

HotSpot Command-Line Flags Overhaul - Design Proposal for JDK-8236988 - WITHDRAWN

Note: this proposal has been withdrawn. It has been superceded by [JDK-8243208](#)

Overview

This is a design document for [JDK-8236988 - Modular Design for JVM Flags](#)

- Lots of files have changed. Please read this document first before heading over to the webreview
- Preliminary webreview: http://cr.openjdk.java.net/~iklam/jdk15/vm_flags_overhaul.008/

Goals

- Improve the implementation of JVM command-line flags (such as `-XX:+UseCompressedOops`)
- Break up monolithic `globals.hpp`
 - No need to include all flags for GC, C1, C2, JVMCI,
 - Before:
 - `#include "runtime/globals.hpp" + globals_shared.hpp` = declare all possible flags
 - After:
 - If you just need flags for ZGC:
`#include "gc/z/z_globals.hpp"`
 - Future: subdivide `globals.hpp` into individual components:
 - (Not in this patch, but possible)
 - `#include "memory/metaspaceshared_globals.hpp" // just flags related to CDS`
- Remove complex macros for flags manipulation
 - Example: `jvmFlags.cpp`, `globals_extension.hpp`, `jvmFlagRangeList.cpp`
- Hotspot command line switches should have multiple type attributes
 - See [JDK-7123237](#)
 - E.g., one can have a "manageable" switch and an "experimental" switch, but not a "manageable_experimental" switch.
- Templatize duplicated flag processing code:
 - E.g., `.jvmFlags.cpp`
- Speed up range/constraint checking for flags
 - Avoid linear search for every `-XX:NumericFlag=value` argument in the command-line (`jvmFlagRangeList.cpp`)

Proposal

(Design contribution by Erik Österlund, Stefan Karlsson, Coleen Phillimore)

NOTE:

- Most of the current patch is generated by a script, so it's easy to change the design
- Summary of changes: 14333 lines changed: 6443 ins; 5024 del; 2866 mod; 34730 unchg

Before

- Flags are specified via macros like `product()`, `develop()`, `diagnostic()`, etc.

```

product(intx, MaxLoopPad, (OptoLoopAlignment-1),      \
        "Align a loop if padding size in bytes is less or equal to this " \
        "value")                                     \
        range(0, max_jint)                           \
\
develop(intx, OptoPrologueNops, 0,                  \
        "Insert this many extra nop instructions "   \
        "in the prologue of every nmethod")         \
        range(0, 128)                                \
\
notproduct(bool, VerifyGraphEdges, false,          \
        "Verify Bi-directional Edges")              \
\
product_pd(intx, InteriorEntryAlignment,           \
        "Code alignment for interior entry points " \
        "in generated code (in bytes)")             \
        constraint(InteriorEntryAlignmentConstraintFunc, AfterErgo) \
\
diagnostic(bool, StressLCM, false,                 \
        "Randomize instruction scheduling in LCM")    \

```

After

- Flags are specified using a pair of files `xxx_globals.hpp`, `xxx_globals.cpp`
- The HPP file is the "declaration" of the flags. It has a description of all the features for each flag.
 - HotSpot developers should consult the HPP for information about the flags (same as before).
- Example: [c2_globals.hpp](#):
 - Syntax of the `XXX_FLAG` macros: see the top of [globals.hpp](#).
 - Implementation of these macros: see [jvmFlags.hpp](#) (around line 537)

```

// PRODUCT_FLAG -- always settable
// DEVELOP_FLAG -- settable only during development and are constant in the PRODUCT version
// NOTPROD_FLAG -- settable only during development and are *not* declared in the PRODUCT version

PRODUCT_FLAG(intx,      MaxLoopPad, (OptoLoopAlignment-1), JVMFlag::RANGE,
        "Align a loop if padding size in bytes is less or equal to this "
        "value");
    FLAG_RANGE(
        MaxLoopPad, 0, max_jint);

DEVELOP_FLAG(intx,      OptoPrologueNops, 0, JVMFlag::RANGE,
        "Insert this many extra nop instructions "
        "in the prologue of every nmethod");
    FLAG_RANGE(
        OptoPrologueNops, 0, 128);

NOTPROD_FLAG(bool,      VerifyGraphEdges, false, JVMFlag::DEFAULT,
        "Verify Bi-directional Edges");

PRODUCT_FLAG_PD(intx,   InteriorEntryAlignment, JVMFlag::CONSTRAINT,
        "Code alignment for interior entry points "
        "in generated code (in bytes)");
    FLAG_CONSTRAINT(
        InteriorEntryAlignment, (void*)InteriorEntryAlignmentConstraintFunc, JVMFlag::AfterErgo);

// NOTE: diagnostic macros are specified by setting the JVMFlag::DIAGNOSTIC bit in the
// flag's attr (see globals.hpp for details).
//
// This allows more flexible specification of attributes. E.g., manageable-experimental flags
// in JDK-7123237 can be specified using JVMFlag::MANAGEABLE | JVMFlag::EXPERIMENTAL,
PRODUCT_FLAG(bool,      StressLCM, false, JVMFlag::DIAGNOSTIC,
        "Randomize instruction scheduling in LCM");

```

Macros in the HPP files are expanded to

```

// The is the main API of this flag for HotSpot code. E.g.,
//     if (StressLCM) {...}
extern "C" bool StressLCM;

// This structure has the meta-information about this flag. E.g., HotSpot code can use it
// to dynamically query the type of this flag.
extern ProductFlag<FLAG_TYPE_StressLCM> FLAG_StressLCM;

// The following inline functions pass meta-information about this flag to the CPP file
inline bool FLAG_DEFVAL_StressLCM() { return false; }
typedef bool FLAG_TYPE_StressLCM;
inline JVMFlag::FlagType FLAG_TYPE_NAME_StressLCM() { return JVMFlag::TYPE_bool; }
inline int FLAG_ATTR_StressLCM() { return JVMFlag::DIAGNOSTIC; }
inline const char* FLAG_DOCS_StressLCM() { return "Randomize instruction scheduling in LCM"; };

<... the expansion of some macros are omitted for clarity>

// Range information are expanded like this:
inline int FLAG_ATTR_MaxLoopPad() { return JVMFlag::RANGE; }
inline FLAG_TYPE_MaxLoopPad FLAG_MIN_MaxLoopPad() { return 0; }
inline FLAG_TYPE_MaxLoopPad FLAG_MAX_MaxLoopPad() { return max_jint; }

```

The main goal of the inline functions is to avoid duplicating the information between the HPP and CPP files, so the CPP files contains just boilerplate code.

- [c2_globals.cpp](#)

```

DEFN_PRODUCT_FLAG(StressLCM);
DEFN_PRODUCT_FLAG(StressGCM);
DEFN_PRODUCT_FLAG(MaxLoopPad); DEFN_PRODUCT_RANGE(MaxLoopPad);
DEFN_PRODUCT_FLAG_PD(InteriorEntryAlignment); DEFN_PRODUCT_CONSTRAINT(InteriorEntryAlignment);

```

Macros in the CPP files are expanded to

```

// Definition of the flag itself
FLAG_TYPE_StressLCM StressLCM = FLAG_DEFVAL_StressLCM();

// Definition of the flag's meta information
ProductFlag<FLAG_TYPE_StressLCM> FLAG_StressLCM(
    FLAG_TYPE_NAME_StressLCM(),
    "StressLCM",
    (FLAG_ATTR_StressLCM() | JVMFlag::C2),
    &StressLCM, FLAG_DOCS_StressLCM());

<... the expansion of some macros are omitted for clarity>

// Range check is implemented like this
JVMFlagRange<FLAG_TYPE_MaxLoopPad> FLAG_RANGE_MaxLoopPad(
    &FLAG_MaxLoopPad, FLAG_MIN_MaxLoopPad(), FLAG_MAX_MaxLoopPad());

// Inside the constructor of FLAG_RANGE_MaxLoopPad, it essentially registers
// itself as the range spec for FLAG_MaxLoopPad,
FLAG_MaxLoopPad._range = &FLAG_RANGE_MaxLoopPad;

```

The consistency between the .hpp/.cpp files are **mostly** statically enforced at compile time. E.g.,

- It's impossible to set a different default or range than as specified in the HPP file.
- It's impossible to define StressLCM as a different type than declared in the HPP file.

However, it's possible to forget to put `DEFN_PRODUCT_RANGE(MaxLoopPad)` in [c2_globals.cpp](#). I can't think of a way to statically check for that, so I added a runtime check (debug builds only). See [JVMFlag::validate_flags\(\)](#).

Meta-information of the flags

As seen above, meta-information about each flag XYZ is stored in the structure `FLAG_XYZ`. This structure can be used directly to access the flag. E.g.,

```
intx new_val = .....;
FLAG_MaxLoopPad.check_range(&new_val, verbose);
```

As a result, we no longer need to do a linear search for range/constraint checking.

Iterating over all flags:

The FLAG_XXXs are constructed using C++ global constructors, and are appended to a global list. This list can be used for iterating over all the flags (e.g., for implementing `-XX:+PrintFlagsFinal`).

```
struct JVMFlag {
    static JVMFlag* _head;
    JVMFlag* _next;
    ...
};

template <typename T>
class ProductFlag : /* a subclass of JVMFlag */ {
    ProductFlagNone(...) {
        /*effectively*/
        this->_next = _head;    // executed as C++ global constructor
        _head = this;
    };
};

#define JVMFLAG_FOR_EACH(f) \
    for (f = JVMFlag::_head(); f != NULL; f = f->next())
```

Alternatives

- Declare flags in template files with custom syntax
 - flag type = bool, name = UseCompressedOops, default = false, help = "Use 32-bit object references in 64-bit VM.", type = product | lp64;
 - Or XML, or
- Auto-generation of header files from templates
- Pros:
 - Can generate more compact code than current proposal (see below)
 - Flags can be pre-sorted, hashed, etc
- Cons:
 - More complex build ([nowhere as bad as .ad files](#), but affects all developers, not just JIT compiler geeks)
 - Not understood by IDE tools