HASH TÁBLÁK TELJESÍTMÉNYÉNEK
MODELLEZÉSE ÉS OPTIMALIZÁLÁSA

MODELING AND OPTIMIZING HASH TABLES

Ph.D. thesis booklet

Ákos DUDÁS

Advisor:
Sándor JUHÁSZ, Ph.D.

Budapest, 2013.
Ákos DUDÁS

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

1117 Budapest, Magyar Tudósok krt. 2. Hungary

e-mail: akos.dudas@aut.bme.hu
tel: +36 (1) 463-3971
fax: +36 (1) 463-2871

Advisor:
Sándor Juhász, Ph.D.
associate professor
1 Preliminaries and objectives

General purpose CPUs have been evolving in the last couple of decades as predicted by Gordon Moore. Chip manufacturers have been steadily increasing the performance of processors by raising the operating frequency speeding up the execution, and by integrating more and more transistors into the cores allowing deep pipelines and sophisticated execution units. Some hardware components have been having a hard time keeping up with this development pace. System memories especially have been falling short of expected performance. The speed gap between the system memory and the CPU had been an issue for at least two decades now [WM95], which have directed chip manufacturers to integrate increasing amounts of cache memories into CPUs, and software engineers to utilize cache memories and lighten the strain put on the system memory.

Processor manufacturers have also been shifting toward new approaches. Raising the operating frequency had stopped due to nonlinear increase in power consumption and cooling problems. Enhancing the instruction throughput of a single core have also slowed down, because it complicates the design and increases the size of the cores, which is a problem due to wire delays [Osk08]. New trends, besides integrating larger caches into the processors, feature parallelism with simultaneous multithreading within a single core, and integrating multiple cores into the same socket [GK06b].

It is argued by Mark Oskin, that “CPU performance must be measured by observing the execution time of real applications” [Osk08]. Observing the algorithmic complexity does not necessarily project the actual performance of an algorithm, as algorithmic complexity does not capture the real behavior of modern computers. Reading data from a nearby cache, for example, is a hundred times cheaper than accessing data from the system memory; these effects are not considered when algorithmic steps are assumed to have a uniform cost.

At the same time multi-core processors represent a significant change in conventional computing and programming [GK06b] and standard performance optimization mechanisms are no longer suitable. Algorithmic complexity fails to capture the concept of parallelism. There are no means to describe synchronizations, blocking, communication overhead between threads, hence these overheads are not represented in standard complexity measures. To achieve good application performance programmers need to adopt new programming models and techniques [Sha11, Osk08] and they need to test and evaluate applications and algorithms by actually executing them and measuring the real-life performance.

To develop good and fast algorithms software engineers need to understand these trends in hardware development and design their products so that they utilize all components of the hardware to their best.

This thesis deals with the optimization of hash tables. Hash tables are basic and widely used data structure in various fields, such as model checking [BR08, LvdPW10], web servers [LM01, VF07], within network routers [BM01, Mit02, SDLT05], and in genome research [NCM01].

Hash tables can be found in most frameworks and software libraries, but despite their popularity, they are mostly black boxes. The implementation details are masked by the literatures main interest in hash functions and probe sequences (number of steps required to find an item in a hash table). This thesis will show, that in fact the implementation, data placement, and data access patterns are just as relevant as the length of the probe sequence. This work follows Song et al., who had argued that “from an engineering perspective designing a good hash table can still be a challenging task with potential for several improvements” [SDLT05].
My research focuses on three aspects of hash table design: optimizing memory layout and access pattern for utilizing cache memories; using data prefetching for masking the latency of data access; and preparing the hash table for thread-safe concurrent access.

The first goal in this thesis is to find the “best” hash table. In order to understand how hash tables behave, and how they can be implemented, I constructed a system describing low-level structures and their memory layout, outlining possible options for creating a hash table. I tested these options for performance and showed that the qall-clock execution time and memory consumption are both important factors. The examined hash tables are evaluated from this point of view.

In order to understand the behavior of the hash tables, and the effect of different implementations on performance, I examined the way the algorithms and data structures use the hardware. My purpose was not only to understand why different implementations behave the way they do, but also to model their behavior capturing the real performance. It has been indicated in the literature that the behavior of the cache memories are a key factor in describing the accurate behavior and performance of hash tables [HL05, Mit02]. I created an analytical model characterizing the cache usage of different types of hash tables.

For memory-intensive applications the technique of “pre-execution” [RMS98, BCS09, Dub04] (also referred to as assisted execution, speculative precomputation, prefetching helper threads) is advocated in the literature. Such techniques have been proven to enhance the performance of data intensive applications by masking data access latency. My goal was to examine its applicability to hash tables. Hash tables are computationally cheap, therefore their performance is dominated by the cost of data access. My aim was to manage this cost with the help of pre-execution.

Good performance in multi-core machines is predicated on multithreaded applications, which require fast and thread safe data structures [Sha11]. The final goal of this thesis is to allow the use of hash tables in fully parallel applications. This requires the hash table to be thread-safe, and also needs it to allow high level of parallel access, and not be a bottleneck. There are two basic approaches for implementing concurrent data structures [HS08]: using mutual exclusion [Dij65] and non-blocking synchronization [Her90]. I examined both to find the best possible option.

Non-blocking implementations can satisfy different “progress conditions” [HS11]. My purpose was to see what types of hash tables satisfy which levels of progress conditions, and how their performance varies in return. Despite reservations in the literature concerning lock-based concurrent data structures, my goal also included providing a highly parallel lock-based hash table implementation with good throughput.

All proposed methods and techniques are subject to performance tests and evaluations with comparison to off the shelf libraries and solutions.

2 Research methodology

The literature of hash tables is vast, and it encompasses a wide range of topics. The goal of my research, as outlined previously, was to enhance the actual performance of hash tables. In order to measure the performance I selected a specific target platform. This platform features a modern single-core or multi-core processor with multiple levels of caches, and has access to
system memory large enough to store both the hash table and all related data without the need of paging data to the hard disk. The amount of data the hash table stores is by two orders of magnitude larger than the size of the caches, therefore the hash table is stored in the system memory, and only a fraction of it fits into the caches at all times.

The two most common operations on hash tables are adding new items and retrieving items. The focus of performance optimization was directed at these two actions. Hash tables also support deletion, but deletion is infrequently used compared to searching and adding, therefore, it was not considered for optimization.

Having the target platform determined, I outlined three aspects to examine: cache behavior, data prefetching, and concurrency. Each aspect is studied separately from the others, but some results are integrated into later theses as well. In each phase, I constructed new structures and algorithms testing ideas to enhance the performance of the hash table in the given context. The performance of each algorithm was measured in terms of wall-clock execution time and required amount of system memory. Research was focused on understanding the behavior of the algorithms and data structures. This invited monitoring of hardware using performance counters to understand its cooperation with the software layers.

Given the new structures, algorithms, approaches, I compared the options to find the best one. I analyzed the results and examined the reasons behind the performance.

In order to fully control the placement of data in memory and avoid overhead of runtime frameworks, the implementations are all in C++ compiled with the same compiler using default optimization. No software libraries or parallel compilers were used to avoid introducing any overhead (with the exception of performance comparison to reference implementations).

3 New scientific results

The results are organized into four theses. Each thesis examines a different aspect of performance optimization. The results are mainly presented in forms of algorithms, always verified by evaluation. Some of the results are formal models of a system, which are validated by simulations.

The first thesis focuses on single-threaded hash table optimization for wall-clock execution time and memory consumption. The thesis presents empirical observations about various implementations of hash tables and shows an analytical model for estimating the performance of hash tables with better accuracy than the expected probe length. The second thesis applies the technique of pre-execution to hash tables for prefetching data into the CPU caches in order to reduce memory access latency. A software method is presented that changes its internal parameters automatically to adapt to the execution environment.

The third and fourth theses are about fully concurrent hash table implementations. The third thesis considers non-blocking solutions while the last thesis focuses on lock-based hash tables. For non-blocking hash tables two new ideas are presented, one satisfying the lock-free, the other one the wait-free progress condition. For lock-based hash tables an adaptive algorithm is given which calibrates fine-grained locking automatically according to the level of parallel access.

The rest of this section lists all four theses in order, citing my publications corresponding to the topic, followed by a general overview of the work, and summary of the results.
THESIS I: Cache-friendly hash tables

The following publications are related to this thesis: [3] [7] [8] [9] [10] [11] [12] [13].

Hash tables are data structures that act like dictionaries, containing items of key-value pairs [Knu73, CLRS01]. The purpose of hash tables is to store and retrieve items. The items are unambiguously identifiable with a key: every item has a unique key, or with other words, if the keys of two items are the same the two items are considered the same. Hash tables are fast containers: when designed appropriately finding an item is an $O(1)$ operation [FKS84].

\textbf{Definition 1 (Hash table)}

A hash table stores items of key-value pairs. The internal structure of a hash table features a directly indexable array; each slot contains a single item or a set of items (Fig. 1). The place of an item in the table array is determined by a hash function.

\begin{center}
\begin{tabular}{|c|c|}
\hline
0 & {[key-value], [key-value], ...} \\
1 & {[key-value], [key-value], ...} \\
2 & {[key-value], [key-value], ...} \\
3 & {[key-value], [key-value], ...} \\
4 & {[key-value], [key-value], ...} \\
5 & {[key-value], [key-value], ...} \\
\hline
\end{tabular}
\end{center}

Figure 1: Abstract structure of a hash table.

Some works have already studied the memory access pattern and the effect of the multi-level memory hierarchy on different algorithms [WM95, HL05] and came to the conclusion, that the last level cache memory has the most significant performance impact on structures with large memory footprint. I integrated these approaches into the performance cost models of hash tables.

\textbf{Subthesis I.1: Hash table structures}

I created a classification system (Fig. 2) for describing the internal structure and memory access pattern of selected hash tables. The system describes data arrangement options and memory access patterns. I gave a common denominator for comparing the performance of hash tables independent of the peculiarities of the different structures: a particular structure is superior to another one, if given the same amount of memory, it completes the same operations faster. I demonstrated with empirical measurements that the number of last level cache misses is a good indicator of this performance.

The two basic types, open hash tables and bucket hash tables must be distinguished. Although these are not new concepts and the literature is full of quasi-definitions, those are sometimes ambiguous, allowing mixed types, and often too much implementation details are specified. My goal is to separate the two types clearly with definitions that are applicable to all
types of hash tables without any doubt. Therefore the definitions below show what I call open- and bucket hash tables. (These are in accordance with the general understanding.)

**Definition 2** *(Open hash table)*

An open hash table stores 0 or 1 item either directly, or linked via a pointer, in each slot of the table body.

**Definition 3** *(Bucket hash table)*

A bucket hash table allows multiple entries to be assigned to a slot of the table. The items may be stored at memory locations external to the table body.

The length of the so called probe sequence is the algorithmic complexity of searching in a hash table. This quantity is often used for judging performance. However, in case of data intensive applications, this can be misleading. In a hash table the cost of moving data is the dominant factor, therefore the number of cache misses is a better indicator of performance. A hash structure should be designed so that it minimizes the number of such events. I showed that among the structures of Fig. 2 item-table-linear-probing and item-table-linked-list are the best in this respect, having best performance at the same time.

**Subthesis I.2: Estimating the number of cache misses**

I obtained analytical results estimating the number of cache misses during a find operation in a hash table. I presented a model for open hash tables with linear probing that allows calculating the length of the probe path, from which the number of cache misses can be derived. The length of the probe sequence is a known result for uniform hashing, which, given its special behavior, is equal to the number of cache misses. The probe length can be calculated knowing the length of the bucket for bucket hash tables, from which I derived the number of cache misses for various structures.

Given a hash table storing \( n \) items and having \( m \) slots (buckets), the load factor of a table is \( \alpha = \frac{n}{m} \). This number is normally used to describe open hash tables, but for the purposes of having a single terminology the same definition is used for bucket hash tables, where, contrary to open hash tables, \( \alpha \) can be larger than 1. For estimating the number of cache misses we must know the number of items that fit in a single cache line; let this be denoted by \( B \).
Let $\mathbb{E}(L_\alpha)$ denote the expected probe length for finding an item using linear probing in an $\alpha$-filled table.

$$\mathbb{E}(L_\alpha) = 1 + \frac{1}{2} \frac{\alpha}{1 - \alpha}$$

(1)

The expected number of last level cache misses for successfully finding an item in an open hash table with linear probing (2), open hash table with uniform hashing (3), bucket hash table with array storage (4), bucket hash table with linked lists and “pointer table” (5) and with “item table” (6), in order, are:

$$\mathbb{E}(C_{\alpha \text{lin}}) = 1 + \frac{1}{2} \frac{\alpha}{1 - \alpha}$$

(2)

$$\mathbb{E}(C_{\alpha \text{uni}}) = -\frac{\ln(1 - \alpha)}{\alpha}$$

(3)

$$\mathbb{E}(C_{\alpha \text{ptwa}}) = 1 + 1 + \frac{1}{B} \frac{1}{2} \alpha$$

(4)

$$\mathbb{E}(C_{\alpha \text{ptwl}}) = 1 + \frac{1}{2} \alpha$$

(5)

$$\mathbb{E}(C_{\alpha \text{itwl}}) = \frac{1}{2} \alpha$$

(6)

From these equations depending on parameter $B$ (which is specific to each task) the number of cache misses can be determined for the different structures given their load factor, and the one with the lowest cache miss count can be chosen.

**THESIS II:**

**Adaptive pre-execution in hash tables**

The following publications are related to this thesis: [1] [5] [14] [15] [16].

Data prefetch mechanisms [BCS08] mask memory latency by preloading data from the main system memory into the CPU caches before they are actually needed by the processing algorithm. These mechanisms speed up execution by eliminating the time the CPU stalls on cache misses. It was argued by Luk [Luk01] that irregular memory access patterns (responsible for increased memory latency and inefficient hardware prefetch) can only be predicted through executing the code itself. Pre-execution [Dub04,KLW+04] runs a second copy of the algorithm itself to calculate the required memory addresses early and load the data ahead of the actual access.

Pre-execution is mostly an abstract idea in the literature, although it has been used effectively [MK06,ZCRS05]. In order to present my proposed method I gave a formal definition for pre-execution using the following terminology.
Definition 4 (Instruction, sequence of instructions)

An instruction is a mathematical operation, a data movement command, or any combination of these, which the processor is able to execute in a few cycles. A sequence of instructions is a series of such instructions. Let $R(\sigma)$ be the set of variables the instructions in $\sigma$ only read, and $W(\sigma)$ the set of variables they read and write. Let $\Sigma = \{\sigma\}$ be the set of these sequences of instructions.

An instruction could be the core of a cycle or a function. These are basic building blocks of algorithms.

Definition 5 (Algorithm)

$\Gamma = \gamma_1, \ldots, \gamma_n$ is an algorithm if $\gamma_i \in \Sigma$, $R(\gamma_i) \neq R(\gamma_{i+1})$ or $W(\gamma_i) \neq W(\gamma_{i+1})$, $R(\gamma_i) \in 2^R$, $W(\gamma_i) \in 2^W \forall i$.

An algorithm is a set of sequences of instructions, where adjacent instruction sequences are distinguished by the memory locations they access. An algorithm is arbitrarily separated into these sequences of instructions. The main thread of execution and the pre-execution must run in parallel, therefore we also make sure that the prefetcher never interferes with the main thread.

Definition 6 (Valid parallel execution of two sequences of instructions)

$\gamma_i$ and $\gamma_j$ ($\gamma_i, \gamma_j \in \Sigma$) can be executed in parallel if $R(\gamma_i) \cap W(\gamma_j) = \emptyset$, $R(\gamma_j) \cap W(\gamma_i) = \emptyset$ and $W(\gamma_i) \cap W(\gamma_j) = \emptyset$. This will be denoted by $\gamma_i \parallel \gamma_j$.

Definition 7 (Valid parallel execution of a sequence of instructions and an algorithm)

A sequence of instructions $\sigma \in \Sigma$ can be executed in parallel with algorithm $\Gamma$ ($\sigma \parallel \Gamma$) if $\sigma \parallel \gamma_1$, $\sigma \parallel \gamma_2 \ldots \sigma \parallel \gamma_n$, that is, $\sigma$ can be executed in parallel with all instructions of algorithm $\Gamma$.

Since the literature gives no formal definition of pre-execution, I created a general one, which can be applied to hash tables and other algorithms as well.

Definition 8 (Pre-execution)

Let $\Gamma$ be an algorithm as described above. The main execution engine completes $\Gamma$ by computing each and every $\gamma_i \in \Gamma$ individually, in order. Pre-execution is another execution engine which, at specified trigger points, schedules $\sigma_j \in \Sigma$ for execution where $\sigma_j \parallel \Gamma$. The $\sigma_j$ instruction sequences are created so that they calculate and preload the memory addresses of the corresponding $\gamma_i$ instruction sequence.

Subthesis II.1: Software-based pre-execution scheme

I have created a pre-execution algorithm (see Alg. 1) for speeding up a hash table by using a prefetching helper thread, which monitors the status of the main execution path and works to load data in advance before the main execution requires them. The algorithm is controlled by two parameters. I demonstrated with experiments that the algorithm is able to improve the performance of a hash table.

The pre-execution engine is realized as a helper thread responsible for the data prefetch. The helper thread communicates with the main thread through shared variables to stay ahead of it. Jumpahead distance keeps a distance between the threads, and the resolution of the synchronization between them (when the distance is enforced) is controlled by parameter workahead size. When the method is applied to hash tables and the parameters are chosen manually, pre-execution can deliver up to 35% increase in performance.
Algorithm 1 Pseudo-code of the proposed pre-execution scheme.

```c
int mainIndex
bool end = false
int jumpAheadDist
int workAheadSize

procedure main(γ₀...γₙ)
for i = 0 to n/workAheadSize do
    mainIndex = i
    for j = 0 to workAheadSize do
        execute γᵢ*workAheadSize+j
    end for
end for
end = true
return

procedure preExecutionEngine(σ₀...σₙ)
while end == false do
    for j = 0 to workAheadSize do
        execute σₘₚ₉(min(mainIndex+jumpAheadDist+j,n)
    end for
end while
return
```

Subthesis II.2: Adaptive pre-execution algorithm

I create an adaptive algorithm (see Alg. 2) that is able to configure the parameters of Algorithm 1 in runtime for best performance. Using the Nelder-Mead method the proposed algorithm constantly monitors the performance and changes the parameters to adopt to the changes in the environment. Through measurements I demonstrated that this algorithm successfully enhances the performance of the hash table through data prefetching and that the adaptive algorithm changes its parameters according to the changes of external circumstances.

The Nelder-Mead method [NM65] is a nonlinear optimization technique for approximating a local optimum of a problem with multiple variables, and it uses only previous values of the function. The variables are the parameters of the algorithm, namely jumpahead distance and workahead size, and the objective function is the performance of the application which we measure in terms of execution time. The proposed solution delivers a performance increase of 10-35% for various hash tables.
Algorithm 2 Pseudo-code of the proposed adaptive pre-execution model

```plaintext
int mainIndex
bool end = false
int jumpAheadDist
int workAheadSize
int sampleSize = 10

procedure main(γ₀...γₙ)
for i = 0 to n/workAheadSize do
  mainIndex = i
  for j = 0 to workAheadSize do
    execute γᵢ∗workAheadSize+j
  end for
end for
end = true
return

procedure preExecutionEngine(σ₀...σₙ)
int samples = 0
int time start = getTime()
while end == false do
  for i = 0 to workAheadSize do
    if samples == sampleSize then
      {jumpAheadDist, workAheadSize} = evaluateNelderMead(getTime() − start)
      start = getTime()
      samples = 0
    end if
    if notDoneAlready(min(idx + jumpAheadDist + i, n)) then
      execute σᵦₘᵢₙ(mainIndex+jumpAheadDist+i,n)
      samples = samples + 1
    end if
  end for
end while
return
```

THESIS III:
Non-blocking parallel hash tables

The following publications are related to this thesis: [4] [6] [17] [19].

Multi-core processors and parallelism change the way data structures are used [Sha11]. Concurrent programs require concurrent data structures guaranteeing thread safety and good performance. Non-blocking parallelism promises good scalability and none of the drawbacks of mutual exclusion [Her90], but at the cost of increased complexity is design. These methods employ, what is called “optimistic synchronization:” changes are always made in local memory and everything is committed into the data structure in a single atomic step with the help of special processor instructions, such as the compare-and-exchange.

Non-blocking hash tables are most often built using non-blocking (either lock-free or wait-free) linked lists. The first hash table I created uses arrays, which are beneficial for the cache usage. The second solution I proposed is a wait-free scheme that is applicable to all types of hash tables and uses no atomic operations within the tables.

Subthesis III.1: Array-based lock-free bucket hash table

I created an array-based non-blocking bucket hash table algorithm (see Alg. 3) supporting fully concurrent insertion, deletion and find operations, all in a lock-free manner, satisfying the linearizability progress condition.

The presented algorithm combines the cache friendliness of arrays with the use of atomic principles for achieving thread-safe non-blocking concurrency. The performance of this method is similar to that of standard non-blocking hash tables using linked lists, and is even better for search-heavy workloads.

Subthesis III.2: Wait-free hash table

I created a parallelism scheme with two implementations (see Algorithm 4 and 5), in which concurrent hash tables can be implemented without the use of mutual exclusion and thread synchronization mechanisms, or atomic instructions. This scheme can be used with arbitrary single threaded hash tables, yet the complete system will be parallel and thread safe. The described algorithm satisfies the wait-free non-blocking progress condition.

The idea of both solutions is to apply multiple hash tables and direct requests to the right table, guaranteeing truly parallel service. The key is connecting incoming requests to the right tables, for which two solutions are presented. The performance of the presented solutions does not measure up to lock-based or the lock-free solutions, but it is compensated for by its wait-freeness.

These wait-free implementations cannot be described by any of the generally used correctness criteria. They differ from traditional concurrent hash tables as the threads need to be aware that they are accessing different hash tables. It is not the responsibility of the hash table to handle parallelism, but it is left up to the threads themselves. This can be viewed as a “relaxation” of the requirements [Sha11].
Algorithm 3 Compare-and-swap based lock-free bucket hash table with arrays.

- buckets[]: pointer to array

**procedure** insert( data )
  
  while true do
    
    h = hash( data )
    myArray = buckets[h]
    
    idx = find_in_array( myArray, data )
    if idx != -1 then
      return myArray[idx]
    
    newArray = reallocate_copy_insert( myArray, data )
    if CompareExchange( buckets[h], newArray, myArray ) then
      return
    
  end while

**procedure** find( data )

myArray = buckets[ hash( data )]

idx = find_in_array( myArray, data )
if idx != -1 then
  return myArray[idx]
else
  return NULL
end if

**procedure** delete( data )

while true do
  
  myArray = buckets[ hash( data )]
  
  idx = find_in_array( myArray, data )
  if idx == -1 then
    return
  
  newArray = reallocate_copy_remove( myArray, data )
  if CompareExchange( buckets[h], newArray, myArray ) then
    return
  
end if

end while
Algorithm 4 Wait-free hash table with a central array.

1: hash_tables[numberOfThreads]: hash table
2: dataSet[]: processed items
3:
4: procedure threadFunction( dataset )
5: for item i in dataSet do
6:   h = hash( i )
7:   if h % numberOfThreads == threadId then
8:     hash_tables[threadId].insert_or_lookup( h, i )
9:   end if
10: end for

Algorithm 5 Wait-free hash table with multiple queues.

1: hash_tables[numberOfThreads]: hash table
2: queues[numberOfThreads]: wait free queue
3:
4: procedure addRequest(data)
5:   h = hash( data )
6:   queues[h].enqueue( data )
7:
8: procedure threadFunction()
9:   while true do
10:     if i = queues[threadId].dequeue() then
11:       hash_tables[threadId].insert_or_lookup( i )
12:     end if
13:   end while
THESIS IV:
Fine-grained locking in bucket hash table

The following publications are related to this thesis: [2] [4] [20] [21].

Mutual exclusion [Dij65, AKH03] is the de facto approach for thread safe data containers. Although the non-blocking solutions discussed in Thesis III are great in scalability and performance, they are complex and require a completely new way of thinking about algorithms and parallelism. Mutual exclusion and the use of locks is a straightforward solution integrated into the design of software, and synchronization of threads supported by design patterns and libraries [MSM04].

Bucket hash tables can implement fine-grained locking to decrease contention for the locks, allowing better throughput, and managing the overhead of locks. The proposed methods below explain how to choose the granularity.

Subthesis IV.1: Fine-grained locking in bucket hash tables

Through empirical measurements I have shown that the performance of bucket hash tables used in parallel environment is effected by the number of locks used by the table internally. For estimating the optimal number of locks I modeled their behavior with a queuing network (Fig. 3). The event when a thread accesses a busy lock I called clashing. From the queuing network I derived the probability of such events, and demonstrated the validity of the model and results with empirical measurements. I gave a heuristic argument describing when a new thread, added to the system, attributes to better throughput, and using this heuristic, gave an upper limit for waiting on a lock.

Multiple threads are accessing a bucket hash table, which is partitioned into $L$ disjoint locked blocks. Before a thread can access any bucket of the table, it must first acquire the corresponding lock. When a lock is busy, the threads perform spin waiting. Threads are allowed to enter the lock in the same order they requested it.

![Figure 3: The queuing network modeling the parallel bucket hash table partitioned into $L$ blocks.](image)

Each of the $L$ locks in the system is chosen with probability $1/L$. Assuming exponential distribution with parameter $\mu$ for the time spent within the critical sections (common for all threads and operations), and exponential distribution of time with parameter $\zeta$ between two queries, the system is as follows. Every lock in this system is a $M/M/1/\infty/\infty/FCFS$ or $M/M/1/\infty/\infty/SIRO$ queue (depending on the chosen service discipline) with arrival intensity $\lambda(L) = \frac{\zeta}{L}$, service intensity $\mu$ and service rate $\rho(L) = \frac{\lambda(L)}{\mu}$.

Knowing the parameters of the queuing network, the probability of the event we call clashing, is $P(\text{clash}) = \rho(L) = \frac{\lambda(L)}{\mu}$. 

15
To capture the delay the threads suffer, waiting time $w(L)$ describes the amount of time a thread has to wait to enter a lock. When two threads are racing for the same shared resource the second thread attributes to better performance as long as there is some action it can perform while the first one is in the critical section protecting the shared resource. Therefore, the system should assure that the $w(L)$ is less than $\frac{1}{\mu}$.

**Subthesis IV.2: Adaptive locking strategy**

I created an algorithm, which calibrates the number of locks used internally by a concurrent bucket hash table. By measuring internal properties of the hash table during runtime, a periodic calibration estimates the parameters of the queuing system described in Subthesis IV.1, and the number of locks is estimated as the lowest number that guarantees that the waiting time for the locks is below the specified threshold. I demonstrated that this algorithm gives a good estimate for the number of locks.

The optimal number of locks is derived from the queuing model after estimating the parameters of the model. The hash table records the average of the length of the queue for the locks $\overline{x}$. Service intensity $\mu$ is calculated by measuring the elapsed time of test queries, and arrival intensity $\lambda(L)$ is calculated as

$$\lambda(L) = \frac{\overline{x} \cdot \mu}{1 + \overline{x}}$$

The number of locks the system should use is then the smallest $L$ such that the probability

$$\mathbb{P}(w(L) < \frac{1}{\mu}) = \left(1 - \frac{\lambda(L)}{\mu}\right) + \frac{\lambda(L)}{\mu} \left(1 - e^{-(\mu - \lambda(L)) \frac{1}{\mu}}\right)$$

is significant (more than 0.95).

### 4 List of publications

**Papers published in international journals**


Book sections


Conference papers published in journals


Papers published in conference proceedings


5 References


