

Analysing and Improving Hash Table Performance

Using usage analysis to improve performance for cache

Tom van Dijk

t.vandijk@student.utwente.nl

ABSTRACT

In this paper we describe three methods for analysing how different algorithms use a dictionary. With these methods, the implementation of the dictionary can be fine-tuned, in order to increase performance. The methods aim at discovering patterns that can be used to achieve more optimal use of hierarchical memory, especially L2 cache. Our research mainly focuses on improving the performance of hash tables used by state search algorithms. We also present some pitfalls that may appear when trying to improve performance.

Keywords

analysis, performance, hash table, dictionary, cache, state search

1. INTRODUCTION

For many if not most applications of computer science, performance is important. Sometimes, even small performance gains can save a lot of time and memory. Improving performance can be done at different levels. For example, an algorithm can be improved by decreasing the number of calculations, by using extra data structures to reduce the number of redundant calculations, or by improving worst case time complexities. Performance can also be improved by optimizing algorithms and datastructures for the environment. For example, the implementation of an algorithm can use special instructions on a specialized processor. A different example is making smart use of cache memory on a processor. Algorithms are often designed with a uniform memory model in mind, in which the access time of memory is constant, while in reality memory is hierarchical, with small fast memory and large slower memory. In this paper we present the results of an attempt to improve the performance of a data structure called a dictionary, used by an algorithm called a state search algorithm. Because a state search involves little calculation, most time is spent accessing the dictionary. Even small improvements could result in much faster searching. Maks Verver [8] investigated this by comparing three implementations of a dictionary: a cache-unaware hash table, which assumes a uniform memory model; a cache-aware binary tree, which can be adjusted for a specific cache configuration; and a cache-oblivious Bender set, which is designed to perform well with any kind of cache. The results of his research were that in his test environment the cache-unaware hash table was fastest, despite not being designed for a cache table. Our research focuses on hash tables and improving their performance in an environment with hierarchical memory.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission.

10th Twente Student Conference on IT, Enschede 23th January, 2009

Copyright 2009, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science

In the introduction we introduce the main subjects of our research. In section 3 we then present three methods for analysis and we demonstrate how to use them. In section 4 we will extend a hash table implementation with a secondary table and use the secondary table to optimize in a specific case. We will then conclude the paper with conclusions and ideas for further research.

1.1 Dictionaries

A dictionary is an abstract data structure used in various applications. A dictionary is a set of keys. Often, but not always, these keys have a value or data structure associated with them. Common operations on a dictionary are *add(key)*, *remove(key)*, *length* and *exists(key)*. A sorted dictionary is a dictionary that can efficiently be browsed with *nth(index)*. There are different ways to implement dictionaries. Example implementations are a linked list, (sorted) binary trees, tries and hash tables. A common approach is to use a chained hash table [4, Ch. 6.4]. The choice of implementation depends on what the dictionary is going to be used for.

1.2 Hash Tables

A hash table is a data structure that scatters data in a flat array according to a hash function. Take for example a hash table with hash function h and n entries. The hash function is a function from a key K to a positive real number in a range $0 < h(x) < M$, $M \geq n$. Usually the hash function generates a deterministic quasi-random number, which can be used as an offset in the table. The entry in the hash table that will be used for key K will then be at offset $h(K) \bmod n$. The hash function generates quasi-random numbers to scatter data as much as possible, so even similar keys will be at totally different offsets.

If two different keys give the same offset ($K_1 \neq K_2, h(K_1) = h(K_2) \bmod n$), the keys will use the same entry in the hash table. This is called a *collision*. Collisions can be dealt with in several ways. Usually the entry in the table will simply point to the start of a linked list, which on access is searched in linear time ($O(n)$). If no collisions happen, a hash table is a flat structure and inserting/deleting/retrieving data could be done in constant time ($O(1)$). If there are many collisions, for example when the table is heavily loaded or there is a bad hash function causing many keys to hash to the same table offset, a hash table search will degrade to a linear search on a linked list, resulting in time complexities worse than $O(1)$. If keys aren't known in advance, it is usually not possible to guarantee no collisions will happen.

There are several ways to tune hash tables. The main way is to invent a very smart hash function, because hash functions directly determine how well data is *scattered* in the hash table. Inventing good hash functions is very hard, especially since very complex hash functions can cause worse performance because they take long to calculate, and because it is hard to invent a good

hash function without knowing the keys we are working with in advance. There are several implementations of hash functions, examples can be found on the Internet [3, 6]. There has been research into perfect hashing functions and minimal perfect hashing functions, which is an effort to create a hash function that guarantees no collisions for a fixed set of keys that is known in advance [1].

There are different ways to work around the $O(n)$ complexity of searching through a linked list when there are collisions, such as using a binary tree instead of a linked list, or by moving the last accessed entry to the front of the linked list (which can increase performance when a few entries are often accessed, but is vulnerable to specific access patterns). There are also alternatives to using an external data structure, for example by chaining in the table with Linear Probing [4, p. 527]. A quite different alternative is called Cuckoo Hashing [5]. Certain operations on a dictionary can be sped up by using additional data structures next to the main structure. A Bloom filter [7] is an example of a data structure that can be used to quickly determine using multiple different hash functions if a certain key is not in the table. Each of these options has advantages, disadvantages and side requirements.

1.3 Tries

A trie is a N-ary tree in which every node is an array with N branches. To find a key in the trie, the key is represented as a sequence of characters. Every character is used as an index in every node. For example, to find the key "920" first the array index "9" is followed, then index "2" and then index "0" to find the key. Tries are described in Knuth [4, Ch. 6.3]. Tries are also called prefix trees.

1.4 Table Compression

Table compression is a method to decrease key size. If a key can be chopped up in smaller pieces, the pieces can be stored in a table and a value assigned to them. For example, a key "abcabcdefabc" can be chopped up in four pieces, "abc|abc|def|abc" and each piece can be put in a table. Lets assume "abc" is assigned to index 1 and "def" is assigned to index 2. The key "abcabcdefabc" can be represented with value "1121". This compressed key is much shorter than the uncompressed variant.

1.5 Hierarchical Memory

Often memory is assumed to be an unsophisticated, flat resource, with simple properties, like a constant access time. In general, this is not the case, because general purpose computers often have four memory layers with different properties. Two of these layers reside on the processor chip, one layer is the RAM memory and one layer is the physical memory, for example a hard disk. The layers on the processor chip are called L1 cache and L2 cache. L1 cache is a very small piece of memory with very high access time, used directly by the processor. L2 cache is slightly slower and much larger than L1 cache. When accessing memory, the CPU first looks in L1 cache, then in L2 cache, and only then in main memory.

Cache memory is structured in cache lines. Every cache line is N bytes long. When transferring data to or from the cache, this is done per cache line, not per byte. A cache controller determines when to store data in cache. There are several possible algorithms for this. A simple way is to always store accessed data in L2, overwriting the least recently used cache line. A com-

puter program may not be able to control what is in cache and what isn't.

Understanding memory hierarchy might be useful to be able to increase the performance of an algorithm or data structure. We call algorithms and data structures that are tailored to specific cache organisation *cache-aware* or *cache-conscious*. Algorithms and data structures that are designed to perform well with caching independent of the cache parameters are called *cache-oblivious*. In order to design cache-aware and cache-oblivious algorithms one must be aware of the caching algorithm used by the cache controller. In other words, the cache-aware and cache-oblivious algorithms must either trick the cache controller or incorporate the caching algorithm.

Cache trashing is a problem that occurs when cached data that is accessed often is overwritten by data that is only rarely accessed. If this happens a lot, cache performances degrades and the usefulness of the extra layer of fast memory is lost.

2. RELATED WORK

Our research is rooted for some part in Maks Verver's investigation of cache-oblivious algorithms [8]. We base our implementation on his framework and the test set he used in his research. One of the conclusions in his paper was that hash tables outperform B-trees and Bender sets in his research, which leads us to focus our investigation on hash tables.

In a paper in 1997 Holzmann reviews several compression methods for reducing the byte length of states in a state search algorithm [2]. Our analysis of the keys inserted into the dictionary by a state search algorithm shows that compressing the keys might be an interesting approach to increase the performance of the dictionary.

3. ANALYSIS

In order to understand how the performance of the dictionary can be improved in specific cases, we need to do analysis. We use a state search algorithm that runs a depth first search (DFS, see listing 1) or a breadth first search (BFS, see listing 2) on a model. The state search algorithm uses a queue and a dictionary as its main datastructure. We analyse the behavior of the DFS and BFS algorithm executed on several different models. Our goal is to gain insight in how the datastructure is used and to discover problems that could cause bad performance. A state search algorithm uses a dictionary to keep track of states the algorithm has visited before. It only uses two dictionary functions: *insert(key)* and *find(key)*. The keys for the dictionary are the binary representations of the visited states.

We execute this algorithm on the models described in table 1.

3.1 Analytical Tools

The three analytical tools we present in this paper all are static tools. This basically means the data we gather and analyse is independent of the implementation of the datastructure. We implement each analysis method by extending a standard hash table. The results of the data is converted to a graph to make it easier to spot patterns. We don't allow the state search algorithm to complete a full search, but have chosen to let it run for N iterations (unique visited states) instead.

Listing 1: Algorithm for depth first search

```

Deque deque;
Set generated;
deque.push_back(initial_state);
while (!deque.empty()) {
    State state = queue.back();
    queue.pop_back();
    for (State s : successors(state)) {
        if (generated.find(s)==false) {
            generated.insert(s);
            deque.push_back(s);
        }
    }
}

```

Listing 2: Algorithm for breadth first search

```

Deque deque;
Set generated;
deque.push_back(initial_state);
while (!deque.empty()) {
    State state = queue.front();
    queue.pop_front();
    for (State s : successors(state)) {
        if (generated.find(s)==false) {
            generated.insert(s);
            deque.push_back(s);
        }
    }
}

```

3.2 State Order Analysis

We can assign every unique state a unique number, based on the order in which they are inserted into the dictionary. We keep track of when every state is used in *insert* and *find* calls. This is visualised in a chart, with time on the horizontal bar and order index on the vertical bar. Time T is a counter that starts at 0 and increases every time *insert* or *find* is called. The algorithm for the analysis can be found in listing 3. The algorithm generates its results in a comma-separated file. This file is processed by a small tool that generates a simple chart. The tool also adds horizontal lines to the chart that can be configured with a parameter. The chart is much smaller than the actual data set; every pixel in the chart represents possibly thousands of states. We use colours to distinguish between areas with low density and areas with high density, by converting the relative density to a HSV colour, with 100% saturation and 100% value and the hue varying from 0 (lowest density) to 360 (highest density).

This chart is useful for getting a general overview of how the algorithm uses the dictionary. We will see a mostly diagonal line going from the bottom-left corner to the top-right corner, which is the *insert line*. All other dots and lines in the chart are from *find* calls on earlier inserted states. Of course, the *insert line* doesn't need to be a straight line: if there are many *find* calls at some point and few *insert* calls, the line will curve. The chart can show that an algorithm visits old states often or not. For example, a chart with only a diagonal line does not visit old states, but only states recently inserted. It can be expected that dense horizontal lines are cache-friendly, because it means (depending on scale of the chart) that the same states are often visited, and will probably be in cache all the time. If there are many *find* calls in many differ-

Table 1: Models

Model name	Description
Eratosthenes	A parallel implementation of the sieve of Eratosthenes which is used to find prime numbers. For every prime number found, a new process is created. Because of this, state size increases during execution.
Leader2	Dolev, Klawe & Rodeh's algorithm for leader election in a unidirectional ring.
Peterson_N	Peterson's solution to the multi-process mutual exclusion problem.
Mobile1	A model of a cell-phone handoff strategy in a mobile network
PFTP	A model of a file transfer protocol
Snoopy	Snooping cache algorithm
Sort	Algorithm that concurrently sorts N random numbers

Listing 3: State order analysis algorithm

```

int counter = 0;
int ordercounter = 0;
FILE *record_file;
void insert(key) {
    if (find(key)) return;
    entry = insertInTable(key);
    entry->order = ordercounter++;
    record(counter++, entry->order);
}
bool find(key) {
    entry = findInTable(key);
    if (entry==null) return false;
    record(counter++, entry->order);
}
void record(int T, int state) {
    fprintf(record_file, "%d,_%d\n", T, state);
}

```

ent places this might indicate a higher chance on cache trashing. There might also be patterns that could indicate that the usual cache algorithm is not efficient for this model. In that case, it might be interesting to investigate possibilities to improve performance.

See figure 1 for an example of a State-T chart. The horizontal axis is T , the vertical axis is the state index. There is a clear diagonal line representing *insert* calls, dots in other areas indicate *find* calls. There are many diagonal lines, representing *find* calls in those areas. These kind of diagonal *find lines* can be expected in a depth first search. After backtracking to earlier states it is not unlikely that a string of successors of new states have been visited before in a similar order.

Figure 2 shows the State-T graph of the Leadership selection model (Leader2) when searched breadth-first. What we see is a single diagonal line. The graph shows that the algorithm appears not to consider states that haven't been visited recently. This is visible in the graph because all *find* calls appear near the diagonal line. A breadth first search does not have backtracking like a depth first search, so certain types of localisation in the state graph do not appear in this figure like they do in figure 2.

Figure 1: State-T chart of Eratosthenes DFS

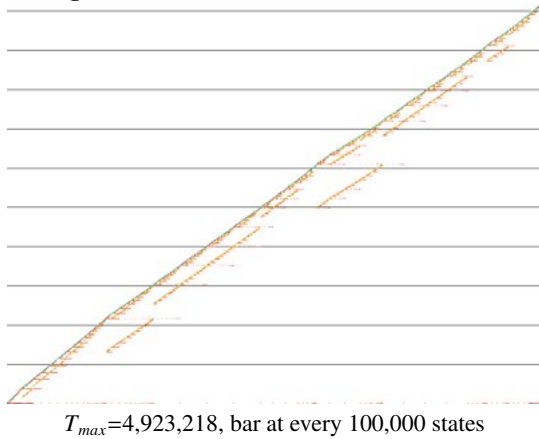


Figure 3: State-T chart of Peterson DFS

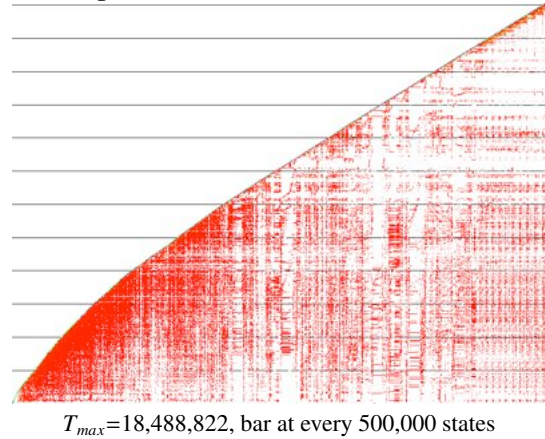
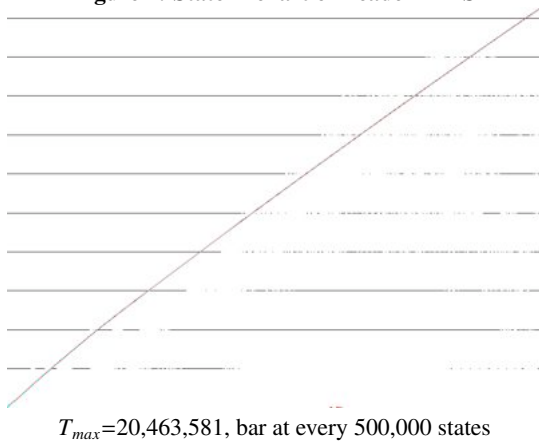


Figure 2: State-T chart of Leader2 BFS



Listing 4: δT analysis algorithm

```

int counter = 0;
FILE *record_file;
void insert(key) {
    if (find(key)) return;
    entry = insertInTable(key);
    record(counter, 0);
    entry->last = counter++;
}
bool find(key) {
    entry = findInTable(key);
    if (entry==null) return false;
    record(counter, counter-entry->last);
    entry->last = counter++;
}
void record(int T, int deltaT) {
    fprintf(record_file, "%d,_%d\n", T, deltaT);
}

```

The State-T chart of the Peterson_N model, searched depth-first, is interesting. From this graph in figure 3 we can see that old states will continue to be revisited (queried with *find*) often. Unfortunately, there does not appear to be an interesting pattern.

3.3 δT Analysis

We can keep track of the last time a state was accessed with *insert* or *find*. Whenever a state is accessed using *find*, we compute $\delta T = T_{now} - T_{last}$, or $\delta T = 0$ in case of an *insert* call. We display this in a chart as well, with time T on the horizontal axis again and δT on the vertical axis. Again, we use color to indicate the density of the chart. See listing 4 for the algorithm. As with the State-T analysis, a comma-separated file is generated which is then processed to create a chart. All *insert* listings are on the horizontal axis ($\delta T = 0$), anything above the axis represents a *find* call.

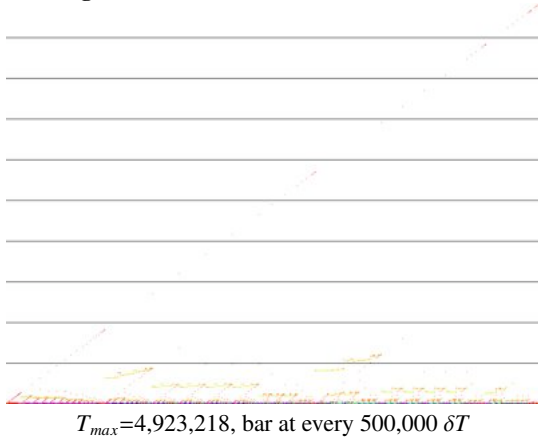
The chart can be used to suggest that the default caching algorithm is likely to be efficient. A bright line at the bottom of the graph (on the horizontal axis) indicates the algorithm often visits recently visited or inserted states again. It is likely that such states already are in L2 cache, depending on the size of L2 cache. If a different pattern appears, for example a line above the horizontal axis, indicating that states are revisited after a certain time,

this might indicate that the cache controller will often cache the wrong data, resulting in much cache trashing.

The chart in figure 4 looks nearly empty, most calls are located near the bottom of the chart, with occasionally a *find* call to an old state forming a dotted diagonal line. The chart shows that most *find* calls have a diagonal pattern in the chart. This can easily be explained by what could be expected in a depth-first search. In some models states will be generated via different paths. After some searching the algorithm will backtrack, eventually going back to the successors of the first generated stats. The state search algorithm will then find states that have been visited earlier. The exact pattern doesn't need to be diagonal, because this depends on the model that is being searched depth-first, but in this case a diagonal pattern appears. Most states are near the horizontal axis and might still be in cache. The same kind of pattern appeared for this pattern in figure 1.

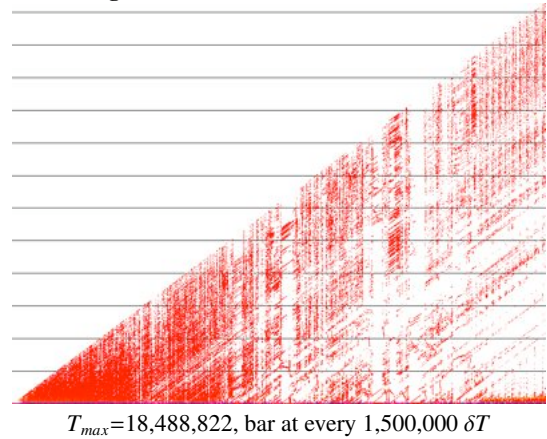
The State-T chart for the leader BFS, figure 2, basically shows only a diagonal line. Figure 5, the δT -T chart, shows detail the State-T chart is unable to show. The pattern here is actually quite interesting; for example it might be possible to try to optimize for the broad line in the bottom and be less optimal for the curved area above. The chart is generated with a horizontal bar every

Figure 4: δT -T chart of Eratosthenes DFS



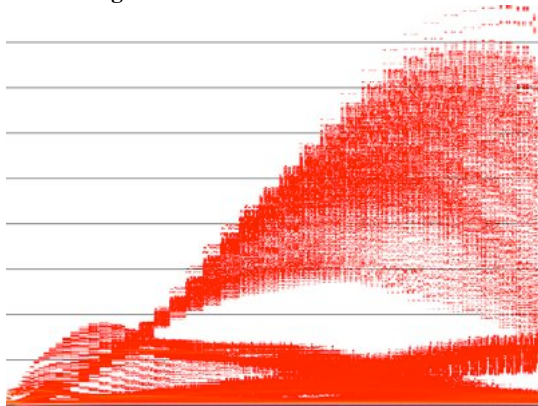
$T_{max}=4,923,218$, bar at every 500,000 δT

Figure 6: δT -T chart of Peterson DFS



$T_{max}=18,488,822$, bar at every 1,500,000 δT

Figure 5: δT -T chart of Leader BFS



$T_{max}=20,463,681$, bar at every 500 δT

500 δT .

There is a very clear difference between figure 4 and figure 5. Figure 4 is nearly empty, because old states are visited all the time. Figure 5 shows the opposite: old states are not visited at all. This may suggest that old states won't be visited at all, but that cannot be guaranteed, because the chart isn't generated for the entire search, only for the first part up to a certain amount of iterations. This is even likely if there are loops in the model.

Figure 6 shows the δT -T chart for the peterson_N depth-first search. The State-T chart already showed that old states will continue to be generated throughout the search and the δT -T chart shows the same.

3.4 Key Analysis

Lastly, we analyse the structure of the states. We keep a count of every byte value of every byte in the keys. In other words, we count how often each byte in the key has a certain value. Assuming the largest key encountered is N bytes long, we will have $256N$ counts. We calculate the diversity in each byte by taking the sum of all values divided by the maximum value. If a certain byte has only one value for every or almost every key, this value will be 1 or close to 1. If a certain byte has M different values

appearing roughly as much as the other, this value will be close to M .

Key analysis can be used to find an optimal reordering of the key to make it suitable for an efficient trie data structure instead of a hash table. For example, bytes with high variety are most useful high in a trie. We did not pursue further research in this. Key analysis is also useful for finding opportunities for table compression, like repeating patterns and low diversity.

The charts generated by a depth-first search and a breadth-first-search of the same model are very similar. An example of this can be seen in the charts in figure 8 and figure 9. The chart in figure 7 is an example of a key analysis that shows repeating patterns and low diversity. All three tables show that many bytes in the files have nearly always the same value. Table compression will result in smaller keys that can be processed faster, either in a trie and in a hash table.

Figure 7: Key analysis of Leader2 BFS

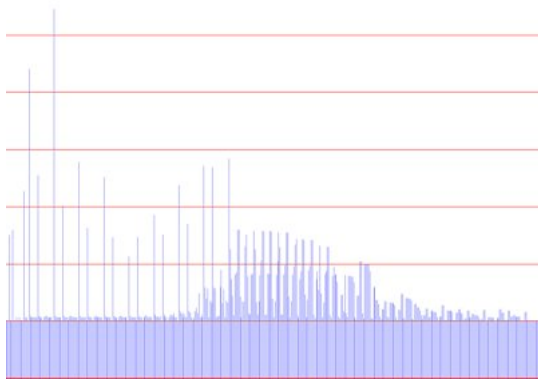


Maximum key length: 257 bytes.

4. IMPROVING PERFORMANCE

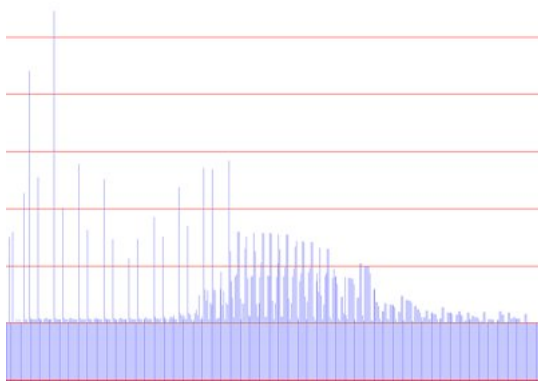
One of the goals of analysis is to find ways to improve performance of dictionaries used by specific algorithms. We focus in our tests on state search algorithms, using the analysis presented earlier. Usually, standard hash tables are used to implement such

Figure 8: Key analysis of Eratosthenes BFS



Maximum key length: 428 bytes.

Figure 9: Key analysis of Eratosthenes DFS



Maximum key length: 428 bytes.

dictionaries. We measure performance by running the tests on a dedicated machine with sufficient resources. By measuring the amount of time elapsed since starting the program, we measure how well it performs. The reason for this decision is that the alternative - measuring the time spent by the processor executing instructions for the process - might be incomplete, because it misses time spent waiting for I/O and time spent by processes on behalf of the executing process. We repeat the test seven times and take the median result to remove possible noise.

The experiments were performed on a 64-bit Linux system (kernel 2.6.18) with eight Intel Xeon E5335 processors (2 GHz, 4 MB cache) and 8 GB of main memory. Our code is single threaded and uses only one core of one processor.

We use a simplified chained hash table implementation as a reference implementation. This implementation only has *find* and *insert* calls. It deals with collisions using a singly linked-list. All memory is preallocated, so it is not needed to consider different strategies to deal with resizing the hash table.

4.1 Hash Tables and Hierarchical Memory

Our attempts to improve performance are based on a hash table. In addition to the main hash table we use an additional, small

hash table we call the *secondary table*. This table is to be in L2 cache all the time. The secondary table deals with collisions by dropping the old value from the table. Every time the hash table is accessed, we first look in the secondary table before continuing in the main table. This should guide the cache controller, so the secondary table is in cache. Having the secondary table in cache would allow us to use our own decision algorithm to store data in cache or not, essentially replacing the cache controller's algorithm with our own. The algorithm of our two-table approach can be found in listing 5.

4.2 Optimizing a Secondary Table

The goal of our first attempt is to determine optimal secondary table sizes and the best general algorithm. We use different sizes of the secondary table and tested this on every model, depth-first search and breadth-first search. Algorithm 1 is very simple and stores every key in the secondary table when it is accessed. The implementation of algorithm 1 is simple and can be found in listing 6. Algorithm 2 is more complex. It is like algorithm 1 when dealing with *insert* calls. After accessing a key in the main table using *find*, it uses a random number generator to decide whether to insert the key in the secondary table or not. The chance depends on the relative size of the secondary table versus the main table. The pseudo-code for this algorithm can be found in listing 7. Algorithm 2 is essentially a probabilistic approximation of the idea that often-accessed data should be in cache. This way we avoid tracking usage statistics for every entry in the hash tables. An often-accessed entry has a larger chance to be promoted back into the secondary cache. By manipulating the modifier MOD, this chance can be tweaked. Our implementation is optimized for speed by using bitwise operations. We tested the data structure and the two algorithms using the Eratosthenes model.

Listing 6: Implementation of Algorithm 1

```
bool promoteThis(_entryMain *entry) {
    return true;
}
```

Listing 7: Implementation of Algorithm 2

```
#define MOD 1
int rndMask = (mainMask >> MOD) & (~secondaryMask)
bool promoteThis(_entryMain *entry) {
    // chance: 2^MOD * (secondarySize / mainSize)
    return (rng() & rndMask) == 0;
}
int rng(); // custom random number generator
```

4.3 Optimizing for the Leader2 Model

Analysis shows that a state search on the leader2 model has a pattern in the δT graph that might be useful for improving performance. The wide line in figure 5 is the area that can be described with $\delta T < 800$. Our algorithm is a modification of the algorithm in the general approach: we keep track of δT in our hash table and insert a key in the secondary table (after a *find* call) only if $\delta T < 800$. We then measure the performance of this datastructure against the performance of the standard datastructure. The algorithm can be found in listing 8.

Listing 5: Hash table with secondary table

```

typedef struct {
    void *key;
    size_t length;
} key_t;

struct _entryMain {
    key_t *key;
    _entryMain *next;
} mainTable[];

struct {
    key_t *key;
    boolean inMain;
} secondaryTable[];

int mainMask, secondaryMask;

// mainSize must be a power of 2
// secondSize must be a power of 2
void init(int mainSize, int secondSize) {
    mainMask = mainSize-1;
    secondaryMask = secondSize-1;
}

void drop(int entry) {
    key_t *key = secondaryTable[entry].key;
    if (!secondaryTable[entry].inMain) {
        int entry1 = hash(key) & mainMask;
        if (mainTable[entry1].key!=null) {
            _entryMain *copy = copy(entry1);
            mainTable[entry1].next = copy;
        }
        mainTable[entry1].key = key;
    }
    secondaryTable[entry].key = null;
}

void promote(_entryMain *entry) {
    key_t *key = entry->key;
    int entry2 = hash(key) & secondaryMask;
    if (secondaryTable[entry2].key!=null)
        drop(entry2);
    secondaryTable[entry2].key = key;
    secondaryTable[entry2].inMain = true;
}

void insert(key_t *key) {
    int entry2 = hash(key) & secondaryMask;
    if (secondaryTable[entry2].key!=null)
        drop(entry2);
    secondaryTable[entry2].key = key;
    secondaryTable[entry2].inMain = false;
}

bool find(key_t *key) {
    int hash = hash(key);
    int entry2 = hash & secondaryMask;
    if (secondaryTable[entry2].key==key)
        return true;
    int entry1 = hash & mainMask;
    _entryMain *entry = &mainTable[entry1];
    while (entry!=null && entry->key!=key)
        entry=entry->next;
    if (entry==null) return false;
    if (promoteThis(entry)) promote(entry);
}

bool promoteThis(_entryMain *entry);
int hash(key_t *key);

```

Listing 8: Implementation of custom algorithm for Leader2 BFS

```

int counter = 1;
void drop(int entry) {
    ...
    mainTable[entry1].key = key;
    // inserted line:
    mainTable[entry1].last =
        secondaryTable[entry].last;
    ...
}

void promote(_entryMain *entry) {
    ...
    secondaryTable[entry2].last = entry->last;
}

void insert(key_t *key) {
    ...
    secondaryTable[entry2].last = counter++;
}

bool find(key_t *key) {
    ...
    if (secondaryTable[entry2].key==key) {
        // inserted line:
        secondaryTable[entry2].last = counter++;
        return true;
    }
    ...
    if (entry==null) return false;
    // replace last line with:
    int deltaT = counter-entry->last;
    entry->last = counter++;
    if (deltaT < 800) promote(entry);
}

```

4.4 Results for Secondary Table Optimization

Table 2 shows that there is no clear optimal configuration. It seems the configuration with a 64k table is best, but the difference is very small. When doing further analysis by counting how many "secondary table misses" (*find* calls only) there are, also shown in table 2, the probabilistic approach seems to be better in theory. Several questions remain unanswered and need to be investigated: is the secondary table actually in cache or not? Is the gain on faster memory access is larger than the loss of extra calculations and extra data fields to maintain?

Our approach is probably the same as what the cache controller already does, so it will probably be slower than the cache controller's own hardware implementation. This could explain why our attempt doesn't have any interesting results.

4.5 Results for Leader2 Optimization

Table 3 shows that there is no improvement. The differences in time are minimal and inconsistent. There are several explanations possible that should be considered. We don't know whether the secondary table is in L2 cache all the time, like we want. The algorithm also has too many indirections: the entry in the main table and the secondary table have a pointer to the actual entry that is in the linked list and these entries each store a pointer to the dynamically sized key. This inefficient use of memory might cause more cache trashing. Perhaps key compression should be used to

Table 2: Results for secondary table optimization

Algorithm	Scnd. size	Time (sec.)	Misses
Mock test		29.974	
Original		6.367	
Alg. 1	16k	6.092	33295
Alg. 1	32k	6.529	29709
Alg. 1	64k	5.583	9264
Alg. 1	128k	6.375	4993
Alg. 1	256k	6.608	2991
Alg. 2 MOD=0	32k	6.685	16951
Alg. 2 MOD=1	32k	6.776	17421
Alg. 2 MOD=2	32k	6.699	18437
Alg. 2 MOD=0	64k	6.730	8725
Alg. 2 MOD=1	64k	6.723	9246
Alg. 2 MOD=2	64k	5.515	10156
Alg. 2 MOD=0	128k	5.713	4500
Alg. 2 MOD=1	128k	6.254	4993

Time values are $T_{elapsed} - T_{mock}$.

Misses are *find* calls to entries not in the secondary table.

Table 3: Results for Leader2 Optimization

Secondary table size	Algorithm 1 (Always promote)	Custom algorithm ($\delta T < 800$)
8k	63.155 sec	63.098 sec
16k	65.628 sec	63.670 sec
32k	62.910 sec	63.111 sec
64k	63.779 sec	65.077 sec
128k	62.952 sec	62.956 sec
256k	63.115 sec	63.009 sec
512k	62.375 sec	62.661 sec

make the keys much shorter, enough to minimize wasted space when removing all indirections from the data structure. Furthermore, states that were visited less than 800 *find* or *insert* calls ago should probably be in L2 cache already. Hence we gain no improvements in performance.

5. CONCLUSIONS

The result of our research so far is that it is unlikely that the performance of hash tables can be improved for state search algorithms on the models we tested, using secondary tables. It is not proven that the secondary table is in L2 cache, because of cache trashing in general and trashing due to following indirections in particular. Applying table compression and removing indirections might alleviate this problem. Our analysis shows that table compression may make keys much shorter. Our analysis also shows that it is likely that visited states are already in cache, or have never been seen yet - especially the δT -T charts show this. There isn't much to gain in those cases.

The cache controller is already performing well. There could be room for improvement if the cache controller is inefficient; how-

ever, this seems not the case. If it would be the case, it is still unclear if trying to trick the cache controller to cache the right data is the best solution to the problem - it might be better to simply replace the cache controller with custom hardware.

5.1 Further Work

It might be interesting to implement a dictionary using a trie instead of a hash table, using key compression for shorter keys. A trie usually has a complexity of $\log(K)$ for (compressed) key size K . Our key analysis shows which bytes of the key are most interesting higher up the trie, due to a high variation in values. Advantages of using a trie are that no hash value needs to be calculated and the key will be parsed only once, instead of every time a possible match is found. Further improvements might even be gained by preallocating one block of memory for the first N levels of the trie. N should be chosen so this block is in L2 cache at all times.

A different approach is finding a model that isn't optimal for current caching strategies, or changing the state search algorithm, or applying the tools for analysis on completely different algorithms with a dictionary as their main data structure.

Our analysis is far from complete and especially lacks an estimation of cache efficiency. It might be interesting to generate a chart that shows which calls to *find* are expected to be cache hits and which are expected to be cache misses. It might be possible to use tools like Valgrind for this.

REFERENCES

- [1] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43:257–264, 1992.
- [2] Gerard J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *In Proceedings of Third International SPIN Workshop*, 1997.
- [3] Bob Jenkins. Hash functions and block ciphers. <http://burtleburtle.net/bob/hash>.
- [4] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, 1998.
- [5] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Journal of Algorithms*, 2001.
- [6] Arash Partow. General purpose hash function algorithms. <http://www.partow.net/programming/hashfunctions/index.html>.
- [7] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *ACM SIGCOMM*, pages 181–192, 2005.
- [8] Maks Verver. Evaluation of a cache-oblivious data structure. 2008.