Costs and Benefits of Non-conformity

Hans-J. Boehm
HP Labs

© 2004 Hewlett-Packard Development Company, L.P.
The information contained herein is subject to change without notice

JVMs vs. Platform ABIs

Runtime conventions used by generated code of most JVMs deviate from C/C++ ABI:

GC-related requirements

- Pointers precisely identifiable
  - In stack, registers.
    - At “safe points”.
  - In heap objects.
  - In static data.
  - Derived pointer restrictions.
- Object headers.

Other common changes

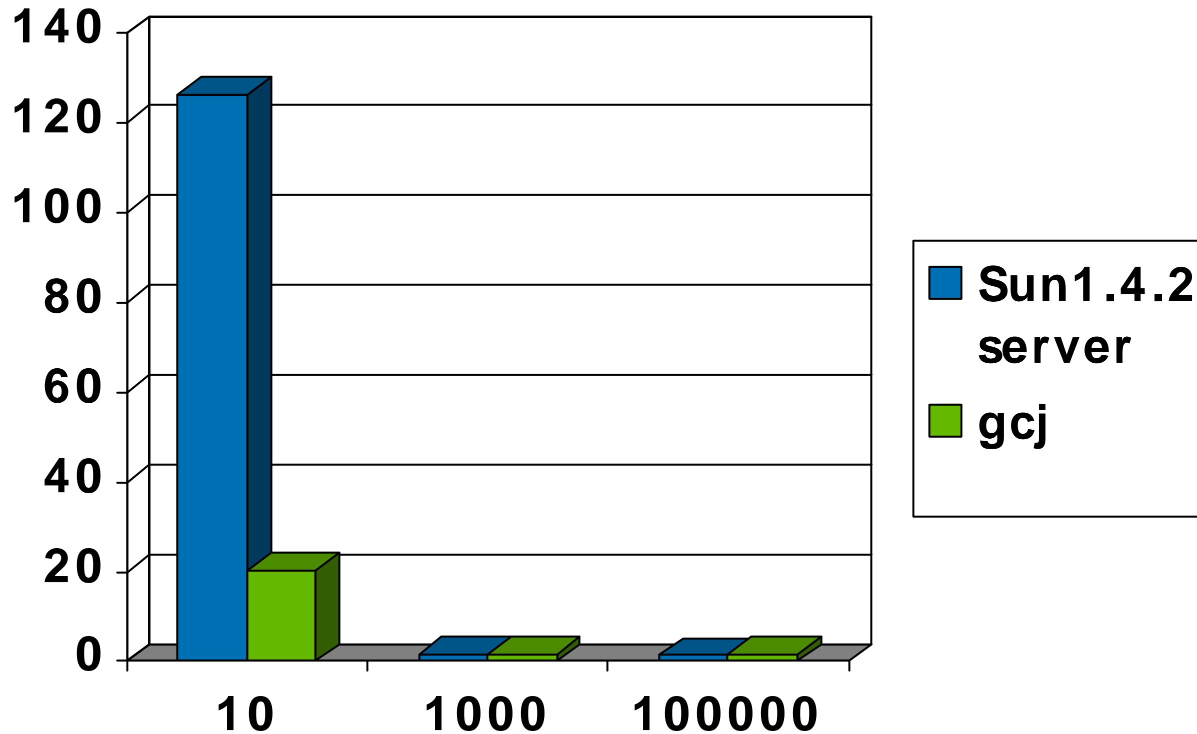
- Different threading system (becoming rare).
- Thread pointer, allocation pointer in register.
- Conventions for calling interpreted code.
- Exception, unwind support (becoming universal).

The costs of varying runtime conventions

- *Native* or *foreign* calls are not just function calls.
  - Platform-provided tools are less useful.
  - User may have to add an explicit translation layer (JNI).
  - System may add translation layer, but semantics can be subtle (CLI?)
  - Significant performance impact, sometimes as a result of native calls are hidden in runtime, or lack of access to platform-optimized libraries.
  - Performance impact can be surprising.

Performance surprise example:

- Time to copy a total of 1 billion `ints` using System.arrayCopy (1.4GHz Itanium2), varying array size:



This Talk

- We look at these runtime differences.
  - Are they still well motivated?
  - Worth the cost?
- Get some insight from gcj.
  - Static Java compiler & interpretive VM.
  - Somewhat bridges the gap.
- This talk is intended to be
  - Half-baked.
    - You probably know more about some of this.
  - Controversial.
- Start with non-GC issues …

Threading system

- Some JVMs have their own thread implementations (Sun classic, Jikes RVM(?), NaturalBridge).
  - Getting interoperability right is tricky.
    - Particularly with thread-hot native libraries.
  - Reasons given include
    - Better performance.
    - Support for more threads.
  - Better platform thread libraries largely invalidate these.
    - And some were based on unfair comparisons anyway.
      - E.g. context switches are faster without processor affinity.
  - I predict these will go away:
    - Fix platform threads instead.

Thread pointer in reserved register

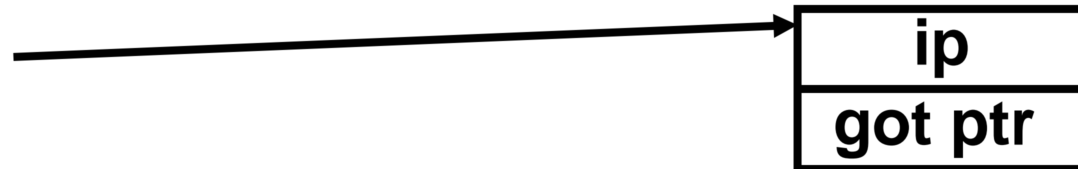
- Important (only?) for lock performance.
- Need fast access to a thread identifier.
  - Need to save it in lock to check for recursive entry.
- Most platform ABIs already have that, or are acquiring it.
  - Except that on x86 & x86-64(?) it requires a load.

Allocation pointer in register

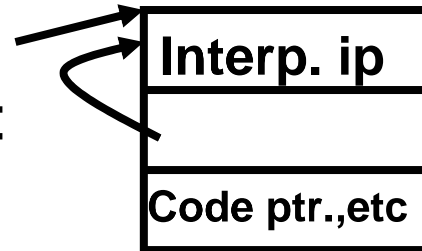
- Important for allocation performance?
- Saves a thread-local variable update per allocation.
- But this should just be a load and a store based on thread pointer register (even on X86).
- Less overhead than free-list vs. pointer bump allocator?
  - < 1% ? [Blackburn, Cheng, McKinley]
  - And can often be cached.
- But cheap to save and restore anyway.

Interpreted code

- Gcj makes interpreted function look like compiled C function. Caller doesn't care.
- Details are platform dependent (libffi). On Itanium:
  - C function:



  - Interpreted function:



- Some architectures need runtime-generated code.

Object headers

- Java objects tend to have more header information than C++ objects.
  - Lock, hash code, etc.
  - Not a real compatibility issue:
    - View as comp.lang.Object fields
  - Now avoided by gcj on most platforms:
    - Only header word is method pointer.
    - 2-3% performance win(?) due to reduced object size, but
    - Objects are never moved.
      - Address can be used as default **`hashCode`**.
    - Locks in fixed size hash table.
      - Entry: "thin lock" plus chain of inflated locks.
      - Current implementation is a bit slow, but
        - Compiler could help more.
        - Still significantly limited by CAS performance.

GC information

- JVMs typically require pointer location information for
  - Objects in GC heap.
  - Static data.
  - Function frames, for call sites and safe-points.
    - Safe-points must be reachable in bounded time.
- Not required by C/C++ ABI.
  - Hard to generate even with C compiler cooperation.
    - Worst problem: Pointer-containing unions & pointers to fields.
- Serious interoperability problem:
  - Need to prevent C/C++ frames from containing GC pointers.

Alternative 1

- Completely Conservative GC
  - No pointer location information.
  - Anything that might be a pointer is considered to be a pointer.
  - Requires minimal C compiler cooperation (at least in theory).
- Costs:
  - May retain more memory.
    - Up to 60% in [HirzelDiwanHenkel02]), mostly liveness info.
    - Unbounded in a few cases, mostly in dense address spaces.
  - No compaction.
    - Free list allocation.
  - Large root set.
  - Pointer verification overhead (if any!)

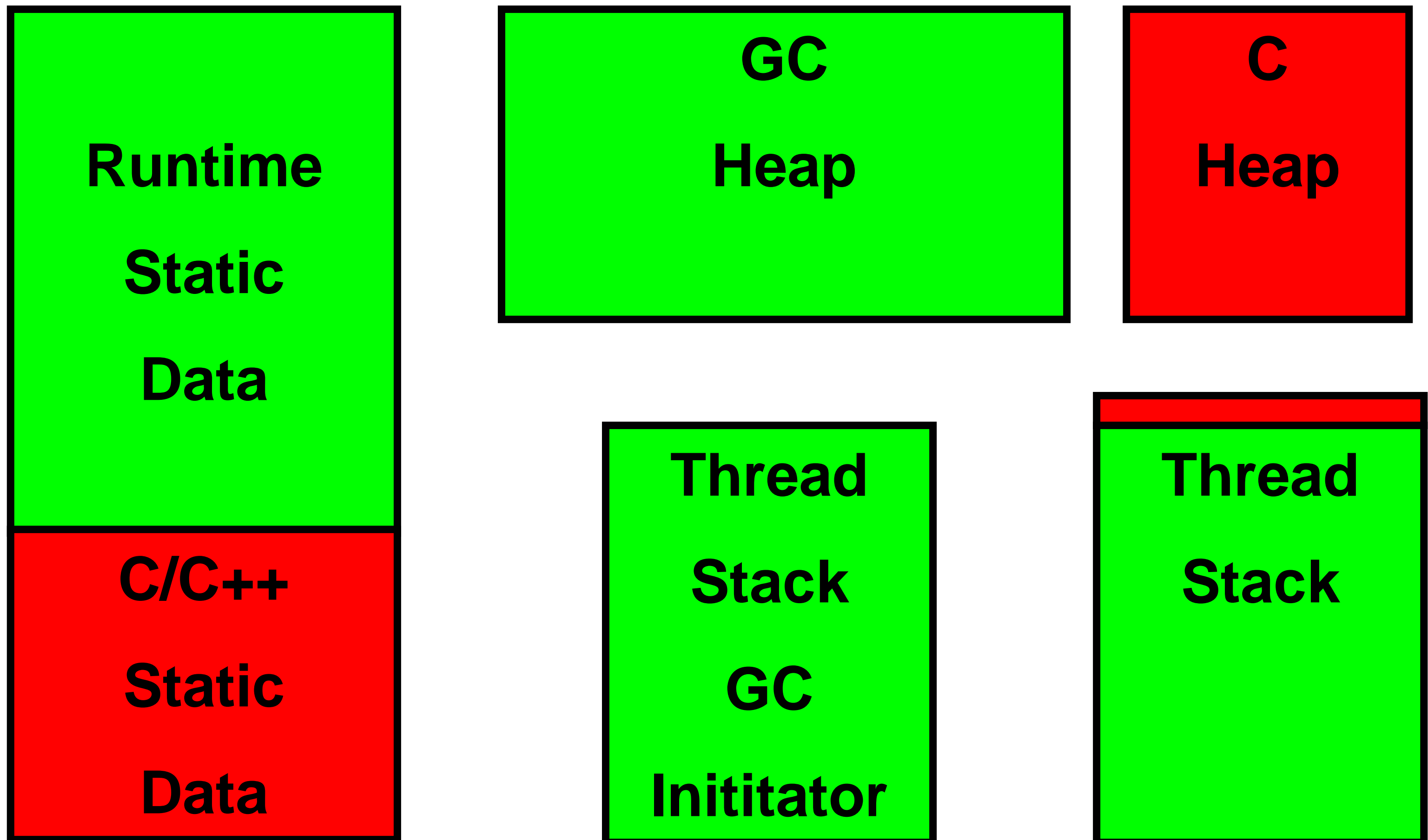
Alternative 2

- Conservative GC, except for Java/managed heap.
  - Type information for Java heap objects is easy to come by.
  - Type information identifies *non-pointers*.
  - Gcj, Mono.
- Costs:
  - May retain more memory.
    - Up to 60% in [HirzelDiwanHenkel02]. (No effect on their worst case.)
    - Unbounded in rare, detectable cases (POPL02, bounded C heap).
    - Worst bounded case is still not good.
  - No compaction(?)
    - Free list allocation.
  - Minimal root size for gcj on IA64: ~4.5MB
    - But that’s silly (exception tables …)

Alternative 3

- [Like Barabash et al, JVM 2001, but less conservative.]
- Optional pointer location information:
  - For static data.
  - For garbage collected heap.
  - For function frames and registers.
- For type-safe (Java, CLI) code:
  - Observe that rapidly changing pointer location information is:
    - Expensive to provide.
    - Unnecessary for preventing long term leaks.
  - Generate pointer info for:
    - Garbage collected heap.
    - Static data (if any).
    - Partial information for function frames and registers.
      - Accurate at call sites.
      - Accurate for non-pointers that are untouched in unbounded loops.
    - Long-lived parts of run-time system.

Alternative 3 illustrated



Alternative 3 vs. Type Accurate

- Benefits
  - No intercallability issues.
  - No safe points.
    - Minimal derived pointer restrictions.
    - Full optimization for small loops.
  - Debug pointer information by binary search.
- Costs
  - For 100% Java/managed code, may retain memory:
    - Long term retention with “probability zero”.
    - Usable worst-case bounds if you need them & can solve other problems.
  - For “native” code:
    - Conservative pointer scanning and/or GC by default.
  - Need at least object pinning in GC. (Costs?)

Conclusions

- Gain from incompatibility with C/C++ ABI seems
  - Minimal in many cases.
  - Unknown in the rest.