CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Efficient String Representation in Erlang/OTP

Master's thesis in Master Programme Algorithms, Languages and Logic

ANDREJ LAMOV

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2016

Master's thesis 2016: Efficient String Representation in Erlang/OTP

Efficient String Representation in Erlang/OTP

ANDREJ LAMOV

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Science and Engineering
Division of Software Technology
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2016

Efficient String Representation in Erlang/OTP
ANDREJ LAMOV

Supervisor: Koen Claessen, Department of Computer Science and Engineering
Examiner: Bengt Nordström, Department of Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Division of Software Technology
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Printed by Chalmers Reproservice
Gothenburg, Sweden 2016

Abstract

A string in Erlang is implemented as a linked list of integers. This leads to a large memory overhead on modern hardware (64 bits) causing each character to take 16 bytes, especially compared to the natural representation where each character takes 1 byte. In this report we show and compare alternative string representations to achieve less overhead. Furthermore, we explore the memory management in Erlang's abstract machine (BEAM), and show how to measure and reason about its memory usage.

As a case-study we use Ericsson's SGSN-MME, a massively concurrent Erlang system running hundreds of thousands of Erlang processes. We have chosen a part of the system where the existing code has performance issues involving hostname processing and representation. In particular we analyse the behaviour of an existing algorithm used for selecting gateway hostnames by performing suffix matching. We give examples of improvements of this, both algorithmic and storage-wise.

Acknowledgements

I would like to thank my supervisors at Ericsson, Urban Boquist and David Wahlstedt, and my supervisor at Chalmers, Koen Claessen. You three possess an enormous amount of knowledge about computer science, and I am very grateful to have had the opportunity to work with you.

Contents

1 Introduction

1.1 Strings in Erlang

A string in the functional language Erlang [1] is implemented as a linked list of integers, and the representation of the string "services.telenor.se" is seen in Figure 1.

```
[115,101,114,118,105,99,101,115, % "services"
 46,                             % "."
 116,101,108,101,110,111,114,    % "telenor"
 46,                             % "."
 115,101]                        % "se"
```

Figure 1: A string in Erlang

The elements of a linked list is known as cons cell, a structure that holds the actual data (a character in this case) and a pointer to the next cons cell.

The first two cons cells in the string "services.telenor.se" are seen in Figure 2. The character `s` is represented by the integer `115` that in binary form is `01110011`, located as the 8 most significant bits in the first cons cell. The second cons cell is "e", represented by the integer `101`, that in binary form is `0110010`. Assuming a char needs 1 byte of storage, and if stored as an integer on a 64-bit machine, the integer and pointer will together take 16 bytes. This creates list entries where 1 byte is actual data and 15 bytes is plain overhead. On a 64-bit machine, each cons cell will contain `93.75%` of wasted space, demonstrated by Figure 2.

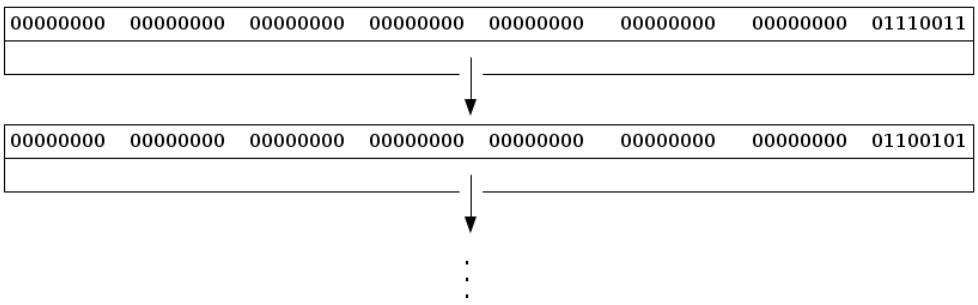


Figure 2: Cons cells of prefix "se" in string "services.telenor.se"

Figure 3 shows how strings can be represented as linked lists of characters, like in Haskell.Since functional languages are based on recursive definitions, and a direct mapping of the recursive data

type to a concrete linked list as seen in Figure 2 is very natural.

```
data List a = Nil | Cons a (List a)

type String = List Char
```

Figure 3: List implementation in Haskell

Linked lists are easy to pattern match, the main technique used to decompose structures in functional languages. It is for example an *O(1)* operation to match out the head and the tail. This ability makes recursion the natural method used for looping.

1.2 Use of strings in Ericsson's SGSN-MME

Ericsson SGSN-MME, short for Ericsson Serving GPRS Support Node and Mobile Management entity, is a network element in the mobile core network of a 2G, 3G and 4G mobile telecommunication system. The node is a massive concurrent system partly implemented in the functional and highly concurrent language Erlang [1].

One such SGSN-MME runs several Erlang virtual machines and can handle 1 000 000 phones concurrently, where each phone is handled by a unique Erlang process.

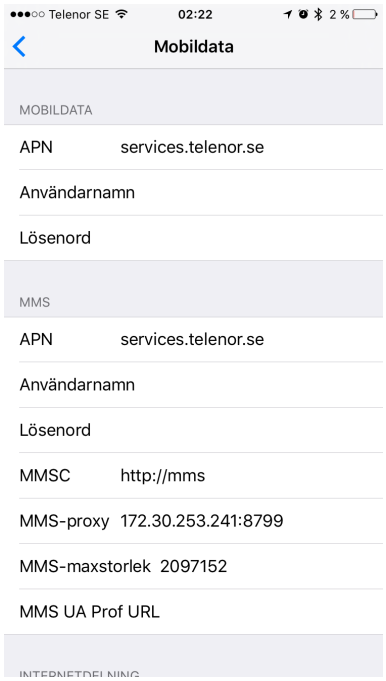


Figure 4: String in a phone

In Figure 4 we see the mobile data configuration in an iPhone, provided by the operator Telenor. The row APN, Access Point Name, with the value `services.telenor.se` serves as a parameter for an SGSN-MME, when selecting a suited gateway in the mobile core network. Figure 5 shows and example of how the APN is used to fetch a set of gateway hostnames, and then a gateway is selected.

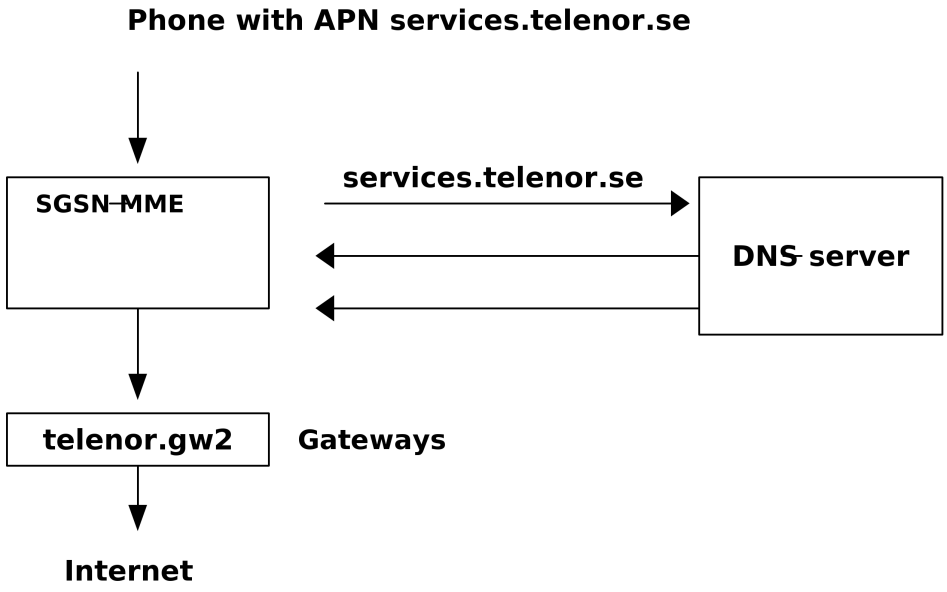


Figure 5: SGSN-MME

The phone can only access the internet through specific gateway, unique for the operator (`services.telenor.se` in this example), services provided by the subscription and the geographical position of the phone. According to the 3GPP, a telecommunication standard, the information about available gateways is stored in on a DNS-server, and each gateway is then identified by a hostname.

An Erlang process needs to fetch the hostnames of the gateways via the DNS-server, process the hostnames and then pick the best suited gateway. The procedure of assigning gateways to a phone is called Gateway selection.

The massive amount of phones handled by the Erlang node, together with a big mobile core network with hundreds of gateways, causes high memory consumption in the system, especially during the gateway selection procedure.

1.3 About this report

In this report we will explore methods to decrease the overhead described in Section 1.1 and in particular in the context of Section 1.2.

We will first give an overview of the Erlang language with a detailed description of some parts of Erlang's virtual machine, the BEAM (Bogdan/Björn's Erlang Abstract Machine) [2]. With this background, we then cover necessary facts about the mobile core network and the use of hostnames by Erlang in the SGSN-MME. This will serve as a constraint for the alternative string representations proposed below.

Our hypothesis is that string processing in the BEAM has room for optimizations in certain respects, both when it comes to CPU and memory behavior. We have explored how the the strings in the DNS code can be represented in alternative ways.

2 Erlang

To fully understand the implementation details that will follow, internal parts of the BEAM has to be described. However, we begin this chapter by giving a brief overview of the key features of Erlang, and then dig into some details of its internal implementation, thus basic familiarity with Erlang will be assumed. A great tutorial for Erlang is `www.learnyousomeerlang.com`.

The overall aim of this chapter is to describe how memory is managed and how Erlang terms are represented during execution of code in a process. Describing such a system is not a straight-forward task, since many of the parts intertwine and depend on each other.

2.1 Overview

Erlang is a high-level, highly concurrent and functional language running on a virtual machine. It is dynamically typed and has an automatic memory management using garbage collection. The ability to provide a platform to build highly concurrent applications comes from the process-centric design. Processes are very light-weight and mainly self-contained. A process has a private heap area, and performs its own garbage collection, without interfering with other processes or stalling the virtual machine.

The communication between processes is done through message passing, with a non-blocking send and a blocking receive. A process is either executing code or waiting in a receive state for on incoming messages. The private heap design enforces a message to be fully copied from the private memory area of the sender to the private memory area a receiver.

Since the processes are isolated and self-contained, they may point to different versions of the same module code. A set of processes may refer to a new version of a module, and another set of processes may refer to the old version of the same module, while the processes are running in the BEAM at the same time. This design resembles very much the design of an operating system, with independent programs running side by side.

Erlang has immutable data and utilizes pattern-matching to deconstruct compound structures. The language has five basic data-types, shown in Figure 6.

In Erlang, entities of data are called *terms*. As example, a small integer in Erlang is a term and a list is a compound term that consists of other terms. [3]

```
-- Every entity of data in Erlang is called a term.
-- Terms can be simple or compound.

-- simple
1                              -- integer
hello                          -- atom
<<1,2,3,4>>                    -- binary

-- compound
{1,2,3}                        -- tuple
[1,2,3]                        -- list
[1,2, {<<1,2,3>>, hello}, []] -- list with different terms
```

Figure 6: Examples of basic types in Erlang

Integers can be of arbitrary size and are internally represented by a composed type when the size exceeds the machine-word size. An *atom* can be best described as an enumeration type, like the one found in C. The big difference is that atoms can be created dynamically. Atoms are created dynamically and are only allocated once; internally they are just represented by unique integers. A *list* is implemented as a linked-list, while a *tuple* is a packed sequence of terms, resembling regular arrays. A *binary* is a sequence of bits. A compound term is a collection of of arbitrary Erlang terms.

Open Telecom Platform (OTP) is a collection of libraries made to build Erlang based applications [1]. BEAM is the virtual machine that can execute Erlang code. ERTS is the Erlang Run-Time System that includes the BEAM and the standard libraries provided by OTP [4].

2.2 Scheduler

Many functions are directly implemented in the BEAM by pure C code. They are referred to as built-in functions (BIFs). Common functions in the standard library, as `lists:reverse`, are BIFs. The BEAM communicates with external processes and services through *ports* and native-implemented functions, *NIFs*, that is functions written in C working with the C versions of Erlang types. From BEAMs perspective, ports make the external environment look like an Erlang process, and NIFs make the external environment look like an Erlang function.

As explained by Virding in [5] a *scheduler* can be viewed as a semi-autonomous BEAM, that is able

to execute code, manage processes, manage memory, provide allocation services for processes, run bifs, nifs and ports.

To achieve concurrency, the BEAM starts several schedulers run on their own operating system thread. The standard configuration is one scheduler per processor core. Each scheduler will then independently select processes from its subset of all running processes, and execute their code in a pseudo-parallel manner. A newly spawned process is handled by the scheduler of the process that initiated the spawn.

The schedulers have two goals. The first is to spread the work-load as evenly as possible on the system, to make sure that parts of the system are not over-loaded or under-loaded. The second goal is to create compact memory locality, so that data can be fetched and written fast. These two goals are essentially conflicting, and the schedulers need to take both into consideration.

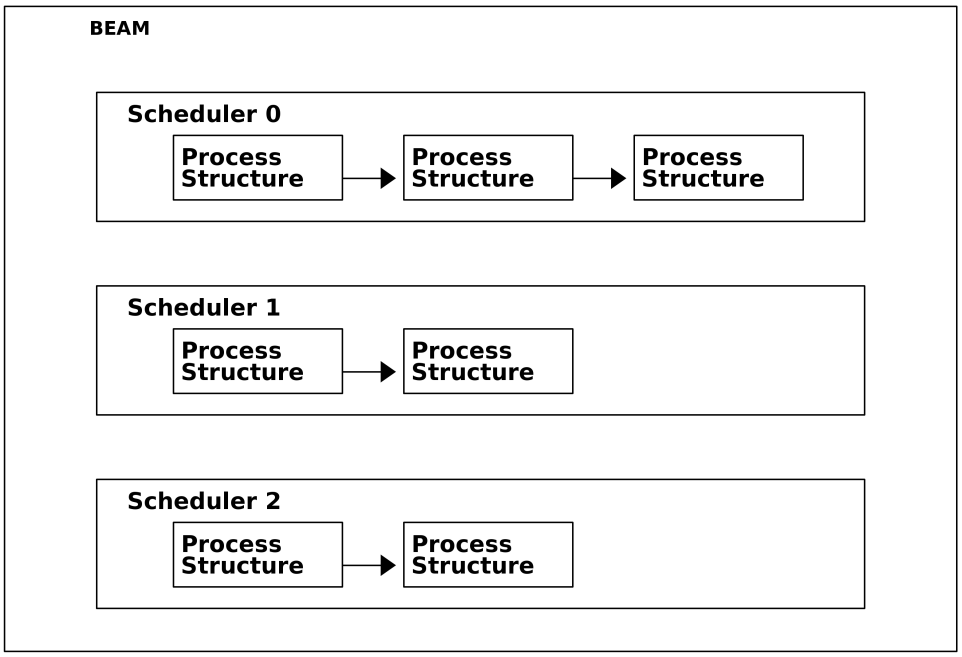


Figure 7: The Beam Schedulers

There are three main methods used by a scheduler to establish load-balance.

The first method is simply where the scheduler decides what process should execute. A scheduler

has a first-in first-out queue of processes that it manages, and a process is rescheduled every 2000 reductions in the virtual machine. Operations such as regular Erlang function calls, garbage collections and BIF/NIF calls will affect the reduction counter, and when the process reaches 2000 reductions, some other process will be selected for execution by the scheduler.

The second method is process stealing. A scheduler that has an empty process queue may steal processes from another scheduler. A scheduler is only allowed to steal runnable processes, since stealing a suspended process would not change the load balance. A not runnable process is for example a process in receive state, without any message in its message queue. Such a process is put on suspend and removed from the scheduler queue. The process is only put back in the queue if a message arrives and is copied to the process heap. Note that even if a process is blocking in a receive statement, the scheduler will not be blocked by it. The scheduler schedules another process that is not in a receive state, to obtain as high load as possible.

The third method is when a scheduler reaches 40000 new reductions, after the BEAM has been initialized. The first scheduler that reaches this amount of new reductions can announce itself as a master scheduler. That master scheduler will inspect how much work the other schedulers are performing, and how much work the schedulers have performed so far. Based on that data, the master can suspend a scheduler and decrease the number of used threads in the BEAM. Handling the same amount of processes but with fewer threads may improve memory locality, and makes it possible for the operating system to utilize the cores for other work.

2.3 Global BEAM Resources

There are global resources in the BEAM shared between all schedulers and processes. A process will have pointers to theses resources, if they are used.

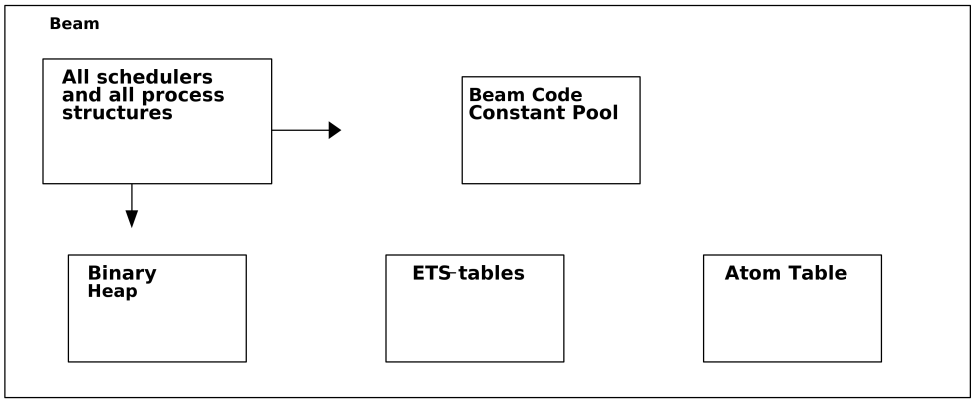


Figure 8: Global BEAM resources outside of process heap

The Beam Code and Constant Pool holds compiled Erlang code and literals, that is, fixed values in the source code, declared in modules. Erlang Term Storage (ETS), described in Section 2.5, are shared key-value tables that can store any term. The *binary heap*, described in Section 16, is the area where binaries larger than 64 bytes, are stored. The *atom table* is a finite table that stores atoms. The table is a mapping between integers and atom names, since atoms are internally represented as integer. This table can only grow and is never garbage collected.

2.4 Virtual Machine Loop

A scheduled process is represented in the BEAM as a structure with references to different areas. The most important areas are the stack and the private memory. Figure 9 shows the different parts involved in the BEAM loop.

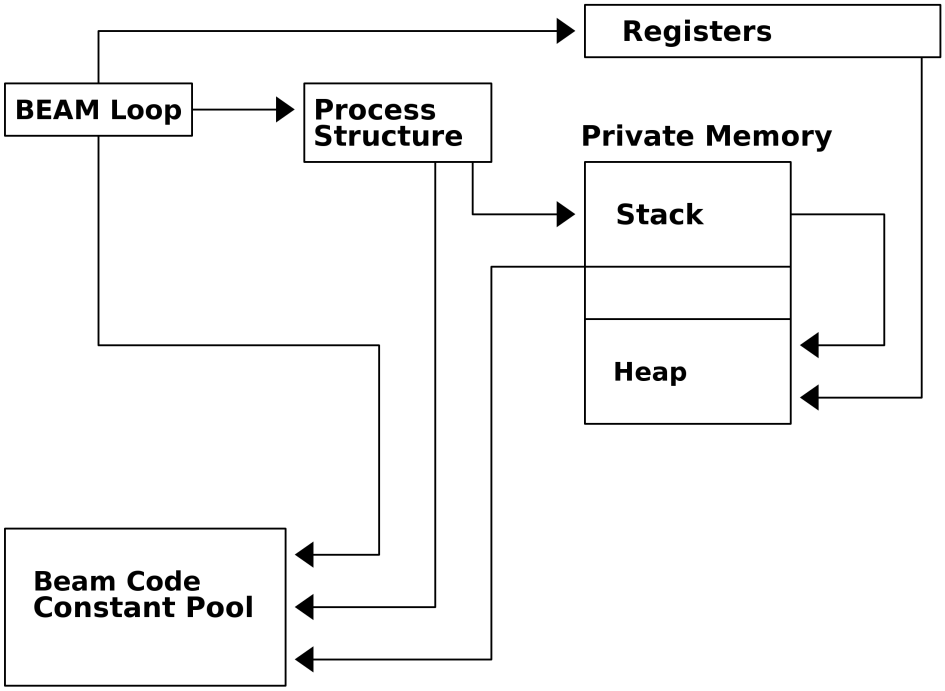


Figure 9: The Beam Loop

As described in Nybloms lectures [6, 7], the private memory consists of a stack and a heap. Variables and single terms (not part of a compound term) such as integers and atoms can be stored directly on the stack, while terms such as binaries, lists, tuples and compound terms are stored on the heap, referenced from the stack to the heap.

The stack also stores the *continuation pointer* that is the return address used to continue execution after a function call returns. Terms stored on the stack may also refer to data in some global BEAM resources, seen in Figure 8, such as atoms, literals and references to ETS-tables.

The stack and the heap are allocated on the same memory block, growing towards each other. This design makes it easy to detect when the private memory block is full and needs to be garbage collected and reallocated, since the pointers to the stack and the heap will overlap. We will go through garbage collection in detail in Section 2.7.

The *Registers* are used by the BEAM to store temporary values during execution, and are not a part

of the process structure, though terms stored in the registers may refer to terms on the heap.

The code reference is either the module and function name, for a newly started process, or a pointer into the actual code. The code, called BEAM-code, is Erlang code compiled to byte-code that can be interpreted by the internal abstract machine.

2.5 Erlang Term Storage (ETS)

As explained by [8] ETS-tables are, together with message-passing, a way for processes to communicate with each other. It acts as a shared memory that uses locks to handle parallel read and write requests. Since Erlang is referential transparent, all data that is requested from an ETS-table must be copied to the process private heap.

2.6 Internal Term Representation

This section describes the type-system and the different types used inside the BEAM. First, the ETerm is introduced, that is the C-representation of an Erlang Term. Before Erlang Terms are shown, the tag-scheme to interpret the types as run-time used in Erlang is introduced. A walk-through of the tag-system should give a deeper understanding of how terms are constructed and used in the virtual machine. This is however not necessery for the understaning of this report. Finally we will explain the different binary types used internally by the BEAM, since binaries are later used when alternative representations are presented.

The BEAM soruce code [9] shows the low-level C representation of a term is called *Eterm*. An Eterm has the size of a machine-word. A simple term is internally represented by one Eterm, while anything that does not fit in a single machine word is a collection of ETerms. Consequently, an Eterm can represent a value, pointer or a header.

```
typedef unsigned int Eterm;
```

Figure 10: Eterm (`sys.h`)

Many parts of the VM operate with ETerms. The stack and heap of a process store Eterms, and the registers of the VM are ETerms.

2.6.1 Types and tags

Erlang uses dynamic typechecking with a tag-scheme, as explained in [10] and [11].

An implementation of such a type-checker is to use *uniform* data representation, and store the type information together with the value, as one unit. The container for such a unit has usually the size of a machine-word, and is called a *handle*. The Eterm in the BEAM is an implementation of a handle, as seen in Figure 10.

The ETerm will consist of a value and a tag, describing what type the value has. Erlang uses a hierarchical tag-scheme, where the tags, residing as the least significant bits of the ETerm, are interpreted in stages. This means that either the tag has direct information of what type the ETerm has, or it indicates that more bits are needed, to get the correct type information. This is examplified in Figure 11, where value A of type *integer* is denoted by one tag, while the value B is of type *long* needs a secondary tag, since a single tag is not enoough to encode all the types that are needed. The first tag will then indicate that the type-information is stored in the next tag.

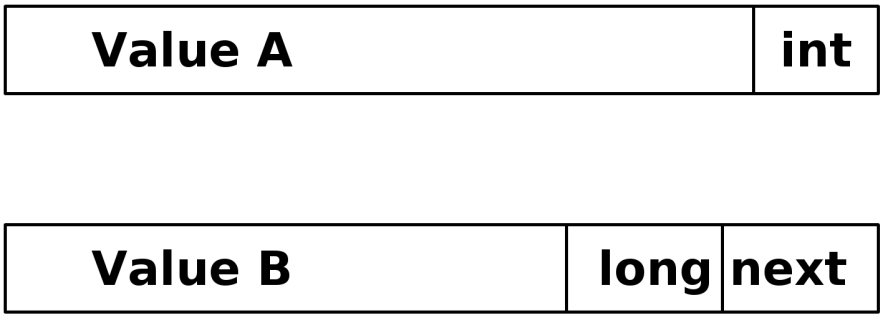


Figure 11: Tag example

The next section will go through the tags used by the BEAM.

Primary tag These are the four primary tag types, stored as the two least significant bits of an Eterm:

- `11` means immediate, values that fit into the handler and have a secondary tag specifying the actual type.
- `01` means list pointer to a cons cell, used to build up lists.

- `10` means boxed, complex types such as tuple, float, big num, binary header, function closure, etc.

- `00` means continuation pointer (return address on stack) or header of a complex structure stored on the heap.

In Figure 12 the four primary tags are shown. The list pointer points to a list with two elements, where the list pointer type is reused in cons cells. The continuation pointer and header Word use the same primary tag. There is no possibility to mix these two up, since the stack pointer only can reside on the stack, and a header word is always allocated in the heap.

The stack may only store immediates, list pointers, boxed pointers and return addresses, that is, the objects on the left side in Figure 12. The heap on the other hand can contain all the different types, besides the continuation pointer, that has the same tag as a header word.

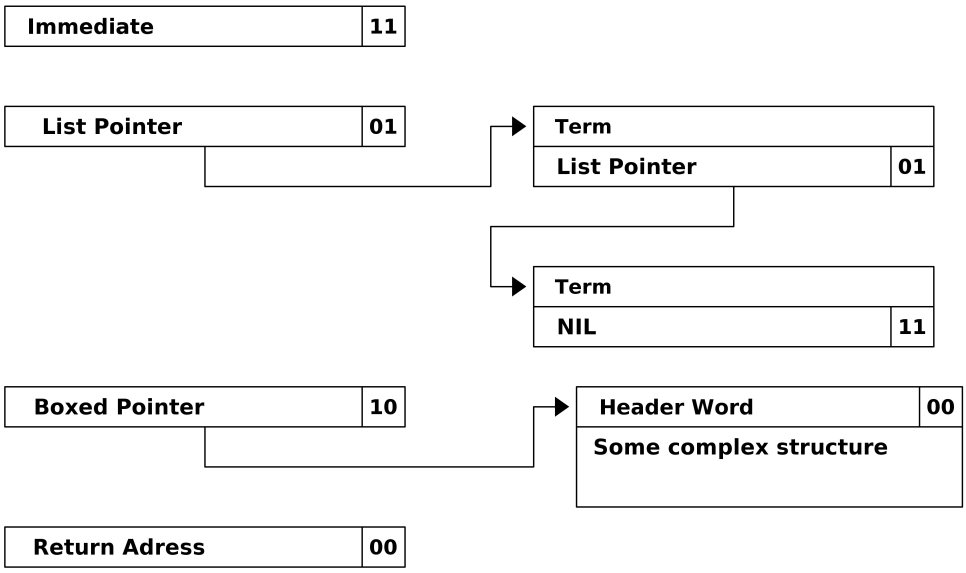


Figure 12: Primary Tags

Immediate tag These are the four *secondary* tags of an Immediate value, taking up two bits after the primary tag `11`:

- `00` means local pid.

- `01` means local port.
- `10` means a secondary immediate tag.
- `11` means small number including sign with the number space of 4 bits less than the word length.

Figure 13 shows an example of how the number `3` would be represented. The number is represented with 28-bits ones' compliment on a machine with 32-bit word size [6].

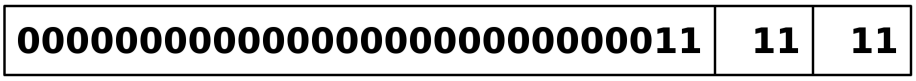


Figure 13: The number 3 represented as a small number

The tag `10` indicated that there is an other two bits tag for the following types:

- `00` means atom.
- `01` means catch address on stack (on a try-catch structure).
- `11` means NIL an empty list.

Figure 14 shows an example of how an atom is encoded on a machine with 32-bits word length. Atoms, that are created dynamically during run-time, are encoded as *enumeration types*, found in languages like C. Let the third introduced atom in some BEAM be `hello`, then `hello` will be encoded as integer `3`, and it gets index 3 in the atom table.

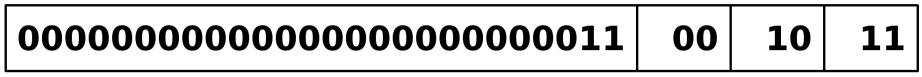


Figure 14: The atom `hello` encoded as the integer `3`

Boxed pointer A header word is the first ETerm referenced by a boxed pointer. The header word describes the type of a compound term. The primary tag is `00`, and the next four bits means the type of the complex structure. [6]

- `0000` means arity of a tuple.
- `001x` means big num (a number that does not fit in a small number, as seen in Figure 13). The last bit `x` indicates if the number is signed or unsigned, where `1` is signed, and `0` unsigned.
- `0101` means function Closure.
- `0001` means internal binary match state, a structure used for pattern matching.
- `1000` means reference counted binary, also known as *refc*, are big binaries stored on the binary heap shown in Figure 8.
- `1001` means heap binary, a binary stored on the heap.
- `1010` means sub-binary, a binary header that points into a heap binary or a refc.

An example of a boxed pointer is seen in In Figure 15, where we show a tuple. The header word has the total tag `000000` and the rest of the bits states the arity of the tuple.

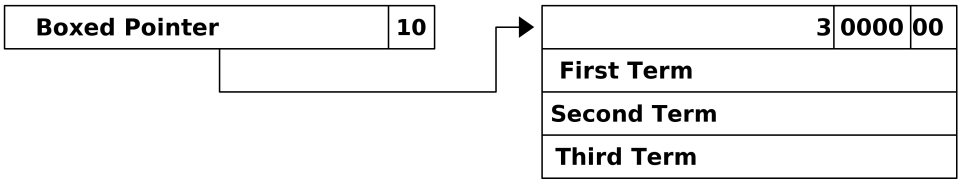


Figure 15: Tuple

Tag Implementation A part of the implementation of the tag-scheme is examplified in Figure 16, that demonstrates how a tuple shown in Figure 15 with arity 3 is created.

```
 1  typedef unsigned int Eterm;
 2
 3
 4  #define ARITYVAL_SUBTAG         (0x0 << _TAG_PRIMARY_SIZE) /* TUPLE */
 5  #define _TAG_PRIMARY_SIZE       2
 6  #define TAG_PRIMARY_HEADER      0x0
 7  #define TAG_PRIMARY_BOXED       0x2
 8  #define _TAG_HEADER_ARITYVAL    (TAG_PRIMARY_HEADER|ARITYVAL_SUBTAG)
 9  #define _HEADER_ARITY_OFFS      6
10
11  #define TUPLE3(t,e1,e2,e3) \
12          ((t)[0] = make_arityval(3), \
13          (t)[1] = (e1), \
14          (t)[2] = (e2), \
15          (t)[3] = (e3), \
16          make_tuple(t))
17
18
19  #define make_arityval(sz)    _make_header((sz),_TAG_HEADER_ARITYVAL)
20  #define _make_header(sz,tag) ((Uint)(((Uint)(sz) << _HEADER_ARITY_OFFS) + (tag)))
21
22  #define make_tuple(x)   make_boxed((x))
23  #define make_boxed(x)   _ET_APPLY(make_boxed,(x))
24  #define _ET_APPLY(F,X)  _unchecked_##F(X)
25  #define _unchecked_make_boxed(x) ((Uint) COMPRESS_POINTER(x) + TAG_PRIMARY_BOXED)
26  define COMPRESS_POINTER(AnUint) (AnUint)
```

Figure 16: Internal creation of a tuple (snippets from `erl_term.h` and `sys.h`)

Binary A *bitstring* term is a sequence of bits. A *bitstring* is a *binary* term if the the total number of bits are a multiple of 8 bits. Bitstrings and binaries have the same internal implementation, and will both be referred to as binaries when the internal implementation is examined.

```
<<1,2,3>> % binary

<<1:33>> % bitstring
```

There are four binary representations used internally in the BEAM: *HeapBin*, *Refc*, *SubBun* and *Match Context*. We will only go through the first three, and explain them by showing its source code from the OTP repository [9].

- *HeapBin* is a binary where the whole data is allocated on the process heap, and treated by the garbage collector as a compound term, similar to a list or tuple. As seen in Figure 17, the metadata together with the pointer takes 3 words.

```
typedef struct erl_heap_bin {
    Eterm thing_word;           /* Subtag HEAP_BINARY_SUBTAG. */
    Uint size;                  /* Binary size in bytes. */
    Eterm data[1];              /* The data in the binary. */
} ErlHeapBin;
```

Figure 17: Heap Binary Struct `erl_binary.h`

- *Refc* is a binary type for large binaries stored on the global binary heap shown in Figure 8. This binary term is built up of two parts. The first part, called *ProcBin*, is the structure stored on the process heap. A ProcBin object refers to the actual binary data stored on the global binary heap. The ProcBin struct in Figure 18 is a bit larger than the heap binary structure, due to the pointers to the actual binary and internal "magic" flags.

```
typedef struct proc_bin {
    Eterm thing_word;           /* Subtag REFC_BINARY_SUBTAG. */
    Uint size;                  /* Binary size in bytes. */
    struct erl_off_heap_header *next;
    Binary *val;                /* Pointer to Binary structure. */
    byte *bytes;                /* Pointer to the actual data bytes. */
    Uint flags;                 /* Flag word. */
} ProcBin;
```

Figure 18: ProcBin Struct `global.h`

- *SubBin* is an object that refers into either a refc binary or heap binary. It is created when a binary is pattern matched into parts or split by `split_binary/2`. A Sub Binary can not refer to another Sub Binary. In Figure 19 the struct contains a pointer to the original binary, the offset, the size and other metadata. The `is_writable` flag indicates if the binary can be

mutated for optimizations done by the run-time system.

```
typedef struct erl_sub_bin {
    Eterm thing_word;           /* Subtag SUB_BINARY_SUBTAG. */
    Uint size;                  /* Binary size in bytes. */
    Uint offs;                  /* Offset into original binary. */
    byte bitsize;
    byte bitoffs;
    byte is_writable;           /* The underlying binary is writable */
    Eterm orig;                 /* Original binary (REFC or HEAP binary). */
} ErlSubBin;
```

Figure 19: Sub Binary `erl_binary.h`

The run-time system performs optimizations on binary creation, and can freely convert between the types described above.

The optimization described below, also examplified in Erlang's documentation [12], is used when appending data to a binary in a sequential manner. Is is important to note that these optimizations are very dependent by the order of the performed procedures. This example demonstrates both an attempt that succeeds and an attempt that fails.

```
Bin0 = <<0>>,                                                    (0)
Bin1 = <<Bin0/binary,1,2,3>>, %% will allocate extra large binary (1)
Bin2 = <<Bin1/binary,4,5,6>>, %% mutates the extra large binary   (2)
Bin3 = <<Bin2/binary,7,8,9>>, %% mutates the extra large binary   (3)
Bin4 = <<Bin1/binary,17>>,    %% can not mutate                  (4)
```

Figure 20: Binary Optimizations

A heap binary is created and assigned to the variable `Bin0` on line 0. On line 1 an append operation is performed. The content of `Bin0` together with `«1,2,3»` is copied into `Bin1`, but the total size of `Bin1` is set to `max(256, 2 * size(Bin1))`. This may convert a heap binary to a refc, and this is exactly what happens to `Bin1`, since its size is 5 bytes. The ProcBin of `Bin1` will point to a 256 byte long binary, with its size field set to `4`. The extra long binary will just be mutated if binary data is appended to it, which saves the run-time system from creating ny copies. On line 2 `Bin1` and `«7,8,9»` are appended. `«7,8,9»` takes 3 bytes, and `Bin1` contains 252 unused bytes, thus `«7,8,9»` is copied into `Bin1` mutating the structure. This does not break immutability, since `Bin1` becomes a sub binary, where the offset and size will point out the correct region. On line 3 the same scenario as on line 2 occurs. On line 4, `Bin1` is used to construct another binary, but there is no way to utilize the allocated empty binary space as done on lines 2 and 3: sub binaries can only refer to continuous blocks, and the content of `Bin1` (together with `«17»`) needs to be copied to a new binary.

2.7 Garbage Collection

The garbage collector used by processes is *copying* and *generational*, as described by [13, 14, 15].

Copying means that the heap is first scanned for live data, and data that is alive, is copied to a new memory area in a packed form, to avoid fragmentation. Generational means that the data is grouped by how many garbage collections it has survived. This split makes it possible for the garbage collector to not traverse data in a smarter way, and analyze the data that has survived many time less often.

There are three different types of garbage collections, and we will show how they are trigged and performed.

We first assume a newly spawned process that has performed some work.

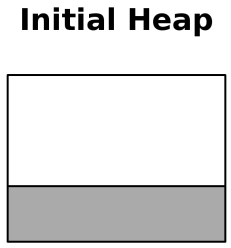


Figure 21: Initial Heap

Let us now assume that the process does more work and fills up its heap. When the heap becomes full the first time, a *first* garbage collection is performed. A new heap is allocated, called *Young Heap*, and the data that is still alive is stored at the bottom of the newly allocated heap refer to this data as *Highwater data* and the mark indicating where Highwater data begins, the *Highwater Mark*. After this the initial heap is discarded and its allocated area is returned to the run-time system.

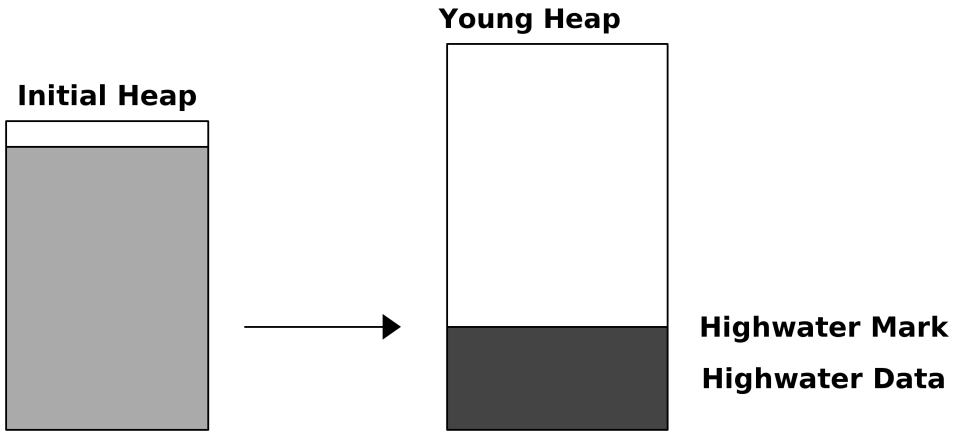


Figure 22: First Collection

Assume now that the process performs even more work and fills the newly allocated Young Heap, so that a second garbage collection is trigged. After the first garbage collection, only two types of garbage collections can be performed, *minor* and *major*. The second collection is a minor.

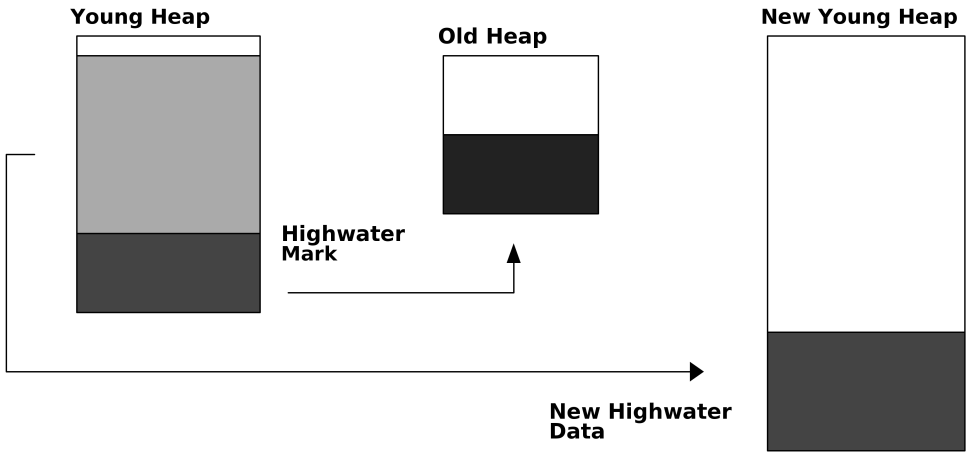


Figure 23: Minor Collection

The live data above in the Highwater mark is copied to the New Young Heap and becomes Highwater data there, because it has survived its first garbage collection. The Highwater data in the current young heap that is still alive is instead copied to a totally new heap area, called *Old Heap*, where data that has survived two garbage collections is stored. The garbage collection design is based on the assumption that the majority of data objects will die young, so the one that are long-lived is put to a different area, so that they don't have to be scanned every time a minor garbage collection is performed. During future garbage collections, only minor collections will be performed, until the Old Heap becomes full. This means that minor collections will allocate new young heaps, but keeping the old heap static. The data in the old heap is never scanned, the old heap block only accumulates data that has survived at least two garbage collections.

When the Old Heap becomes full, a major collection is performed. In a major collection all data is scanned, both data in the Old Heap and the New Heap. The data that is alive is stored as Highwater data in the New Young Heap. The Old Heap is discarded, and the process has only one heap again.

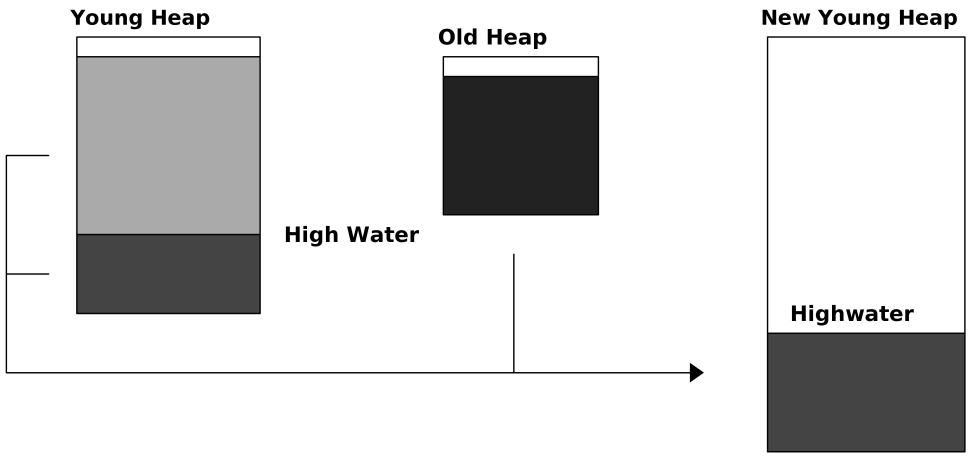


Figure 24: Major Collection

Next time the Young Heap becomes full, a minor collection is run, allocating a new Old Heap to store alive objects from Highwater data area.

This type of garbage collection is called two *generational semi-space stop-and-copy* garbage collection technique, while binaries on the binary heap are reference counted, and not managed by a collection in a process. This section will focus on the garbage collection performed locally in a process.

During a semi-space stop-and-copy garbage collection active objects are copied from a *from-space* to a *to-space*. This implies that all references stored in objects need to be rewritten to keep the objects structures intact. The from-space can then be reclaimed by the system. During the garbage collection procedure, the program is stopped, and can only continue its execution when all objects are reallocated on the to-space, in a sane condition.

A generational collector uses the *weak generational hypothesis* that most objects are short-lived and will not survive several garbage collections. The amount of objects that survive are assumed to be long-lived, and should be inspected less frequently. This hypothesis is implemented by using two heap areas, *old generation* and *young generation*.

3 Domain Description

3.1 Packet-Core and SGSN-MME

We give a simple and restricted overview of the mobile packet core network and the nodes relevant for our task. The mobile packet core network is the core of a mobile telecommunication system, acting as a center point that handles, provides and controls the ability of a mobile phone to use different services, such as web browsing or voice-over-IP-calls. One important node in it is the SGSN-MME. In addition to the SGSN-MME, this report will also describe the gateways, which the SGSN-MME assigns to phones, to get access to the internet.

3.1.1 Network overview

We begin with a simple overview over the Packet-Core Network. This simplified figure shows the SGSN-MME.

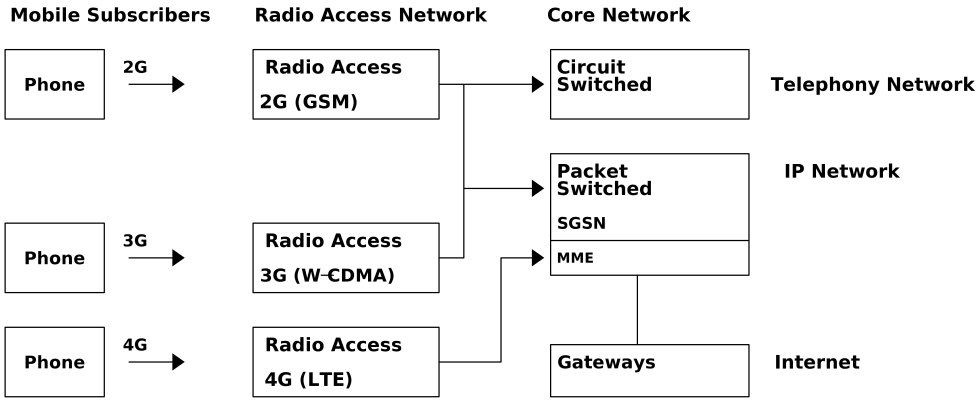


Figure 25: 2G, 3G and 4G Mobile Core Network

Let us begin by looking at how data flow through the core network from a 4G connected phone. As explained by Boquist [16], the phone in Figure 25, represented as a Mobile Subscriber, MS, in the core network, is first connected to the radio access network. The radio access network acts as a bridge to the Mobile Core Network. At the core network data from the phone is inspected, routed and forwarded. The last endpoints are the gateways, that forward the payload from the phone to the internet.

The core network consists of two main parts, a circuit switched network for telephony functions, such as calls and SMS. For each call, a virtual circuit is allocated during the whole duration of the call. The packet switched network is instead based around IP, and handles IP data sent from for example a browser or e-mail client running on the MS.

An SGSN can handle several radio access networks, enabling the core network to also support 2G and 4G. When handling 4G, the SGSN is extended with a MME node.

The gateways forward the phone to the internet and provide the ability for mobile operators to control the services a phone is able to use, apply policy enforcement, packet filtering and charging support.

The core network can have a high number of gateway nodes, depending on the terrain of the country where the core network is installed, the load the core network can handle and the ability to group subscribers to certain sets of gateways, to have a more fine-grained control over the load in the network.

The capacity of an SGSN-MME is measured in the number of phones it can serve at the same time. The recent Ericsson SGSN-MME, that is built of several Intel Xeon processors, including special

purpose hardware, can handle in the order of 10 millions of phones at the same time.

3.1.2 Architecture

The two main functionalities the SGSN-MME must provide is *control signaling* and *user traffic*. Control signaling is used to configure the phone and is not visible to the user (the human using the MS). The signaling may configure the phone internally in the network, but also signaling to the phone that it needs to reconfigure. Some examples of control signaling is when the phone is started or changes a base station in its radio access network, for example when driving a car and the geographical position is changed. The configuration of what services the phone may use, the assigning of gateways, is done by the control plane, taking into consideration the internal state of the network for load balancing. User traffic is instead data sent by applications on the phone. These packets have nothing to do with the network, they just need to be forwarded to the internet. The answer from hosts on the internet should then be routed back to the corresponding mobile phone.

The architecture of an SGSN-MME resembles these two functionalities. The SGSN-MME is divided into two planes, the control plane (CP) and a payload plane (PP), that performs user traffic routing.

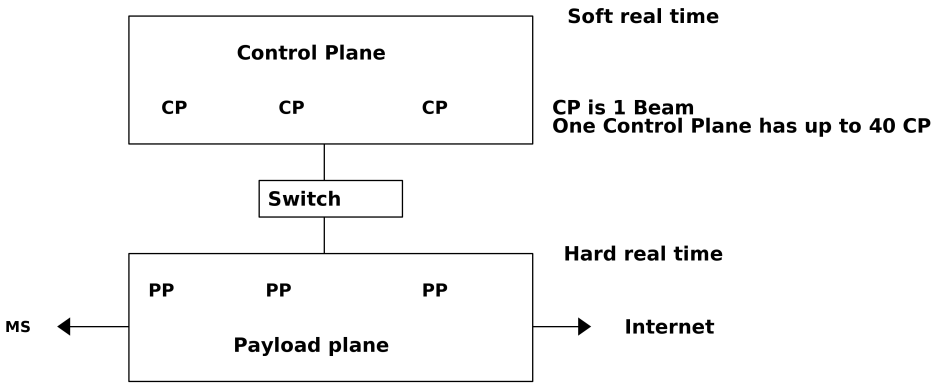


Figure 26: Architecture

The user plane has hard real-time requirements. User data needs to be routed through the node as fast as possible to maintain a good user experience. The control plane has a soft real-time requirement, due to a very high code complexity, where thousands of GPRS standards need to be implemented. The control plane keeps a state of each connected MS and when needed send signals to the MS through the Payload plane, which has the physical connectivity to the radio

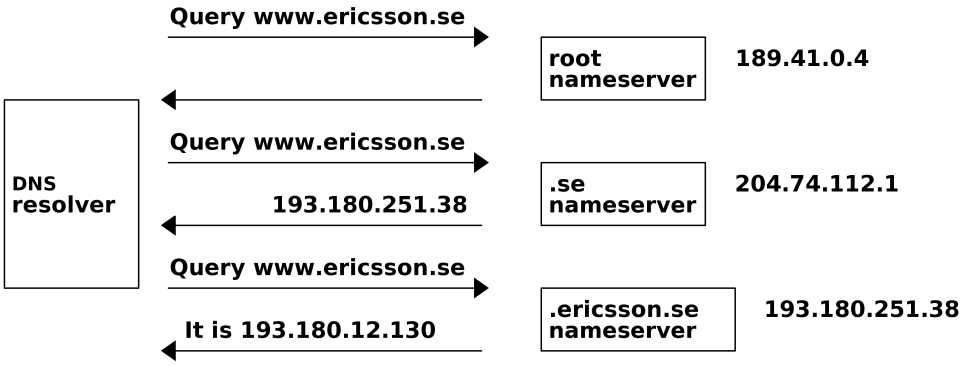
network.

3.2 Domain Name System

In this section we describe the Domain Name System, DNS, as explained by Kurose [17]. We first show how DNS is used in general and what functionality it provides on the internet. We then describe how DNS is used in the mobile communication network, in particular by the SGSN-MME. The section is completed with an example of a DNS flow between the SGSN-MME and the DNS-server, where we also inspect the use DNS-records used in the network.

DNS is, in it simplest form, a system for translating human-friendly hostnames into IP adresses. Generally a DNS-server either maps a hostname to an ip-adress, or refers to another DNS-server, who may possess the correct mapping.

Let us follow a simple example of how a DNS-resolver finds the IP of a hostname.



The DNS-resolver (usually a part of an operating system) used by applications like the web browser, needs to obtain a ip-adress of `www.ericsson.se`. The DNS resolver performs a request to the root nameserver, to get the IP of the `.se` nameserver. The `.se` nameserver points the resolver to the `.ericsson.se` nameserver, that has a mapping of the IP.

We say that the DNS system is hierarchical and decentralized. It is hierarchical because name servers are ordered in a tree-like structure, and an iterative search for the correct IP is a traversal of the tree structure, based on the order of labels in the hostname, read from right to left, separated by the dots. It is decentralized because the responsibility is spread over several DNS-servers.

There are many types of different DNS queries and many response types, called Records. The records used in the SGSN-MME context are NAPTR, SRV, A and AAA. Here is a brief explanation of them:

- A-record maps IPv4 address to a hostname.
- AAA-record maps IPv6 address to a hostname.
- NAPTR-record, contains weights, interfaces and protocols mapped to a hostname.

These records may come in so called record sets, where an answer to a query may contain several records. A record set with naptr records must be sorted accoring to the weights and continue the iterative lookup in a ceratin order.

One functionality that DNS handles is to store the mapping between phone parameters to two types of gateways in the core network, Packet Gateway (PGW) and Serving Gateway (SGW). The gateways are end nodes used used to connect the phone to the internet. The used parameters are APN from Figure 4, that maps to a PGW, and Tracking Area Identifier, TAI, that is geographical information of the phone, that maps to a SGW.

The procedure the SGSN-MME has to do is to traverse the DNS-servers with queries based on APN and TAI and lookup all the valid PGWs and SGWs for that particular APN and TAI. The end result is a candidate list of gateway nodenames. The procedure of retrieving the list is called Straightforward-NAPTR.

The types of records indicates that a complex network of DNS-servers can be configured, when an initial query results in several new queries. The access point name, such as "services.telenor.se", the services provided by the subscription and the geographical location are mapped to DNS-records with gateway hostnames.

The reason for chosing DNS as protocol is that DNS-servers are generally high capacity, have high reliable network sense and are load shared. The fact is that internet has been using DNS as key-value store for hostnames since at least 1986. The protocol is light weight and supported on almost every OS.

3.3 Gateway Selection

In this section we describe the current implementation of the gateway selection procedure. We begin by showing an overview over the involved parts, then inspect the DNS cache and finally analyze how an Erlang process is processing the string data it fetches from the cache.

3.3.1 Overview

Figure 27 shows that the gateway selection procedure is performed by each Erlang process that is servicing a phone.

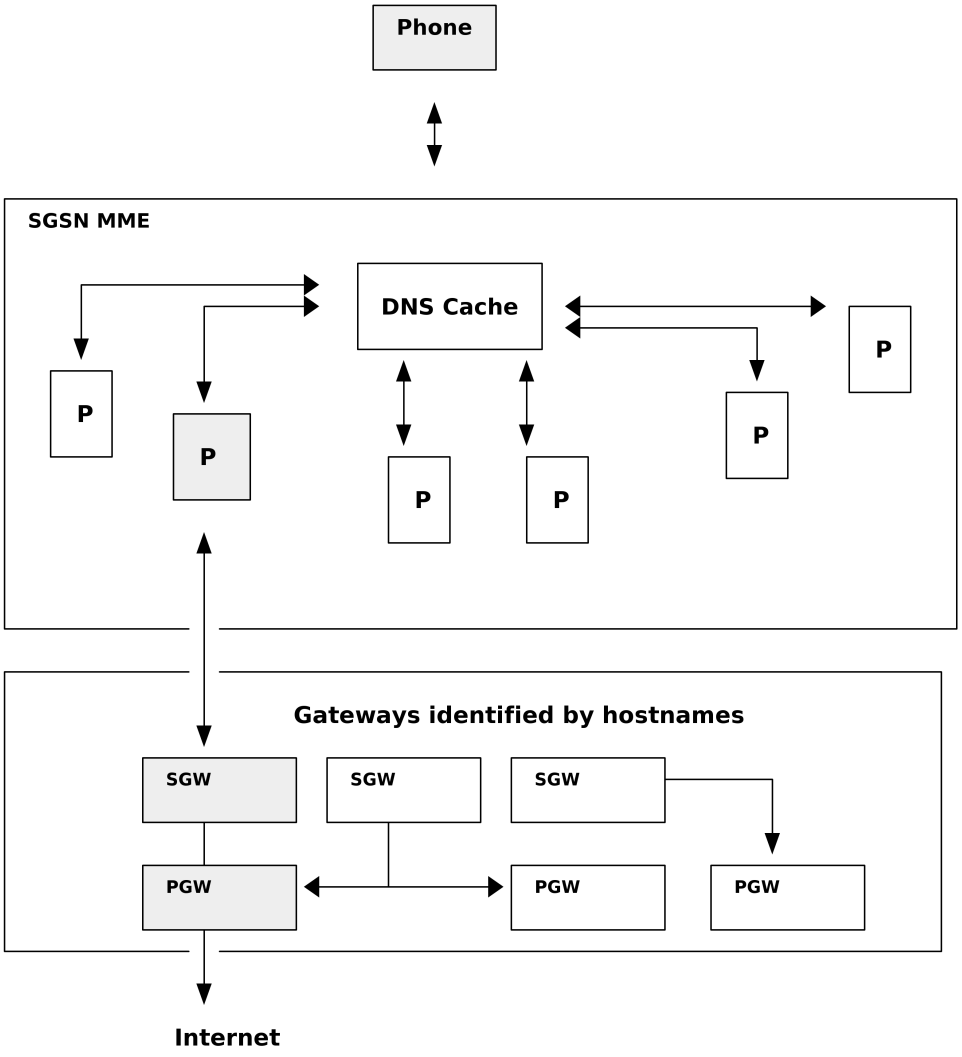


Figure 27: Gateway selection overview

The grey nodes illustrates how a phone is handled by one Erlang process, that fetches the list of all available gateways from the DNS cache, and then chooses a gateway pair. A phone needs two gateways, a Serving Gateway (SGW), and a Packet Gateway (PGW).

As described in the 3GPP specification [18], there are two alternatives for selecting a gateway pair. It is either based on the order in which the list of available gateways is built or by topological closeness. The first first alternative is controlled by the DNS-records, that beside containing the hostnames, also contain weights for load-balancing, and this weight will affect the order in what ordet the hostnames will be returned from the DNS cache. The second option, that is based on topological closeness, forces the Erlang process to inspect all hostnames to find a pair that is as close as possible. A gateway that both has a built-in SGW and PGW will be seen as *colocated* and have the highest priority, since the SGW and PGW are as close as possible.

3.3.2 DNS-Cache

The majority of the DNS-requests are mainly looked up in the DNS cache that is implemented as an ETS-table.

Assume we perform a DNS-request for the address `telenor.apn.epc.mnc456.mcc123.3gppnetwork.org.`, as seen in Figure 26. This address is configured by the mobile operator to find the correct gateways for that particular network. The overview figure shows the simple flow of looking up a hostname, and if it is not present in the cache, doing a real resolve and then updating the cache.

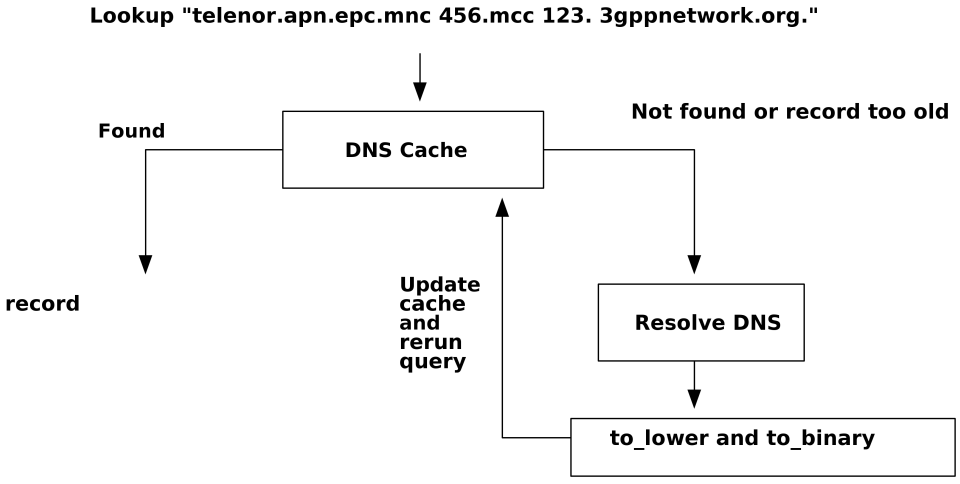


Figure 28: DNS-Cache

If the record for the domain name is not found, Erlang's resolver is run and a record set is obtained. An example of two records are seen in the code below. The two records holds hostnames for a potential gateway pair, "topon.eth0.sgw.a.b" and "topon.eth2.pgw.d.c.b".

```
([{dns_rr,"telenor.apn.epc.mnc456.mcc123.3gppnetwork.org",naptr,in,0,
         0,
         {1,1,[],"x-3gpp-sgw:x-s5-gtp:x-s5-pmip",[],"topon.eth0.sgw.b.a"},
         undefined,[],false},
 {dns_rr,"telenor.apn.epc.mnc456.mcc123.3gppnetwork.org",naptr,in,0,
         0,
         {1,1,[],"x-3gpp-pgw:x-s5-gtp:x-s5-pmip",[],"topon.eth2.pgw.d.c.b"},
         undefined,[],false}],
 naptr)
```

Figure 29: Received naptr-record in Erlang

The records in this set will be put in the DNS cache. First `to_lower` is applied to the payload and the the whole record is converted to a binary to compress the string data in the payload. The binary compresses the string by packing each character (represented by an integer) into a sequence of bytes. Recall Figure 2 that showed the amount of unused space in a string, illustrating the prefix "se". If we apply `term_to_binary` on "se", both characters are stored in a single word, in the payload chunk of a heap binary.

00000000	00000000	00000000	00000000	00000000	00000000	01110011	01100101

Figure 30: "se" as binary

This is done for a more compact storage of the string data in the DNS cache, that is an ETS-table.

3.3.3 Pair priority

The key operation in the gateway selection procedure is to find the best pair from the list of available gateways. The best pair is the one with the highest topological closeness. Or more general, the longest common suffix labelwise.

First the process does iterative DNS-lookups that result in a set of records, each fetched from the DNS cache. Each record needs to be decompressed with `binary_to_term` so that the hostname can be recreated as a string. The resulting candidate list of hostnames is shown in Figure 31. It

consists of one lists with all valid SGWs and one list with all availed PGWs.

```
Sgws = ['topon.eth0.sgw.a.b.c', 'topon.eth2.sgw.b', 'topon.eth0.sgw.a.c', ... ]
Pgws = ['topon.eth0.pgw.b.c', 'topon.eth1.pgw.c.b', 'topon.eth0.pgw.a.c', ... ]
```

Figure 31: List of SGW and PGW domain names

Then an SGW is paired with a PGW. The case when an SGW has the `topon` prefix, it has to be matched with a PGW with a `topon` prefix by counting the topological closeness, that is, the number of equal labels counted on the reversed nodename. The format is seen in Figure 32. We call the substring between two dots a label.

```
<topon|topof>|interface|nodename>

"topon.eth0.telenor.se.3gpp.org"
```

Figure 32: 3GPP Hostname Format

In Figure 33 we see the involved operations for counting the topological closeness of an SGW and PGW pair. First the string is split by dot, then if the prefix is `topon`, the prefix and interface is removed. The rest, that is the hostname, is then reversed, and number of equal labels is counted.

```
"topon.eth0.sgw.a.b.c",
"topon.eth0.pgw.b.b.c"


% string:tokens
["topon", "eth0", "sgw", "a", "b", "c"]
["topon", "eth0", "pgw", "b", "b", "c"]

% match out node name by removing "topon" and "eth0"
["sgw", "a", "b", "c"]
["pgw", "b", "b", "c"]

% lists:reverse()
["c", "b", "a", "sgw"]
["c", "b", "b", "pgw"]

% priority is 2 since the two first labels are equal
```

Figure 33: String to list of labels

A pair has the highest priority if the nodenames are identical. That kind of pairs are called colocated. Practically it means that the SGW and PGW are on the same physical node.

3.3.4 Implementation

The current implementation of finding the pair with the highest priority is by generating all possible pairs and calculate their priority, as in Figure 34.

```
Combinations =
    [{get_pair_priority(Sgw, Pgw), Sgw, Pgw} || Sgw <- Sgws, Pgw <- Pgws],
```

Figure 34: Generate list of pairs

All possible pairs are generated by a list-comprehension. The function `get_pair_priority` is defined in Figure 35.

```
get_pair_priority([_Topon, _Interface1 | SGW], [_Topon, _Interface2 | PGW]) ->
    count_equal_labels(lists:reverse(SGW), lists:reverse(PGW)).

    count_equal_labels([X|XS],[X|YS],I) ->
    count_equal_labels(XS,YS,I+1);
count_equal_labels(_,_,I)->
    I.
```

Figure 35: Helper functions

The `Combinations` are then sorted by the priority, and then the priority is filtered out. Both operations are implemented by using list-comprehensions. Then the first pair is matched out, the `Head` variable in this case.

```
Sorted = lists:sort(fun ({M,_,_},{N,_,_}) ->, M >= N end, Combinations),
CandidateList = [{Sgw,Pgw}|| {_,Sgw,Pgw} <- Sorted],
[Head | _Rest] = Result
```

If the best pair for some reason is not reachable in the network, the rest of the candidate list has to be searched again, to find the next best pair. If an SGW is down, then the pair need to have a new SGW, and if the PGW is down, a new PGW.

The bad SGW is filtered out. The method to do it is again by using list-comprehension.

```
[{Sgw, Pgw} || {Sgw, Pgw} <- CandidateList,

Sgw#gw_selection_candidate.hostName /= BadHost];
```

Figure 36: Maybe retry

3.3.5 Problems

Generating every pair is an $O(n^2)$ operation (where n is the number of hostnames fetched from the DNS cache) both in memory and time. Given a large mobile core network with hundreds of SGWs and PGWs, this extensive generation of pairs can be expensive. We will inspect the behavior of memory in more detail in Section 4 and by simple improvements change the memory complexity of the algorithm. With this change, we then reason about actual string representation of the hostnames.

4 Contributions

The contribution of this report can be divided into three parts:

- Method for memory measurement

We examine and explore how the process heap behaves during certain string operations to be able to spot patterns with negative influence on memory consumption and CPU performance.

- Gateway selection algorithm

We analyze the gateway selection algorithm with our method for measuring memory to detect flaws and improve the algorithm. The improved version will then serve as a reference implementation against which other representations are measured.

- Different string representations

We present alternative string representations and analyze their memory and time performance.

4.1 Methods for Memory Measurement

In this section we will explain how we measure the heap of a process. We will go through what a modified version of the built-in Erlang function `process_info` measures and how we use it to follow and explore the heap behavior of a process.

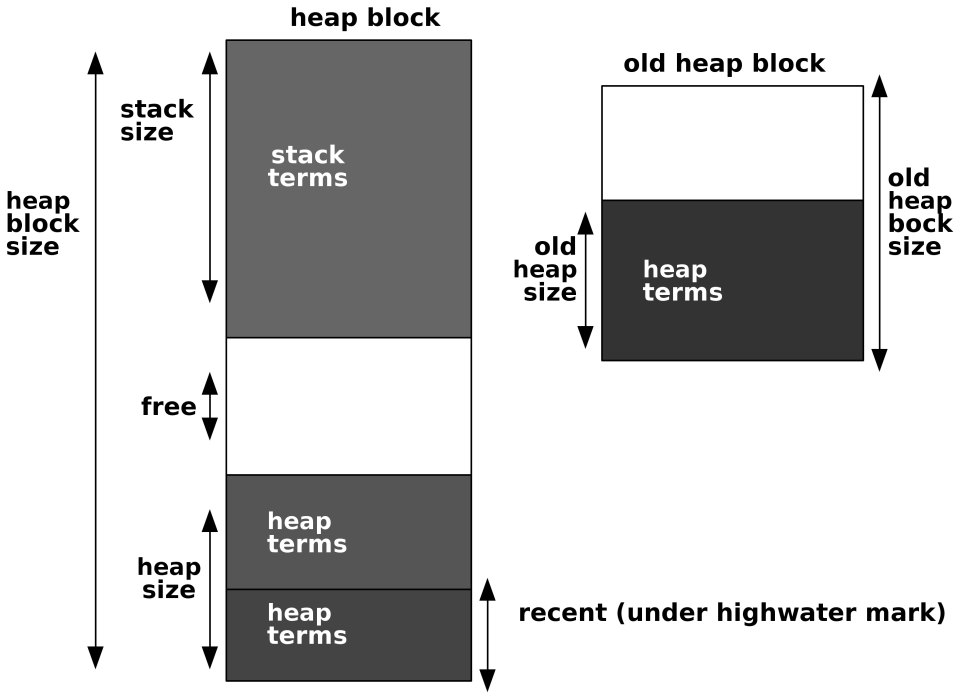


Figure 37: `process_info` performed on the shell process.

The Erlang built-in function `process_info` inspects the process structure in real-time, and performs simple calculations with the heap adresses stored in the process structure. The functions outputs the values seen in Figure 37. We use a modified version of `process_info` that adds a counter for major garbage collections, since only a counter for minor garbage collections is present. Our out put can be seen in Appendix A, and the process structure can be seen in Appendix B. The major counter is not present in the original version, since it would most probably overflow in a large application, making it very hard to interpret. But the examples in this section are small and isolated, and the there is no risk to read inconsistent data.

We use `process_info` in combination with a simple blocking function `block`.

```
block() ->
    receive
        _ ->
            continuing
    end.
```

Figure 38: `block()`

The function `block` can be added to source code to pause the execution of a process. While the execution is paused, another process that possess the process identifier of the blocked process can inspect its process structure with `process_info`. When that is done, a message is sent to the blocking process so that it can continue its execution.

Running `block` will affect the stack of the process, since the continuation pointer used to return to the function in which `block` was called, need to be stored. This will add an extra word to the stack in our measurements. After the return, the pointer is popped from the stack.

```
simple() ->
    A = f(),
    B = f(),
    C = f(),
    D = f(),
    block(), # measure here
    {A,B,C,D}.

arguments(A, B, C, D) ->
    Res = {A,B,C,D},
    block(), # measure here
    Res.

f() ->
    1.

Pid_simple = spawn(simple, [])
Pid_arguments = spawn(arguments, [1,2,3,4])
```

Figure 39: `simple` and `arguments`

In Figure 39 two functions are examined, `simple` and `arguments`. The process structure is inspected at the points where `block()` is run. The important values is seen in Table 1.

Table 1: `simple` and `arguments` info

function	stack_size	heap_size
simple/0	5	0
arguments/4	2	13

`simple` calls `f` and assigns its return value, `1`, four times, then `block/0` is called. Since `f/0` returns a small integer that is only stored on the stack, the stack has four small integers and the continuation pointer used by `block/0`, resulting in 5 words on the stack. In the second function `arguments/4`, the heap allocation is much bigger than the one of `simple/0`. Since processes are self-contained in

Erlang, the arguments are copied to the heap in the form of a list. When `arguments` is spawned, `A, B, C` and `D` are placed on the heap in a list, taking 8 words. The creation of the tuple then takes 5 words on the heap, and 1 word on the stack. Together with the continuation pointer, 1 extra word is pushed to the stack. The result is 13 words on the heap, and 2 words on the stack.

```
literal1(A) ->
    B = "hello hello hello hello" ++ A,
    block(),
    B.

literal2(A) ->
    B = A ++ " hello hello hello hello",
    block(),
    B.
```

Figure 40: Literates

Next example is about literals, shown in Figure 40. `literate1/1` needs to make a copy during concatenation, while `literate2/1` is just referencing to the constant pool. Both functions are spawned with the argument "hello". Even if the the string returned by both functions is the same `hello hello hello hello hello`, the `heap_size` differ a lot, as seen in Table 2.

Table 2: `Literals info`

function	`stack_size`	`heap_size`
`literal1/1`	2	58
`literal2/1`	2	22

`literal1/1` has to create a copu of the literal "hello hello hello hello" to create a new list who has the `A` list at the end. This is done to maintain referential transparency if the literal is used again. `literal2/1` only copies the string in the function argument, to create a new string where the last cons-cell in the argument string points to the constant pool.

The final example demonstrates garbage collection. The code is seen in Figure 41.

```
trigger_gc() ->
    blockt(), % #0
    List1 = lists:seq(1, 114),
    block(), % #1
    List2 = lists:seq(1, 70), % 160 words
    block(), % #2
    List3 = lists:seq(1,5),
    block(), % #3
    Struct = {List1, List2, List3},
    block(), % #4
    Struct.
```

Figure 41: Trigger garbage collection

By this example we demonstrate how `List1`, `List2` and `List3` float through the different heap areas during garbage collection. First `List1` is allocated, as seen in Figure 42 at `#1`. With its 228 words, and some overhead stack from this measurement, it will fill upp the initial 233 words large heap block. The next list, `List2`, will force an initial garbage collection, since the heap is full after the allocation of `List1`, moving `List1` to the recent area, under the highwater mark, seen in Figure 43.

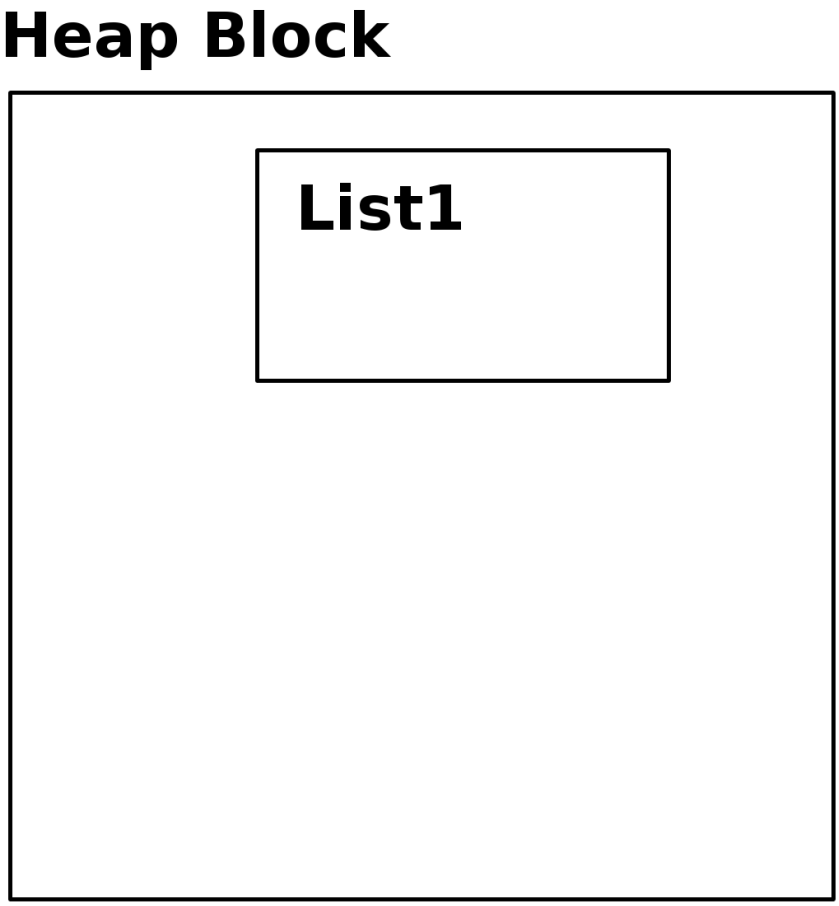


Figure 42: Heap at `measure:report #1`

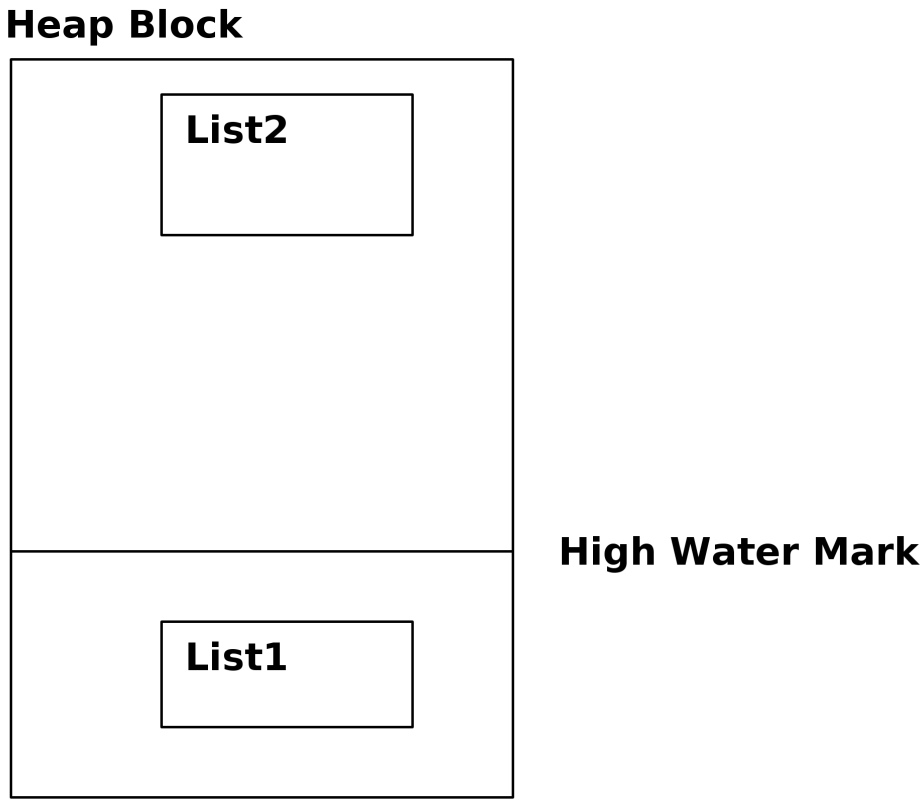


Figure 43: Heap at `measure:report #2`

The third list, `list3`, will force a first minor collection, moving the data in the recent area to a newly allocated old heap. Each list is now in a separate heap area, shown in Figure 44

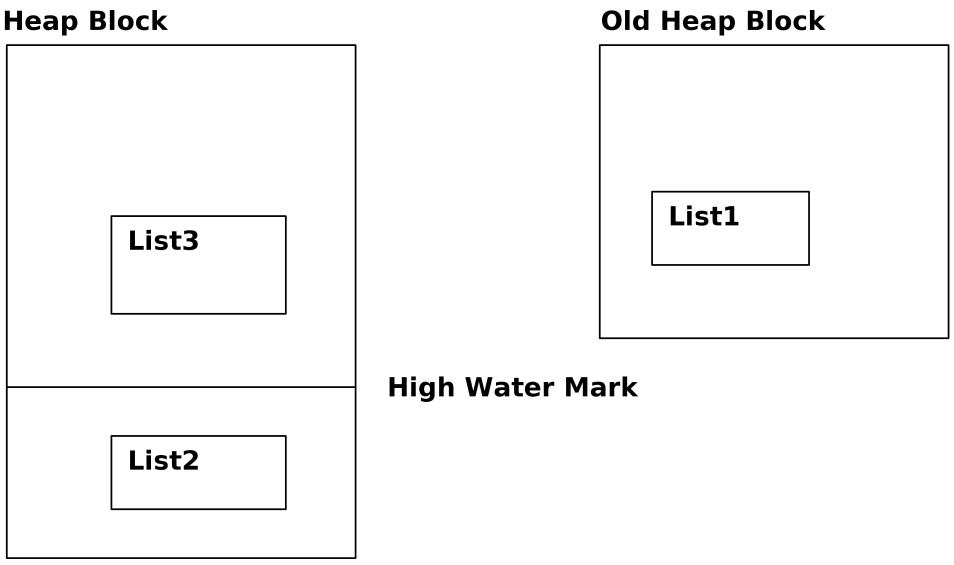


Figure 44: Heap at `measure:report #3`

By putting the three lists in a `Struct`, we see how we now have a structure with references to all the areas, shown in Figure 45.

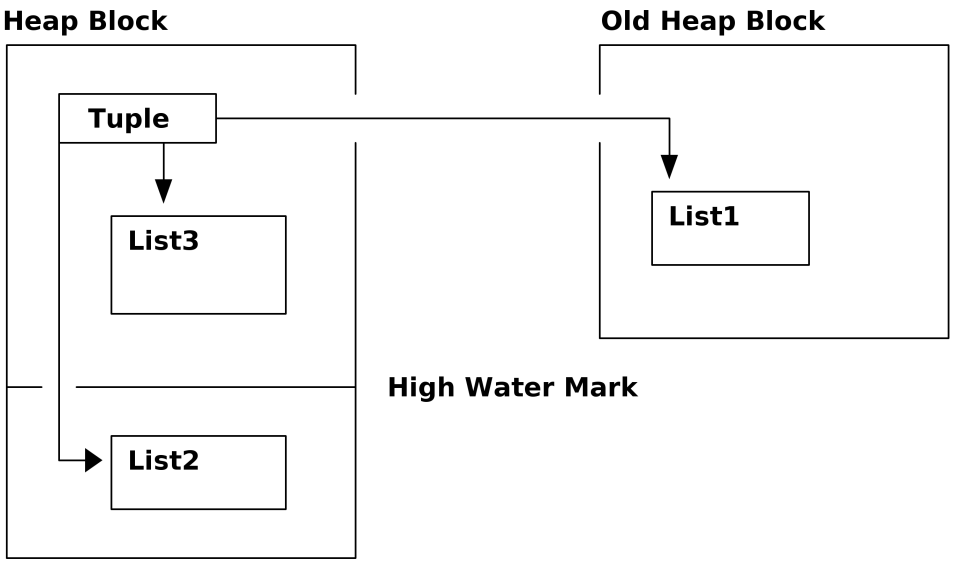


Figure 45: Heap at `measure:report #4`

We can also follow the heap movement in the graph in Figure 46, where we see the different heap sizes. The values in the legend are illustrated in Figure 37.

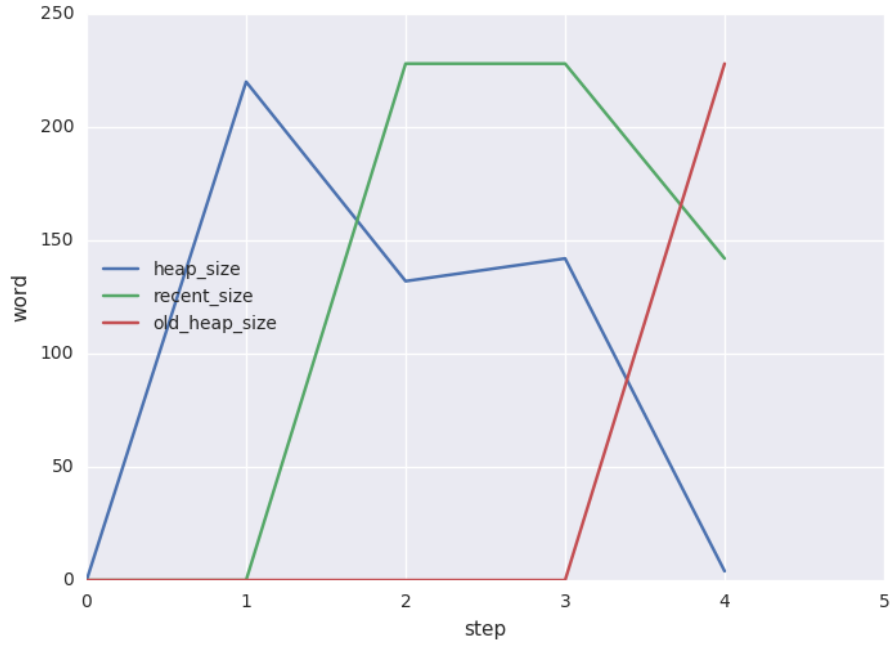


Figure 46: Heap behaviour during `gc_trigger`

4.2 Gateway Selection Algorithm

We explore the heap behavior in the $O(n^2)$ list-comprehension described in Section 3.3 and understand memory-wise why a high complexity is bad. As input we use real DNS-data from a very large network containing 150 SGWs and 150 PGWs. A short refresher of the first steps in the gateway selection algorithm is seen in Figure 47.

```
Combinations =
    [{get_pair_priority(Sgw, Pgw), Sgw, Pgw} || Sgw <- Sgws, Pgw <- Pgws],
```

Figure 47: Generate list of pairs

We will explore the code in Figure 34 in the following steps:

- We first show the relevant values from the heap illustration in Figure 37 during execution of the list-comprehension.
- We sport a stack-growth and rewrite the `get_pair_priority` function.

- Since we only need the best value, there is no need to generate all pairs. Instead we just iterate over all pairs and keep the best one in memory.

The algorithm modified by these steps will later serve as a reference implementation when we explore different string representations.

4.2.1 Heap behavior in list-comprehension

We begin by inspecting the heap information while running the code from Figure 47.

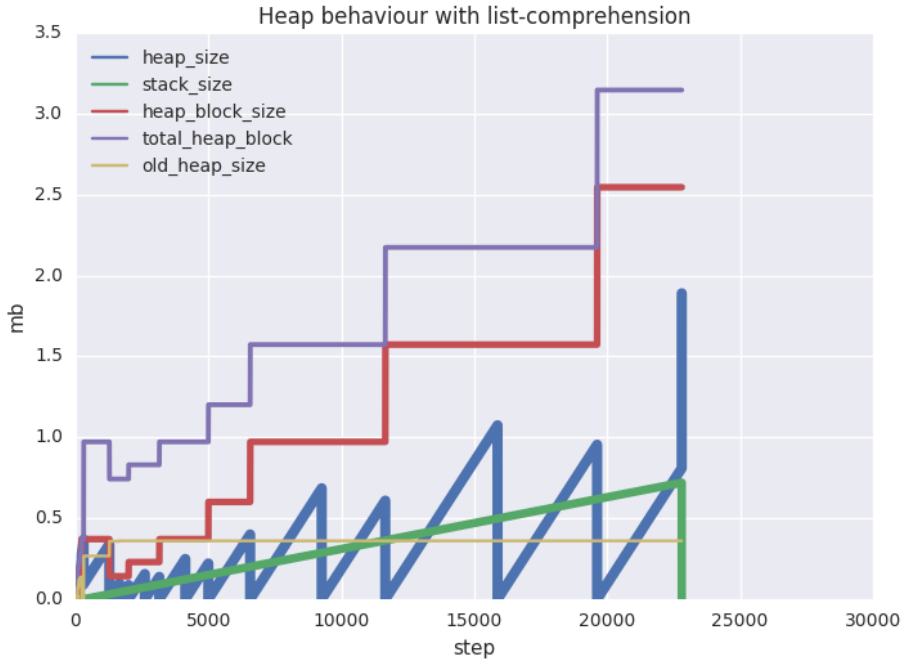


Figure 48: Heap behaviour with list-comprehension

In Figure 48 we see some interesting behavior. First, due to the fact that list-comprehensions are not tail-recursive, but body-recursive, and grow on the stack during the creation of the `Combinations` list, we see a linear growth in stack size. The heap size have a chain saw pattern where every fall indicates a minor garbage collection, since the old heap size is constant, we assume that there are no major garbage collections.

We confirm our assumption about major garbage collection by Figure 49, where only the minor gc

counter is incremented with each fall of the heap size line in Figure 48.

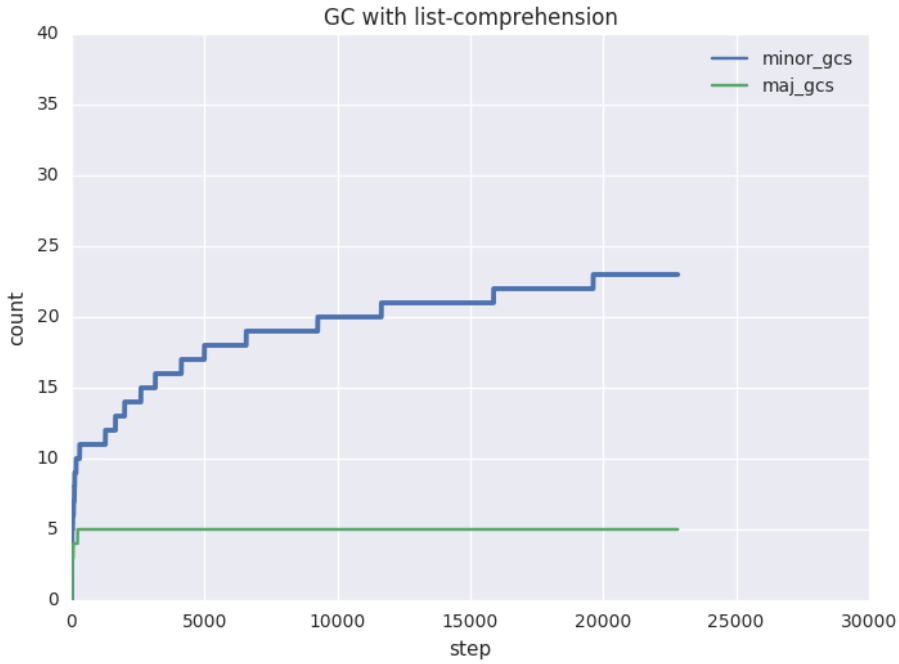


Figure 49: GC with list-comprehension

The total heap block size allocated needs to fit the growing stack plus the heap size. When the list-comprehension is done, the list is copied to the heap, seen in the last heap size spike at approximately 1.9 MiB in Figure 48, while the stack is popped to zero. The heap block size is then 2.6 MiB.

An extensive activity of the heap together with the stack forces the process to allocate a rather large heap block, and the process will be stuck with this large heap block if the process only do light work from now on.

The size of `Combinations` is also independent of the string representation. 150 SGWs and 150 PGWs are 300 list with strings, and when building a list, every tuple in the list will have a reference to one of the 150 SGWs and one reference to the 150 PGWs. The elements in `Combinations` are then tuples containing three simple terms, `{Priority, Sgw, Pgw}`, that is a priority integer, a pointer to a SGW and a pointer to a PGW.

If the resulting list really is a list of pointers to shared data, what is then creating so much short-lived garbage and the saw tooth pattern, forcing so many minor garbage collections? The answer is in the `get_pair_priority` function seen in Figure 35, that must reverse each hostname to count

equal labels. Reverse is a copying operation and the hostname is only needed in reversed form a very short while.

By avoiding `list:reverse` in `get_pair_priority` we may change the pattern of the heap size, and the garbage collector will only need to allocate a heap block that may fit the stack, built by the body recursive list-comprehension.

We do the same heap measurements but change the implementation `get_pair_priority`. We now do not use `lists:reverse`, but just iterate over the original list of labels keeping a current count of equal labels, resetting the counter if labels do not match. The last sequence of matching labels is the correct priority. The code is presented in Figure 50.

```
count_equal_labels(A, A) ->
    256;
count_equal_labels(A, B) ->
    count_equal_labels_reversed(A, B, 0).

count_equal_labels_reversed([_X|XS],[_X|YS],I) ->
    count_equal_labels_reversed(XS,YS,I+1);
count_equal_labels([_X|XS],[_Y|YS], _) ->
    count_equal_labels_reversed(XS, YS, 0);
count_equal_labels_reversed(_,_,I)->
     I.
```

Figure 50: `count_equal_labels` with lighter reverse

The heap behavior is presented in Figure 51.

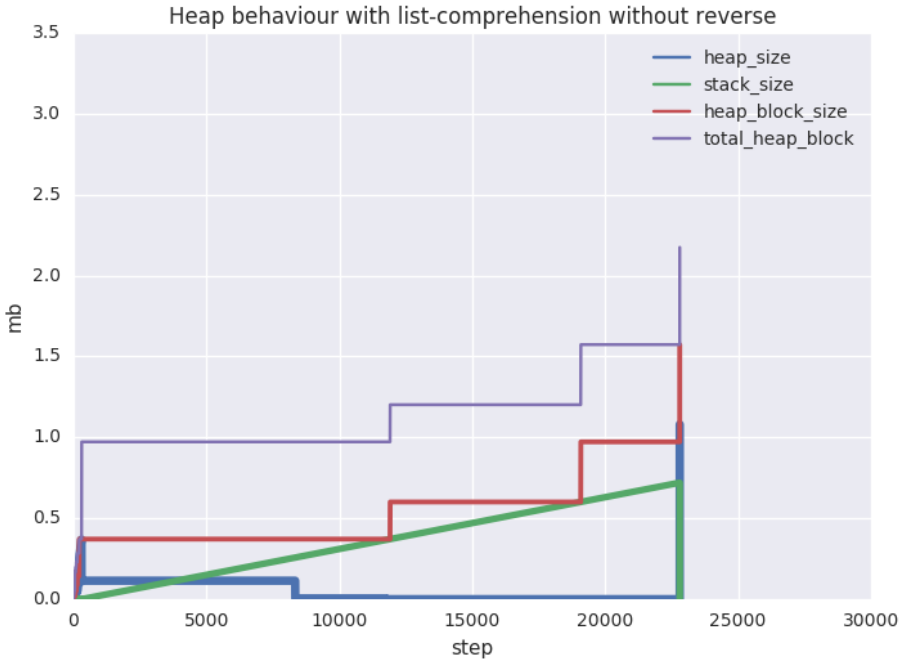


Figure 51: Heap behaviour with list-comprehension without reverse

With no garbage generated from `lists:reverse` we see that the heap block size is now proportional to the stack sizes generated from from the body-recursive list-comprehension. The heap block size is decreased by 1 MiB, from 2.6 MiB with `lists:reverse` to 1.5 MiB.

We confirm the lower garbage collection activity by Figure 52.

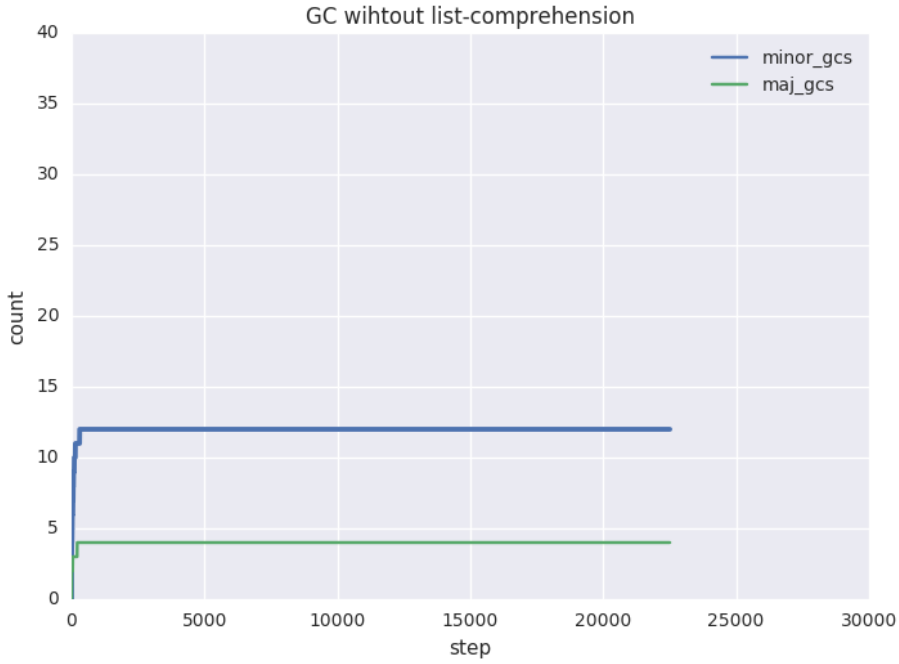


Figure 52: GC without list-comprehension

We now remove the quadratic memory consumption and then tackle the string representation. The fix is rather simple, we iterate over all pairs, but only keep the best in memory. During the iteration, the extra memory needed (beside the set of hostnames) is the best pair so far. By doing so we go from $O(n^2)$ of memory complexity to $O(n)$. The code is seen in Figure 53.

```
gw_best_pair([Sgw|SGWs], [Pgw|PGWs]) ->
    Res = lists:foldl(fun(S, Acc) ->
                              A = lists:foldl(fun(P, InnerAcc) ->
                                                      {P0, _, _} = InnerAcc,
                                                      P1 = get_priority(S, P),
                                                      if P1 > P0 -> {P1, S, P};
                                                         true    -> InnerAcc
                                                      end
                                              end, Acc, PGWs),
                              A
                      end, {get_priority(Sgw,Pgw), Sgw, Pgw}, SGWs),
    Res.
```

Figure 53: Without list-comprehension

And the resulting heap behavior is showed in Figure 54

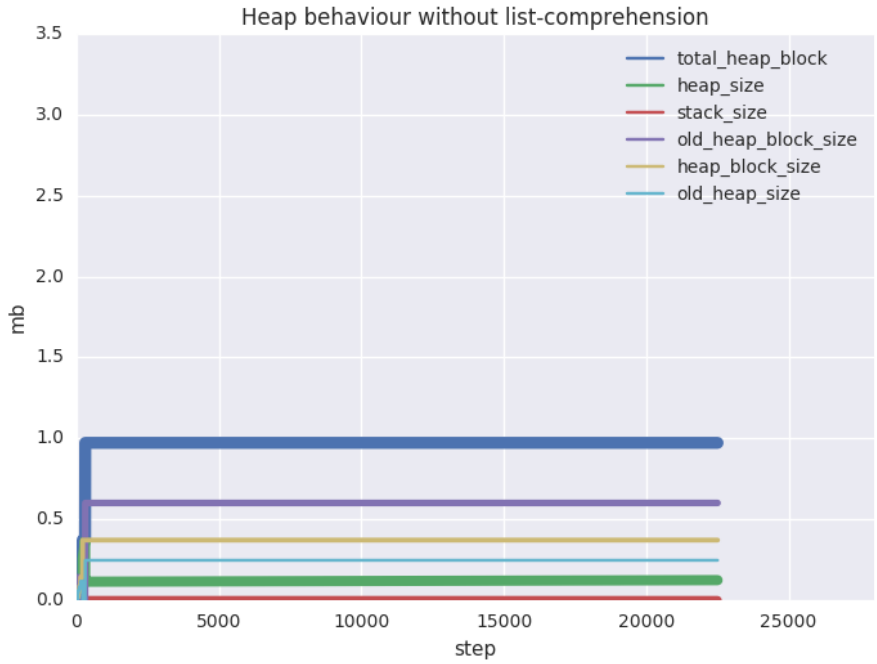


Figure 54: Heap behaviour with Binaries on normal cache

The heap values are now constant, which gives us a more stable and predictable garbage collection activity. Constant values are also easier to reason about and put into perspective, when analyzing performance analytically.

Even if the total heap block size is 1 MiB, the set with 300 hostname does not fill up the whole heap. We measure the size of the string data and see with the help of the bar chart in Figure 55, that it takes 0.36 MiB.

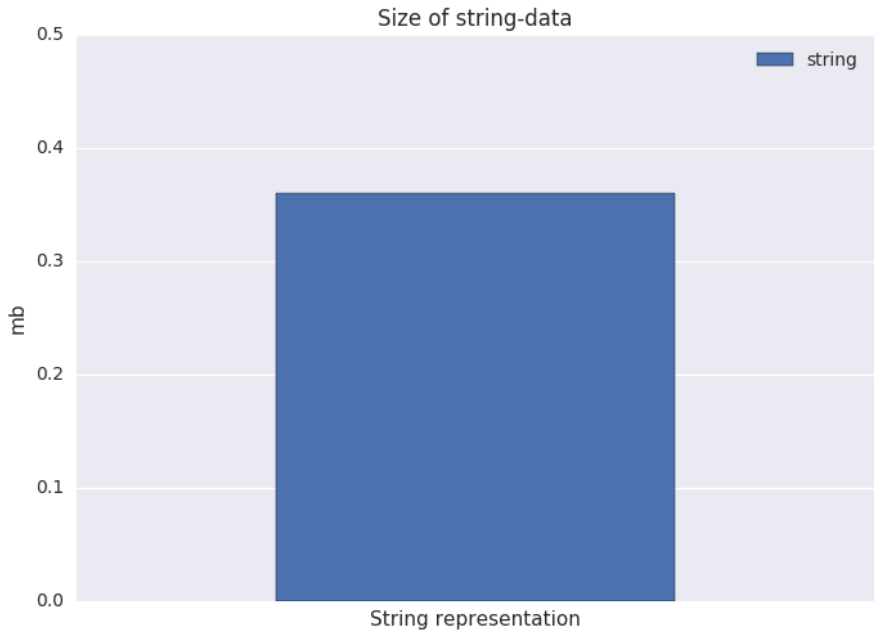


Figure 55: Size of all hostnames as list of string

We can now explore alternative string representations and use the size of the hostname set, that is 0.36 MiB, as a reference representation. We also use the total heap allocation from Figure 54 as a reference, when we analyze the heap activity of the alternative representations.

4.3 Alternative String Representations

In this chapter we try out different string representations to lower the memory consumption and the time to process the structures holding the strings.

We will present six different methods, with focus on the following types:

- *Heap binary list* We pack each character in a string as a sequence of bytes, creating small binaries that are stored in the process heap.

- *Big Integer list* We cast a sequence of bytes to a big integer, a type that has less overhead than Heap Binaries.

- *Atom list* Each label (the hostname parts separated by dot) is represented by an atom, which means that the list of labels only need to store an atom reference. This introduces sharing, since the same reference is re-used by all processes.

- *Atom tuple* Same as atom list but stored in tuples instead of lists, to remove the links beteween labels.

- *Atom trie* We store all labels in two tries [19], and find the best pair via an intersection operation. This can be efficient if the hostnames have shared suffixes.

- *Big binary* Each hostname is stored as a big binary on the global binary heap. This also introduces sharing, but for the whole hostname and not just individual labels.

Before we go through the representations in detail, we first describe the input data.

4.3.1 Input data

As input data, we will use a real life DNS cache taken from a mobile operator. A typical host name in the cache is around 10 labels long and the total length is around 75 characters. The hostnames share approximately 7 suffix labels. The format of such a hostname is shown in Figure 56, where each label is anonymized. The cache contains 150 SGW hostnames and 150 PGW hostnames, that is, 300 hostnames in total.

```
Normal = "topon.interface.aaaaa.bbbbbbbbbbbbbbbbbb
              .cccc.ddd.eee
              .ffffff.gggggg.hhhhhhhhhhh
              .iii".
```

Figure 56: Hostname `normal`

To see how the different representations scale, we also use two variations of the DNS cache. One cache with hostnames with twice as long labels, shown in Figure 57, and one cache with hostnames that have twice as many labels (of the original size), shown in Figure 58.

```
Longer_labels = "topon.interface.bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
                  .ccccccc.dddddd.eeeeee
                  .fffffffffff.ggggggggggg.hhhhhhhhhhhhhhhhhhhhh
                  .iiiiii",
```

Figure 57: Hostname `longer_labels`

```
Longer_list = "topon.interface.
                .bbbbbbbbbbbbbbbbbb.bbbbbbbbbbbbbbbbbb
                .cccc.cccc
                .ddd.ddd
                .eee.eee
                .ffffff.ffffff
                .gggggg.gggggg
                .hhhhhhhhhhh.hhhhhhhhhhh
                .iii.iii".
```

Figure 58: Hostname `longer_list`

We refer to the three hostname variations as type `normal`, `longer_labels` and `longer_lists`, and we refer to the caches containing the hostnames as normal cache, longer-labels cache and longer-list cache.

4.3.2 Reference implementation

The reference representation of a hostname - a list of strings - is examplified in Figure 59.

```
A = ["topon","interface","aaaaa","bbbbbbbbbbbbbbbbbbb"
                        ,"cccc", "ddd", "eee"
                        ,"ffffff", "gggggg", "hhhhhhhhhhh"
                        "iii"],
```

Figure 59: String encoder

We now explore alternative string representations with the above representation as reference representation, with reference size shown in Figure 55 and reference heap values shown in Figure 54.

4.3.3 Alternative Representations

Heap Binary List Our first approach is to pack the host names by representing the label as a binary. Since the length is often shorter than 64 bytes, the binary will be allocated on the process heap. The code converting the hostname to the heap binary format is seen in Figure 60, where the function `encode` is applied to a hostname from the `normal-cache`.

```
encode(Hostname) ->
    Labels = string:tokens(A, "."),
    BinaryLabels = lists:map(fun(L) -> term_to_binary(L) end, Labels),
    BinaryLabels.
```

```
A = "topon.interface.aaaaa.bbbbbbbbbbbbbbbbbb.
                          .cccc.ddd.eee
                          .ffffff.gggggg.hhhhhhhhhhh
                          .iii",

encode(A) -> [<<131,107,0,5,116,111,112,111,110>>,
              <<131,107,0,9,105,110,116,101,114,102,97,99,101>>,
              <<131,107,0,5,97,97,97,97,97>>,
              ...]
```

Figure 60: Heap binary encoder

In Figure 61 we see how much memory the hostname lists take with the heap binary representation and different caches. This value is the amount of heap space a process needs at least to hold the corresponding structures.

Note that the heap binary representation scale better with longer labels than longer lists. Adding more characters to labels does not affect the size, because there is no need to allocate new binaries to fit the added data. The characters are just added to the unused space in the last word of the binary sequence.

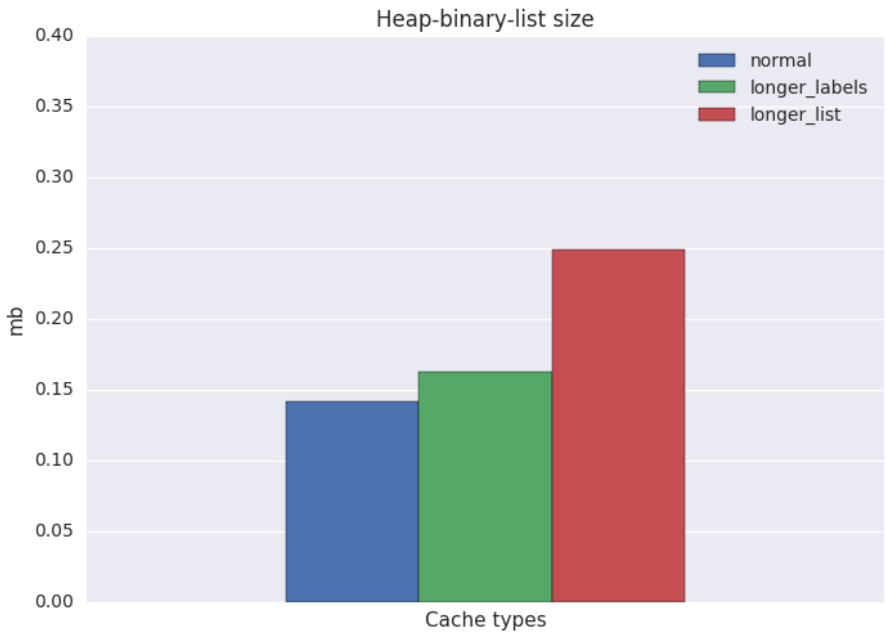


Figure 61: Heap-binary-list size

In Figure 62 we see the heap behavior during the gateway selection procedure with the normal cache.

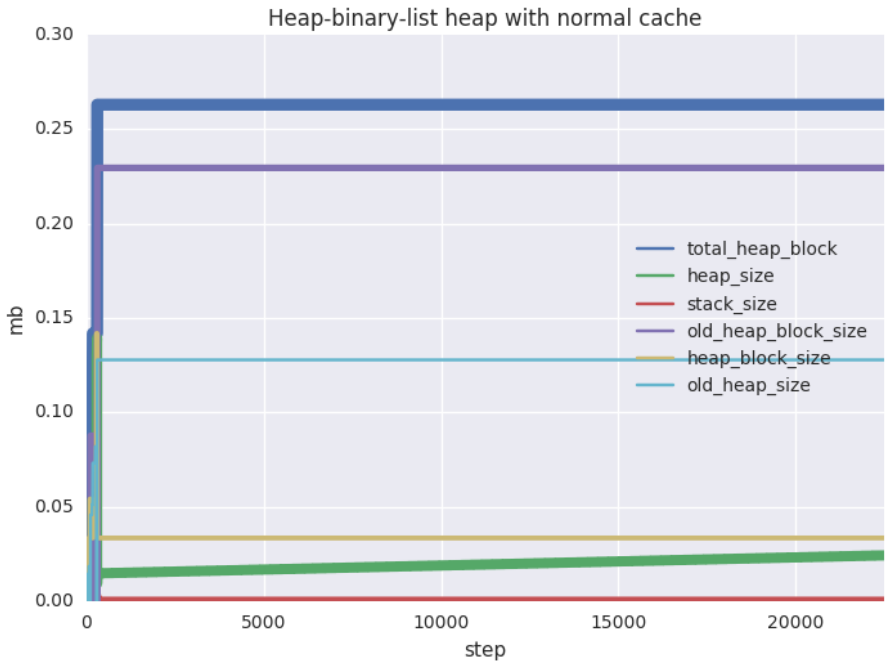


Figure 62: Heap-binary-list heap with normal cache

The values in the graph are stable, and to see when they become stable, we zoom into the first 800 steps, seen in Figure 63. Note that the stable values seen in Figure 62 are taking form at step 300. At step 300 all the hostnames have been fetched from the cache, and `heap_size` drops while `old_heap_block_size` and `old_heap_size` is increases roughly the same amount - the hostnames are moved from the heap to the old heap.

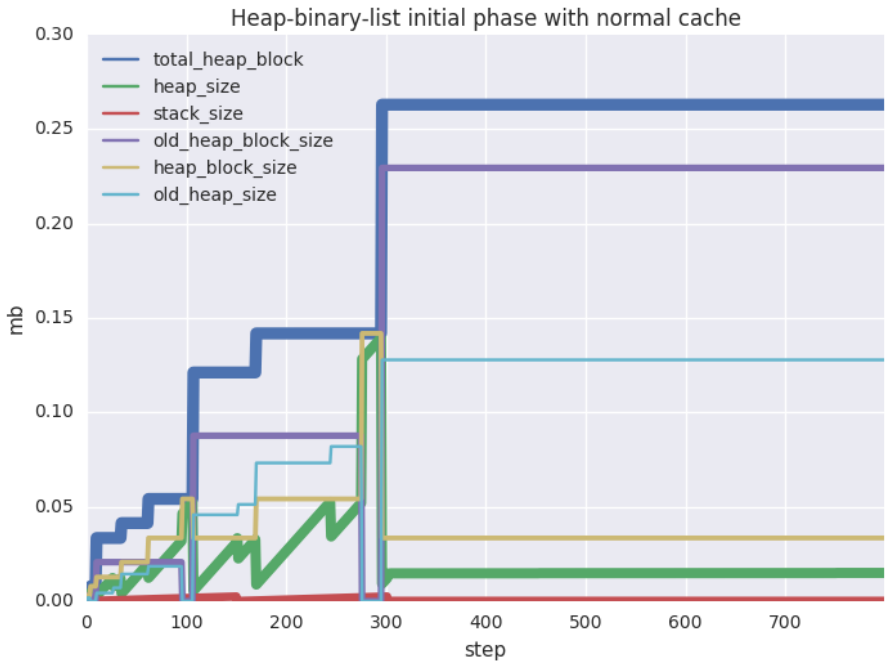


Figure 63: Heap-binary-list initial phase with normal cache

We now show how the `total_heap_block` reacts on the different types of cache data. Figure 64 shows that `normal` and `longer_labels` have almost the same heap values, while the much larger structure `longer_list` has a twice as big heap impact.

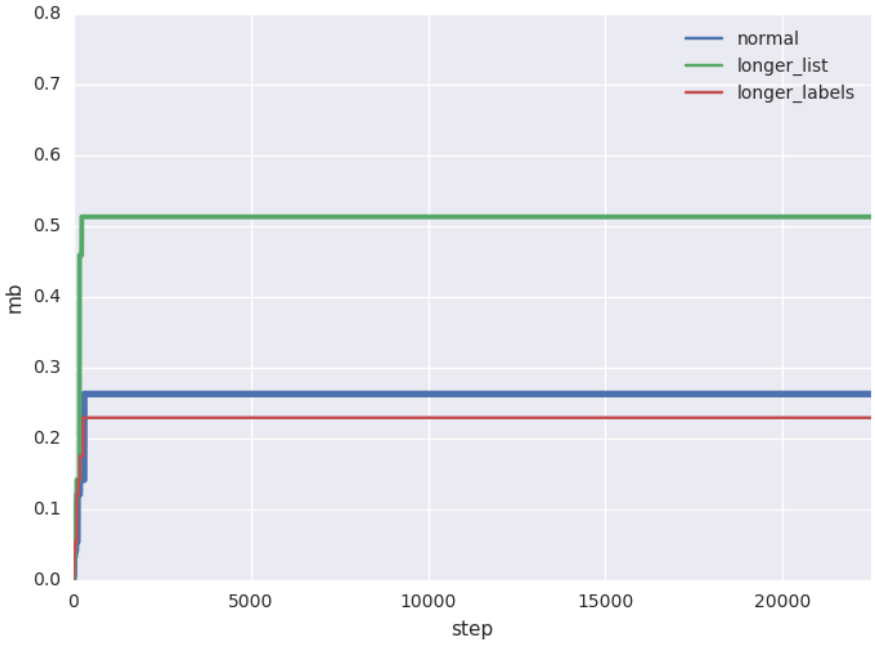


Figure 64: Heap-binary-list `total_heap_size` scaling

The `longer_labels` on the other hand has a slightly larger `heap_block_size` than `normal`, and does not directly correlate to the relation in 61. The reason for this is garbage collection timing. The larger-labels cache will populate the `heap_block` in such a way that the allocation of the `heap_block_size` is larger, allowing the `heap_size` to grow and by chance fit the allocated block better, than the `heap_size` of the `normal` case.

Big Integer List The next variant of data structure to examine is where we cast each string label to a big integer. The conversion is seen in Figure 65.

```
toBig(L) ->
    B = term_to_binary(L),
    BitSize = size(B) * 8,
    <<Big:BitSize>> = B,
    Big.

encode(Hostname) ->
    Labels = string:tokens(Hostname),
    lists:map(toBig, Labels).
```

```
A = "topon.interface.aaaaa.bbbbbbbbbbbbbbbbbb.
                          .cccc.ddd.eee
                          .ffffff.gggggg.hhhhhhhhhhh
                          .iii",

encode(A) -> [2424233642215653535598,
              ,10412004229865677718657607754597
              ,2424233642133813289313
              , ...]
```

Figure 65: Big-integer-list encoder

Big integers have a smaller overhead than binaries, which is seen by comparing Figure 66 with structures containing big integers with the heap binary equivalent from Figure 61. Figure 66 is misleading with regard to scaling, because it may seem that this structure scale in a different way than heap binaries. The reason is that a big integer, as well as a binary, increment with one word each eight bytes. Depending on the packing, the increments may appear at different stages. In this case, the `longer_labels` bar seem to scale worse than the one with a heap binary representation.

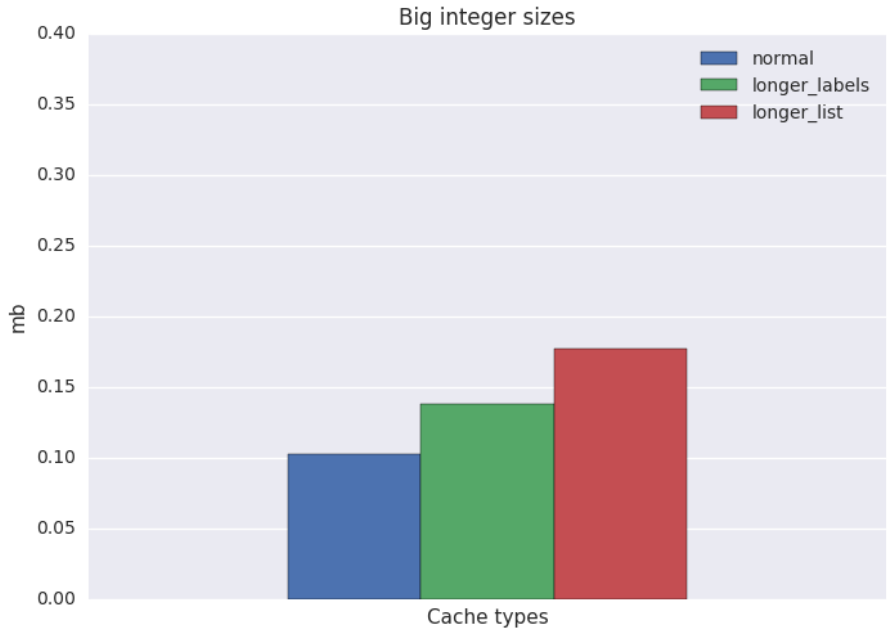


Figure 66: Size of the big-integer-list

The heap behavior with the normal cache is seen in Figure 67, where the `total_heap_block` is 0.7 MiB smaller than the heap binary version from Figure 62.

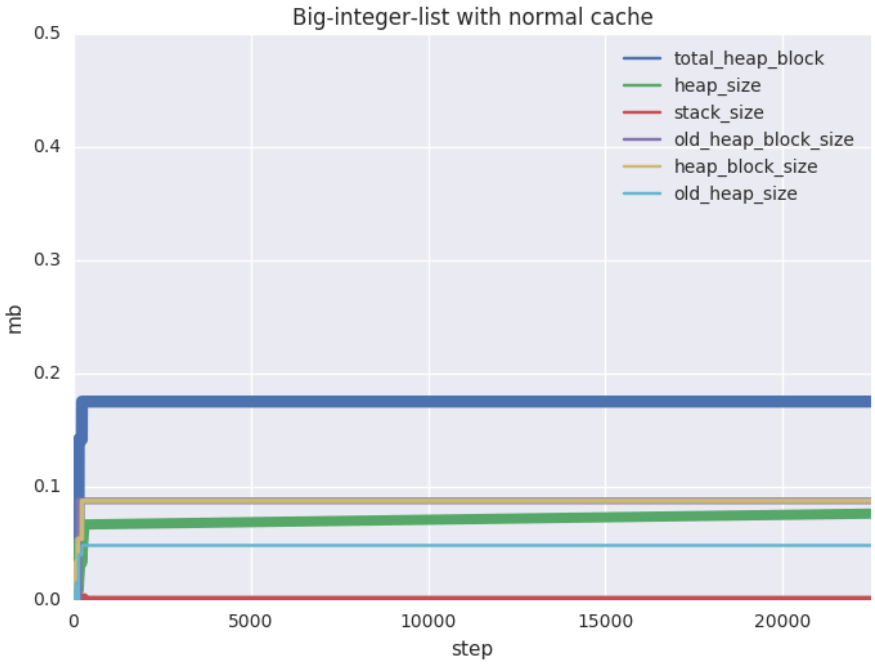


Figure 67: Big-integer-list with normal cache

As with heap binaries, the same stability is seen, and we zoom into the first 800 steps, showed in Figure 68.

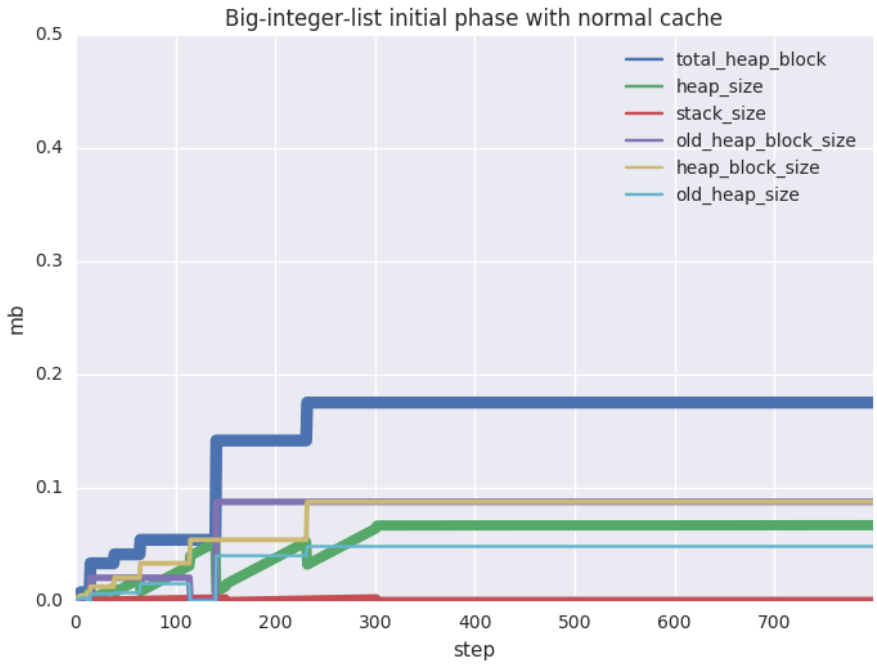


Figure 68: Big-integer-list initial phase with normal cache

The growth on the heap is not as big as with heap binaries, and this leads to smaller allocations and less movement between the heap and the old heap.

We see how the `total_heap_block_size` scale with the different caches in Figure 69. The structures of `longer_labels` and `longer_list` force the garbage collector to allocate a 0.1 MiB larger `total_heap_block` than the `normal` version.

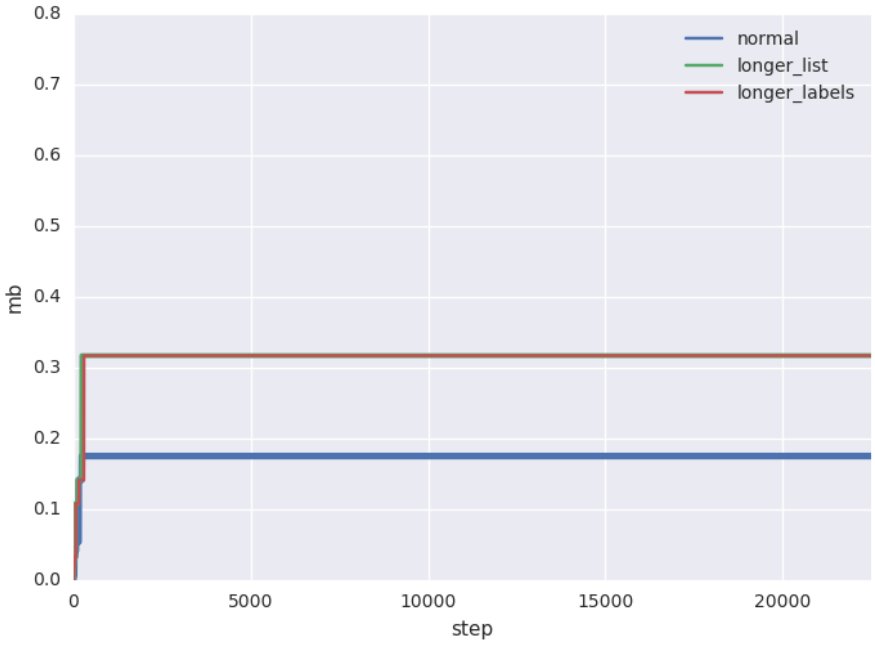


Figure 69: Big-integer-list `total_heap_size` scaling

We inspect the the 1000 first steps for the `largel_labels` case, and compare the heap values to the `normal` case. Figure 70 shows the heap values for `larger_labels` and we see the same kind of `heap_size` drop at step 300 as with heap binaries.

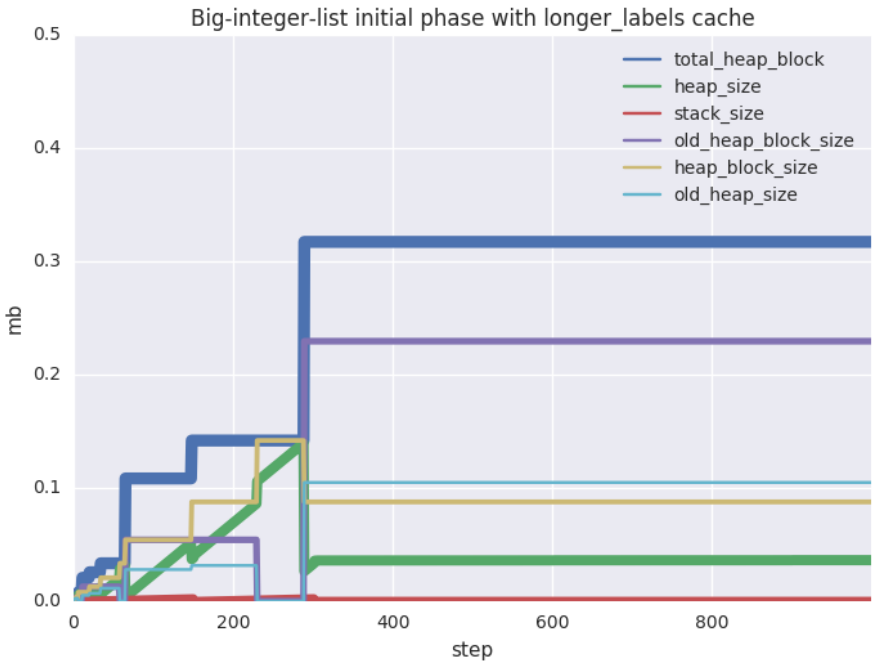


Figure 70: Big-integer-list initial phase with $\text{longer}_{\text{labels}}$ cache

Atom List We now introduce shared labels by representing each label as an atom. Creating atoms dynamically is controversial, since atoms are not garbage collected, and an uncontrolled dynamic creation of atoms can lead to memory issues. The encode function is seen in Figure 71.

```
encode(Hostname) ->
    Labels = string:tokens(Hostname),
    Atoms = lists:map(fun(L) ->
                      list_to_atom(L)
                      end, Labels),
    Atoms.

A = "topon.interface.aaaaa.bbbbbbbbbbbbbbbbbbb.
                           .cccc.ddd.eee
                           .ffffff.gggggg.hhhhhhhhhhh
                           .iii",


encode(A) -> [topon, interface, aaaaa, ... ]
```

Figure 71: Atom-list encoder

In Figure 72 we see that lists with atoms only use halft the space compared to big integers. This is expected since we get one word per label, when using atoms.

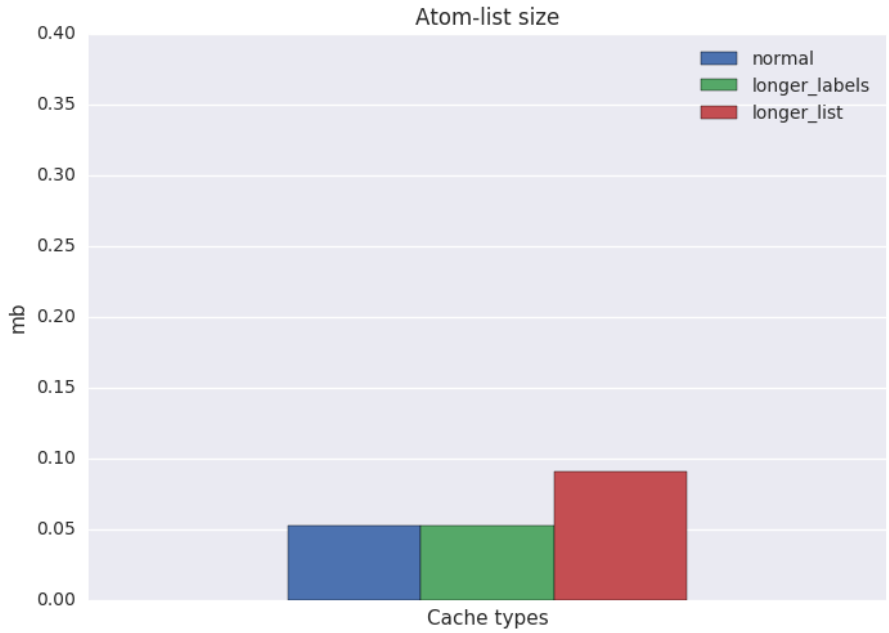


Figure 72: Size of atom-list

The heap development, in Figure 73, with the normal cache reflects this by approximately allocating half the memory of what the heap with big integers are doing.

Figure 73: Atom-list-heap-with normal cache

The scaling in Figure 74 also corresponds well to the sizes in Figure 72.

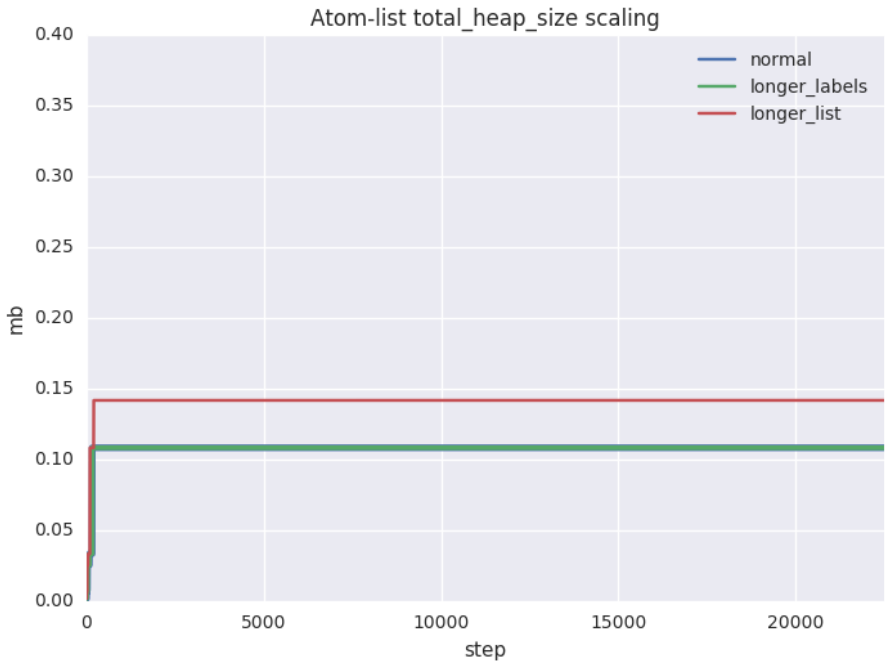


Figure 74: Atom-list `total_heap_size` scaling

Atom Tuple By storing atoms in a tuple instead of a list, we will get rid of a pointer in every cons-cell. An element in a tuple does not have any overhead, and the only overhead in a tuple is the header. The header just stores the size of the tuple.

The created tuple consists of two other tuples. The first tuple contains the topon/topoff and interface information, and the second tuple contains the labels in the nodename. This is seen in Figure 75.

```
{{topon, interface}, {aaaaa, bbbbb, ccccc, ...}}
```

Figure 75: Atom-tuple format

The encoding and decoding of a tuple with atoms can be seen in Figure 76.

```
encode(Hostname) ->
    Labels = string:tokens(Hostname),
    Tuple  = case Labels of
                 ["topon",Interface|CanonicalName] ->
                     {topon,
```

```
                     list_to_atom(Interface),
                     list_to_tuple(lists:map(fun(L) -> list_to_atom(L) end, CanonicalName))};
                ["topoff",Interface|CanonicalName] ->
                    {topoff,
                     list_to_atom(Interface),
                     list_to_tuple(lists:map(fun(L) -> list_to_atom(L) end, CanonicalName))};
                [Interface|CanonicalName] ->
                    {list_to_atom(Interface),
                     list_to_tuple(lists:map(fun(L) -> list_to_atom(L) end, CanonicalName))}
            end,
    Kind.


A = "topon.interface.aaaaa.bbbbbbbbbbbbbbbbbbb.
                          .cccc.ddd.eee
                          .ffffff.gggggg.hhhhhhhhhhh
                          .iii",


encode(A) -> {{topon, interface}, {aaaaa, bbbbbbbbbbbbbbbbbb, cccc, ...}}
```

Figure 76: Atom-tuple encoder

With tuples a different approach is used to perform the topological matching, since the elements in

a tuple are accessed by using indices.

```
get_priority({X,XS}, {Y,YS}) ->
    count_equal_labels_reversed(XS, YS).


% Count equal labels to match topological closeness
count_equal_labels(A, A) ->
    256;

count_equal_labels(A, B) ->
    count_equal_labels_reversed(A, B, 0).

count_equal_labels_reversed(AS, BS, Offset)
  when (size(AS)-Offset < 1) or (size(BS)-Offset < 1) -> Offset;
count_equal_labels_reversed(AS, BS, Offset) ->
    A = element(size(AS)-Offset,AS),
    B = element(size(BS)-Offset,BS),
    case A == B of
        true ->
            count_equal_labels_reversed(AS, BS, Offset+1);
        false ->
            Offset
    end.
```

Figure 77: `count_equal_labels` with lighter reverse

The overhead mentioned in Figure 75 is reflected in Figure 78, where the `longer_labels` cache is the one which gains most from the change to tuples, since the overhead is smaller in proportion to the number of labels.

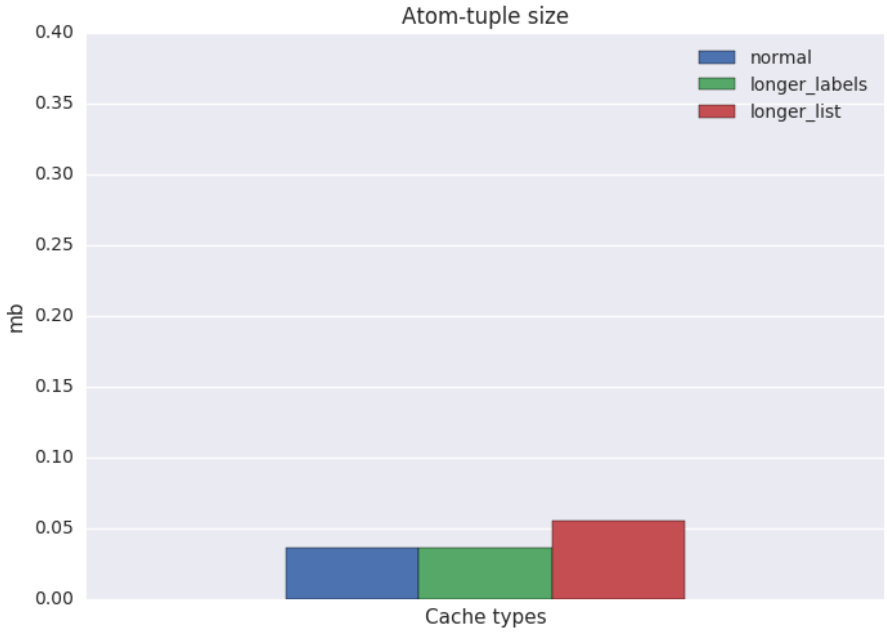


Figure 78: Size of the atom tuple

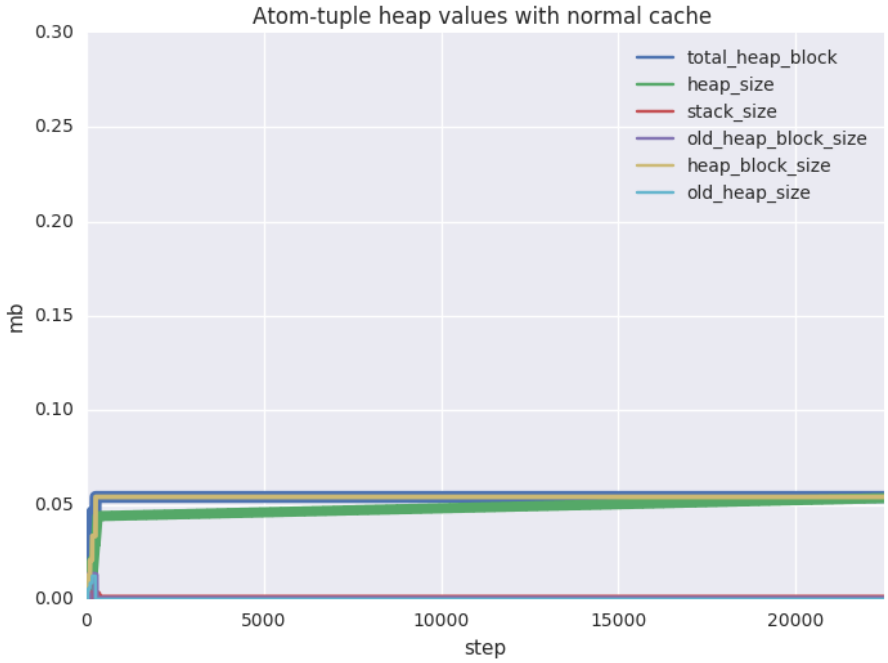


Figure 79: Atom-tuple heap values with normal cache

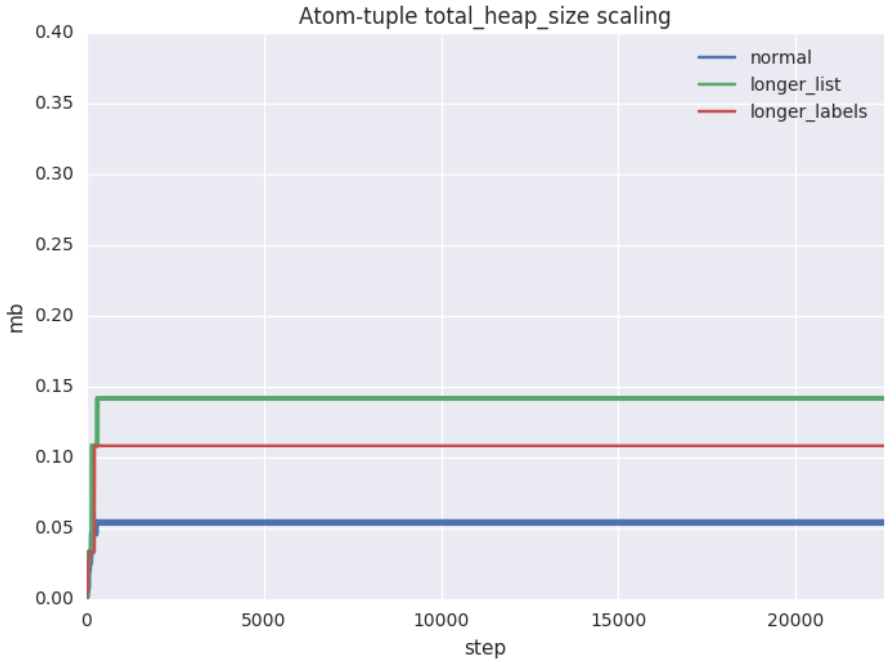


Figure 80: Atom-tuple `total_heap_block` scaling

Atom Trie We now store the hostname labels in tries. A trie is a tree-like structure with sharing of prefixes, as described in [19].

We use one trie for the SGWs and one trie for the PGWs, and then find a pair with the common longest suffix by finding the longest path in the intersection of the two tries. This leads to the creation of three tries in total, one PGW trie, one SGW trie and one intersection trie. In each node we also store the number of children, this gives us the opportunity to quickly find the longest path in a trie. The whole trie code can be seen in Appendix C.

Given n hostnames of constant size (which they approximately have in our DNS cache), the following time complexities apply:

- insertion of a hostname into a trie is *O(1)*
- intersection of two tries is *O(n)*
- finding a longest path in a trie is *O(n)*.

By using this approach, we get rid of the $O(n^2)$ time-complexity when comparing each pair combination.

The code snippet in Figure 81 illustrates a trie built from a list of labels. Each node in the trie also holds the the depth (from that node) of its nearest leaf, this value will be later used to quickly traverse the trie to find the longest path.

```
A = [[aaa, bbb, ccc],
     [aaa, bbb, ddd, eee, fff]].

B = [[aaa, bbb, ccc, ddd],
     [aaa, bbb, ddd", eee, ggg],
     [eee, fff, ggg]].

trie:build(A) ->
    [{2,aaa,
      [{1,bbb,
        [{0,ccc,[]},{2,ddd,[{1,eee,[{0,fff,[]}]}]}]}]}].

trie:build(B) ->
    [{3,aaa,
      [{2,bbb,
        [{1,ccc,[{0,ddd,[]}]},
         {2,ddd,[{1,eee,[{0,ggg,[]}]}]}]}]},
     {2,eee,[{1,fff,[{0,ggg,[]}]}]}].
```

Figure 81: Build a trie

Given the two tries `A` and `B` from Figure 81, the longest common paths can be found by intersecting the two tries. This operation can be seen in Figure 82.

```
trie:intersect(A, B) ->
  [{2,aaa,
  [{1,bbb,[{0,ccc,[]},
   {1,ddd,[{0,eee,[]}]}]}]}]
```

Figure 82: Trie intersection

By following the counter that holds the number of nodes to the nearest left stored in each node, we follow the path with the highest amount of children, to find the longest path.

When the longest path is found, we use it to match out the two paths from their original tries.

Figure 83 shows that the trie method is as effective as atom-list when using the normal and longer-labels cache. There is a lot of overhead in the trie implementation, since every node is a tuple with

lists. We also see that the trie performs a lot worse memory-wise when the shared suffix between the hostnames is significantly smaller, as in the case with the longer-list cache.

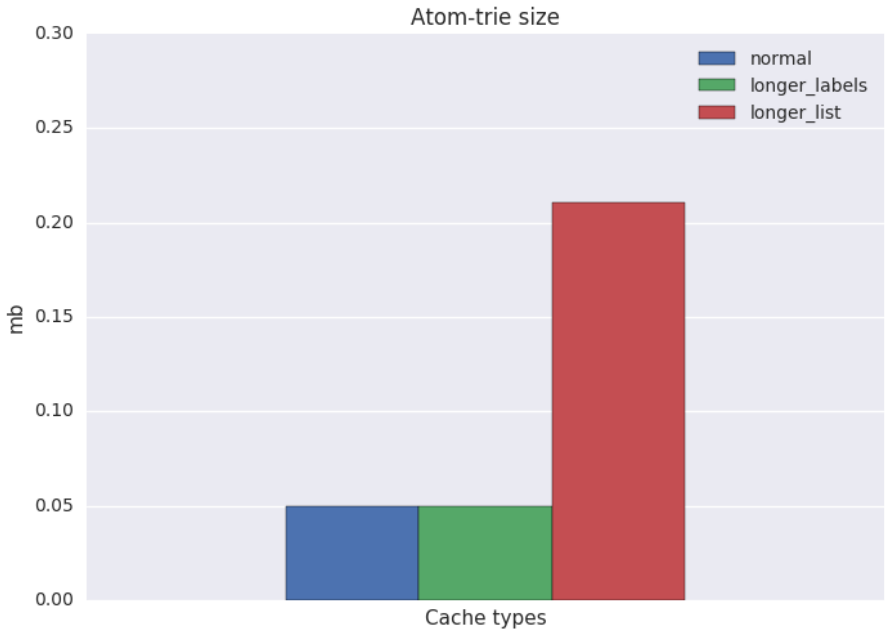


Figure 83: Atom-trie size

Figure 84 illustrates the heap behavior with the normal cache. Note that the whole operation takes 4000 steps and not 22500 steps as with all other representations.

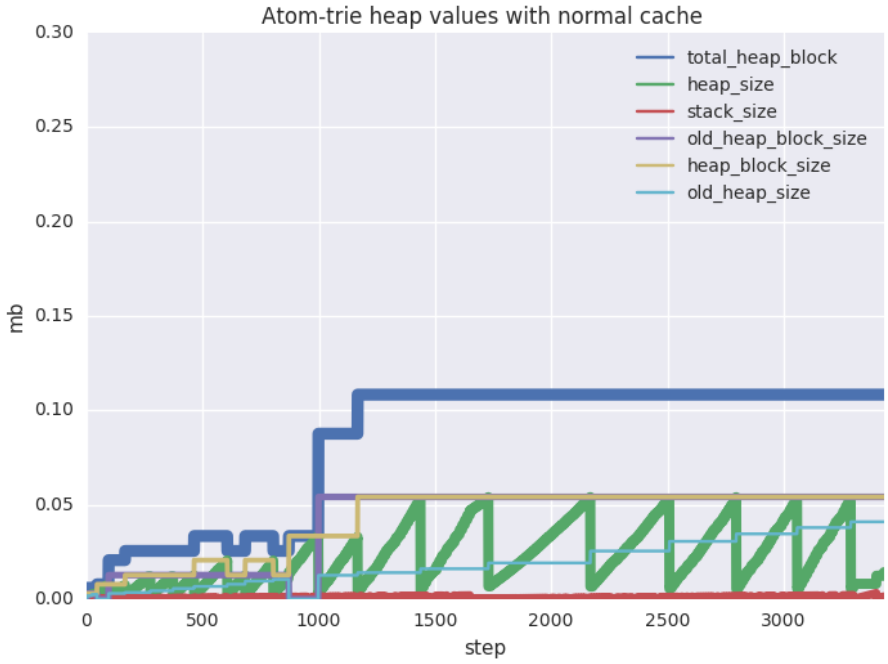


Figure 84: Atom-trie heap values on normal cache

Big Binary The final variant of data structure to examine is where we introduce sharing on the whole host name, by using the binary heap. Recall that binaries larger than 64 bytes are stored at a global heap, and they are shared between processes. This means that the DNS cache only have to store the reference to the binary heap, and a process will get the reference when querying the

DNS cache.

```
make_refc_binary(Bin) ->
    Size = size(Bin),
    %% If heap binary, add padding to make it a refc binary
    case Size < 64 of
        true ->
            Padding = 8 * (64 - Size),
            %% Create big binary, but point to the actual payload
            %% to make comparison easier
            Refc = <<Bin/binary, 0:Padding>>,
            <<Match:Size/binary,_/binary>> = Refc,
             % The Match Context type?
            Match;
        false ->
            Bin
    end.

encode(String) ->
    Labels = string:tokens(String, "."),
    Header = [lists:map(fun(T) -> length(T) end, Labels)],
    BinHeader = list_to_binary(Header),
    BinData = make_refc_binary(list_to_binary(String)),
    {BinHeader, BinData}.


A = "aaa.bbb.ccc"

encode(A) ->
         {<<3,6>>, <<"aaa.bbb.ccc">>}
```

Figure 85: Using big binaries

To count topological closeness we make use of the built-in function `binary:longest_common_suffix/1`. This function takes a list of binaries and returns the length (in bytes) of the common suffix of all binaries in the given list. To know how many labels are within the range that `longest_common_suffix` returns, we attach meta-data in a header. The header is stored together with the binary in a tuple. The format can be seen in Figure 85, at the last line. Two hostnames may have a common suffix that ends in the middle of a label, and that label is then not a part of the common suffix. We need a way to calculate the number of whole labels that exists within the range of bytes that is the

common suffix. The function that performs the matching is seen in Figure 86.

```
get_priority({H, B}, {H, B}) ->
    255;
get_priority({H1, B1}, {H2, B2}) ->
    CommonBytes = binary:longest_common_suffix([B1, B2]),
    get_priority(H1, H2, 0, CommonBytes, 0).
get_priority(H1, H2, Offset, CommonBytes, Acc) ->
    Idx1 = size(H1) - Offset - 1,
    Idx2 = size(H2) - Offset - 1,

    Size1 = binary:at(H1, Idx1),
    Size2 = binary:at(H2, Idx2),

    case Size1 == Size2 andalso Acc + Size1 =< CommonBytes of
        false ->
            Offset;
        true ->
            get_priority(H1, H2, Offset+1, CommonBytes, Acc+Size1 + 1)
    end.
```

Figure 86: Matching big binaries

The sizes of the big-binary representation are constant disregards of cache version. This is because the process only stores a minimal binary header and a pointer to the big-binary heap, seen in Figure 87.

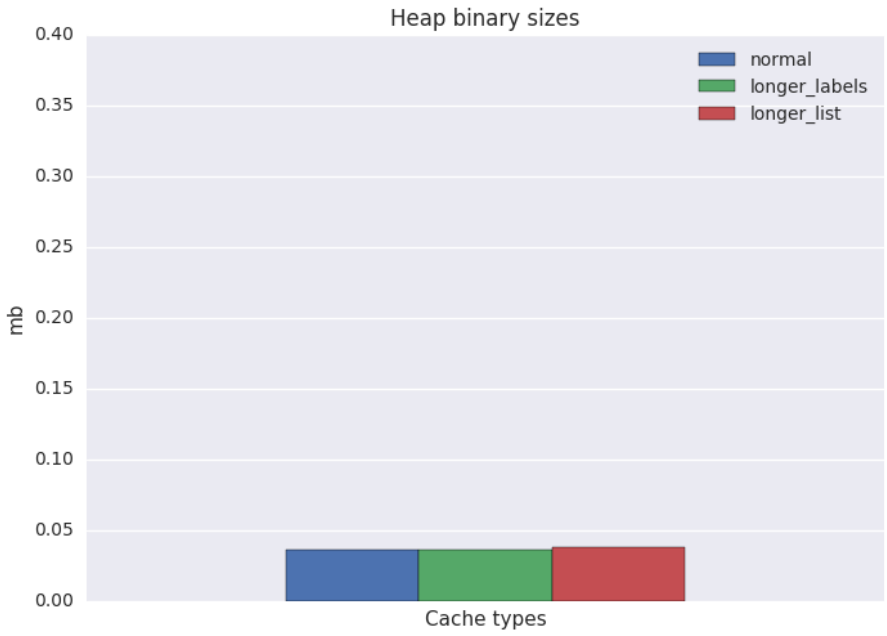


Figure 87: Size of big binary

The heap values are shown in 88 where the heap activity comes from the fact that `binary:longest_common_suffix/1` takes a list of binaries as argument, then a new small list has to be created every time `binary:longet_common_suffix` is called.

Figure 88: Big binary heap values on normal cache

Figure 89 is misleading, since we expect the heap blocks to have equal size. The variation in total heap size is caused by the timing of the garbage collection. The $\text{longer}_{\text{list}}$ version has less heap because the longer header that holds the length of each label is a heap binary that, when growing, may trigger a garbage collection, that frees up the heap block.

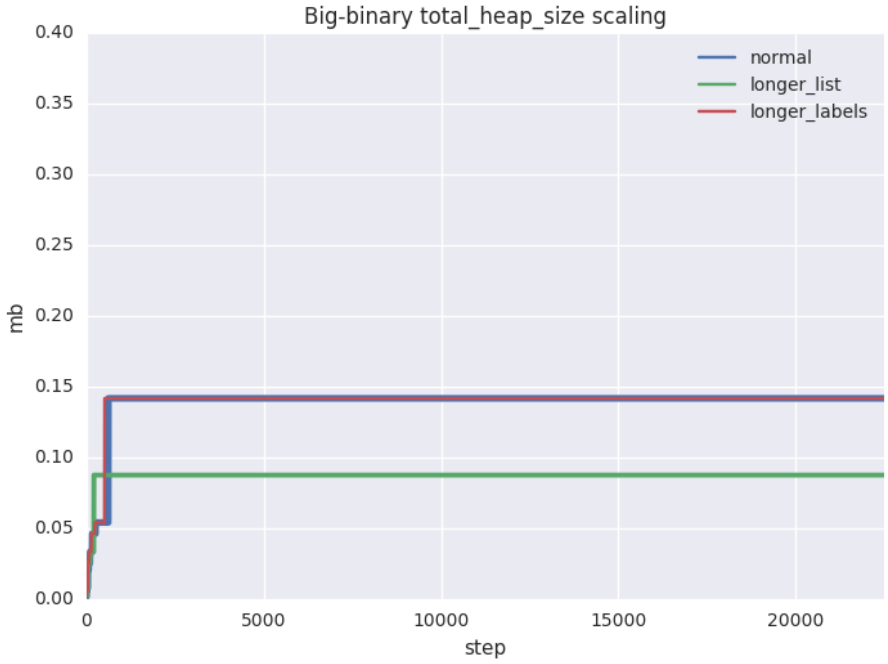


Figure 89: Big-binary `total_heap_block` scaling

The reason why we are not matching out the labels with sub binaries from Section 16, implemented in Figure 19, is because of their large overhead.

It is tempting to split a big binary by the dots with `binary:split/3`, that returns a list of sub binaries that points into the original big binary.

```
binary:split(<<"aaa.bbb.ccc">>, [<<".">>],[global]) ->
      [<<<"aaa">>, <<"bbb">>, <<"ccc">>]
```

But the overhead seen in the C structure in Figure 19 and the extra work for the garbage collector results in the following heap values, shown in Figure 90. We inspect the 5000 first steps that are representative for the whole execution. By splitting the big binary into smaller sub binaries, the total heap block in increased by 0.5 MiB, compared to the more constant heap block size in Figure 88.

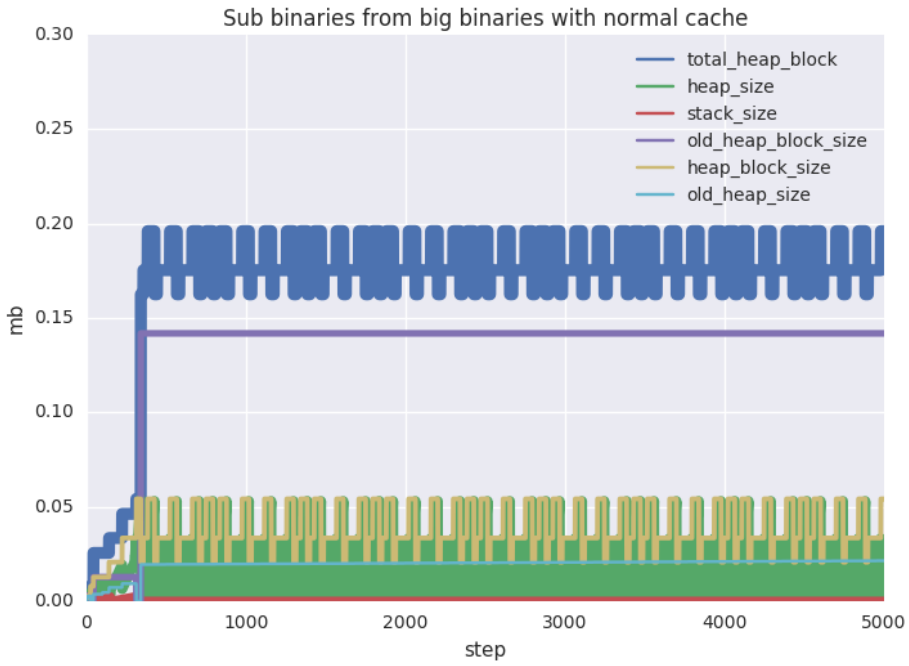


Figure 90: Big binary heap values on normal cache

4.3.4 CPU

We inspect the time it takes to perform the gateway selection procedure by the different representations in Figure 91.

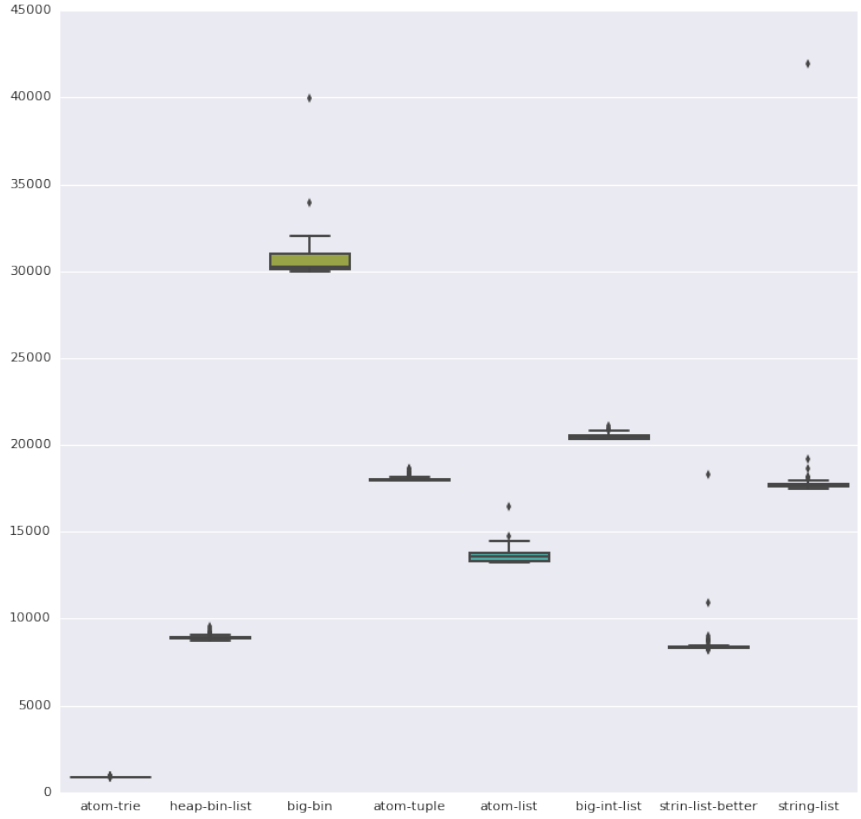


Figure 91: Time performance of different representations

The trie solution is the fastest and the solution with big binaries is the slowest.

4.3.5 Discussion

In Figure 92 we show a summary over the memory consumption with the normal cache, and Figure 93 we show a summary over the memory consumption with the $\text{longer}_{\text{list}}$ cache.

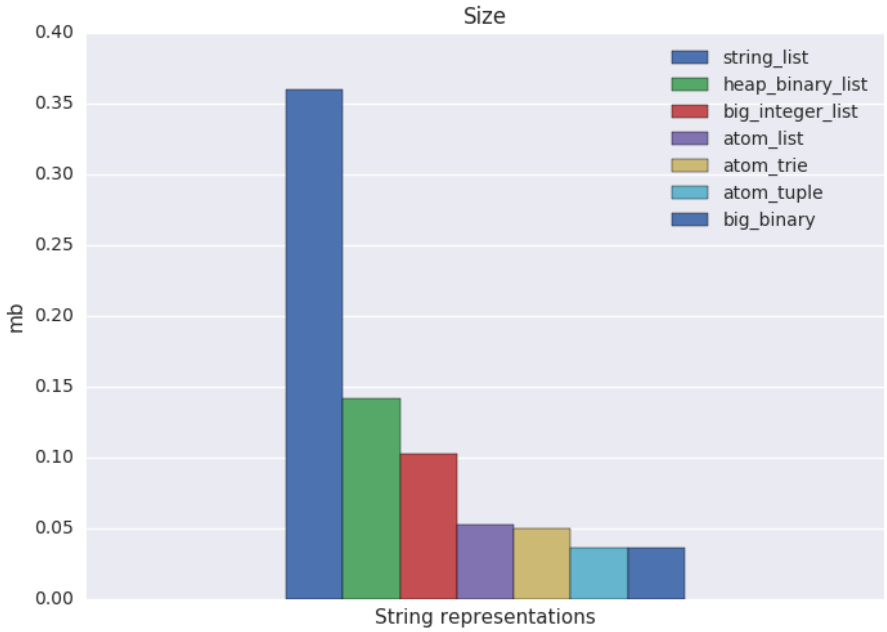


Figure 92: Overview over different representations with normal cache

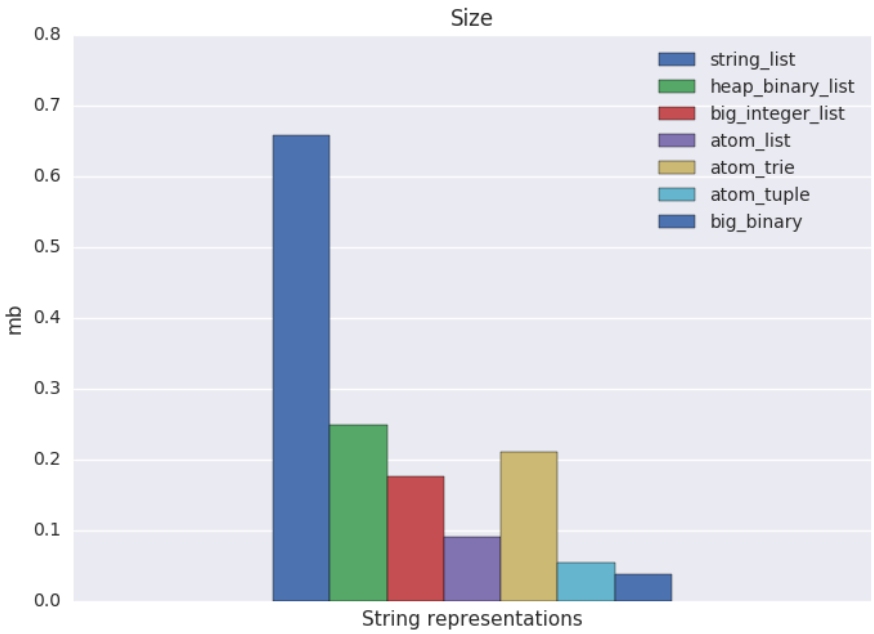


Figure 93: Overview over different representations with longer list cache

As we see in Figure 91 the trie solution is the fastest, due to its lower time complexity. But the trie solution does not scale as well when the labels do not share suffixes, as seen in Figure 93. The best scaling memory wise is achieved with big binaries, but since they are stored outside of the process heap, all operations on them are expensive, since the data is far away from the processor.

The solution with atoms is almost as good as big binaries, and has better cpu performance. The only downside with atoms is that they are created dynamically and can overflow the fixed atom table. If the input data is known, and it can be assured that the atom table will have enough space, then this is a good solution. The difference between tuples and lists in the atom case gets larger when the input data have longer lists.

We see that heap binaries have a bigger overhead than big integers, but heap binaries are a bit faster.

5 Conclusion

The purpose of this work was to explore a more efficient string representations in Erlang, to tackle the overhead found in Erlang's native string implementation, as well as algorithmic improvements for certain string operations, like finding two strings with the longest common suffix. As use-case we use Ericsson's SGSN-MME, in particular the string operations on hostnames öfound in the gateway selection algorithm. Our contribution is mainly the way we measure memory to understand memory inefficient code, and the comparison of different string representations.

We present a deterministic method to measure the heap values over time in an Erlang process. This gives us the ability to study in detail how the heap behaves and spot patterns such as garbage collection and stack growth, that can help us understand high memory consumption. By using this method we find and improve the gateway selection algorithm and go from the space complexity of $O(n^2)$ to $O(n)$, while keeping the same time complexity of $O(n^2)$. This gives us a situation where the set of input strings is the only bottleneck memory-wise. We can now compare different string representations by using other Erlang types and more sophisticated data structures to represent the hostnames.

All representations we present have both strengths and weaknesses, but we conclude that for the explored set of hostnames, the most memory efficient method is to store the hostnames as big binaries. This forces the string to be stored outside of the process heap, and the process heap only needs to store a pointer and some meta-data, where both can be considered to take constant space. This method scales well, since alternation in hostname length does not affect the size of the pointer to the binary. The downside is memory locality, since the virtual machine has to fetch the data from a global binary heap to perform any computations.

The best performance time-wise is achieved by using tries, since hostnames share a large part of their suffix. The operation of building up the trie structure and finding the best pair has the time complexity of $O(n)$. The memory consumption is slightly higher than the one with big binaries, but the trie does not scale as well as binaries, and if the set of hostnames does not have any suffix sharing, the trie performs poorly memory-wise, since sharing is lost.

The other methods are variations of packing where we take advantage of the fact that each character in a hostname can be represented with 1 byte, unlike the native string representation in Erlang, where a character that is part of a string, uses 16 bytes.

Finally, the choice of representation is dependent on the string data, but the important contribution of this work is to show that the knowledge about how Erlang stores the different types internally, matters for the different design choices a programmer has to make. Even if Erlang is a high-level language, the understanding of the low-level aspects of its virtual machine can be necessary.

6 Related Work

The paper [20] identify memory inefficiencies concerning strings in the Java Virtual Machine (JVM). Specifically, the authors improve how the JVM handle *duplication*, the situation when many strings objects have the same value, and *fragmentation*, when there is a unused space in the character arrays holding a string. Both problems are solved by a special String Garbage Collection, that restructure the string data on the heap during a garabge collection. Duplicated strings objects containing the same string values are unified by pointing to a single instance of the string.

The paper [21] introduces a string type for Haskell named *ByteString*. This type uses an 8-bit unsigned integer type internally, and a string then becomes an array of unboxed bytes, instead of a linked list with boxed words.

Elixir is a language that compiles to BEAM byte code that runs on Erlang's virtual machine. In Elixir every string is represented as a bitstrings, instead of Erlang's native represenation with linked lists. This approach is taken to save space by packing the characters and support UTF-8. In UTF-8 a letter can be encoded with more than one byte, and will need some header information with length information. Managing all this with integers is cumbersome. The native string representation with linked lists can still be used, with special syntax. The old style is used to better interact with Erlang libraries using the native string representation. [22].

A process_info/2

Output from modified process_info/2.

```
> process_info(self(), [garbage_collection_info, garbage_collection]).
[{garbage_collection_info,[{old_heap_block_size,610},
                           {heap_block_size,376},
                           {mbuf_size,112},
                           {recent_size,73},
                           {stack_size,24},
                           {old_heap_size,401},
                           {heap_size,332},
                           {under_water,73},
                           {bin_vheap_size,0},
                           {bin_vheap_block_size,46422},
                           {bin_old_vheap_size,0},
                           {bin_old_vheap_block_size,46422}]},
 {garbage_collection,[{min_bin_vheap_size,46422},
                      {min_heap_size,233},
                      {fullsweep_after,65535},
                      {major_gcs,1},
                      {minor_gcs,5}]}]
```

Figure 94: Output from process info/2

B struct process

A process structure with addresses and values that can be inspected.

```
#  define HEAP_START(p)     (p)->heap
#  define HEAP_TOP(p)       (p)->htop
#  define HEAP_LIMIT(p)     (p)->stop
#  define HEAP_END(p)       (p)->hend
#  define HEAP_SIZE(p)      (p)->heap_sz
#  define STACK_START(p)    (p)->hend
#  define STACK_TOP(p)      (p)->stop
#  define STACK_END(p)      (p)->htop
#  define HIGH_WATER(p)     (p)->high_water
#  define OLD_HEND(p)       (p)->old_hend
#  define OLD_HTOP(p)       (p)->old_htop
#  define OLD_HEAP(p)       (p)->old_heap

struct process {

   Eterm* htop;         /* Heap top */
   Eterm* stop;         /* Stack top */
   Eterm* heap;         /* Heap start */
   Eterm* hend;         /* Heap end */
   Uint heap_sz;                /* Size of heap in words */
   Uint min_heap_size;         /* Minimum size of heap (in words). */
   Uint min_vheap_size;        /* Minimum size of virtual heap (in words). */

   Process *next;               /* Pointer to next process in run queue */

   ErlMessageQueue msg; /* Message queue */

   Eterm *high_water;
   Eterm *old_hend;            /* Heap pointers for generational GC. */
   Eterm *old_htop;
   Eterm *old_heap;
   Uint16 gen_gcs;              /* Number of (minor) generational GCs. */
   Uint16 max_gen_gcs;          /* Max minor gen GCs before fullsweep. */
   ErlOffHeap off_heap; /* Off-heap data updated by copy_struct(). */
   ErlHeapFragment* mbuf;       /* Pointer to message buffer list */
   Uint mbuf_sz;                /* Size of all message buffers */
   ErtsPSD *psd;                /* Rarely used process specific data */

}
```

Figure 95: Fragment from `struct process`

C Trie

```
%% Add path to Trie
%%
%% Store (tail length - 1) in each node.
%% The number of nodes to the nearest leaf.
add([], Tries) ->
    Tries;

add([L|LS], []) ->
    [{length(LS), L, add(LS, [])}];

add([L|LS], [{C, L, TS}|TSS]) ->
    Len = min(length(LS), C),
    [ {Len, L, add(LS,TS)} | TSS];

add(LS, [TS|TSS]) ->
    [TS | add(LS,TSS)]. % Append new node

%% Build Trie from a list of paths
build(LSS) ->
    lists:foldl(fun(LS, Acc) -> add(LS, Acc) end, [], LSS).


%% Intersect two Tries
intersect([], []) ->
    [];
intersect([], _) ->
    [];
intersect(_, []) ->
    [];
intersect([{C0,L,TS1}|TSS1], [{C1,L,TS2}|TSS2]) ->
    [{min(C0,C1), L, intersect(TS1,TS2)} | intersect(TSS1, TSS2)];

intersect(TSS1, TSS2) ->
    [ {min(C0,C1),L,intersect(TS1,TS2)}
      || {C0,L1,TS1} <- TSS1, {C1,L2,TS2} <- TSS2, L == T].
```

```
%% Find longest path
find_longest([L|LS]) ->
    find_longest(LS, L).
find_longest([], Best) ->
    Best;
find_longest([{C,_,_}| Rest], {C1, _, _}=Best) when C =< C1 ->
    find_longest(Rest, Best);
find_longest([L | LS], _) ->
    find_longest(LS, L).

%% Path to reversed list
path_to_rlist(TSS) ->
    path_to_rlist(TSS,[]).

path_to_rlist({_,L,[LS]}, Acc) ->
    path_to_rlist(LS,[L|Acc]);
path_to_rlist({_,L,[]}, Acc) ->
    [L|Acc].

%% Find longets pair from two sets of strings
find_longest_pair(XSS, YSS) ->
    Inter = intersect(build(XSS), build(YSS)),
    Path = find_longest(Inter),
    path_to_rlist(Path, []).
```

References

[1] Erlang;. Available from: https://www.erlang.org/.

[2] Ericsson AB. Erlang – Implementation and Ports of Erlang;. Available from: http://erlang.org/faq/implementations.html.

[3] Erlang – Data Types;. Available from: http://erlang.org/doc/reference_manual/data_types.html.

[4] Readme.md of OTP;. Available from: https://raw.githubusercontent.com/erlang/otp/maint/README.md.

[5] Robert Virding. Hitchhiker's Tour of the BEAM;. Erlang Factory. Available from: http://www.erlang-factory.com/upload/presentations/708/HitchhikersTouroftheBEAM.pdf.

[6] Patrik Nyblom. The Halfword Heap Emulator, Stockholm Lecturep;. Erlang Factory. Available from: http://www.erlang-factory.com/upload/presentations/467/Halfword_EUC_2011.pdf.

[7] Patrik Nyblom. The Halfword Heap Emulator, San Francisco Lecture;. Erlang Factory. Available from: http://www.erlang-factory.com/upload/presentations/569/Halfword_Erlang_Factory_SF_2012.pdf.

[8] Klaftenegger D, Sagonas K, Winblad K. On the scalability of the Erlang term storage. ACM Press;. p. 15. Available from: http://dl.acm.org/citation.cfm?doid=2505305.2505308.

[9] erlang/otp. Erlang Open Telecom Platform;. Available from: https://github.com/erlang/otp/tree/maint-18.

[10] Papaspyrou N, Sagonas K. On preserving term sharing in the Erlang virtual machine. In: Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop. ACM;. p. 11–20. Available from: http://dl.acm.org/citation.cfm?id=2364493.

[11] Pettersson M. A staged tag scheme for Erlang;. Available from: https://it.uu.se/research/publications/reports/2000-029/2000-029-nc.pdf.

[12] Ericsson AB. Erlang – Constructing and Matching Binaries;. Available from: http://www.erlang.org/doc/efficiency_guide/binaryhandling.html.

[13] Wilson PR. Uniprocessor garbage collection techniques. In: Memory Management. Springer; 1992. p. 1–42.

[14] Wilhelmsson J. Exploring Alternative Memory Architectures for Erlang: Implementation and Performance Evaluation. Master's thesis, Uppsala University. 2002;Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.3682&rep=rep1&type=pdf.

[15] Wilhelmsson J. Efficient memory management for message-passing concurrency, Part I: Single-threaded execution;Available from: http://www.diva-portal.org/smash/record.jsf?pid=diva2:117278.

[16] Urban Boquist. THE ERICSSON SGSN-MME;. Erlang Factory. Available from: http://www.erlang-factory.com/upload/presentations/597/sgsn.pdf.

[17] Kurose JF. Computer Networking: A Top-Down Approach Featuring the Internet, 3/E. Pearson Education India; 2005.

[18] 3GPP. 3rd Generation Partnership Project, Technical Specification Group Radio Access Network, Release 13;.

[19] Cormen TH. Introduction to algorithms. MIT press; 2009.

[20] Häubl C, Wimmer C, Mössenböck H. Compact and efficient strings for Java;75(11):1077–1094. Available from: http://linkinghub.elsevier.com/retrieve/pii/S0167642310000791.

[21] Coutts D, Stewart D, Leshchinskiy R. Rewriting haskell strings. In: International Symposium on Practical Aspects of Declarative Languages. Springer; 2007. p. 50–64.

[22] Core-Developer of Elixir AM;. personal communication.