

# **Thread and Interrupt Safe Method Dispatch in PCL**

July 2007, Nikodemus Siivola  
Steel Bank Studio Ltd  
(for the 4th European Lisp Workshop)

<http://sb-studio.net/pub/>

## **Abstract**

Efficient Method Dispatch in PCL (Kickzales and Rodriguez, 1990) describes some of the strategies used in the original PCL implementation of Common Lisp Object System and MOP. We informally discuss the modifications done in SBCL version of PCL to provide thread and interrupt safe method dispatch.

## Introduction

Optimized CLOS method dispatch á la Kickzales and Rodriguez is inherently state-full: dispatch functions need to be updated as new methods are called, and as mappings are added to memoization caches.

The SBCL version of PCL was recently (SBCL 1.0.7) modified to make both memoization cache and dispatch function updates thread and interrupt safe. Additionally an interrupt safety issue affecting metacircle detection during applicable method computation has been identified and addressed. The work described here does not purport to be complete: other thread and interrupt safety issues may still exist in SBCL's method dispatch. Our discussion focuses on informal exposition of the new cache algorithm, and lightly touches on some other thread and interrupt safety issues.

## Memoization Cache

The memoization cache lies in the heart of many of the method dispatch code paths in PCL, even though there are multiple alternative strategies. It is essentially a table mapping lists of class wrappers to precomputed effective methods, slot offsets, and constant return values: when a method is called, the class wrappers of its arguments are obtained and used to probe the cache. If there is a hit, it is the desired effective method; in the event of a miss the effective method is computed and stored in the cache so the the next call with similar arguments need not compute it again.

The above synopsis glosses over several details, such as invalidating wrappers when classes are redefined and invalidating caches when methods are added to generic functions, but these aspect remain unchanged from the original and have little bearing on the cache algorithm, so we do not delve on them.

In the original PCL implementation a modicum of thread safety was provided by using a modification counter in the cache: the counter was checked before and after a read, and incremented for each modification. If the counter changed during a read it was treated as a cache miss. By the time SBCL forked from CMUCL the bit-rot had already set in: PROBE-CACHE calls didn't check the counter, all modifications didn't increment the counter, and there were code paths where modifications to the cache in the middle of a read could lead to type-errors.

Instead of fixing the modification counter code we opted for a new strategy, both in the interest of rewriting the cache code for clarity, and in hopes of making cache reads (the overwhelmingly common operation) cheaper. We can probably declare at least partial success on both counts: the new cache implementation is shorter and arguably easier to read; while cache performance was not drastically improved by the new implementation, it also did not degrade in any of the benchmarks used and some benchmarks show modest improvements.

There are two kinds of caches: those storing only keys, and those storing both values and keys. Again, this distinction is not interesting from the point of view of the thread and interrupt safety, so we ignore it in this discussion, but interested readers wishing to explore the SBCL code base should bear this in mind. The number of keys used per value is constant for any given cache: a method with a single required argument uses a 1-key cache, while a method with N required arguments uses an N-key cache.

## *Cache Layout*

The cache contains a cache vector, in which both keys and values are stored. Sub-sequences of the cache vector are called cache lines. An empty cache vector is filled with sentinel values. The first N-elements of a filled cache line in N-key cache are the keys, and the N+1th element of a filled cache line is the value.

To map N keys to to a cache line, they are hashed, and their hash values mixed together to produce a combined hash value. The primary cache line starting index is then computed from this:

```
(mod (* hash line-size) (length cache-vector)) => index
```

The N keys starting from this location in the cache are then compared to the desired ones: if all keys match the line is the correct one. If any key is a mismatch the next cache line is tried, and this is repeated until the correct one is found, or maximum probe depth is reached -- meaning the keys do not exist in the cache.

## **Cache Reads**

Cache reads proceeds exactly as described above, with the additional condition that if any key or value read from the cache is the sentinel value, it is treated as an immediate miss. Reading a sentinel value can happen when the probing reaches an empty cache line, when the cache line being read is in the process of being written to but is not yet complete, and when the cache write to that line was interrupted and didn't complete the line.

## **Cache Writes**

Writing to cache start similarly: the primary cache line is computed. Instead of comparing the contents of the cache line to the desired keys, however, the implementation attempts to write it's own keys to the cache line using compare-and-swap:

```
(compare-and-swap (svref cache-vector index) +sentinel+ key) => old
```

The compare-and-swap is implemented using an atomic hardware instruction (LOCK CMPXCHG on x86-family of processors): it succeeds only if the old value at the index is EQ to +SENTINEL+, in which case it replaces it with our key. If it is not, nothing is done. In both cases the original value is returned: either the sentinel that was replaced, or an already written key.

If the returned value is the sentinel (meaning the write was a success), or if it is EQ to the key we tried to write (meaning an EQ key was already in place) we continue; in both cases the end result is the same: our key at the index we wanted to have it in. If we succeed in ensuring all our keys in successive indices in the same cache line, we need only write our value after them: that we can write heedless of what is there beforehand, as long as the write itself is atomic (so no thread can see a "partially written" value). As our values are word-sized lisp objects, this is not an issue on any realistic hardware platform today as long as the cache vector is properly aligned.

However, if the value returned by COMPARE-AND-SWAP is not the sentinel or EQ to our key we abandon the attempt and move to the next cache line, starting over with it. We continue in this manner until we either succeed ensuring a full line, or till we reach the maximum allowed probe depth of the cache -- in which case we abandon the effort and allocate a new larger cache, which replaces the original one.

When replacing a cache with new one, we first allocate an empty cache vector initialized with sentinel values, then we write our new entry there, followed by copying all the complete and valid cache lines that fit from the old cache. This order is necessitated by our desire to limit the maximum size of caches -- if nothing else, we are limited by ARRAY-DIMENSION-LIMIT. If we have reached the maximum cache size we must write the new entry first to ensure that it will fit and the write will succeed. Since the cache is a memoization cache possibly dropping old entries is harmless from correctness point of view, and efficiency of normal cases is unaffected. It needs to be said that we are not entirely satisfied with this arrangement: dropped entries should be randomized to avoid ping-pong cases where successive method calls cause each other's entries to be dropped from the maximally grown vector, and actually shrinking the vector once it is fully grown might also be worth in some cases.

## ***Thread Safety***

Cache reads are trivially thread safe as long as keys stored in the cache vector remain unchanged.

Writing values using the above algorithm never changes a key once it has been written, so parallel readers and writers that have already seen it are safe.

Values can change, but since they do not affect consistency thread safety is unaffected. Reads of incomplete cache lines result in misses and trigger new writes: this is a source of inefficiency, but does not violate consistency as the write will then complete the interrupted line.

## ***Interrupt Safety***

Cache reads are trivially interrupt safe.

If an asynchronous interrupt and unwind happens at any time during a write, an incomplete cache line will be left in place. This is harmless, as the next write using wrappers matching the ones what were written will complete it.

## **Applicable Method Computation**

CLOS applicable method computation must detect metacircles. That is, it must notice if the computation uses the method being computed.

Classic PCL implementation uses special variable holding a stack of methods being computed to detect this: an error is signalled if an applicable method computation is initiated with the method in question already on the stack. The stack is implemented as a list on a special variable: to push a value on the stack it is consed on the front and the variable is rebound to the new list head.

If an interrupt arrives during applicable method computation and the handler uses the method being computed, this approach detects a bogus metacircle -- unless steps are taken to prevent this.

The solution SBCL uses is to rebind the stack to NIL for interrupt handlers. Real metacircles occurring in interrupt handlers are detected normally, the detection merely starts from scratch. Another alternative would be to disable interrupts for the duration of applicable method computation, but this is not desirable: disabling and re-enabling would incur a cost for the common case of no interrupts during the computation.

## **Discriminating Function Updating**

When the memoization cache is expanded, the discriminating function must be updated to have a reference to the expanded cache. Associated information must also be stored in the generic function for other parts of the system to access, resulting in a two-step process:

```
;; This glosses over much – see SBCL sources for details.  
(setf (gf-dfun-state generic-function) (cons dfun cache))  
(set-funcallable-instance-function generic-function  
  (compute-discriminating-function generic-function))
```

This is prone to race conditions: if threads T1 and T2 happen to update the same generic function in an interleaved manner, the generic function may be left in an inconsistent state, where the discriminating function closes over state not accessible directly via the generic function.

The transient inconsistency in effect during the snippet above is to our current analysis relatively harmless: it will not cause misbehaviour, but may cause cycles to be wasted as operations need to be redone – and the situation will soon be rectified. The potential effects of the same inconsistency left in place for an unbounded time are more alarming.

To solve this issue we use a per generic function lock, to ensure that two threads do not try to update the same generic function in parallel. We also disable interrupts for the duration of the updating to ensure that it completes, and we will not be left with generic function in an inconsistent state.

This area is still in need of more work, however. Reverse engineering the assumed invariants in PCL is hard enough that our assumption of both the harmlessness of the transient inconsistency and the undesirability of the unbounded inconsistency are both in need of revision. The possibility of using COMPARE-AND-SWAP on the funcallable-instance-function to provide needed atomicity is also intriguing. (This confessional note is included because our dissatisfaction with the work done here so far has arisen only lately, and we have prior to this already boasted of having a thread and interrupt safe dispatch function updating working. We still believe that is correct, but proof and test-cases are sorely lacking.)

## **Adding and Removing Methods**

ADD-METHOD and REMOVE-METHOD are highly statefull operations. If they happen in parallel methods may be lost from the GENERIC-FUNCTION-METHODS list, or duplicates may be inserted. Similarly, if they are interrupted the generic function may be left in an inconsistent state, with a method incompletely removed or added.

To avoid these problems we utilize the same per generic function lock as dispatch function updating does, and disable interrupts for the sensitive parts.

No details are offered, as they are not very interesting, and quick perusal of SBCL sources will satisfy curiosity of those who despite all odds find themselves piqued.

## **Future Work**

In addition to the already mentioned issues there are several implementation paths worth exploring, especially in the memoization cache:

- The original PCL method dispatch paper notes that it is possible to obtain multiple hash values from a single one, by extracting only the required number of bits from it. This in turn can be used to reduce collisions when caches are expanded by trying multiple fields and selecting the one that yields the lest collisions. We have made two trials at this, but on both times there was no real benefit to be seen: in most cases the number of collisions was unaffected, and in some cases their number was even greater. Understanding why this is so would be desirable – or implementing this correctly if it turns out that theory is correct and the results were skewed by our tests, or if our initial implementations of this were incorrect.
- Instead of storing keys directly in the cache vector we would like to experiment with storing keys in lists or vector stored in the cache vector. This would waste less space for generic functions with many required arguments, whose caches can be relatively sparse. It would also allow cache writes to happen with just a single COMPARE-AND-SWAP, instead of one per key. The effect on memory locality is harder to predict: less wasted space improves the locality, but we would also be taking several more memory indirect, which may have an adverse effect.
- In addition to the chained-vectors approach mentioned earlier we would like to experiment with non-linear probing of the cache.

The other method dispatch strategies also need to be vetted for thread and interrupt safety, in particular the discrimination net method.

Thread safe class redefinitions are also something we would like to support, even if it means stopping the rest of the world while the update is in progress.

## Appendix: Linear Time Slot Definition Finding

Some, thankfully relatively rare, operations require the slot definition of a class to be accessed by name, which is in PCL performed by FIND-SLOT-DEFINITION.

The classic implementation of this was a linear search of CLASS-SLOTS list, which is not only  $O(N)$  in relation to the number of slots in the class, but also incurs a generic function call (or instance slot access) per slot definition looked at, as SLOT-DEFINITION-NAME must be obtained for each slot definition looked.

We have implemented  $O(1)$  FIND-SLOT-DEFINITION which also avoids the SLOT-DEFINITION-NAME access overhead by storing the slot definitions in a simple special purpose hash table, which is updated whenever CLASS-SLOTS change: we allocate a SIMPLE-VECTOR of  $2 + \text{number of slots elements}$ , and store the slot definitions in plists (using the slot names as indicators) stored in the index indicated by the SXHASH of the slot name modulo length of the vector.

Using a vector with two “extra” elements not only reduces number of collisions, but also allows slotless classes to be handled identically, with no need to check for an empty vector: the computed index will always hold a NIL, resulting in FIND-SLOT-DEFINITION returning NIL.

In our experiments this improves the performance of SLOT-VALUE with a variable slot name by approximately 50% even for classes with only a few slots – for classes with many slots the effect is even more pronounced.

## Thanks

We are indebted to the SBCL committers and #lisp regulars for advice, SBCL users for patience when bugs in the new cache implementation broke their code, Cliff Click for his comments on the cache write strategy and pointing out its needless complexity, Jeremy Brown for bringing the  $O(N)$  characteristic of FIND-SLOT-DEFINITION to our attention, and you our reader for bearing with our misuse of the third person pronoun.