include <inttypes.h>
#include <assert.h>
#include <numeric>
#include <cstring>

#include "types.h"
#include "atomic.h"
#include "vsimstat.h"
#include "vsimconfig.h"
#include "misc.h"

namespace vsim {

using namespace vespalib;

VSimStat::VSimStat(VSimConfig &vcfg) :
    _vcfg(vcfg),
    _hs(vcfg.getNumPartitions(), 0),
    _parths(vcfg.getNumPartitions(), 0),
    _numfinished(0),
    _fetchertimes(vcfg.getNumPartitions(), 0),
    _partitionqueuetimes(vcfg.getNumPartitions(), 0),
    _totalmaxfetchertime(0),
    _totalqueryqueuetime(0),
    _totalhandlertime(0)
{ }

void
VSimStat::report(std::vector<uint64_t> &fetchertimes, std::vector<
    uint64_t> &partitionqueuetimes, uint64_t maxfetchertime, uint64_t
    queuetime, uint64_t handlertime)
{
    Atomic::postInc(&_numfinished);
    for (partid_t i = 0; i < fetchertimes.size(); i++) {
        Atomic::add(&_fetchertimes[i], fetchertimes[i]);
        Atomic::add(&_partitionqueuetimes[i], partitionqueuetimes[i]);
    }
    Atomic::add(&_totalmaxfetchertime, maxfetchertime);
    Atomic::add(&_totalqueryqueuetime, queuetime);
    Atomic::add(&_totalhandlertime, handlertime);
}

void
VSimStat::reportHits(HitStat hs)
{
    if (hs.size() != _hs.size()) {
        std::cout << "Cannot merge stats of different size, ours: " <<
             _hs.size() << " real: " << hs.size() << std::endl;
        return;
    }
    for (unsigned int i = 0; i < _hs.size(); i++) {
        _hs[i] += hs[i];
    }
    _hslock.lock();
    for (unsigned int i = 0; i < _parths.size(); i++) {
        _parths[i] += hs[i];
    }
    _hslock.unlock();
}

#include <cstdlib>

/**
 * Update partition spec based on hit statistics.
 */
PartitionSpecSP
VSimStat::updatePartitionSpec(const PartitionSpecSP &ps)
{
    _hslock.lock();
    HitStat hs(_parths);
    _hslock.unlock();

    assert((*ps).size() == hs.size());

    docid_t totalhit = 0;
    for (unsigned int i = 0; i < hs.size(); i++) {
        hs[i]++;
        totalhit += hs[i];
    }

    docid_t numslots = _vcfg.getNumSlots();
    docid_t idealhit = totalhit / hs.size();
    docid_t slothit = idealhit / numslots;
    std::vector<docid_t> numslots_per(hs.size(), numslots);
    PartitionSpecSP newspec(new PartitionSpec(*ps));

    for (unsigned int i = 0; i < hs.size(); i++) {
        // We need more slots, take one from our neighbour
        if (hs[i] < idealhit) {
            docid_t remainder = idealhit - hs[i];
            docid_t remainder_slots = remainder / slothit;
            for (unsigned int j = hs.size() - 1; j > i &&
                remainder_slots > 0; j--) {
                PartitionRange &mine((*newspec)[j]);
                docid_t available_slots = MIN(mine.size(),
                    remainder_slots);
                // Move these backwards
                PartitionRange &prev((*newspec)[j - 1]);
                prev.insert(prev.end(), mine.begin(), (mine.begin() +
                    available_slots));
                mine.erase(mine.begin(), (mine.begin() +
                    available_slots));
                numslots_per[j] -= available_slots;
                numslots_per[j - 1] += available_slots;
            }

        } else if (hs[i] > idealhit) {
            docid_t remainder = hs[i] - idealhit;
            docid_t remainder_slots = remainder / slothit;
            for (unsigned int j = i; j < hs.size() - 1; j++) {
                PartitionRange &mine((*newspec)[j]);
                PartitionRange &next((*newspec)[j + 1]);
                docid_t available_slots = MIN(remainder_slots, mine.
                    size());
                // Move these forward
                next.insert(next.begin(), (mine.end() - available_slots
                    ), mine.end());
                mine.erase((mine.end() - available_slots), mine.end());

                numslots_per[j] -= available_slots;
                numslots_per[j + 1] += available_slots;
            }
        }
        hs[i] = idealhit;
    }
    _hslock.lock();
    _parths = hs;
    _hslock.unlock();
    return (newspec);
}

void
VSimStat::sleepUntilFinished(void)
{
    VSimTime now;
    now.recordTime();
    unsigned long diff = (now.getTimeSeconds() - _startSim.
        getTimeSeconds());
    std::cout << "Sleeping for " << (_vcfg.getSimTime() - diff) << "
        seconds\n";
    sleep(_vcfg.getSimTime() - diff);
}

void
VSimStat::startSimulation(void)
{
    _startSim.recordTime();
    getrusage(RUSAGE_SELF, &_rusage_start);
}

void
VSimStat::stopSimulation(void)
{
    _stopSim.recordTime();
    getrusage(RUSAGE_SELF, &_rusage_end);
}

VSimReport
VSimStat::generateReport(const PartitionSpec &spec)
{
    uint64_t totalfetchertime = 0;
    uint64_t totalpartqueuetime = 0;
    for (unsigned long i = 0; i < _fetchertimes.size(); i++) {
        totalpartqueuetime += _partitionqueuetimes[i];
        totalfetchertime += _fetchertimes[i];
    }
    totalpartqueuetime /= _partitionqueuetimes.size();
    totalfetchertime /= _fetchertimes.size();

    VSimTime usertime_start(_rusage_start.ru_utime);
    VSimTime usertime_end(_rusage_end.ru_utime);
    VSimTime systime_start(_rusage_start.ru_stime);
    VSimTime systime_end(_rusage_end.ru_stime);

    VSimReport report;
    report.numpart = _fetchertimes.size();
    report.numhandlers = _vcfg.getMaxConnections();
    report.numhits = _vcfg.getNumHits();
    report.numclients = _vcfg.getNumClients();
    report.heapsize = _vcfg.getHitHeapSize();
    report.numdocs = _vcfg.getMaxDocId();
    report.numattr = _vcfg.getNumAttributes();
    report.numslots = _vcfg.getNumSlots();
    report.distribution = VSimReport::distNameToInt(_vcfg.
        getSlotDistribution());
    report.numqueries = _numfinished;
    report.simtime = _startSim.getDiffMilliSeconds(_stopSim) / 1000;
    report.cputime = (usertime_start.getDiffMilliSeconds(usertime_end)
         + systime_start.getDiffMilliSeconds(systime_end)) / 1000;
    report.qps = (_numfinished / report.simtime);
    report.qps_weighted = (_numfinished / report.cputime);
    report.time_queue = (_totalqueryqueuetime / _numfinished);
    report.time_exec = (_totalhandlertime / _numfinished);
    report.time_fetcher = (_totalmaxfetchertime / _numfinished);
    report.time_partitionqueue = (totalpartqueuetime / _numfinished);
    report.time_handler = report.time_exec - report.time_fetcher -
        report.time_partitionqueue;
    report.fraction_handler = (double)100.0 * (double)report.
        time_handler / (double)report.time_exec;
    report.hitstats = _hs;
    for (docid_t i = 0; i < report.hitstats.size(); i++) {
        report.partitionqueuetimes.push_back(_partitionqueuetimes[i] /
             _numfinished);
        report.fetchertimes.push_back(_fetchertimes[i] / _numfinished)
            ;
    }
    report.attributeindices = _vcfg.getAttributeIndices();
    report.numindices = report.attributeindices.size();
    for (docid_t i = 0; i < spec.size(); i++) {
        const PartitionRange &pr(spec[i]);
        report.partitionscheme.push_back(pr.size());
    }
    return (report);
}

} // end vsim
```

C.1.49 vsimstore.h

```
#ifndef _VSIM_VSIMSTORE_H_
#define _VSIM_VSIMSTORE_H_
#include <vector>
#include "vsimconfig.h"
#include "types.h"

namespace vsim {

/**
 * A VSimStore contains a set of attributes belonging to documents.
    The
 * attribute arrays containing attribute data are initialized and
    accessed
 * through this class when asking for the rank of a document.
 */
class VSimStore
{
public:
    VSimStore(const VSimConfig &);
    ~VSimStore();

    /**
     * Make sure that the attributes are not swapped to disk.
     */
    void memlock(void);
    void memunlock(void);

    /**
     * Generate the attribute values.
     */
    void seed(unsigned long);
    void seedRange(unsigned long, docid_t, docid_t);

    /**
     * Calculate the rank of a document.
     */
    rank_t getRank(docid_t) const;

    static const unsigned long MAXRANK = 1000000;
private:
    docid_t _maxdocid;
    AttributeIndexVector _attrindex;
    AttributeVector _attributes;
};

} // end vsim
#endif
```

C.1.50 vsimstore.cpp

```
#include <iostream>
#include <algorithm>
#include <sys/mman.h>
#include <boost/random/linear_congruential.hpp>
```

```
#include <boost/random/uniform_int.hpp>
#include <boost/random/variate_generator.hpp>
#include "vsimconfig.h"
#include "vsimstore.h"

#define CACHE_LINE_SIZE 64

namespace vsim {

typedef boost::minstd_rand base_generator_type;

VSimStore::VSimStore(const VSimConfig &vcfg) :
    _maxdocid(vcfg.getMaxDocId()),
    _attrindex(vcfg.getAttributeIndices())
{
    // Create memory area for all attributes
    for (unsigned int i = 0; i < vcfg.getNumAttributes(); i++) {
        attrptr_t astore = 0;
        if (posix_memalign((void **)&astore, CACHE_LINE_SIZE, (
            _maxdocid * sizeof(attr_t))) != 0) {
            std::cout << "Error allocating aligned memory for
                attributes" << std::endl;
            exit(-1);
        }
        _attributes.push_back(astore);
    }
}

void
VSimStore::memlock(void)
{
    for (unsigned int i = 0; i < _attributes.size(); i++) {
        attrptr_t astore = _attributes[i];
        mlock(astore, (_maxdocid * sizeof(attr_t)));
    }
}

void
VSimStore::memunlock(void)
{
    for (unsigned int i = 0; i < _attributes.size(); i++) {
        attrptr_t astore = _attributes[i];
        munlock(astore, (_maxdocid * sizeof(attr_t)));
    }
}

void
VSimStore::seed(unsigned long s)
{
    seedRange(s, 0, _maxdocid);
}

void
VSimStore::seedRange(unsigned long s, docid_t start, docid_t end)
{
    base_generator_type generator(s);
    boost::uniform_int <attr_t> dist(1, MAXRANK);
    boost::variate_generator<base_generator_type&, boost::uniform_int<
        attr_t> > vgen(generator, dist);

    for (docid_t i = start; i < end; i++) {
        for (unsigned int j = 0; j < _attributes.size(); j++) {
            attrptr_t astore = _attributes[j];
            astore[i] = vgen();
        }
    }
}

VSimStore::~VSimStore()
{
    // Create memory area for all attributes
    for (unsigned int i = 0; i < _attributes.size(); i++) {
        attr_t *astore = _attributes[i];
        free(astore);
    }
}

rank_t
VSimStore::getRank(docid_t docid) const
{
    const unsigned int numrounds = 20;
    rank_t rank = 0;

    // Busy loop tuned to blow some cycles here
    for (unsigned int i = 0; i < numrounds; i++) {
        for (unsigned long a = 0; a < _attrindex.size(); a++) {
            attrptr_t astore = _attributes[_attrindex[a]];
            rank += astore[docid] + i;
        }
    }
    return (rank);

}

} // end vsim
```

C.1.51 vsimtime.h

```
#ifndef _VSIM_VSIMTIME_H_
#define _VSIM_VSIMTIME_H_
#include <time.h>

namespace vsim {

/**
 * The VSimTime class is a wrapper around the POSIX time keeping
    classes and
 * provides methods for aquiring the current time as well calculating
    time
 * difference.
 */
class VSimTime {
public:
    VSimTime();
    VSimTime(struct timeval &);
    void recordTime(void);

    static const unsigned long NANO_PER_MILLI = 1000000;
    static const unsigned long MILLI_PER_SECOND = 1000;
    static const unsigned long NANO_PER_MICRO = 1000;
    static const unsigned long MICRO_PER_SECOND = 1000000;
    unsigned long getTimeSeconds(void);
    unsigned long getTimeMilliSeconds(void);
    unsigned long getTimeNanoSeconds(void);

    unsigned long getDiffMilliSeconds(VSimTime &);
    unsigned long getDiffMicroSeconds(VSimTime &);
private:
    struct timespec _value;
};

} // end vsim

#endif
```

C.1.52 vsimtime.cpp

```
#include <time.h>
#include <sys/time.h>
#include "vsimtime.h"

namespace vsim {

VSimTime::VSimTime()
{ }

VSimTime::VSimTime(struct timeval &val)
{
    _value.tv_sec = val.tv_sec;
    _value.tv_nsec = (val.tv_usec * 1000);
}

void
VSimTime::recordTime(void)
{
    clock_gettime(CLOCK_MONOTONIC, &_value);
}

unsigned long
VSimTime::getTimeSeconds(void)
{
    return (_value.tv_sec);
}

unsigned long
VSimTime::getTimeMilliSeconds(void)
{
    return (_value.tv_sec * 1000);
}
```

```
unsigned long
VSimTime::getTimeNanoSeconds(void)
{
    return (_value.tv_nsec);
}

unsigned long
VSimTime::getDiffMilliSeconds(VSimTime &t2)
{
    unsigned long t2sec = t2.getTimeSeconds();
    unsigned long t2nsec = t2.getTimeNanoSeconds();
    unsigned long t1sec = _value.tv_sec;
    unsigned long t1nsec = _value.tv_nsec;

    unsigned long millidiff = 0;
    if (t2sec > t1sec) {
        millidiff = (t2sec - t1sec) * MILLI_PER_SECOND;
        if (t2nsec > t1nsec) {
            millidiff += ((t2nsec - t1nsec) / NANO_PER_MILLI);
        } else {
            millidiff -= ((t1nsec - t2nsec) / NANO_PER_MILLI);
        }
    } else if (t2sec < t1sec) {
        millidiff = (t1sec - t2sec) * MILLI_PER_SECOND;
        if (t2nsec > t1nsec) {
            millidiff -= ((t2nsec - t1nsec) / NANO_PER_MILLI);
        } else {
            millidiff += ((t1nsec - t2nsec) / NANO_PER_MILLI);
        }
    } else {
        if (t2nsec > t1nsec) {
            millidiff = ((t2nsec - t1nsec) / NANO_PER_MILLI);
        } else {
            millidiff = ((t1nsec - t2nsec) / NANO_PER_MILLI);
        }
    }
    return (millidiff);
}

unsigned long
VSimTime::getDiffMicroSeconds(VSimTime &t2)
{
    unsigned long t2sec = t2.getTimeSeconds();
    unsigned long t2nsec = t2.getTimeNanoSeconds();
    unsigned long t1sec = _value.tv_sec;
    unsigned long t1nsec = _value.tv_nsec;

    unsigned long microdiff = 0;
    if (t2sec > t1sec) {
        microdiff = (t2sec - t1sec) * MICRO_PER_SECOND;
        if (t2nsec > t1nsec) {
            microdiff += ((t2nsec - t1nsec) / NANO_PER_MICRO);
        } else {
            microdiff -= ((t1nsec - t2nsec) / NANO_PER_MICRO);
        }
    } else if (t2sec < t1sec) {
        microdiff = (t1sec - t2sec) * MICRO_PER_SECOND;
        if (t2nsec > t1nsec) {
            microdiff -= ((t2nsec - t1nsec) / NANO_PER_MICRO);
        } else {
            microdiff += ((t1nsec - t2nsec) / NANO_PER_MICRO);
        }
    } else {
        if (t2nsec > t1nsec) {
            microdiff = ((t2nsec - t1nsec) / NANO_PER_MICRO);
        } else {
            microdiff = ((t1nsec - t2nsec) / NANO_PER_MICRO);
        }
    }
    return (microdiff);
}

} // end vsim
```

C.1.53 printreport.cpp

```
#include <iostream>
#include <stdlib.h>
#include "vsimreport.h"

using namespace vsim;

int
main(int argc, char **argv)
{
    std::vector<VSimReport> reports;
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " samplefile" << std::
            endl;
        exit(EXIT_FAILURE);
    }
    reports = importReportBatch(argv[1]);
    for (unsigned int i = 20; i < reports.size(); i++) {
        reports[i].exportReport("test.bin");
    }
    return 0;
}
```

C.2 Scripts

C.2.1 Simreport Python interface for sample files

```
#!/usr/bin/env python
import struct
import stat
import os
import sys
import math
```

```
# The simreport module is able to read simulator output reports
    generated by
# Vsim and provides an object oriented interface towards the reports.
    Moreover,
# it contains support for generating mean values of multiple runs
    together with
# standard deviations of these values. The script may also be run
    standalone
# with a filename to the sample file as argument.
def ReadInt(handle):
    return struct.unpack_from('>Q', handle.read(8))[0]

def ReadDouble(handle):
    return struct.unpack_from('d', handle.read(8))[0]

class SimReport:
    def __init__(self, reportNum, filename=None):
        self.numpart = 0
        self.numhandlers = 0
        self.numhits = 0
        self.numclients = 0
        self.heapsize = 0
        self.numdocs = 0
        self.numattr = 0
        self.numslots = 0
        self.distribution = 0
        self.numqueries = 0
        self.simtime = 0
        self.cputime = 0
        self.qps = [0]
        self.qps_mean = 0
        self.qps_stddev = 0
        self.qps_weighted = 0
        self.time_queue = 0
        self.time_exec = 0
        self.time_fetcher = 0
        self.time_partitionqueue = 0
        self.time_handler = 0
        self.fraction_handler = 0.0
        self.hitstat = []
        self.partitionqueuetime = []
        self.fetchertimes = []
        self.numindices = 0
        self.attributeindices = []
        self.partitionscheme = []

        self.reportnum = reportNum
        if filename == None:
            return
        handle = open(filename, mode='rb')
        self.importData(handle)
        handle.close()

    def importData(self, handle):
        self.numpart = ReadInt(handle)
        self.numhandlers = ReadInt(handle)
        self.numhits = ReadInt(handle)
        self.numclients = ReadInt(handle)
        self.heapsize = ReadInt(handle)
        self.numdocs = ReadInt(handle)
        self.numattr = ReadInt(handle)
        self.numslots = ReadInt(handle)
        self.distribution = ReadInt(handle)
        self.numqueries = ReadInt(handle)
        self.simtime = ReadInt(handle)
        self.cputime = ReadInt(handle)
        self.qps[0] = ReadInt(handle)
        self.qps_weighted = ReadInt(handle)
        self.time_queue = ReadInt(handle)
        self.time_exec = ReadInt(handle)
        self.time_fetcher = ReadInt(handle)
        self.time_partitionqueue = ReadInt(handle)
        self.time_handler = ReadInt(handle)
        self.fraction_handler = ReadDouble(handle)
        self.hitstat = []
        self.partitionqueuetime = []
        self.fetchertimes = []
        self.attributeindices = []
        self.partitionscheme = []
        for i in range(self.numpart):
            self.hitstat.append(ReadInt(handle))
            self.partitionqueuetime.append(ReadInt(handle))
            self.fetchertimes.append(ReadInt(handle))
        self.numindices = ReadInt(handle)
        for i in range(self.numindices):
            self.attributeindices.append(ReadInt(handle))
        for i in range(self.numpart):
            self.partitionscheme.append(ReadInt(handle))

    def equalConfig(self, report):
        return (self.numpart == report.numpart and self.numhandlers ==
             report.numhandlers and self.numhits == report.numhits and
             self.numclients == report.numclients and self.heapsize ==
             report.heapsize and self.numdocs == report.numdocs and
            self.numattr == report.numattr and self.numslots == report
            .numslots and self.distribution == report.distribution and
             self.numindices == report.numindices)

    def add(self, report):
        self.qps.extend(report.qps)
        self.qps_weighted += report.qps_weighted
        self.time_queue += report.time_queue
        self.time_exec += report.time_exec
        self.time_fetcher += report.time_fetcher
        self.time_partitionqueue += report.time_partitionqueue
        self.time_handler += report.time_handler
        self.fraction_handler += report.fraction_handler
        for i in range(min(self.numpart, report.numpart)):
            self.hitstat[i] += report.hitstat[i]
            self.partitionqueuetime[i] += report.partitionqueuetime[i]
            self.fetchertimes[i] += report.fetchertimes[i]
            self.partitionscheme[i] += report.partitionscheme[i]

    def printreport(self):
        print self.numhandlers, self.numpart, self.numhits, self.
            qps_mean, self.qps_stddev, self.time_queue, self.time_exec
            , self.time_fetcher, self.time_partitionqueue

    def average(self, howmany):
        self.qps_mean = sum(self.qps) / howmany
        for num in self.qps:
            self.qps_stddev += math.pow(num - self.qps_mean, 2)
        self.qps_stddev = math.sqrt(self.qps_stddev)
        self.qps_weighted /= howmany
        self.time_queue /= howmany
        self.time_exec /= howmany
        self.time_fetcher /= howmany
        self.time_partitionqueue /= howmany
        self.time_handler /= howmany
        self.fraction_handler /= howmany
        for i in range(self.numpart):
            self.hitstat[i] /= howmany
            self.partitionqueuetime[i] /= howmany
            self.fetchertimes[i] /= howmany
            self.partitionscheme[i] /= howmany

def importReportBatch(filename):
    s = os.stat(filename)
    size = s[stat.ST_SIZE]
    handle = open(filename, 'rb')
    reports = []
    i = 0
    while handle.tell() < size:
        report = SimReport(i)
        report.importData(handle)
        reports.append(report)
        i += 1
    handle.close()
    return reports

def averageReports(reportlist):
    avglist = []
    numavg = []
    for report in reportlist:
        found = False
        for i in range(len(avglist)):
            if report.equalConfig(avglist[i]):
                avglist[i].add(report)
                numavg[i] += 1
                found = True
                break
        if not found:
            avglist.append(report)
            numavg.append(1)
    for i in range(len(avglist)):
        avglist[i].average(numavg[i])
    return avglist

def importAvgReports(filename):
    return (averageReports(importReportBatch(filename)))
```

```
if (len(sys.argv) > 1):
    reports = importReportBatch(sys.argv[1])
    for report in reports:
        report.printreport()
```