THE HALFWORD HEAP EMULATOR

EXPLORING A VIRTUAL MACHINE

PATRIK NYBLOM, ERICSSON AB

pan@erlang.org

ERICSSON

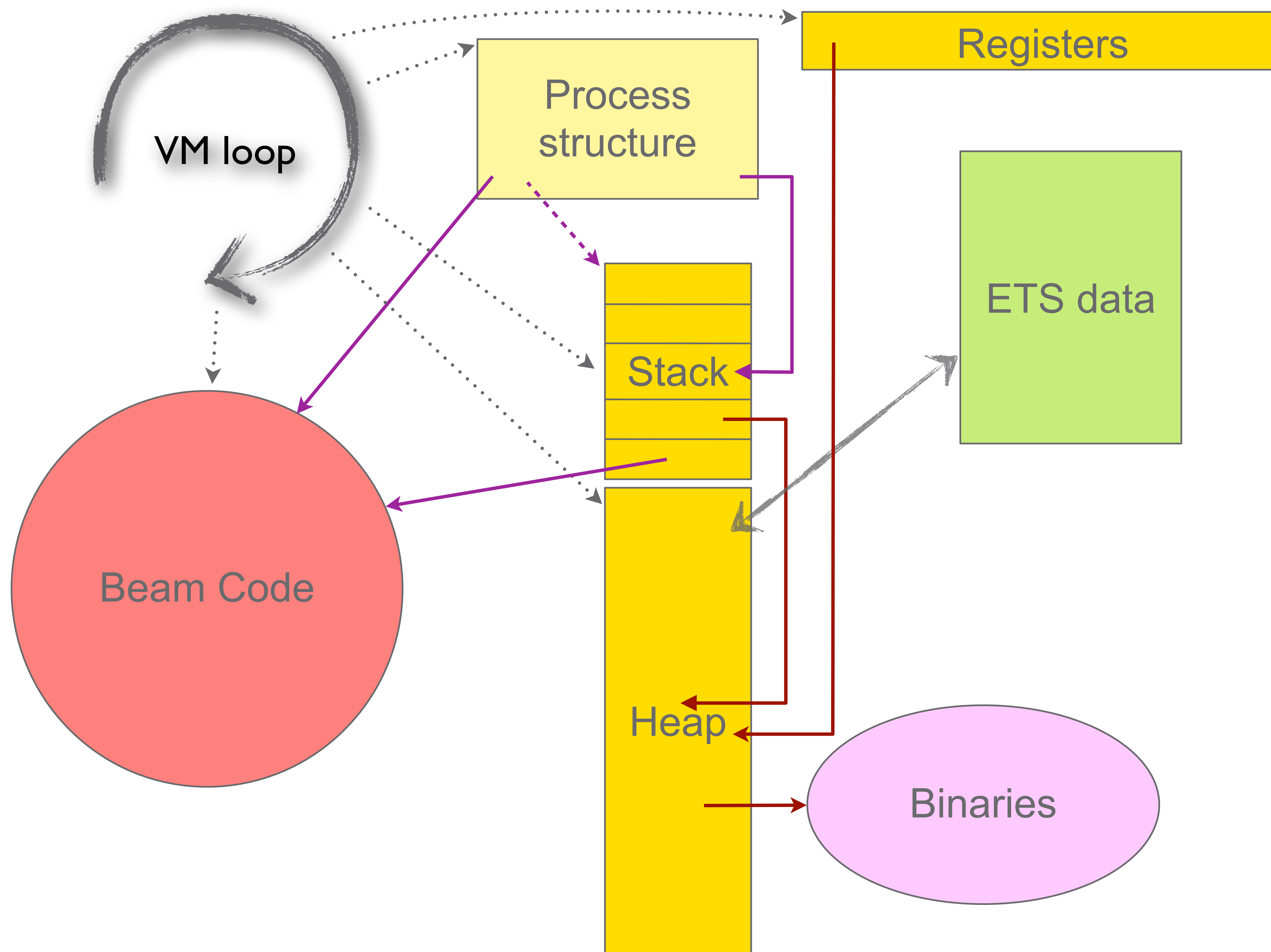
THE BEAM VIRTUAL MACHINE

- **B**jörns/Bogdans **E**rlang **A**bstract **M**achine
- Has evolved over the years and is a joint effort by many developers, within and outside of the Erlang OTP team
- A virtual register machine with 1024 virtual registers
- Garbage collection is done per process with a generational copying GC (two generations + a special nursery used only by native code functions)
- Has a constant pool that is not garbage collected
- Large binaries are stored outside of the heap and may be shared among processes

THE BEAM VIRTUAL MACHINE (CONTINUED)

- The most important datatype in the VM code is the `Eterm`, one (possibly complex) Erlang term
- An `Eterm` is normally one word (`sizeof(void *)`) large
- The heap of a process is an array of `Eterm`'s
- The data in ETS tables are stored as `Eterm`'s
- The registers are `Eterm`'s
- The stack in the VM consists of `Eterm`'s
- When a complex term is constructed in the VM, a number of `Eterm`'s are allocated on the process heap, to represent e.g. tuple's, list (cons) cells etc
- The variables in the program are stored on the stack or in a register
  - If the term the variable refers to is complex, the stack/register will contain a pointer to the heap, so the `Eterm` has to be able to contain pointers

THE BEAM VIRTUAL MACHINE (CONTINUED)



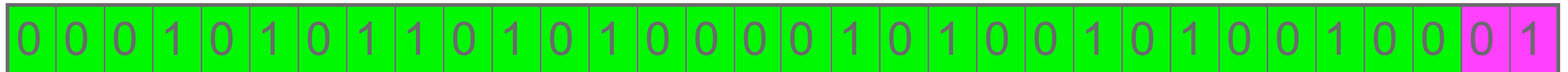
REPRESENTING ERLANG TERMS

› Erlang is a dynamically typed language

› The virtual machine determines the type of a value on the stack or in registers by looking at *tags*

› A *tag* is in reality a couple of bits stolen from the data to encode the type information needed

› Erlang has a hierarchical tag system where the primary tags are placed in the two least significant bits of each word:

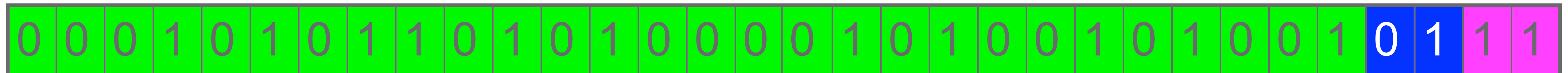


› The primary tags are (currently)

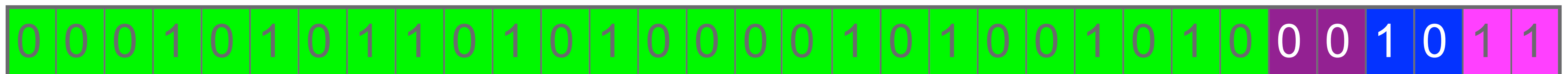
- 00 = Continuation pointer (return address on stack) or header word on heap
- 01 = Cons cell (list)
- 10 = Boxed (tuple, float, bignum, binary, external pid/port, exterrnal/internal ref ...)
- 11 = Immediate (the rest - secondary tag present)

REPRESENTING ERLANG TERMS (CONTINUED)

- The immediate values have more tag bits



- The current encoding is
  - 0011 = Pid (local)
  - 0111 = Port (local)
  - 1011 = Immediate 2 (the rest, even more tag bits)
  - 1111 = Small number (fixnum), up to 28 bits including sign
- And then there's the rest of the immediates, with two more tag bits:

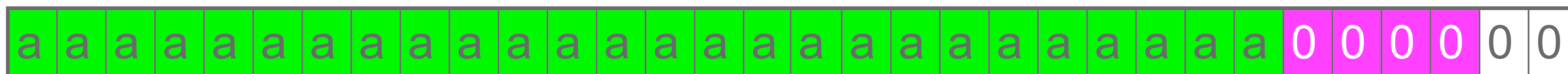


- The current encoding is:
  - 001011 = Atom
  - 011011 = Catch (on stack)
  - 111011 = NIL

ERICSSON

Representing Erlang terms (continued)

› The boxed values on the heap (primary tag 10) all begin with a header word, also containing tags and 26 bits for arity



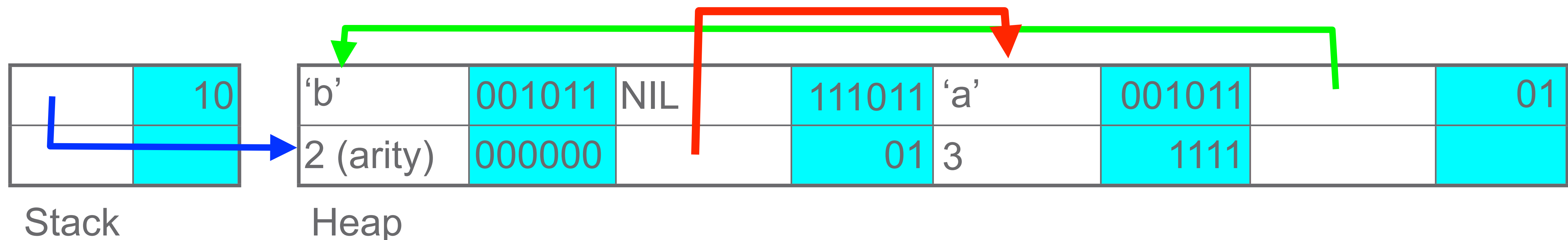
› The tags here are currently
- 0000 = Tuple
- 0001 = Binary match state (internal type)
- 001x = Bignum (needs more than 28 bits)
- 0100 = Ref
- 0101 = Fun
- 0110 = Float
- 0111 = Export fun (make_fun/3)
- 1000 - 1010 = Binaries
- 1100 - 1110 = External entities (Pids, Ports and Refs)

WHEN A TERM IS CREATED

- So - when a term is created...
  - You write something like this in your program:

```
A = {[a,b],3}
```

  - The virtual machine puts the following on the heap:
    - A pair containing the atom ‘b’ as it’s first element and NIL as it’s second
    - A pair containing the atom ‘a’ as it’s first element and a pointer to the previous pair (tagged as *Cons cell*) as it’s second
    - A header word followed by a pointer to the previous pair (tagged as *Cons cell*) and the (*Small*) number 3
  - And then, in a register or on the stack, a pointer tagged as *Boxed* is representing the whole term, referring to the header word of the tuple



ERICSSON

THE HEAP IS AN ARRAY OF WORDS

- Each cell on the heap needs to be able to contain a tagged pointer (a Boxed or a Cons)
  - So does each cell on the stack and each register
- Even a small number (like 3) takes a cell (a word) on the stack or the heap
- If the size of a pointer changes, the whole size of the heap and stack follows
  - 64bit pointers means more or less twice as large heap as with 32bit pointers
  - Not entirely true, larger numbers and binary data can be contained in one cell on the heap, but almost true...
  - The overhead for strings is overwhelming in a 64bit VM, almost 16 times the size of a corresponding binary (at least for long latin1 strings)...

CPU SPEED AND MEMORY BANDWIDTH

ERICSSON

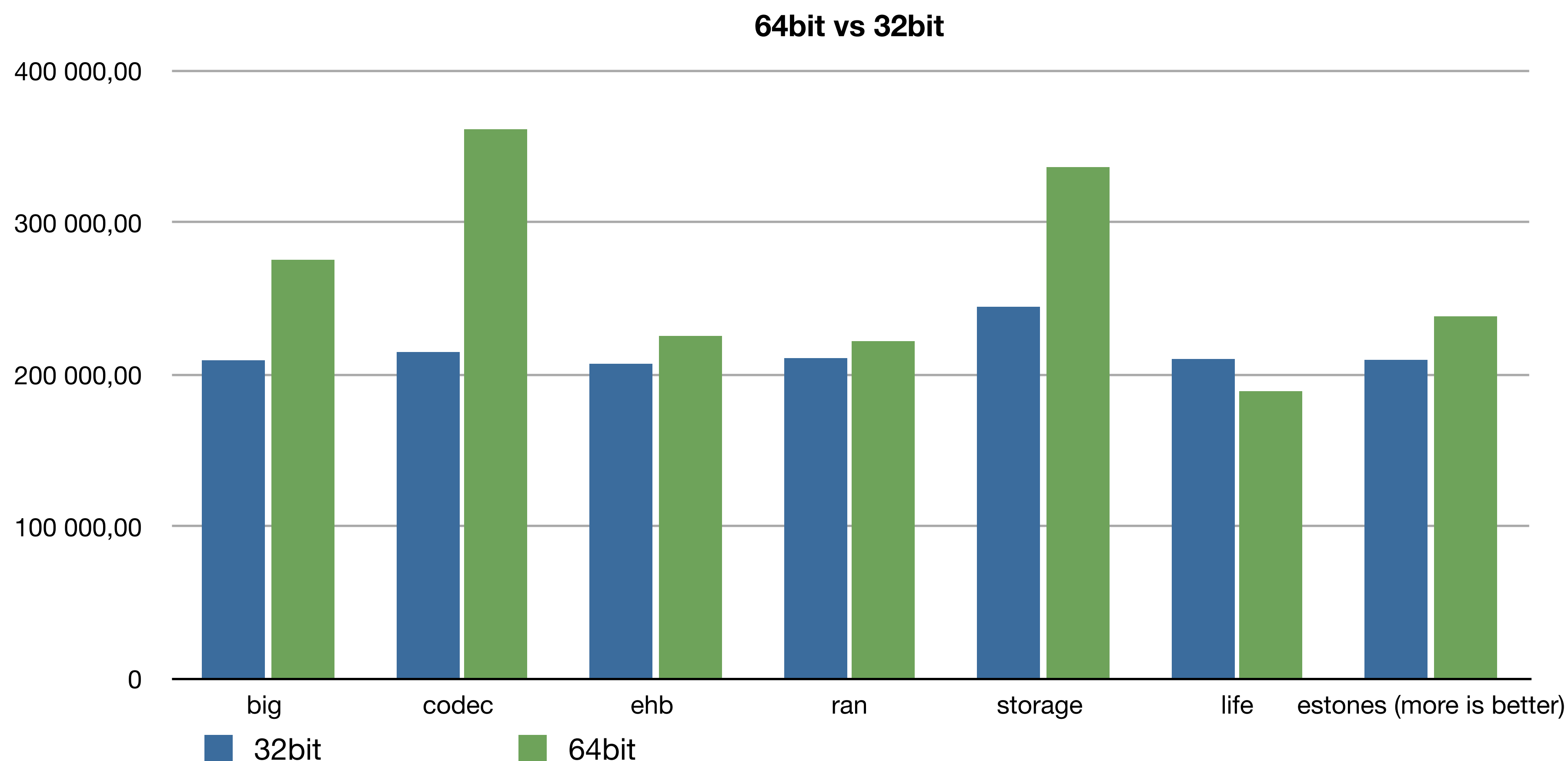
- The combined speed of the cores in modern computers still increases rapidly
- Except for the leap in memory bandwidth in the Core i7 architecture (which Intel representatives describe as a “one time gift”), memory bandwidth can not keep up with the CPU speeds
- Utilizing the CPU’s potential despite the slow memory is one of the most interesting challenges we have
- Memory consuming programs will continue to be slow, even with faster and more cores
- A program that consumes twice the amount of memory will have to wait twice the time for data to be available from memory
  - except for when it’s already in the cache, but the cache will be exhausted twice as fast...

IA-32 VS X86-64

- The x86_64 processors are extended IA-32 (x86) processors
  - Twice as large general purpose registers
  - Twice the amount of GPR's
  - Twice the amount of XMM registers
  - 48 bit address space (in the future extendable to 64 bit)
- The x86_64 processors can execute in 32 bit legacy mode
  - But then we are back in the 32 bit world and lose 8 GPR's among other things
- For programs that do not need several gigabytes of memory, 32 bit mode might increase performance
  - if you have a lot of pointers...

THE PROBLEM

- The Erlang VM does use a lot of pointers...
- Erlang programs also tend to be large, consuming more than 2GB of memory
- When you need to switch from a 32bit VM to a 64bit, you suddenly start consuming nearly twice the amount of memory
- ...while execution speed falls when not number crunching:



THE ALTERNATIVES

- Stick to the 32 bit machine
  - If the program is to large, spread over several virtual machines on a 64bit OS
  - Pros:
    - Execution speed in the 32bit VM is usually higher
    - Combined memory footprint is small
  - Cons:
    - The system needs to be redesigned (ouch)
    - Execution speed may fall due to internode communication
- Buy more memory and faster machines, then switch to 64 bit
  - Pros:
    - No redesigning
    - The way to go eventually anyway...
  - Cons:
    - Costly if at all possible
    - May still be slower

ERICSSON

THE ALTERNATIVES (CONTINUED)

- A virtual machine where heap data is referred to as 32 bit offsets into the heap
  - Sounds nice, but heap data is scattered over two generations of heaps, temporary storage, constant pools etc
  - Also return addresses on the stack needs full pointers
  - Requires huge rewrite and would result in performance degradation
- A virtual machine where each process has it's own 4 GB address space
  - Nice idea, would work if we had the full 64 bit address space to work with
    - Unfortunately we do not, we have a maximum of 48 bits for addressing and would have to limit the number of processes
  - By letting processes share memory areas and relocating processes when one 4GB area gets full, we could make this work
    - This however would require serious rewrites of almost every built in function in the VM
- A virtual machine where all heap data is kept in a 32bit world but the rest of the VM can operate in the full 64bit address space
  - This is definitely feasible

THE SOLUTION

- All Erlang process data (heap and stack) is kept in one 4GB address range
  - This means that the sum of all the heaps and stacks (+ beam code) in the system may not exceed 4GB (or something slightly less, as the data section may interfere)
  - ETS data and shared binaries are *not* included in those 4 GB
  - For an application with a moderate process sizes and numbers, but with large ETS datasets and/or large binary datasets this approach may reduce the memory footprint severely
  - By also compacting the ETS data we get almost the same memory footprint as with a 32 bit VM (but are able to have more data in memory)
  - This is also fairly simple to implement on 64 bit Linux

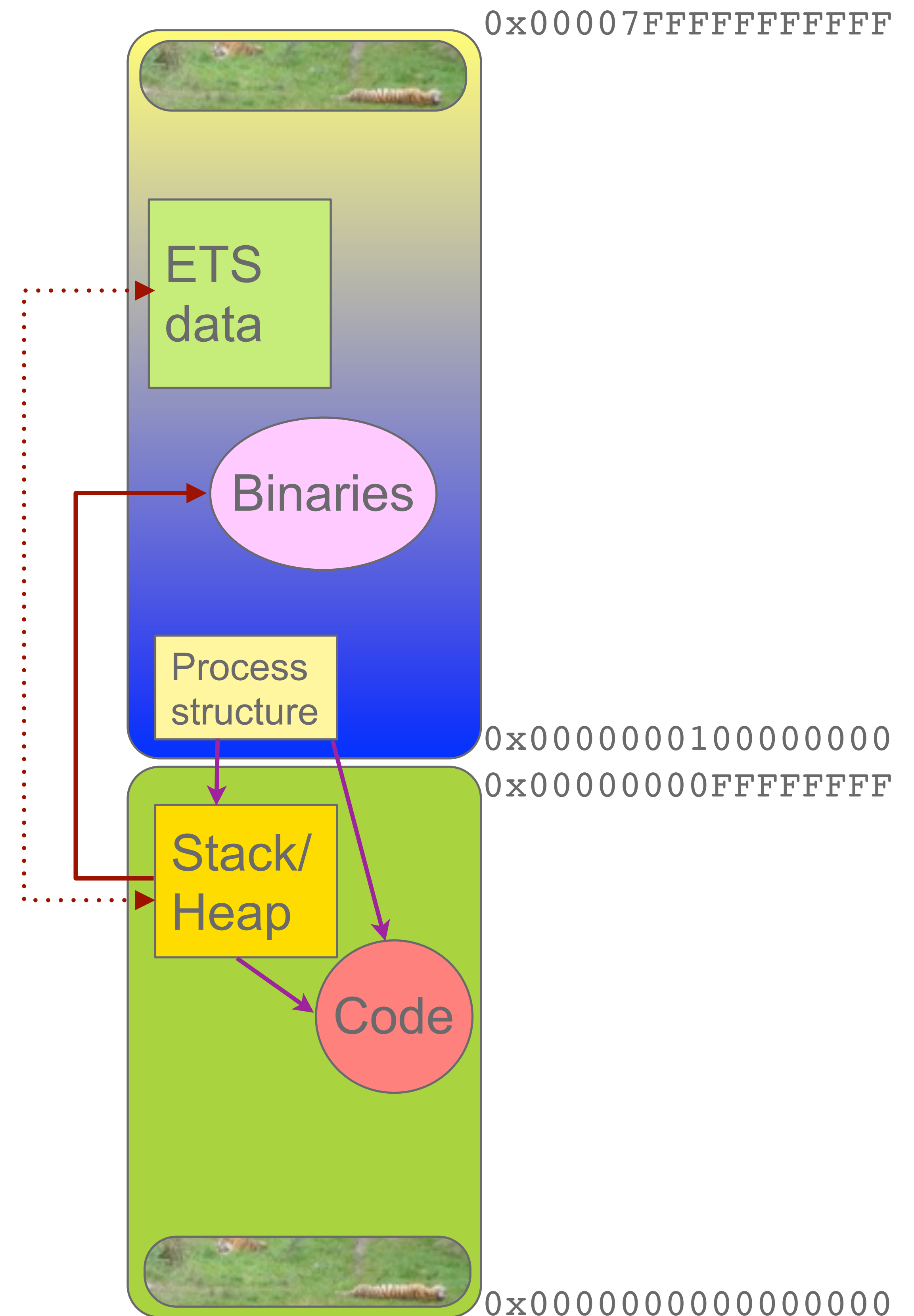
ERICSSON

THE IMPLEMENTATION

- We need a way to reserve the lower 4GB of the address range for Erlang process data
  - This can be done on Linux in a fairly simple way, using mmap and MAP_NORESERVE to take control of the "low" memory
  - An memory page allocator for the lowest 4 GB of memory was implemented that gives processes memory pages from the "low" range
  - As all memory allocations in the VM specify the kind of data (i.e. process heaps are allocated with `erts_alloc(`**`ERTS_ALC_T_HEAP`**`,size)` etc), it's quite easy to direct the heap data to pages allocated with the special "low" memory mmap allocator
- We also need to truncate pointers to 32 bit when stored in the process and expand them to 64 bit when we are to follow the pointers
  - This is done when tags are added/removed - almost no extra cost!

THE IMPLEMENTATION (CONTINUED)

- A few other things needed to be tampered with
  - Code needs to be in “low” memory, but threaded instructions are 64 bits as actual instructions are pointers into the VM executable
  - Pointers to binary data takes two “halfwords” on heap as binaries are in “high” memory
  - Code that expects a regular pointer to fit in an Erlang term variable needed rewriting
  - Code that stored Erlang terms on the C stack needed rewriting



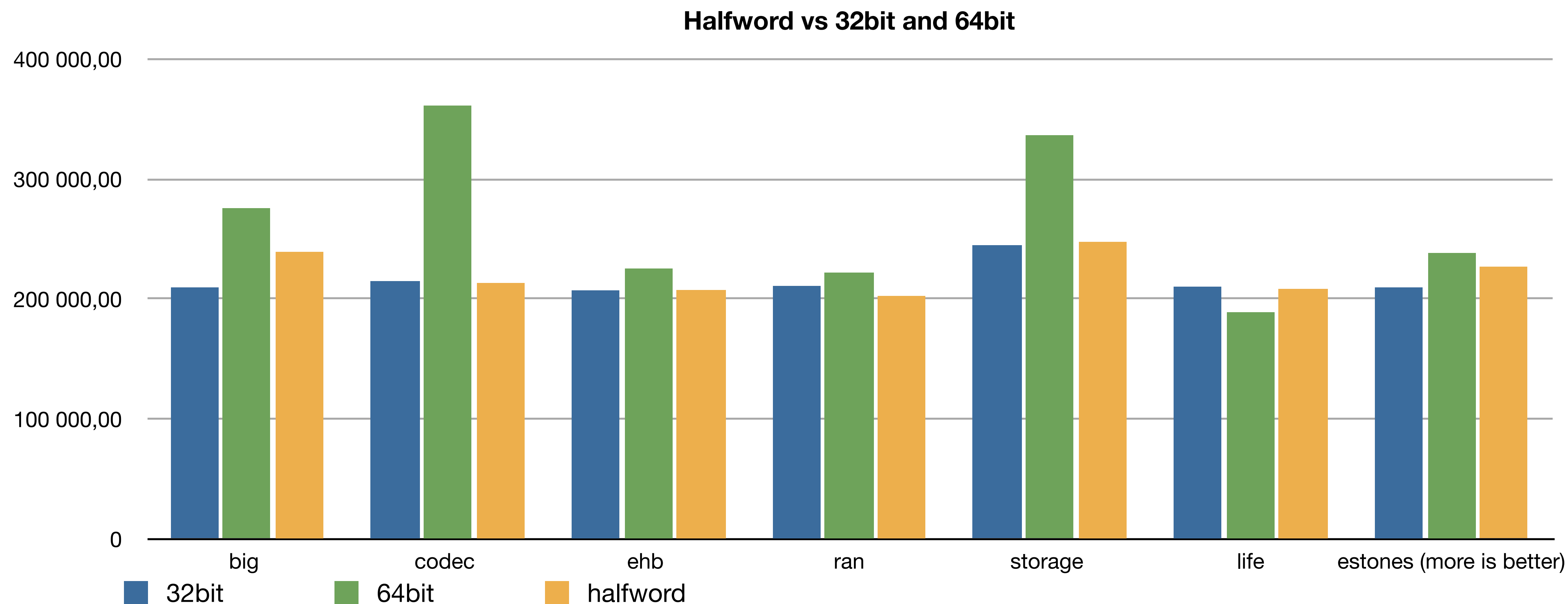
ERICSSON

THE IMPLEMENTATION - ETS

- ETS data is stored with each object in a separately allocated area
- The Erlang term data is rewritten when copied into the table so that pointers are replaced by offsets from the start of the object instead of absolute addresses
- Each ETS object can then be up to 4GB
- The ETS objects (and all metadata) can be stored in “high” memory
- Copying to/from the ETS tables require special copy routines, but it’s more or less as fast as in a regular VM
- Some operations (like matching) are a little trickier, but it could mostly be solved without performance penalties

WHAT'S GAINED?

- Compared to a 32 bit VM (given that we have a 64 bit OS)
  - We can break the 4GB barrier as long as we keep the regular processes below 4GB - huge amounts of ETS and binary data is possible
  - Usually more place for process heaps as most of the "low" 4GBs are free in 64 bit mode
  - Number crunching is slightly faster
  - Drivers etc can be written as 64 bit code
- Compared to a 64 bit VM (given that we can use it...)
  - Faster
  - Smaller memory footprint



WHAT'S LOST?

- Compared to a 32 bit VM
  - The ability to run on 32 bit OS'es
  - The ability to run on anything except Linux (without resorting to pure 64 bit machine)
  - HiPE (for now)
- Compared to a 64 bit VM
  - The possibility to have processes that together (or by themselves) exceed 4GB of combined stack and heap
  - Some number crunching speed
  - The ability to run on anything except Linux
  - HiPE (for now)

ERICSSON

CAN THIS BE USED AT ALL?

- Yes - many applications have nowhere near the heap sizes that requires a pure 64 bit VM
- Yes - many applications have huge amounts of ETS data that really requires a 64 bit address space while they still don't need much space for heaps
- Yes - it exists and has existed since R14B02 and is tested daily, just like any other supported platform
- Yes - many applications run on 64bit Linux
- ...and you can always try, no adaptation of your source code is needed. If your application fit's the requirements, you will gain memory and speed without changing one single line of code

CAN AND WILL THIS BE IMPROVED?

- The halfword virtual machine can certainly be improved:
  - Porting to more platforms - win64, FreeBSD and MacOS for example
  - Spread processes over more 4GB ranges
    - Requires extensive rewrites and some inventing, but is feasible
    - May result in lower performance, but more applications could use it
  - Code could be moved out of the “low” range (fairly easy, but may not be worth it)
  - There may be room for optimizations
- It is supported/maintained and will continue to be so if there’s “enough interest” (i.e. if it’s used...)
- If more applications use it, more developing effort will of course be put into further development...

ERICSSON

SUMMARY

- Pure 64 bit VM's tend to waste a lot of memory and may perform worse than the 32 bit VM's
- By reducing memory footprint, we gain speed as well as memory
- Compacting the heap data may gain speed enough to compensate for the extra cost of not using full size pointers
- Many applications could benefit from using the halfword VM, but few know about it yet
- You could try it as an alternative, both to a 32 bit VM and to a 64 bit one, as long as your OS is a 64 bit Linux
  - Just replace your current installation with the halfword build and give it a try
  - `./configure --enable-halfword-emulator --disable-hipe`

ERICSSON