Cycles to Recycle: Garbage Collection on the IA-64

Richard L. Hudson
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052-8119
Rick.Hudson@intel.com

J. Eliot B. Moss
Dept. of Computer Science
Univ. of Massachusetts
Amherst, MA 01003-4610
moss@cs.umass.edu

Sreenivas Subramoney
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052-8119
Sreenivas.Subramoney@intel.com

Weldon Washburn
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052-8119
Weldon.Washburn@intel.com

ABSTRACT

The IA-64, Intel's 64-bit instruction set architecture, exhibits a number of interesting architectural features. Here we consider those features as they relate to supporting garbage collection (GC). We aim to assist GC and compiler implementors by describing how one may exploit features of the IA-64. Along the way, we record some previously unpublished object scanning techniques, and offer novel ones for object allocation (suggesting some simple operating system support that would simplify it) and the Java "`jsr` problem". We also discuss ordering of memory accesses and how the IA-64 can achieve publication safety efficiently. While our focus is not on any particular GC implementation or programming language, we draw on our experience designing and implementing GC for the Intel Java Virtual Machine for the IA-64.

1. INTRODUCTION

Intel's new 64-bit instruction set architecture (ISA), the IA-64, introduces a number of interesting architectural features. We have been involved in designing and implementing the memory management and garbage collection (GC) portions of Intel's Java (TM)[1] Virtual Machine (JVM) for the IA-64, and have thereby gained experience in how the IA-64's features relate to GC. We hope this paper will help other GC implementors as they tackle designing, implementing, or porting for the IA-64. We have implemented on early engineering and sample versions of the Itanium (TM)[2] processor hardware most of the techniques we describe, and the current system runs well-known benchmarks, ones modeling servers as well as clients.

We proceed by first describing the features of the IA-64 that we believe to be relevant for GC and memory management. Note that for the most part we keep the discussion at the level of the instruction set architecture, rather than considering any particular implementation of it, such as the Itanium processor. We next point out the few IA-64 features that appear to be *most* significant with respect to GC. Given the background of IA-64 features, we consider a number of topics related to GC and memory management, and for each topic consider implications of the IA-64 feature set for the implementation of that topic. We close with further discussion of the most significant features in light of the more detailed GC implementation topics.

2. FEATURES OF THE IA-64 ISA

Here is a description of the aspects of the IA-64 relevant to memory management. A reference work is available for the interested reader [9].

Data types: The IA-64 directly manipulates 8, 16, 32, and 64 bit signed and unsigned integer quantities, as well as floating point numbers, etc.

General registers (GR): There are 128 general purpose 64-bit registers. GR0 is hard-wired to the value 0. There is a separate file of 128 floating point registers, and up to 128 more special purpose ("application") registers.

Predicate registers (PR): Rather than having a single set of "condition code" registers, the IA-64 has 64 1-bit predicate registers; PR0 is hard-wired to 1 (true). Comparisons and other tests generally produce results into two named predicate registers. Every instruction has a 6-bit predicate field, and is executed only if the corresponding predicate register contains 1.[3] This is called *predication*, and it allows short if-then and if-then-else sequences to execute efficiently without conditional branches.

Address space: Addresses are 64 bits and can address individual 8-bit bytes, as well as 16, 32, and 64 bit integer quantities, and several floating point sizes up to 128 bits. Accesses should be aligned for best performance. The IA-64 supports a comprehensive variety of paging and protection features, including multiple pages sizes from 4Kb to 256Mb.

Register stack: The general registers are divided into a *static* subset (GRs 0–31) and a *stacked* subset (32–127). The stacked registers operate such that each stack frame in a program's call stack has up to 96 GRs. A frame allocation instruction, generally executed near the beginning of each procedure, indicates the number of registers to use for the procedure's local variables and for its arguments to procedures it calls. This is similar to the "register window" mechanism of some other processors, but operates with finer granularity. The loading/storing of overflowing/underflowing registers is normally carried out by a hardware Register Stack Engine rather than traps to software fault handlers. Only the GRs are stacked (not the PRs, etc.), and only the first 32 GRs are static, so only they can be used for dedicated purposes.

Addressing modes: Computation on the IA-64 is mostly register-to-register, and one uses explicit loads and stores to access memory (but with automated storing and loading of some registers via the register stack mechanism). The only memory addressing mode is *register*

[1] Java is a trademark of Sun Microsystems, Inc.

[2] Itanium is a trademark of Intel Corporation.

[3] Actually, a *few* instructions cannot be predicated, i.e., execution of those instructions ignores the predicate register's value.

ISMM 2000 Minneapolis, MN USA
Copyright 2000 ACM 0-89791-88-6/97/05 ..$5.00

indirect: there is no *base register plus offset* or *base register plus index register* form, and all address arithmetic is explicit.[4] The architecture has the ability to add a 14-bit immediate value, to add a 22-bit immediate value (with restrictions on the source register), and to load a 64-bit immediate (using two instruction slots), with a single instruction.

Register conventions: Software conventions further restrict the number of registers available for dedicated use. The available general registers are GR4–GR7 (preserved, i.e., callee-save) and GR14–GR31 (scratch, i.e., caller-save). Of the predicate registers, PR1–PR5 are preserved and PR6–PR15 are scratch. Branch register BR0 receives return addresses on calls, while BR1–BR5 are preserved and BR6–BR7 are scratch.

Instruction format: Instructions are grouped into *bundles*. Each 128-bit bundle contains 3 41-bit instruction slots plus 5 bits of format information giving the bundle's *template*, namely what kind of instruction is in each slot. Not all combinations of instructions are allowed in a single bundle (giving rise to some interesting instruction scheduling issues). The template bits also indicate *instruction group* boundaries where a later instruction may have a resource dependence on an earlier instruction (e.g., consume its result). Instruction groups may be arbitrarily long, crossing bundle boundaries, or as short as one instruction. The essence of the model is that instructions in the same group might execute concurrently, exploiting whatever degree of instruction-level parallelism is offered by the processor implementation. Branches are always to the first instruction of a bundle, but a taken branch does not execute the remaining instructions of its own bundle.

Integer instructions available: Integer/logical register to register instructions include: add, subtract, and, and with complement, or, exclusive or, shift left and shift right (by fixed or variable count, signed or unsigned for shift right), shift left (by 1, 2, 3, or 4 bits) and add, compare (less than, less than unsigned, equal, not equal)[5] with various ways of combining the result into two predicate registers, test bit (fixed bit position), shift right pair by a fixed count (which can implement rotation), bit field extraction and insertion (fixed position and size), population count (number of 1 bits in a register), and compute zero index (the position of the first all 0 8-bit byte or 16-bit word, from either the left or the right end of a 64-bit register). The floating point unit implements fixed point multiply and reciprocal approximation (i.e., the integer unit does not support multiply or divide). The architecture supports various moves between registers of the different kinds, etc.

Memory—Control speculation: Control speculation is the speculative execution of instructions that may not be reached in the true flow of control (e.g., hoisting code up out of an if-then-else). The IA-64 supports *speculative loads*, which do not cause a fault if they encounter a problem. Rather, the register target of the load is specially marked. GRs are marked with a 65th bit called the NaT (Not a Thing) bit, and FRs with a special IEEE floating point value, NaTVal. Most computational instructions set the NaT bit of their target registers if the NaT of any input register is 1. If an instruction sets something other than a GR or FR and has a NaT input, then the instruction faults. Generally, one checks for NaTs explicitly with a special conditional branch instruction. It would normally branch to code that would execute the load(s) and following computation, but non-speculatively. The register stack mechanism and register spill/fill instructions handles NaTs naturally.

Memory—Data speculation: A compiler cannot always prove that a given load and store access different memory locations, yet it may be always, or almost always, true that a later load does not access the same location as an earlier store. Data speculation consists in moving such a load instruction earlier than the store (perhaps to reduce stalls while waiting for the load's results from memory), but somehow checking that the store did not in fact update the location read by the *advanced load*. The IA-64 supports data speculation with advanced load instructions, an Advanced Load Address Table (ALAT), and advanced load checks. The advanced load instruction enters its load address into the ALAT, which has some small fixed size. All stores check the ALAT, and if they find a matching entry, they *invalidate* it. The advanced load check, which is generally inserted where the original load instruction was (i.e., before it was advanced), verifies that the advanced load's entry is still in the ALAT. One kind of check simply re-executes the load if there is no matching ALAT entry; another kind branches to recovery code, and can re-load and re-do any additional calculations depending on the loaded value.

One may combine control and data speculation using speculative advanced load instructions.

Memory—Hierarchy Control: In the IA-64 there are three ways to control memory hierarchy actions: locality hints, explicit prefetching, and implicit prefetching. Locality hints indicate whether a load, store, or line fetch instruction's data is (a) temporally local (keep in level 1 cache), (b) non-temporal at level 1 (but keep in level 2), (c) non-temporal at level 2 (and non-temporal at level 1, so keep in level 3), or (d) non-temporal at all cache levels. The line fetch instruction has faulting and non-faulting versions, and with the hints can load data into specific levels of cache. There is a separate hint to indicate that a loaded value is likely to be updated (i.e., in a multiprocessor one should acquire exclusive access to the cache line, etc.).

Memory—Atomic Update: The IA-64 supports three atomic read-update-write memory operations: exchange, compare-and-exchange (sometimes called compare-and-swap), and fetch-and-add.

Memory—Access Ordering: While a processor's own data references obey read-after-write, write-after-read, and write-after-write orderings with respect to the order of instructions in the instruction stream, reads and writes may be perceived by other processors in different orders. Where relative ordering of accesses to different locations is important, loads, stores, and atomic update instructions may specify additional ordering restrictions. In particular, the IA-64 supports *acquire/release* ordering semantics [5] with optional acquire semantics on loads, release semantics on stores, and acquire or release on atomic updates; it also offers a memory fence instruction.

3. MOST RELEVANT FEATURES

We identify four features of the IA-64 of greatest relevance to GC:

Many general registers: There are enough general purpose registers that it is reasonable, at least more so than on many other processors, to dedicate several to special purposes for GC, such as an allocation pointer. We will see several instances where we advocate dedicating general purpose or predicate registers to specific functions.

Predication: Predication makes it easy to conditionalize a group of instructions without branching. We can exploit it to build atomic instruction sequences in ways not possible without predication.

Acquire/release memory ordering: Some architectures, such as the Sun SPARC (running with Total Store Order), order all stores on multiprocessors; others such as the Compaq Alpha, allow much reordering of loads and stores, and offer only a strict *memory fence* to force ordering when it is required. The IA-64's acquire/release offers the best of both models: it allows much reordering for efficiency in multiprocessor memory hardware while imposing less overhead than fences when some degree of ordering is necessary. We exploit the acquire/release model in implementing Java synchronization and in

[4] The register indirect form supports optional post-incrementation of the base register by either a signed 9-bit immediate constant or (for loads only) the contents of another GR.

[5] Comparisons produce results into *two* predicate registers, so one obtains the opposite sense by switching the two registers.

```
      add rba = 8, ro          // ro = obj base, rba gets bounds address
      add rfe = 16, ro         // rfe = address of first element
      ld8 rb = [rba]           // rb gets bound
      shladd ra = ri, 3, rfe   // ra (element address) gets (ri << 3) + rfe
      cmp.ltu pt, pf = ri, rb  // compare index ri against bound
              // the less-than-unsigned test checks whether 0 <= ri < rb
              // pt == 1 iff the test is true, and pf == NOT pt
(pt)  ld8 res = [ra]           // load the 8-byte element (if pt is true)
(pf)  br throwexception        // throw out-of-bounds exception (if pf is true)
```

Figure 1: Example code for array indexing

```
      add rfa = 20, ro         // ro = obj base, rfa gets field address
      ld4 rf = [rfa]           // load a 4-byte field, offset 20
      sxt4 rf = rf             // sign extend from 4 to 8 bytes
```

Figure 2: Example code for object field access

achieving publication safety.

Exploiting instruction level parallelism (ILP): This concern is common also to VLIW and superscalar machines, but is nevertheless deserving of discussion. For example, a GC implementor might hope that some operations, such as GC write barriers, might have small *incremental* overhead in that they fill otherwise "empty" pipeline slots. We will reconsider this issue after we have presented example instruction sequences.

4. SUPPORTING GC ACTIONS

We begin with a series of topics directly related to GC and memory management, and later cover the memory access ordering features of the IA-64 relevant to memory semantics issues such as publication safety, locking, and `volatile` variables. A point we do no discuss further is that one may more easily manage a 64-bit address space in creative ways than one can a 32-bit address space.

4.1 Accessing Fields and Dispatching

The main thing to note here is that, given the IA-64's single register-indirect memory addressing mode, object references should point to the most commonly accessed word of an object. This most likely is the virtual function table pointer. Other fields may be accessed equally conveniently at positive and negative offsets from the object base, except arrays are best laid out at positive offsets. The shift-left-and-add instruction is convenient for array accesses if the element size is 2, 4, 8, or 16 bytes. Figure 1 gives an example Java array access code sequence; we make no claim of optimality! Note that some of the values, such as the bound `rb` and the address of the first element `rfe`, might be kept in registers for repeated use; it is good that the IA-64 offers plenty of registers!

The first column indicates the predicate register used to predicate execution of an instruction; if there is no register mentioned, then it means to use PR0, which is always 1 (true). Rather than conditionalizing the load of the element, one could use an unconditional speculative load, which would allow the load to be placed *before* the bounds check, which might improve instruction scheduling, depending on the surrounding code. In either case an IA-64 feature (predication or speculation) is useful. Clearly there are many possible variations, and obtaining good schedules depends on specifics of processor implementation as well as surrounding code.

A 4-byte signed integer field access might look as in Figure 2. It could be helpful to manage an "interior" pointer, adjusting it at each field load, using the base-register-update addressing mode, so as to point to the next field needed.

A significant implication for GC is that it is desirable to support interior pointers (pointers that do not point directly to the base of an object). Diwan, et al., explored issues in supporting such *derived pointers* [4]. A derived pointer is a function of some number of base pointers and a fixed or variable integer offset, such that one can recover the offset given the derived pointer and all the base pointers. An interior pointer points to a field within an object, and is a special case of a derived pointer constructed from a single base pointer by adding the offset.

A GC strategy Diwan, et al., proposed is: before GC, convert derived pointers to offsets from corresponding base pointers; during GC, relocate the base pointers as necessary; after GC, convert the offsets back to possibly relocated derived pointers. A nice effect of this strategy is that it shields the majority of the GC code from interior pointers. The strategy requires that each derived pointer have an associated base pointer, which the compiler must keep available somewhere for the GC. Given the number of registers on the IA-64, the occasional retention of additional base pointer values is less of a problem than it might be on other architectures.

Summary: Field access sequences will typically be quite short, with minimum possible memory accesses. The absence of a register-offset addressing mode means that, compared with other architectures, we need to do more address arithmetic adds on the IA-64. However, one should note that an IA-64 add-then-load will likely execute with delay comparable to a register-offset addressing mode on another architecture, since both must do the add somewhere. Further, the IA-64 sometimes avoids the add, or folds it into an update addressing mode on an earlier load or store, and thus may start some accesses sooner. The add does require an extra instruction slot, but early experience with object-oriented code suggests that it will take very aggressive compiler optimization to fill slots tightly. In addition to performance considerations, the IA-64 style of addressing suggests that a JVM will need to manage interior pointers.

4.2 Object Allocation

There are three issues we consider related to object allocation: how to avoid lock overhead on allocation in systems supporting concurrent threads; how to zero allocated memory; and visibility of initializing writes on multiprocessors. The latter issue we defer to the section on memory access ordering.

4.2.1 Zeroing Memory

Zeroing memory is easy to deal with: our experience to date (albeit mostly under the IA-32 architecture) is that it is best to zero allocation areas in bulk, using supplied library routines (e.g., `memset`), immedi-

```
      // swt and swf are predicate registers, always holding opposite values
      //// swt means a task switch and resumption have happened
      //// swt and swf are thread-local
      // ap is the allocation pointer
      // vt is the new object’s vtable pointer value
      // sz is the new object’s size in bytes
      // np receives the address of the new object
top:
(swf) mov np = ap              // indicate address of new object
(swf) st8 [ap] = vt            // store vtable pointer
(swf) add ap = ap, sz          // bump allocation pointer
(swt) br redo                  // task switched, so retry whole sequence
      ....
redo: // reset pred regs
      cmp.eq swf, swt = r0, r0 // set swf true, swt false
      br top                   // try again
```

Figure 3: Example interruptible atomic allocation sequence

```
      // st, sf, ap, vt, sz, and np are as before
      // lp is the limit pointer
top:
(swf) mov np = ap              // indicate address of new object
(swf) st8 [ap] = vt            // store vtable pointer
(swf) add ap = ap, sz          // bump allocation pointer
(swf) cmp.le swf, swt = ap, lp // merge limit test result into swf, swt
(swt) br redo                  // task switched or past limit
      ....
redo:
      cmp.eq swf, swt = r0, r0 // set swf true, swt false
      cmp.le 0, pgt = ap, lp   // redo limit test to discriminate
(pgt) br.call rp = gc          // call gc
      br top                   // try again
```

Figure 4: Interruptible allocation sequence including limit check

ately before starting to use an area for allocation (this can be avoided on the first use of demand-zero pages). We observed this to be better than zeroing immediately after GC or upon each allocation. This is particularly likely to be true on a multiprocessor where one processor does the GC work, since it not only removes the zeroing from the critical path of allocating and using objects, but also overlaps the zeroing with other work, rather than simply moving it to another time and gaining economy of scale by doing it in bulk. Bulk zeroing is a potential performance advantage of linear allocation over free-list techniques.[6]

4.2.2 *Avoiding Lock Overhead: Can the OS Help?*

The general strategy we propose for avoiding lock overhead on allocation has been done before: provide an separate allocation area for each processor, and lock only when a processor's area fills up and it needs another large chunk from a global pool. This eliminates possible interference from other processors. But another atomicity problem remains: atomicity with respect to threads run by the *same* processor.

What we desire is this: if we receive a time-slice interrupt, at any point in the allocation sequence, to be able to switch threads, have the new thread perform allocation from the same allocation area, and when we switch back either for the interrupted allocation to be ok or to have failed detectably, so that we can retry it.

The OS support we would like is simple: after a task switch, before resuming the interrupted task, set a pair of predicate registers indicating that a task switch has occurred.

The code sequence in Figure 3 exploits this feature, keeping the allocation pointer in a register. This register's value is propagated from thread to thread (within the same OS process) when we thread switch on the same processor, which, from the thread's point of view, means the allocation pointer can "jump" at any time.

To understand this code sequence better, consider the effect of a task switch (and resumption) at each possible location:

At `top` (or before): The next three instructions are not executed (their predicate is false); we branch to `redo` and start over. Note that retry is fairly cheap, which is why we do not start the sequence with setting the `swf` and `swt` predicates every time.

After the `mov`: There are no effects visible to other threads, and since we retry, we overwrite the possibly stale value in `np`.

After the `st8`: In this case we have stored the vtable pointer to memory, but any allocation in a thread we switch to will overwrite it, and we will retry and allocate elsewhere, or execute the store again in the same place.

After the `add`: Here, since we have changed `ap`, the allocation has effectively occurred as far as other threads are concerned. However, we cannot bump `ap` and check the task switch flag together atomically, so we will retry. The net effect is to leave a garbage object behind—not quite as nice as we would like, but not harmful except perhaps to performance. With suitable grouping of the instructions into bundles, this case can be made quite rare, perhaps impossible on some implementations of the architecture (i.e., they will not interrupt between the `add` and `br`).

After the `br`: We have committed the sequence.

Figure 4 shows a similar sequence that includes a limit check; we omit the reasoning as to the correctness of interruption at each point,

[6] This raises an additional point: allocating objects in the stack has different zeroing cost, too, so it is not obvious that allocating in the stack is faster than linear allocation and a well-tuned garbage collector.

```
(sf)  mov np = ap              // (as before)
(sf)  st8 [ap] = vt, sz        // store vtable pointer, bump ap by sz
(sf)  cmp.le sf, st = ap, lp   // (as before; only if need limit check)
(st)  br redo                  // (as before)
```

Figure 5: Allocation using post-incrementing store

```
      // ap POINTS TO the allocation pointer in memory
      // vt, sz, and np are as before
retry:
      ld8 np = [ap]            // get address for new object
      mov ar.ccv = np          // set compare value register
      add tmp1 = np, sz        // bump by size
      xor tmp2 = tmp1, np      // check for crossing block boundary;
      shr tmp2 = tmp2, k       // also checks BIG sizes, which flag
      cmp.eq PR0,p1 = tmp2,R0  // ...  the special cases
(p1)  br check-for-gc          // go handle overflow and special cases
      cmpxchg8.acq tmp2 = [ap], tmp1, ar.ccv
           // exchange, writing tmp1 to [ap] if [ap] equals np
      cmp.eq PR0,p1 = np,tmp2  // see if value read (tmp2) equals np
(p1)  br retry                 // retry if cmpxchg failed
      st8 [np] = vt            // proceed to set up object
```

Figure 6: Allocation using compare-and-exchange

```
      // obj holds a reference to the object modified
      // f is the offset of the field being updated
      // p is the reference being stored
      // ct holds the virtual base of the card table:
      //// the location that would hold the mark for the card at address 0
      shr.u ry = obj, k        // form card index (k is a constant)
      add rx = obj, f          // form field address
      add ry = ct, ry          // form address of card byte
      st8 [rx] = p             // store the pointer
      st1 [ry] = GR0           // store the constant 0 in the entry
```

Figure 7: Pointer store with card marking

since the argument is very similar to the previous sequence.

Both sequences can be improved if the size of the new object is known at code generation time and will fit in the 9-bit signed immediate field of a post-incrementing store instruction, as shown in Figure 5.

Something else to keep in mind concerning the atomicity of these sequences is that, while there is nothing in the instruction set architecture specification indicating the groups of instructions that will actually be executed concurrently, *specific implementations* might in fact happen to exhibit stronger atomicity, i.e., that interrupts will not happen at certain places because multiple instructions are either all executed or none, just because of the way the pipelines, instruction issuing, etc., work.

Another technique we use in allocation is worth mentioning: if an object has special allocation requirements, e.g., an alignment restriction, finalization, or weak pointer properties, we set a high order bit of the size information in the class—the word called `sz` in the code sequences. Adding such a size will violate the limit, and thus send us to the "slow path". In this way we get very fast inline allocation for the common case, with the rest handled by a subroutine. This helps maintain separation of responsibility among the class loader (which creates size information), the JIT, and the GC. It also removes the need for the JIT to generate allocation code for these more complex cases.

The notion of possibly abandoning an allocated object, avoiding more costly interlocking for atomicity in allocation, is new. Shivers, et al., [11] offer a good survey of atomic allocation techniques, but they focus on list pairs, rather than objects of different sizes, which need a vtable pointer stored for the GC to be able to interpret them.

4.2.3 *Alternative Approach to Allocation*

An obvious alternative approach that does *not* require new OS support is to use an atomic instruction. The obvious candidate is fetch-and-add, but it allows only certain small and fixed increments. Thus we offer in Figure 6 a sequence based on compare-and-exchange. In this case we use a limit test based on going past the end of an aligned block of size 2^k bytes.

This sequence could be shortened a bit if `cmpxchg` were available for register operands. The virtue of our previously proposed sequence is that it is faster, since it has fewer and cheaper memory operations.

4.3 Write Barriers

Many garbage collectors employ *write barriers* to detect when user code creates a pointer from one region to another. In particular, generational collectors use write barriers to detect the creation of pointers from older to younger generations. One form of write barrier is *card marking* [12], in which one associates with each aligned 2^k byte region (called a *card*) a mark indicating whether any object starting in that region has experienced a pointer store. Figure 7 shows a code sequence for card marking on the IA-64.

This sequence is not in itself particularly subtle, though there is flexibility in scheduling the `st8` instruction later if that produces a better schedule. However, we observe that this sequence marks the card corresponding to the address of the object's *header*, not the ad-

```
      // obj holds a reference to the object modified
      // f is the offset of the field being updated
      // p is the reference being stored; m is a mask of k low-order ones
      //// (it’s constant, but too big for an immediate)
      // s is the sequential store buffer pointer
      add rx = obj, f          // form field address
      andcm ry = p, m          // round p down to start of block
      st8 [rx] = p             // store the pointer
      cmp.lt px, py = obj, ry  // compare source and target addresses
(px)  st8 [s] = rx, 8          // store rx to SSB, increment s by 8
```

Figure 8: Address order write barrier, with sequential store buffer

```
      // obj holds the reference to check
      tbit.nz pnz,PR0 = obj,0  // test bit 0 of obj
(pnz) br.call rp = rdbarrier   // call read barrier if it’s 1
```

Figure 9: Example read barrier sequence

```
      // obj holds the reference to check
      ld8.s vt = [obj]         // start spec load of vtable word
      tbit.nz pnz,PR0 = obj,0  // test bit 0 of obj
(pnz) br.call rp = rdbarrier   // call read barrier if it’s 1
      chk.s vt, redo           // check if load worked
join: ...

redo: ld8 vt = [obj]           // non-speculative load, done when
      br join                  //// obj is ok but ld8.s failed for
                               //// another reason, e.g., TLB miss
```

Figure 10: Read barrier with interleaved speculative load

dress of the *updated slot*, as suggested by Hölzle [6]. The sequence of the figure comprises three instruction groups, consisting of the first two instruction, then the next two, and finally the last instruction. A sequence that does card marking based on the slot address will have more groups, because it must do further calculations dependent on the result `ry`, and thus may take more cycles to execute.

The sequence for an address-order write barrier [13, 14] is perhaps more interesting. In Figure 8 we use a block size of 2^k; we also record the interesting stores using a *sequential store buffer* [8], to illustrate that feature.

This sequence takes advantage of the large register set (dedicating registers to hold the block mask `m` and the SSB pointer `s`), of the large address space (in using the address-order write barrier, which, though it can work in smaller address spaces, is particularly suited to large address spaces), and of the predicated execution and auto-increment addressing mode features of the IA-64. This write barrier is just as long as the card marking one, and has the same number of instruction groups, but records the actual location updated, which may speed up processing in the GC code (though it does not absorb duplicate updates as nicely as card marking does).[7]

Chilimbi and Larus [3] used an SSB to log certain object accesses, to help reorder objects at GC time and improve cache performance. Again, the IA-64 makes this technique attractive, since it takes just one instruction and there are likely enough registers that dedicating one to the SSB does not hurt other things. Further, the cache control features of the IA-64 suggest useful extensions to Chilimbi and Larus’s approach, adding prefetching both in application code and in the GC.

4.4 Read Barriers

Some garbage collectors use *read barriers* to detect accesses to certain objects (called *node marking* [7]), or via certain pointers (*edge marking*), which is what we consider here. Read barriers are also useful in supporting persistence, to detect accesses to objects not currently resident. Since the IA-64 is byte-addressed, and since objects will naturally be aligned on 8-byte boundaries, normal object references will have the low three bits zero, so we can use one of those to mark the interesting references. The IA-64 can test the bit, and then branch (or execute other code predicated on the test), quite efficiently, as shown in Figure 9.

This code sequence will tend to execute with minimal delay (e.g., one pipeline tick between the two instructions). Even better, one can start loads via `obj` *before* this check, using IA-64 speculative loads, which implies minimal impact on the timing of the normal case critical path. Figure 10 gives a (trivial) illustration of this.

Note that after the call, either `vt` was loaded all right to begin with, or the `rdbarrier` routine fixed it up, or the `ld8.s` failed for some other reason, e.g., TLB miss. In the case where the address should be ok but the `ld8.s` failed, we retry the load non-speculatively, to force handling of soft faults (or reporting of hard ones). In any case speculation can help hide the cost of a read barrier.

4.5 Object Scanning

During garbage collection one must scan objects in the heap, in order to process their pointer fields and the targets of those pointers. We found the following technique to produce efficient code. We associate with each non-array class a table giving the offsets of the pointer fields for objects of that class. We mark the end of the table with a 0 word. This scheme is undoubtedly well known, but to our knowledge is not in the literature. It is faster (as tested on the IA-32) than scanning

[7] Our experience with benchmarks is that duplicates are rare, though it is trivial to write a program that updates the same slots many times.

```
      // obj holds the reference to the object
      // t points to the first table entry; off gets the offset
next: ld4 off = [t], 4         // 32-bit offsets; bump ptr, too
      cmp.ne pgo,PR0 = off,GR0 // check for end
      add rfa = obj, off       // form field address
(pgo) ld rf = [rfa]            // load field
(pgo) ...                      // additional processing
(pgo) br next                  // loop if more
                               // fall through when done
```

Figure 11: Object scanning code sequence

```
      // obj holds the reference to the object; p will scan it
      add p = obj, 8           // form address of size field
      ld8 sz = [p], 8          // get size; bump p to first element
      mov LC = sz              // get size in the loop count reg
      br test                  // branch to test at end of loop
next: ld8 elem = [p], 8        // fetch element and bump p
      ...                      // additional processing
test: br.cloop next            // decr LC and branch if not 0
```

Figure 12: Scanning an array with a counted loop

a bit-vector indicating pointer and non-pointer fields for objects of each class. Our scanning code looks something like what is shown in Figure 11.

For handling arrays of pointers, we can scan using a *counted loop* and the `LC` (loop count) application register, as shown in Figure 12. (We could use the same approach for the field offset table if it happens to run faster.)

4.6 Stack and Register Tracing

For accurate GC one needs to find exactly those locations in the thread stacks, registers, globals, and heap that contain pointers. We described how to scan heap objects for pointers in the previous section. The general approaches one would use for finding pointers in stacks and registers are the same as for other architectures, and there are software conventions, particularly those supporting exception handling, that help in "decoding" the stack so as to find individual stack frames. On the other hand, determining how to handle each stack slot and register is a bit more subtle, because of the IA-64's predication and speculation features.

Control speculation, i.e., speculative loads, presents three cases. First, the load may have failed, leaving a NaT in the register. Second, the load may have succeeded, but read a value that the program is not going to use, e.g., by speculatively loading a value off the end of an array. This presents a difficulty since such values can be arbitrary, and thus should not be treated as pointers by the GC. Worse, the predicate determining if the value will be used may not yet have been computed. The third case is that the load may have succeeded in loading a value that will be used (and thus is type safe).

One way to handle speculatively loaded values is to make the load to appear to have failed, and thus not need to handle the value that may have been loaded. The JIT should therefore produce tables from which the GC can determine which registers contain speculatively loaded values that might possibly be pointers. The GC will then set the NaT bits for those registers, and when the thread is resumed, either the value will never be used or the thread's checks will redo the loads as necessary. The thread will therefore see the new values the GC may have produced as it moved objects in the heap.

Data speculation, i.e., advanced loads. Again, the issuing thread checks these, and it is concerned with possible aliasing by stores between the advanced load and the check. Thread switching invalidates advanced load information (the ALAT), so one need not consider other threads (except perhaps on a multiprocessor, but that raises many concurrency issues beyond the ALAT). If running GC on the same thread, it is correct (and easiest) simply to invalidate the ALAT.

Predication encourages a code generation style in which if-then and if-then-else code is produced without branches, having the conditional code predicated. Such predicated code would then be interleaved, with the then-clause instructions mixed in with the else-clause instructions, but only those instructions corresponding to the proper clause actually executed. Now suppose that on one branch of an if-then-else a particular register contains a pointer, and on the other branch a non-pointer value. It is clear that the GC must consult the predicate to determine whether the register contains a pointer, since the program counter value cannot tell us which clause is being executed.

This kind of situation was anticipated by Diwan, Moss, and Hudson [4], but they found it to be very rare in their code, whereas for the IA-64 it may be more common. This leads to a style of associating a predicate register with each general register, indicating whether that general register contains a pointer. We can use PR0 in the unconditional case. (We also need a "sense", i.e., whether having the PR be true means that the GR holds a pointer or means that the GR holds a non-pointer.) Note that in these cases there is no overhead in the mutator, since the predicate is already being used. Note further that usually the same predicate can be associated with more than one GR.

This association of predicate registers with general registers to indicate which contain pointer values leads to a novel way of handling the "`jsr` problem" [1]. In Java, a typical way to generate code for a `try-finally` block is to emit the `finally` clause as a local subroutine, called with the Java `jsr` bytecode. This `finally` block is called from the normal case and the exception case, and those two contexts may use the same local variable slot differently, one for a pointer and the other for a non-pointer. If we use a predicate register to distinguish the normal and exception cases, then our register decoding mechanism handles `jsr` routines nicely.[8]

4.7 GC Safe Points

In most garbage collected systems there are points in the code where

[8] Since the number of predicate registers is finite and nesting of `try-finally` blocks is not bounded, this scheme might (in principle, probably not in practice) need an overflow mechanism.

```
// Thread T1                  | // Thread T2
// v has the vtable value     | // g points to the global
// p has the object address   | // p gets the object address
// g points to the global     | // v gets the vtable value
st8 [p] = v                   | ld8.acq p = [g]      // .acq not required!
st8.rel [g] = p               | ld8 v = [p]
```

Figure 13: Object initialization / publication safety example

a GC is ok—and other points where it is not, because some important invariant is temporarily violated, e.g., between a write and its corresponding write barrier. In the Intel JVM for the IA-64 we used the same approach we did for the IA-32: to produce suitable register and stack frame mapping tables for essentially every code position in code generated by the JIT compiler [15]. Native routines that might lock or take a long time have associated GC tables. However, not all routines coded in C that use references have tables, so collection is disallowed when a thread is executing there. We make write barriers GC-atomic by performing them in such a routine. We get threads into GC-safe states by continuing and interrupting them, repeating as necessary, until they are in GC-safe code. The delay until we get a thread to a GC-safe point varies statistically, but the approach seems to work well in practice. One could also have the GC interpret forward through short stylized sequences, such as write barriers. In any case, the GC-safe-almost-everywhere approach works fine on the IA-64.

We also considered, but did not implement, a complementary approach: GC-safe only at certain chosen spots. This also appears to be easy to support on the IA-64. (In fact, one can use the same technique, but if the safe spots are relatively rare, the expected number of times one needs to allow a thread to advance will be large.) One simply dedicates a predicate register to indicate that a GC (or other interruption of normal control) is desired, and plants a predicated call or branch instruction at each safe spot. At first blush this polling may not seem attractive, but in practice one likely has many choices of where to place the polling instructions, and can choose to put them in otherwise unused slots. Also, since the predicate they are testing is essentially never set in nearby code, they do not involve data dependences that reduce parallelism.

Note that if a thread is executing a “foreign” code subroutine, i.e., code that does not necessarily obey the same register conventions, etc., then rather than using a predicate register to signal the need for an interruption of control, one might use a (probably per-thread) memory location, checked on the way out of the foreign code. This shows that polling via predicate registers must use *local* rather than *global* registers, and that one would thus need to manipulate each thread’s register set independently (versus setting a single global value).

There are undoubtedly many other techniques. Our point here is that for the IA-64 it is more reasonable to consider dedicating a register for this kind of purpose.

4.8 Memory Access Ordering Issues

In order to achieve the best hardware implementation performance, many modern multiprocessors do not guarantee that memory accesses become visible to other processors in the same order they are performed locally. In those cases in which software algorithms require certain orderings for correctness, one uses special instructions to enforce the required ordering. The IA-64 supports the *acquire/release* model of ordering. A *load acquire* guarantees that its load appears to happen at the memory (i.e., to other processors) before later memory accesses by the same processor; a *store release* guarantees that its store appears to happen at the memory after earlier memory accesses by the same processor.

While it is admittedly hard to get used to at first, ordering is quite distinct from atomicity. Atomic read-update-write operations affect a single memory location, and guarantee that no other read or write happens to that location in the middle. Acquire/release instructions enforce ordering with respect to *other* storage locations in addition to the one accessed, but do not of themselves guarantee atomicity.

An important case that has come up in discussions about Java concerns initializations of objects [10]. Suppose thread T1 allocates an object, initializing the vtable, and then stores into a global variable the address of the new object. A little later, thread T2 reads the global variable, obtains the address of the new object, and reads the vtable pointer. We would like to insure that T2 sees the right vtable pointer value. Figure 13 shows code that will work.

It works because the `st8.rel` forces the `st8` to occur first, the `ld8.acq` forces the `ld8` to occur later. Thus, if the `ld8.acq` obtains the value stored by the `st8.rel`, the `ld8` will obtain the value stored by the `st8`.

The possible problem here is that ordered memory operations can be slower. The case of initialization is not necessarily that bad. We need a store release only when storing a pointer that might be read by another thread. Thus, entirely local objects do not even need it. Since the two stores are not otherwise related, we need the store release to enforce the ordering if the object may be accessed by other threads. So this is the best we can do from T1’s side.

T2’s side is more worrisome, though, since it would seem to imply that we need a load acquire every time we load a pointer from the heap, to an object whose fields we access, unless we can prove the object is private. However, unlike T1, T2’s loads are *related*: the `ld8`’s address (in `p`) depends on the result of the `ld8.acq`. It turns out that the IA-64 will enforce an ordering on these two particular loads even if we use a non-acquiring load. This is good news, since load acquires force ordering with respect to *all* memory accesses, not just the dependent ones, whereas all we need here is for the dependent ones to be ordered.

There are two other cases where memory access ordering is a particularly prominent issue: locking, and volatile variables. These are both handled fairly nicely on the IA-64. When locking a (non-private) object, one uses an atomic operation, such as compare-and-exchange, with an acquire tag to force acquire ordering semantics. Once one has the lock, one uses ordinary loads and stores to access the locked object’s (or objects’) fields, then uses another compare-and-exchange, but with a release tag to obtain release semantics. This guarantees that all access to the locked object are suitably “bracketed” by the lock/unlock operations. Figure 14 shows code for this approach.

The code shown in Figure 14 is for the common case of a synchronization implementation strategy known as *thin locks* [2]. In this strategy the common cases record all relevant lock state information directly in the lock word. However, if other threads wait for the lock, they enqueue themselves by changing the lock state to refer to their queue entries. To do that enqueuing, a thread *locks the lock word*, i.e., it obtains the *meta-lock*. When a thread obtains the meta-lock, no other thread is allowed to change the lock word until the meta-lock is released. Thus, an `st.rel` instruction can be used to release the meta-lock since the lock word’s value cannot change while the meta-

```
      // obj holds a reference to a lock field
      // cv has the expected current value of the field
      // nv has the desired new value
      mov ar.ccv = cv          prepare to compare-and-exchange
      cmpxchg8.acq tmp = [obj], nv, ar.ccv
      cmp.eq PR0,p1 = cv,tmp   // see if value read is expected
(p1)  br.call slowlock         // handle less common "lock" cases
      ...                      // access with ordinary loads/stores
      mov ar.ccv = nv          now expect the "new" value
      cmpxchg8.rel tmp2 = [obj], tmp, ar.ccv
      cmp.eq PR0,p1 = nv,tmp2  // see if value read is expected
(p1)  br.call slowunlock       // handle less common "unlock" cases
```

Figure 14: Code sequence for Java synchronized access

```
      // obj holds a reference to a lock field
      // cv has the expected current value of the field
      // nv has the desired new value
      mov ar.ccv = cv          prepare to compare-and-exchange
redo: cmpxchg8.acq tmp = [obj], nv, ar.ccv
      cmp.eq PR0,p1 = cv,tmp   // see if value read is expected
(p1)  br redo                  // (one could add back off, etc.)
      // here we hold the metalock
      //// we can change the lock info with ordinary loads/stores
      ...
      // end with nv holding the "metalock-released" value
      st8.rel [obj] = nv
```

Figure 15: Code sequence for metalock usage

lock is held. We show this in Figure 15. The flip side of this protocol is that releasing an ordinary lock requires a compare-and-exchange since the lock word's value can change while the lock is held, e.g., to enqueue a thread that is requesting the lock.

Java volatile variables need to be read with acquire semantics and written with release semantics, to enforce suitable memory ordering. Furthermore, unless the Java memory model is revised, in order to guarantee sequential consistency of accesses to volatile variables, stores to volatiles need to be followed by memory fence instructions before the next read of a volatile by the same thread.

We observe that some other architectures offer *only* a memory fence to enforce ordering, and that it results in potentially much greater overhead than the acquire/release model. The IA-64 property that dependent loads are ordered may improve efficiency considerably as well.

5. MOST RELEVANT FEATURES AGAIN

Having surveyed a range of GC-related features and code sequences, we reconsider the four most relevant IA-64 features.

Registers: We suggested keeping an allocation frontier pointer and corresponding limit pointer in general registers, keeping an SSB pointer in a general register, keeping task switch flags in predicate registers, and that retaining (in general registers) base pointers corresponding to derived pointers would not likely be a problem.

Predication: We showed how predication helps in constructing atomic sequences, e.g., for allocation, and we also suggested using predicate registers to help determine which general registers contain pointers at particular code points.

Memory access ordering: We showed how to use the IA-64's acquire/release model to support publication safety fairly cheaply, and described how it avoids full memory fences for most synchronizations and accesses (except stores to volatile variables).

Exploiting ILP: This topic deserves further discussion. The general experience in the community is that object-oriented (OO) code, or at least pointer-oriented, code, which we take as being largely correlated with use of GC, does not offer as much ILP as (say) Fortran array code. While the optimization techniques are different and (some might argue less mature) for OO code, the difference in ILP appears also to be more fundamental: OO programs follow chains of dependent pointers. The IA-64 does not directly “fix” this problem, though predication and speculation help.

An appropriate question to ask is: *Does inserting GC-related code sequences make the situation worse?* We consider three examples.

First, we presented two sample write barrier code sequences above (Figures 7 and 8). Each consists of five instructions in three instruction groups. Clearly, if the write barrier is in the vicinity of just the right other code, it can be interleaved into that other code's instruction groups and may not result in any slow down. If the other code's groups have no-ops that the write barrier can fill, then one would not even be increasing the size of the groups. Even if one needed to make the groups larger and use other bundle templates, the number of clocks needed to fetch, issue, and execute the code might not change, though the slight increase in code size might impact instruction fetch bandwidth and instruction cache footprint. How *often* and how effectively write barriers can be merged with surrounding code is an interesting quantitative question beyond the scope of this paper.

Second is the card marking write barrier. As mentioned before we mark based on the object header rather than the address of the updated slot. Since the slot address must be calculated from the header address, using the slot address implies delaying calculation of the card to mark, which increases the number of instruction groups.

A third example is polling for rare conditions, such as a request for a GC in a polling implementation of GC safe points. One would insert predicated (i.e., conditional) call instructions. In this case, we can be opportunistic and search for no-op slots suitable for such calls.

Since there is no nearby setting of the predicate register tested by the instruction, there is no nearby data dependency, so one can imagine implementations of the IA-64 instruction set architecture that eliminate such conditional branches early and with no disruption to pipeline flow. On the other hand, the succeeding instructions *are* control dependent on the conditional branch's not being taken, and thus there may be one or more cycles of delay, depending on the issue logic and pipeline design. While such polling instructions can *conceivably* be essentially "free," they may still have some overhead. This overhead might be minimized if they are placed at the end of instruction groups, which are likely to incur a delay anyway.

We offer a final observation about pipeline slots: other parts of systems beyond the GC would like to use any "spare" slots. Examples include low overhead profiling and assertion testing.

6. SUMMARY AND CONCLUSION

The Intel Java Virtual Machine now runs on engineering sample IA-64 hardware and software. The GC is robust and supports generational copying GC, a large object space, the Train Algorithm (mature object space), finalization, weak references, threads, and synchronization. The JVM includes a JIT. We look forward to measuring the system and comparing implementation strategies once production systems become available, but our main point is that most of the techniques we have described are implemented and known to work.

The contributions of this paper are as follows. Most obviously, we have illustrated how various GC-related code kernels appear on the IA-64, simplifying GC implementation for compiler and run-time writers. We described how GC can handle code that exploits the speculation features of the IA-64. We devised a task-switch signal from the operating system that simplifies and speeds up atomic object allocation—and challenge the OS community to support it. We recorded for the literature an object scanning technique, tested to be faster than scanning bit vectors. We offer a new solution to the Java `jsr` problem, using IA-64 predicate registers and giving a uniform solution to modeling which registers contain pointers for GC. We described how to achieve publication safety on the IA-64—cheaply.

In our survey of techniques and code fragments we identified four features of the IA-64 as particularly relevant to GC: the large register set allows more GC-specific items to reside in dedicated registers; predication makes it easier to construct atomic instruction sequences for object allocation; acquire/release semantics and ordering of dependent loads give the prospect of better performance for synchronization primitives and publication safety; and instruction grouping and multi-way issue likely provide significant opportunity to "hide" the overhead of GC-related operations such as write barriers.

Our conclusion is, as our examples have shown, that the IA-64 provides superior capabilities for implementing GC.

Acknowledgments

Many members of the Intel Java Virtual Machine group also worked in the implementation, including Michal Cierniak, Jesse Z. Fang, Andrew Hsieh, Guei-Yuan (Ken) Lueh, and Tatiana Shpeisman. The anonymous reviews were helpful.

7. REFERENCES

[1] O. Agesen, D. Detlefs, and J. E. B. Moss. Garbage collection and local variable type-precision and liveness in Java(TM) virtual machines. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, volume 33 of *ACM SIGPLAN Notices*, pages 269–279, Montreal, Québec, Canada, June 1998. ACM Press.

[2] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *1998 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 258–268, Montreal, Quebec, June 1998. ACM Press.

[3] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *The 1998 International Symposium on Memory Management*, Vancouver, BC, Oct. 1998.

[4] A. Diwan, J. E. B. Moss, and R. L. Hudson. Compiler support for garbage collection in a statically typed language. In *Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992. SIGPLAN, ACM Press.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 15–26. ACM Press, May 1990.

[6] U. Hölzle. A fast write barrier for generational garbage collectors. In E. Moss, P. R. Wilson, and B. Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, Oct. 1993.

[7] A. L. Hosking. *Lightweight Support for Fine-Grained Persistence on Stock Hardware*. PhD thesis, University of Massachusetts at Amherst, MA 01003, Feb. 1995.

[8] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, Sept. 1991.

[9] Intel Corporation. *The IA-64 Architecture: Software Developer's Manual*. Santa Clara, CA, Jan. 2000. URL: http://developer.intel.com/design/ia-64/manuals/.

[10] W. Pugh. Fixing the Java memory model. In *Java '99: Proceedings of the 1999 ACM Conference on Java Grande*, pages 89–98, San Francisco, CA, June 1999. ACM Press.

[11] O. Shivers, J. W. Clark, and R. McGrath. Atomic heap transactions and fine-grain interrupts. In *Proceedings of the 1999 ACM International Conference on Functional Programming (ICFP)*, Paris, France, Sept. 1999.

[12] P. G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers, 1988. B.S. Thesis, Dept. of EECS, Massachusetts Institute of Technology, Cambridge.

[13] D. Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, Amherst, MA, Feb. 1999.

[14] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *Proc. 1999 ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, November 1-5, 1999*, pages 379–381. ACM, Nov. 1999.

[15] J. M. Stichnoth, G.-Y. Lueh, and M. Cierniak. Support for garbage collection at every instruction. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 118–127, May 1999.