A staged tag scheme for Erlang*

Mikael Pettersson (mikpe@csd.uu.se)
High-Performance Erlang Group (HiPE)
Computing Science Department
Uppsala University, Sweden

October 26, 2000

Abstract

The runtime systems in Ericsson's implementations of the Erlang programming language, up to and including release R6B, use a simple tag scheme which allows for fast computation of an object's type. However, the tag scheme also restricts all Erlang objects to a 28- or 30-bit address space. This is problematic when Erlang is ported to new systems, and reduces reliability for applications needing large amounts of memory.

This paper describes the development of a new *staged* tag scheme, which was designed to *not* restrict the range of pointers, and thus eliminate the source of the abovementioned problems. Secondary benefits also followed: the staged tag scheme is more flexible, and, perhaps surprisingly, is actually more efficient.

The staged tag scheme has been integrated into Ericsson's Erlang code, and is a standard feature starting with release R7A.

1 Introduction

Erlang is a dynamically-typed programming language. Instead of associating types with variables and functions at compile-time, as statically-typed languages do, dynamically-typed languages associate types with values at runtime. In addition, each primitive operation, such as `+`, `hd`, and `tl`, checks at runtime that it is defined for the types of its actual arguments. Some operations are generic, e.g. `+` which is defined for all combinations of integer and floating-point arguments. Other languages having similar properties include Lisp and Prolog.

Since the type of a value is not statically known and must be retrievable at runtime, the standard solution is to use a *uniform* data representation with embedded type information. A common approach is to represent each value by a machine-word sized *handle*. A small value can be stored directly in its handle (a so-called *immediate*), while a larger value is placed in memory (*boxed*) and referenced via a pointer in its handle.

A *tag* is a runtime meta-value which denotes a type. To enable correct interpretation of handles, each handle includes a tag which describes its type

*This work was conducted in the context of the High-Performance Erlang project (HiPE) at Uppsala University, supported by the Advanced Software Technology competence centre (ASTEC) and Ericsson Utvecklings AB.

and physical layout. A handle containing a tag and a pointer to a boxed object is called a *tagged pointer*. The acts of attaching and removing tags from data is called *tagging* and *untagging*, respectively, and a *tag scheme* is the overall approach taken to represent tags and tagged values in a programming language implementation.

This paper describes the development of a new, flexible, and robust tag scheme for Ericsson's Erlang runtime system.

The existing tag scheme is based on a simple idea: a value is represented by a 32-bit handle, partitioned into a 4-bit type tag and a 28-bit type-dependent data field. A small value is stored directly in the data field, and a large value is placed in memory and referenced via a 28-bit address. This approach is simple and makes it very cheap to compute the type of a value. However, this simplicity comes at a cost: the set of possible types is restricted, and the range of tagged pointers is limited, which causes portability and reliability problems.

The new tag scheme was primarily designed to remove the limitation to the range of tagged pointers. It does this by describing types in *stages*: a small *primary* type tag is stored in the handle's low bits, and remaining type information is stored in the pointed-to object (for boxed values) or in further tag bits in the handle (for immediate values). This is in fact a standard approach which is often used to implement dynamically-typed Lisp-like languages.[1]

Since a staged tag scheme represents some types piecemeal, there is some concern that type checking operations may incur increased runtime overheads compared to a non-staged tag scheme. For instance, an additional indirection is needed for some boxed objects since parts of their types have been moved from the handles to the objects themselves. Dispatching on type is also more expensive with a staged tag scheme. This paper will show that a runtime system may actually *gain* performance by employing a carefully designed and optimised staged tag scheme – the Erlang R7 runtime system gained 5–10% in performance thanks to the new tag scheme. The key is to design a tag scheme in which the most frequent operations can be optimised, even if this entails overheads in less frequent operations.

This paper is not a tutorial on the design and implementation of tag schemes for Erlang or other dynamically-typed languages. The reader is referred to [2] for a comprehensive introduction to the subject. For an introduction to Erlang, see [1]. The source code for the Open Source Erlang systems is available at `www.erlang.org`.

The rest of this paper is organised as follows. Section 2 first describes Erlang R4B and the original tag schemes used by the JAM and BEAM virtual machines. Then a new tag scheme is developed to eliminate their shortcomings. Some problems with the new tag scheme are identified and an improved second version is developed. Section 3 first describes Erlang R6B and the original tag scheme used by the BEAM virtual machine. Then it describes how the second version of the new tag scheme for R4B is adapted for R6B. Finally, Section 4 describes how the new tag scheme was integrated in Erlang R7. The new tag scheme is a standard feature in Ericsson's Erlang system starting with R7A.

[1] For instance, the author's Bifrost Scheme system, developed 1989–1993, used a staged tag scheme similar in spirit to the one developed here.

```
 3         2         1
10987654321098765432109876543210
```
Primary values:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaa0000  FRAME, address:28
iiiiiiiiiiiiiiiiiiiiiiiiiiii0001  FIXNUM, int:28
aaaaaaaaaaaaaaaaaaaaaaaaaaaa0010  BIGNUM, address:28
aaaaaaaaaaaaaaaaaaaaaaaaaaaa0011  FLONUM, address:28
000000000000################0100  ATOM, num:16
nnnnnnnn################cc0101  REF, node:8, num:18, creat:2
nnnnnnnn0000000000########cc0110  PORT, node:8 num:8, creat:2
sss###############nnnnnnnncc0111  PID, serial:3, num:15, node:8, creat:2
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1000  TUPLE, address:28
00000000000000000000000000001001  NIL
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1010  CONS, address:28
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1011  ARITYVAL, arity:28
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1100  MOVED, address:28
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1101  CATCH, address:28
0000s0000000aaaaaaaaaaaaaaaa1110  THING, sign:1, arity:16
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1111  BINARY, address:28
```

Figure 1: OTS4J representation

2 Erlang R4B / 47.4.1

2.1 JAM (OTS4J)

This section describes the data representation used by the JAM virtual machine in Erlang R4B, a.k.a. Erlang 47.4.1. The representation is detailed in Figure 1.

2.1.1 General comments

An Erlang value is encoded a 32-bit word, containing a 4-bit type tag and 28 bits of type-specific data.

A value referring to a memory object is constructed by shifting the address up 4 bits and inserting a type tag in the low 4 bits. This limits the usable address space to $[0, 2^{28} - 1]$.

FRAME and CATCH values are tagged pointers into the JAM stack.

A BIGNUM or FLONUM heap object starts with a header, formatted as a THING value. The header's arity gives the size (number of data words following the header) of the object. For a BIGNUM object, the header's sign bit indicates the sign of the number.

A TUPLE heap object starts with a header, formatted as an ARITYVAL value. The header's arity gives the size of the tuple.

A CONS heap object does not have a header. When the runtime system's generational copying garbage collector [4] forwards objects, it does not separate CONS from non-CONS objects. Therefore, when scanning *tospace* the collector does not know in advance whether the next object has a header or not. The THING tag exists to allow the collector to identify objects containing binary data which should not be scanned. Both THING and ARITYVAL need to exist as uniquely typed meta-values.

```
 3          2          1
10987654321098765432109876543210
```
Primary values:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaa0000  CP0, address:32
aaaaaaaaaaaaaaaaaaaaaaaaaaaa0001  CONS, address:28
aaaaaaaaaaaaaaaaaaaaaaaaaaaa0010  TUPLE, address:28
sss###############nnnnnnnncc0011  PID, serial:3, num:15, node:8, creat:2
aaaaaaaaaaaaaaaaaaaaaaaaaaaa0100  CP4, address:32
nnnnnnnn0000000000########cc0101  PORT, node:8, num:8, creat:2
nnnnnnnn##################cc0110  REF, node:8, num:18, creat:2
000000000000################0111  ATOM, num:16
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1000  CP8, address:32
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1001  FLONUM, address:28
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1010  ARITYVAL, arity:28
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1011  BIGNUM, address:28
00000000000000000000000000011011  NIL
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1100  CP12, address:32
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1100  MOVED, address:28
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1101  CATCH, address:28
0000s0000000aaaaaaaaaaaaaaaa1101  THING, sign:1, arity:16
aaaaaaaaaaaaaaaaaaaaaaaaaaaa1110  BINARY, address:28
iiiiiiiiiiiiiiiiiiiiiiiiiiii1111  FIXNUM, int:28
```
BEAM register-operand encoding:

```
oooooooooooooooooooooooooooo0000  XREG, offset:28
oooooooooooooooooooooooooooo0001  YREG, offset:28
00000000000000000000000000000010  RREG
```

Figure 2: OTS4B representation

When the collector forwards an object, it leaves a forwarding address behind in the first word of the original object. The forwarding address is tagged as MOVED, permitting the collector to identify already forwarded objects.

BINARY objects are referenced via tagged pointers, but are allocated outside of the process' heaps. They are mark-sweep collected and have no tagged header words.

2.2 BEAM (OTS4B)

This section describes the data representation used by the BEAM virtual machine in Erlang R4B. The representation is similar to OTS4J, but differs in details as described below. The representation is detailed in Figure 2.

2.2.1 Continuation Pointers

BEAM stores untagged continuation pointers (pointers to 32-bit aligned words containing pointers to machine code) on the stack. These continuation pointers implicitly use the four CP tags, which are analogues of JAM's FRAME tag.

To make room for the four CP tags, BEAM changes the order of the other tags, and uses special cases and context-sensitive overlays to compress them to the 12 remaining values:

- NIL is a special case of BIGNUM, with address 1.
- Tag 12 is CP12 on the stack and MOVED on the heap.
- Tag 13 is CATCH on the stack and THING on the heap.

2.2.2 BEAM instruction operand encoding

BEAM allows generic operands to virtual-machine instructions to be either atomic Erlang values (FIXNUM, ATOM, or NIL) or virtual-machine registers. In these BEAM instructions, the operand is a single word containing either an atomic Erlang value or a register descriptor. BEAM encodes these cases by redefining tags 0–2 to denote different register classes when found in generic instruction operands. The register tags must therefore be disjoint from the tags of allowable atomic types.

2.2.3 Optimised CONS/TUPLE type tests

BEAM uses certain properties of the tag assignment in order to implement more efficient CONS and TUPLE type tests. An ordinary type test is implemented as follows:

```
#define is_TYPE(x) ((x & 0xF) == tag_TYPE)
```

This requires a mask and a compare, often followed by a conditional jump. BEAM assigns single-bit values to the CONS and TUPLE tags, and no ordinary value has tag zero. Thus, any value which is *not* a CONS must have at least one tag bit set apart from the CONS tag, and similarly for TUPLE. BEAM therefore implements these type tests as follows:

```
#define is_not_cons(x) ((x & (0xF - tag_cons)) != 0)
#define is_cons(x) !is_not_cons(x)
```

This can often be implemented by a mask followed by a conditional jump, without the need for an explicit compare operation.

2.3 Comments on R4B

2.3.1 Address space problems

The usable address space is limited to the $[0, 2^{28}-1]$ range. This causes problems because portable C code (the Erlang runtime systems are written in C) cannot make assumptions about *where* in the virtual address space memory will be allocated. In Unix-like operating systems, `malloc` will use either the `brk` or the `mmap` system call to allocate memory. However, `mmap` often returns memory "high" in the virtual address space, which is incompatible with the tag scheme. Workarounds may be possible on a system-by-system basis, but this is clearly undesirable.[2]

The Erlang runtime systems contain a function `safe_alloc` which allocates "safe" memory. If this function finds that a new memory block was allocated

[2] For instance, the port to Windows NT contains initialisation code which pre-allocates the "high" address range, which forces future allocations to the "safe" range.

outside of the permissible address range, it prints an error message and immediately terminates the runtime system. This can, and in the author's experience sometimes does, occur even though the total amount of *allocated* memory is much less than 2^{28} bytes, perhaps 50–100MB. This is obviously contrary to Erlang's aspiration as a language for building robust systems.

It is also imperative that all allocation sites in the runtime system use `safe_alloc` if tagged pointers to that memory will be constructed, since those pointers will otherwise be destroyed by the tag and untag operations. Unfortunately, in both Erlang R4 and R6 at least one allocation site is not safe, viz. in `fix_alloc` which is an optimised allocator for small fixed-size memory blocks.[3]

2.3.2 Field access overheads

Accessing a field via a tagged pointer requires a separate untagging step, since general-purpose processors do not have "shift down and add constant" instructions or addressing modes. Similarly, to construct a tagged pointer from the object's address requires two operations: an up-shift followed by an add.[4]

2.3.3 Unoptimised fixnum arithmetic

It would be beneficial if FIXNUM could be given an all-zero tag, since that would allow common fixnum operations to be optimised by eliminating the need for explicit tag manipulations [2]. If the FIXNUM tag could be reduced to just two bits, then the SPARC's `tadd` instruction [5] could be used to optimise fixnum arithmetic even further.

2.4 New tag scheme #1 (NTS4 #1)

In this section a new tag scheme developed for Erlang 47.4.1 by the author is described. The primary motivation for developing this representation was to allow the full 32-bit address space to be used, without any limitations imposed by the Erlang runtime system itself. Some secondary benefits also followed, as mentioned below.

Basics

- The primary objective is to allow Erlang objects to be placed anywhere in the 32-bit address space. This implies that pointers must not be tagged by shift operations, since they discard significant address bits. It is reasonable to require all object addresses to be 32-bit aligned, however.
- Due to the 32-bit alignment property, the low two bits of every object's address will be zero. It is therefore possible to store a *primary tag* in the low two bits, without loss of information.
- A two-bit primary tag can express four different cases, of which at least one must be left for immediate values, leaving room for at most three types of tagged pointers. However, JAM has 8 different types of tagged pointers, and BEAM has 10.

[3] This problem was found while debugging the port of the HiPE system to R4B [3].

[4] Alpha and x86 processors can perform this in a single instruction, but only for shifts up to 3. This is intended for indexing integer and floating-point arrays.

Reducing the number of pointer types

- A MOVED value can only occur in the first word of a forwarded heap object, while the collector is running. Note that:
  - For objects with a header, the header is either ARITYVAL/THING or MOVED. Instead of tagging the forwarding pointer as MOVED, it can be given the same tag as the original reference to the object. That is, the header of a TUPLE is either ARITYVAL or a forwarding pointer *tagged as TUPLE*, and similarly for BIGNUM and FLONUM objects.
  - A CONS cell has no header. Instead the CAR can be overwritten with a marker, and the forwarding pointer can be stored in the CDR, tagged as CONS.

  Therefore, the MOVED pointer tag can be eliminated.

- The call and catch frames on the JAM stack are linked together, which means that they can be identified without looking for words tagged as FRAME or CATCH. The JAM therefore does not need these two tags.

  It is also possible to modify the BEAM interpreter to eliminate its need for the CP and CATCH tags (see section 2.4.7).

  However, there are still five types of tagged pointers (CONS, TUPLE, BIGNUM, FLONUM, BINARY), which will not fit in the primary tag.

- Hence it is necessary to fold some of these five cases together, and to add a secondary type tag in the pointed-to object. It would be awkward and expensive to add headers to CONS cells, but the other four types already have header words.

- NTS4 therefore uses two primary tags for tagged pointers, one for CONS cells without headers, and one for BOXED objects (TUPLE, BIGNUM, FLONUM, BINARY) who start with tagged header words.[5] Headers are extended with a subtype field.

Re-encoding the immediate types

- There are two primary tags left, and eight types of immediate values: ATOM, FIXNUM, REF, PORT, PID, NIL, ARITYVAL, and THING. Again, some form of folding is necessary. In NTS4 #1, the all-bits-zero tag is used for FIXNUM, and the last primary tag is used as a first-stage tag for the seven remaining immediate types. The purpose of this assignment is to facilitate an optimised implementation of fixnum arithmetic.

- Three bits would be required to distinguish between the seven remaining immediate types. This would only leave 27 bits for data, although some types need 28 bits.

[5] Since BINARY objects are located outside of the collected heaps, their headers do not contain any type information in OTS4. In NTS4, their headers are changed to start with a THING word.

- A close inspection of Erlang 47.4.1 reveals that some of these seven types actually have less than 28 bits of data: ATOM, PORT, NIL, and THING have unused bits, and ARITYVAL could also be shrunk slightly. REF and PID, however, require all 28 data bits.

- NTS4 #1 therefore uses a new two-bit tag (bits 2 and 3) to distinguish between REF, PID, THING, and other immediates. A further two-bit tag (bits 4 and 5) distinguishes between the remaining cases (ATOM, PORT, NIL, ARITYVAL).

- THING needs a sub-tag to distinguish between BIGNUM, FLONUM, and BINARY objects. It would be possible to use a two-bit sub-tag for these cases, and to format the remaining data bits as in the original representation. However, this would mean that THING and ARITYVAL headers have different number of bits available for arity, and that THING headers have a BIGNUM sign bit in the data part which must be masked out.

- Instead, NTS4 moves the BIGNUM sign bit into the THING sub-tag itself. There are now four cases in a THING sub-tag: FLONUM, BINARY, BIGNUM, and negative BIGNUM. The THING and ARITYVAL data portions are now both 26 bits wide, and can be accessed without masking.

The resulting representation is detailed in Figure 3.

2.4.1 The representation of NIL

NIL is all-bits-one, i.e. -1. This choice is deliberate, since it is often cheaper to create a small-magnitude negative value than a larger-magnitude positive value.

2.4.2 The non-value

The Erlang 47.4.1 runtime system makes an undocumented assumption that all-bits-zero is a non-value, i.e. that no proper Erlang value has the all-bits-zero representation. The runtime system frequently uses zero to indicate error or the absence of a value.

This assumption fails when FIXNUM is given the all-zero tag.[6] NTS4 introduces an explicitly named non-value constant, and encodes it as a FLONUM header with arity 0.

2.4.3 Encoding BEAM instruction operands

As described before, BEAM allows generic instruction operands to be atomic Erlang values or virtual-machine registers. It encodes these cases by using the tags of three non-atomic Erlang types to denote registers. This was not very explicit in the R4B source code, which lead to major difficulties during the development of the new data representation.[7] Once this interdependency had been discovered, the BEAM register encoding was changed to use the CONS primary tag, followed by a two-bit register class tag and a 28-bit offset field.

[6] Identifying and fixing this previously undocumented assumption was a major effort.

[7] In particular, using a non-zero tag for FIXNUM worked, while using the zero tag caused BEAM to crash.

```
 3         2         1
10987654321098765432109876543210
```
Primary values:

`iiiiiiiiiiiiiiiiiiiiiiiiiiiiii00`	FIXNUM, int:30
`aaaaaaaaaaaaaaaaaaaaaaaaaaaaa01`	CONS, address:32
`aaaaaaaaaaaaaaaaaaaaaaaaaaaaa10`	BOXED, address:32
`xxxxxxxxxxxxxxxxxxxxxxxxxxxxtt11`	IMMED1, data:28, tag:2

First-stage immediate values:

`nnnnnnnncc##################0011`	REF, node:8, creat:2, num:18
`nnnnnnnnccsss################0111`	PID, node:8, creat:2, serial:3, num:15
`aaaaaaaaaaaaaaaaaaaaaaaaaaatt1011`	THING, arity:26, tag:2
`xxxxxxxxxxxxxxxxxxxxxxxxxxtt1111`	IMMED2, data:26, tag:2

THING header values:

`aaaaaaaaaaaaaaaaaaaaaaaaaa001011`	FLONUM, arity:26 (always 2)
`aaaaaaaaaaaaaaaaaaaaaaaaaa011011`	BINARY, arity:26 (always 0)
`aaaaaaaaaaaaaaaaaaaaaaaaaa1s1011`	BIGNUM, arity:26, sign:1

Second-stage immediate values:

`nnnnnnnncc################001111`	PORT, node:8, creat:2, num:16
`##########################011111`	ATOM, num:26
`aaaaaaaaaaaaaaaaaaaaaaaaaa101111`	ARITYVAL, arity:26
`11111111111111111111111111111111`	NIL

BEAM register-operand encoding:

`ooooooooooooooooooooooooooo0001`	XREG, offset:28
`ooooooooooooooooooooooooooo0101`	YREG, offset:28
`00000000000000000000000000001001`	RREG

The non-value:

`00000000000000000000000000001011`	THE_NON_VALUE

Figure 3: NTS4 #1 representation

2.4.4 Polymorphic access macros

The property that all values have a type tag in the low four bits is deeply embedded in the original runtime system. In particular, the runtime system makes several assumptions based on this:

- A single `unsigned_val` macro is used for untagging FIXNUM, ATOM, and register descriptor values. Although there is a separate `arityval` macro for accessing ARITYVAL headers, `unsigned_val` is sometimes used for this purpose too.
- A single `ptr_val` macro is used for untagging all tagged pointer types, except BEAM continuation pointers.
- A single `tag_val_def` macro is used for retrieving the 4-bit type tag from any value. This is commonly used in places where data is traversed.

A significant part of the implementation of the new data representation was to replace all uses of the untyped `unsigned_val` and `ptr_val` macros with type-specific ones, i.e. `atom_val`, `tuple_val`, `port_node`, etc. The garbage collector was rewritten to use the new representation directly, which actually simplified it by eliminating many special cases. To avoid having to rewrite non performance-critical code, `tag_val_def` was re-implemented as a compatibility function.

2.4.5 Pointer tag and field access operations

In OTS4, two operations are required to tag a pointer (an up-shift followed by an add). In NTS4, a single operation suffices (an add).

The cost for untagging a pointer is the same in OTS4 (a down-shift) and NTS4 (a subtraction). Note, however, that a pointer is usually untagged so that fields in the pointed-to object can be read. The read is often implemented by a load instruction with a "register plus constant" addressing mode. In NTS4, the tag subtraction can be folded into the addressing mode of the load instruction.

2.4.6 Forwarding pointers

As described before, the MOVED tag is not necessary for representing forwarded objects. To indicate that a BOXED object has been forwarded, its header is overwritten with the object's new location, tagged as a BOXED pointer. To indicate that a CONS cell has been forwarded, its CAR is overwritten with THE_NON_VALUE, and its CDR is overwritten with the cell's new location, tagged as a CONS pointer.

When the collector finds an already-forwarded object, it uses the forwarding address as-is. In the original runtime system, the collector had to change the tag of the forwarding address from MOVED to that of the object's type.

2.4.7 Eliminating the BEAM CP/CATCH tags

NTS4 #1 makes no provisions for the BEAM CP and CATCH tags. BEAM uses both these tags for its implementation of exception handling. In JAM, the catch frames are linked together in reverse history order, and the process control block contains a pointer to the youngest frame. In BEAM, the youngest

handler is found by scanning the stack for a word containing a code pointer tagged CATCH. This word will be a local variable in the stack frame of the function which implements the handler. The bottom-most word in a stack frame contains the saved continuation pointer from the caller, so it has one of the CP tags. After the CATCH word has been found, a reverse scan is performed to find a word tagged CP; the address of this word is the stack pointer to use.

The CP-tagged words are also used by the module-unload check, which needs to find all code pointers embedded in the stack in order to determine if any code is currently executing in the given module.

The CP and CATCH tags can be eliminated by adding structure to the stack, or by adding a meta-data table which associates stack layout information with each return address.[8] However, generating the stack layout table would require non-trivial support from the BEAM compiler or loader.

For simplicity, the implementation instead added structure to the BEAM stack. The BEAM loader and interpreter were changed to add a size field at the bottom of stack frames. The BEAM compiler and interpreter were changed to expand catch frames with a pointer field to link catch frames together, and a field containing the offset from the catch frame to the bottom of the stack frame in which it is located.

2.5 New tag scheme #2 (NTS4 #2)

The NTS4 #1 tag scheme was a success since it eliminated the 28-bit address space limitation, and its implementation also identified and corrected several other problems. The fixnum representation permitted some optimisations, although these were not implemented.

However, the elimination of the BEAM CP and CATCH tags caused undesirable incompatibilities (BEAM code generated from OTS4B does not work in NTS4 #1 due to the catch frame change) and runtime overheads (an additional field must be initialised in each activation record).

This section describes a successor to NTS4 #1, which was designed to eliminate the need to change the BEAM compiler. Unfortunately, it also sacrifices the ability to optimise fixnum arithmetic since the all-bits-zero tag is no longer available for fixnums.

The essential idea behind NTS4 #2 is to reintroduce the CATCH, CP, and FRAME tags. This solves the BEAM-related problems, but requires further changes to make room for the new tags. The new representation is described in Figure 4. Important details are discussed in the following subsections.

2.5.1 All-ones FIXNUM tag

In instructions for binary arithmetic operators, the BEAM interpreter uses a macro `is_both_small(x,y)` to test if two operands are both fixnums. Either the all-bits-zero or the all-bits-one fixnum tag can be used to implement this test efficiently. In NTS4 #2, the all-bits-zero tag is used for CP values, as in OTS4B, so the all-bits-one tag was chosen for FIXNUM.

[8] Only two words are needed: one word containing the size of the stack frame, which allows the stack to be traversed, and one word containing the address of the currently active exception handler, or the null pointer.

```
 3         2         1
10987654321098765432109876543210
```
Primary values:

`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa00`	CP, address:32 (BEAM STACK)
`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa00`	FRAME, address:32 (JAM STACK)
`aaaaaaaaaaaaaaaaaaaaaaaaaaaattt00`	HEADER, arity:27, tag:3 (HEAP)
`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa01`	CONS, address:32
`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa10`	BOXED, address:32
`xxxxxxxxxxxxxxxxxxxxxxxxxxxxtt11`	IMMED1, data:28, tag:2

First-stage immediate values:

`nnnnnnnncc##################0011`	REF, node:8, creat:2, num:18
`nnnnnnnnccsss###############0111`	PID, node:8, creat:2, serial:3, num:15
`xxxxxxxxxxxxxxxxxxxxxxxxxxtt1011`	IMMED2, data:26, tag:2
`iiiiiiiiiiiiiiiiiiiiiiiiiiii1111`	FIXNUM, int:28

Second-stage immediate values:

`nnnnnnnncc################001011`	PORT, node:8, creat:2, num:16
`##########################011011`	ATOM, num:26
`iiiiiiiiiiiiiiiiiiiiiiiiii101011`	CATCH, index:26 (BEAM STACK)
`ooooooooooooooooooooooooo101011`	CATCH, offset:26 (JAM STACK)
`11111111111111111111111111111011`	NIL

Header values:

`aaaaaaaaaaaaaaaaaaaaaaaaaaaa00000`	ARITYVAL, arity:27
`00000000000000000000000001000100`	FLONUM, arity:27 (always 2)
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa01s00`	BIGNUM, arity:27, sign:1
`00000000000000000000000000010000`	BINARY, arity:27 (always 0)

BEAM register-operand encoding:

`oooooooooooooooooooooooooooo0000`	XREG, offset:28
`oooooooooooooooooooooooooooo0001`	YREG, offset:28
`00000000000000000000000000000010`	RREG

The non-value:

`00000000000000000000000000000100`	THE_NON_VALUE (DEBUG)
`00000000000000000000000000000000`	THE_NON_VALUE

Figure 4: NTS4 #2 representation

2.5.2 Immediate CATCH values

NTS4 #2 reintroduces a distinct CATCH tag. However, there is no room among the four primary tags for another type of tagged pointer. Therefore, catch values are encoded as non-pointers in NTS4 #2.

In OTS4J, a catch value is a tagged pointer to a catch frame. In NTS4 #2, the value is instead the *offset* from the start of the process' stack to the catch frame. This offset can be represented in fewer bits than a general pointer. Thus, NTS4 #2 represents it as an immediate type with a 26-bit data field. This change adds very little to the cost of exception handling in JAM[9], and it actually reduces the cost of stack-relocation (to increase the size of the stack) since catch frame links no longer need to be adjusted.

In OTS4B, a catch value is a tagged pointer to the code which implements the exception handler. The key observation here is that there will be many fewer catch handlers than code addresses. Therefore, each handler can be stored in a table, and be represented by its table index.[10] In NTS4 #2, the BEAM module loader adds all catch handlers to a table and rewrites each `catch` instruction to contain its handler's index. There is no additional runtime overhead for entering or leaving a catch expression; the only overhead occurs when an exception is thrown and the CATCH word has been found: an indirection via the table must be performed. Compared to the stack scan, this cost is negligible. When a module is unloaded, all its catch handler entries in the table are deallocated and made available for new modules.

2.5.3 Overlay CP/FRAME and HEADER tags

Previous tag schemes used distinct tags for THING and ARITYVAL. The only exception is OTS4B, which used tag 13 to denote CATCH when found in the stack, and THING when found in the heap.

Since NTS4 #2 adds FIXNUM and CATCH as new subtypes under primary tag 3 (immediate values), a redesign of the representation of the immediate types is necessary. Instead of increasing the number of tag bits or introducing new levels of subtyping, NTS4 #2 changes the ARITYVAL and THING tags to use primary tag 0 instead. The interpretation of tag 0 is now dependent on the location of the value (on the stack it is a CP or FRAME word, on the heap it is a header word), but this causes no problems in the runtime system.

This rearrangement permits FIXNUM and CATCH to be new cases under primary tag 3 with only a minor tag adjustment: since FIXNUM is given the all-bits-one tag, the second-stage immediate types are moved to the tag previously used by THING. NIL is no longer the all-bits-one value, but it is still a small negative value (-5) which should be inexpensive to construct.

2.5.4 HEADER layout

The layout of header words (THING and ARITYVAL) was changed to allow more efficient type tests and case analysis than in NTS4 #1. NTS4 #1 made it easy to implement basic type tests such as `is_arity_value`, `is_thing`, `is_big`,

[9] An additional pointer subtraction is required when creating a catch frame, to compute its offset from the base of the stack.

[10] Björn Gustavsson at Ericsson Utvecklings AB suggested this solution to me.

and `is_binary`. However, it did not permit efficient case analysis since the different cases did not have consecutive values (ARITYVAL was special), causing overheads in the `tag_val_def` emulation function.

In NTS4 #2, the representation has a single HEADER type with a sub-type field and an arity field. The sub-type tags are assigned[11] so that:

- The tags use consecutive numbers from 0 to 4, which simplifies case analysis in the `tag_val_def` emulation function.
- A single bit in the header (bit 2) differentiates between positive and negative bignums, which simplifies `is_big`.
- ARITYVAL is given tag zero, to make the important `is_arity_value` and `is_tuple` type tests as efficient as possible. The `is_thing` type test becomes slightly more expensive (check that primary tag is zero, then that the header's sub-tag is non-zero), but this is a reasonable tradeoff since `is_thing` is very infrequently used.

2.5.5 BEAM register operand encoding

Since FIXNUM no longer occupies primary tag 0, the BEAM register operand encoding was changed to use the OTS4B encoding again. (See section 2.2.2.)

2.5.6 Optimised CONS/BOXED type tests

Section 2.2.3 describes how the OTS4B representation permits efficient tests for the CONS and TUPLE types, since no ordinary Erlang value has an all-zero type tag in OTS4B. Since FIXNUM no longer occupies primary tag 0, the same optimisation can be applied in NTS4 #2, for CONS and BOXED type tests.[12]

2.5.7 The non-value

Since primary tag 0 is no longer used for ordinary Erlang values, the non-value object can be given the all-bits-zero representation again. When the runtime system is compiled for debugging, an illegal form of FLONUM header is used instead. The purpose of this is to prevent code in the runtime system from making unwarranted assumptions about the representation of the non-value.

3 Erlang R6B / 4.9.1

3.1 Erlang R6B original tag scheme (OTS6B)

This section describes the data representation used by the BEAM virtual machine in Erlang R6B / 4.9.1. (The older JAM virtual machine was dropped in this release.) This representation is a successor to OTS4B (see section 2.2), with changes to accommodate an enlarged address space, larger 'reference' values, new kinds of binaries, and a new type for function closures. The representation is detailed in Figure 5.

[11] The assignment is constrained by an unknown problem in the runtime system which prevents assigning tag 0 to BINARY.

[12] It is unfortunate that the CONS/BOXED type test and fixnum arithmetic optimisations are mutually exclusive.

```
 3         2         1
10987654321098765432109876543210
```
Primary values:

`aaaaaaaaaaaaaaaaaaaaaaaaaaaa0000`	CP0, address:32
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa0001`	CONS, address:30
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa0010`	TUPLE, address:30
`sss###############nnnnnnnncc0011`	PID, serial:3, num:15, node:8, creat:2
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa0100`	CP4, address:32
`nnnnnnnn################cc0101`	PORT, node:8, num:18, creat:2
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa0110`	REF, address:30
`00000000####################0111`	ATOM, num:20
`00000000000000000000000000000111`	NIL
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa1000`	CP8, address:32
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa1001`	FLONUM, address:30
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa1010`	ARITYVAL, arity:28
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa1011`	BIGNUM, address:30
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa1100`	CP12, address:32
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa1100`	MOVED, address:30
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa1101`	CATCH, address:30
`0aaaaaaaaaaaaaaaaaaattttttt1101`	THING, arity:19, tag:8
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa1110`	BINARY, address:30
`iiiiiiiiiiiiiiiiiiiiiiiiiiii1111`	FIXNUM, int:28

THING header values:

`0aaaaaaaaaaaaaaaaaaa000000011101`	REFC_BINARY
`0aaaaaaaaaaaaaaaaaaa000000101101`	HEAP_BINARY (NYI)
`0aaaaaaaaaaaaaaaaaaa000000111101`	SUB_BINARY (NYI)
`00000000000000000110000001001101`	FUN_BINARY (arity 6)
`00000000000000000010000001011101`	FLONUM (arity 2)
`0aaaaaaaaaaaaaaaaaaa000001101101`	POSITIVE_BIGNUM
`0aaaaaaaaaaaaaaaaaaa000001111101`	NEGATIVE_BIGNUM
`0aaaaaaaaaaaaaaaaaaa000010001101`	REF

BEAM operand encoding:

`oooooooooooooooooooooooooooo00`	XREG, offset:32
`oooooooooooooooooooooooooooo01`	YREG, offset:32
`00000000000000000000000000000010`	RREG
`xxxxxxxxxxxxxxxxxxxxxxxxxxxxx11`	LITERAL

Figure 5: OTS6B representation

3.1.1 Addresses widened to 30 bits

The R6B runtime system uses the fact that all objects and BEAM instructions are 32-bit aligned. Instead of shifting addresses up four bits when creating tagged pointers, they are shifted only two bits. To untag a tagged pointer, it is shifted down two bits, and then the low two bits are cleared. This extends the usable address space from 28 to 30 bits, but also increases the cost for untagging a tagged pointer.[13]

3.1.2 New types and THING subtyping

R6B adds two new kinds of binaries, and a new function closure type to replace the tuple-based implementation used in previous releases. Since there are no unused primary tags, these new types all use the BINARY primary tag. To distinguish between the four types which use the BINARY tag, THING header words were changed to include a subtype field. Redundant subtypes were also added for flonums, bignums, and references.

Overloading the BINARY tag to also represent function closures unfortunately requires additional tests in the runtime system whenever binaries are accessed.

3.1.3 Reference values enlarged

In R6B, the 'reference' type was widened to incorporate a 96-bit 'number' field. Hence, a reference value is now a tagged pointer to a five-word record, containing a header (THING, subtag REF), a head ('node' and 'creat' fields formatted as an old-style reference), and a 96-bit 'number' field occupying 3 words.

3.1.4 Port number field widened

The 'number' field in ports is 18 bits, as in OTS4. However, instead of a hard-coded upper limit of 2^8 distinct ports at runtime, the limit is set based on the maximum number of file descriptors. In practise, the limit will still be much lower than the 18 bits permitted by the field width.

3.1.5 The representation of NIL

In OTS6B, NIL is encoded as an ATOM with index 0. This does not eliminate the special-casing of NIL in the runtime system (NIL was a special case of BIGNUM in OTS4B), but it is apparently cheaper to special-case atoms than bignums.

3.1.6 BEAM operand encoding shortened

In OTS6B the tag used to distinguish registers from literals in BEAM operands was shortened to 2 bits. This was possible because all the permissible types of literals (FIXNUM, ATOM, NIL) have type tags whose low 2 bits are 1.

The BEAM interpreter uses this to implement faster access to registers. In OTS4B, to access an operand which denoted an X- or Y-register required the following steps: (1) mask out the low 4 bits and do a dispatch, (2) retrieve a

[13] This change was introduced in R5, an internal version not released as Open Source.

byte offset by shifting the operand down 4 bits, (3) add the byte offset to the X- or Y-register array and access the addressed word.

In OTS6B, this becomes: (1) mask out the low 2 bits and do a dispatch, (2) retrieve a byte offset by subtracting the specific register tag, (3) add the byte offset to the register array and access the addressed word. Since step (2) is a subtraction of a constant instead of a shift, steps (2) and (3) can often be folded into the addressing mode of the final load or store instruction.

3.2 New tag scheme (NTS6B)

NTS6B is a straightforward adaptation of NTS4 #2 for Erlang R6B. The new representation is detailed in Figure 6.

3.2.1 Header layout

The header layout is similar to the one in NTS4 #2, with minor changes to accommodate the new types in R6B; in particular, the subtag field was widened from 3 to 4 bits.

Since several types satisfy the `is_binary` test, the subtags are assigned so that a single bit (bit 5) in the header distinguishes between binaries and non-binaries. Since `is_binary` must inspect the header's subtag in the new tag scheme, the subtag assignment was made so that function closures no longer are considered special cases of binaries. This change permitted a cleanup of the runtime system by removing function closure type tests from places where binaries are accessed.

3.2.2 The representation of NIL

As in NTS4 #2, NIL is assigned its own type tag and is represented as a small negative value (-5). This change permitted a cleanup of the runtime system by removing NIL type tests from places where atoms are accessed.

4 Closing the circle: Erlang R7

The staged tag scheme was initially developed for Erlang R4B, during the summer of 1999. At a meeting between the HiPE group and members of Ericsson's Erlang development group in February 2000, the author presented the staged tag scheme (NTS4 #1). The Ericsson representatives showed great interest in the new tag scheme, but also expressed concerns about the incompatible changes made to the BEAM compiler. A solution to the problem was suggested (change CATCH to use an index instead of a pointer).

After the meeting, the author redesigned NTS4 #1 into NTS4 #2, and then ported it to Erlang R6B (NTS6B). At this point, a snapshot of the Erlang R7 development source code was made available to the author, and NTS6B was ported to R7. The new code was then adopted by Ericsson's Erlang development group in April 2000.

During the next month a number of adjustments were made to the tag assignment and access macros to improve performance.[14] The author then conducted

[14] These changes were backported to NTS6B and NTS4 #2 and are included in this paper's description of those tag schemes.

```
 3         2         1
10987654321098765432109876543210
```
Primary values:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa00  CP, address:32 (BEAM STACK)
aaaaaaaaaaaaaaaaaaaaaaaaaatttt00  HEADER, arity:26, tag:4 (HEAP)
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa01  CONS, address:32
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa10  BOXED, address:32
xxxxxxxxxxxxxxxxxxxxxxxxxxxxtt11  IMMED1, data:28, tag:2
```
First-stage immediate values:

```
nnnnnnnnccsss###############0011  PID, node:8, creat:2, serial:3, num:15
nnnnnnnncc#################1011  PORT, node:8, creat:2, num:18
xxxxxxxxxxxxxxxxxxxxxxxxxxtt1011  IMMED2, data:26, tag:2
iiiiiiiiiiiiiiiiiiiiiiiiiiii1111  FIXNUM, int:28
```
Second-stage immediate values:

```
##########################001011  ATOM, num:26
iiiiiiiiiiiiiiiiiiiiiiiiiii011011  CATCH, index:26 (BEAM STACK)
11111111111111111111111111111011  NIL
```
Header values:

```
aaaaaaaaaaaaaaaaaaaaaaaaaa000000  ARITYVAL, arity:26
aaaaaaaaaaaaaaaaaaaaaaaaaa000100  FLONUM, arity:26 (always 2)
aaaaaaaaaaaaaaaaaaaaaaaaaa001s00  BIGNUM, arity:26, sign:1
aaaaaaaaaaaaaaaaaaaaaaaaaa010000  REF, arity:26
aaaaaaaaaaaaaaaaaaaaaaaaaa010100  FUN, arity:26 (always 6)
aaaaaaaaaaaaaaaaaaaaaaaaaa100000  REFC_BINARY, arity:26
aaaaaaaaaaaaaaaaaaaaaaaaaa100100  HEAP_BINARY, arity:26
aaaaaaaaaaaaaaaaaaaaaaaaaa101000  SUB_BINARY, arity:26
```
BEAM operand encoding: (unchanged)

```
oooooooooooooooooooooooooooo00  XREG, offset:32
oooooooooooooooooooooooooooo01  YREG, offset:32
00000000000000000000000000000010  RREG
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx11  LITERAL
```
The non-value:

```
00000000000000000000000000000100  THE_NON_VALUE (DEBUG)
00000000000000000000000000000000  THE_NON_VALUE
```

Figure 6: NTS6B representation

extensive profiling on the runtime system, which revealed that the `tag_val_def` emulation caused significant overhead in the term comparison operators `eq` and `cmp`. These where rewritten and optimised for the most frequent cases. Björn Gustavsson at Ericsson Utvecklings AB rewrote the term copy routines to use the new tag scheme instead of accessing data via the `tag_val_def` emulation.

The combination of the new tag scheme and the optimisations described above has actually resulted in a noticeable performance improvement compared to the original Erlang R6B/R7 runtime system. On the author's machine, the ESTONE[15] result increased from 18000 to 19800, indicating a 10% improvement to overall system performance. This shows that a staged tag scheme can be introduced to reduce portability problems and improve flexibility and reliability, without sacrificing performance.

References

[1] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

[2] David Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, University of Arizona, Department of Computer Science, October 1993.

[3] Erik Johansson, Mikael Pettersson, and Konstantinos Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM, September 2000.

[4] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic memory management*. John Wiley & Sons, 1996.

[5] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. Prentice-Hall, 1994.

[15] ESTONE is a synthetic benchmark for Erlang implementations. See `www.erlang.org`.