CLOS Efficiency: Instantiation

On the Behavior and Performance of LISP, Part 2.1

Didier Verna

EPITA Research and Development Laboratory
14–16 rue Voltaire, 94276 Le Kremlin-Bicêtre, France
didier@lrde.epita.fr

Abstract

This article reports the results of an ongoing experimental research on the behavior and performance of CLOS, the COMMON-LISP Object System. Our purpose is to evaluate the behavior and performance of the 3 most important characteristics of any dynamic object-oriented system: class instantiation, slot access and dynamic dispatch. This paper describes the results of our experiments on instantiation. We evaluate the efficiency of the instantiation process in both C++ and LISP under a combination of parameters such as slot types or classes hierarchy. We show that in a non-optimized configuration where safety is given priority on speed, the behavior of C++ and LISP instantiation can be quite different, which is also the case amongst different LISP compilers. On the other hand, we demonstrate that when compilation is tuned for speed, instantiation in LISP becomes *faster* than in C++.

Keywords C++, LISP, Object Orientation, Performance, Instantiation

1. Introduction

More than 15 years after the standardization process of COMMON-LISP (Steele, 1990; ANSI, 1994), and more than 20 years after people really started to care about performance (Gabriel, 1985; Fateman et al., 1995; Reid, 1996), LISP still suffers from the legend that it is a slow language.

As a matter of fact, it is too rare to find efficiency demanding applications written in LISP. To take a single example from a scientific numerical calculus application domain, image processing libraries are mostly written in C (Froment, 2000) or in C++ with various degrees of genericity (Duret-Lutz, 2000). Most of the time, programmers are simply unaware of LISP, and in the best case, they falsely believe that sacrificing expressiveness will get them better performance.

We must admit however that this point of view is not totally unjustified. Recent studies (Neuss, 2003; Quam, 2005) on various numerical computation algorithms find that LISP code compiled with CMU-CL can run at 60% of the speed of equivalent C code. People having already made the choice of LISP for other reasons might call this "reasonable performance" (Boreczky and Rowe, 1994), but people coming from C or C++ will not: if you consider an image processing chain that requires 12 hours to complete (and this is a real case), running at a 60% speed means that you have just lost a day. This is unacceptable.

Hence, we have to do better: we have to show people that they would lose *nothing* in performance by using LISP, the corollary being that they would gain *a lot* in expressiveness.

This article is the second one in a series of experimental studies on the behavior and performance of LISP. In the first part (Verna, 2006), we provided a series of micro-benchmarks on several simple image processing algorithms, in order to evaluate the performance of pixel access and arithmetic operations in both C and LISP. We demonstrated that the behavior of equivalent LISP and C code is similar with respect to the choice of data structures and types, and also to external parameters such as hardware optimization. We further demonstrated that properly typed and optimized LISP code runs as fast as the equivalent C code, or even faster in some cases.

The second step in our experiments deals with CLOS (Keene, 1989), the object-oriented layer of COMMON-LISP. Our purpose is to evaluate the behavior and efficiency of instantiation, slot access and generic dispatch in general. This paper describes the results of our experiments on instantiation. Studies on both C++ via GCC and COMMON-LISP via 3 different compilers are presented, along with cross-comparisons of interesting configurations.

2. General Overview

This section gives a general overview of the conducted experiments. Although the terminology varies a great deal be-

tween languages, we choose to stick to the LISP jargon and only mention once each equivalent term used by the C++ community.

2.1 Protocol

The main purpose of our experiments is to test the behavior and performance of instantiation in as many configurations as possible. This has lead us to consider a number of parameters described below. Some of these parameters were not expected to have a major impact on performance *a priori*, but despite the combinatorial explosion of the test cases, we tried to remain neutral and avoid filtering out some of them based only on presumption.

2.1.1 Involved Parameters

Class Size Our experiments use 3 kinds of classes, containing respectively 1, 7 and 49 slots (rather called data members in C++). In addition, when the slots are not initialized (see below), an empty class is also used for timing reference.

Class Hierarchy For each class size N, the actual class(es) are constructed in 3 different ways, defined as follows.

- *Plain:* A single class containing all the N slots.
- *Vertical:* A hierarchy of $N + 1$ classes containing 1 slot each, and inheriting from one upper class at a time, the toplevel class being a class with no slot.
- *Horizontal:* A set of N simple classes containing 1 slot each, and a final class with no slot, inheriting (via multiple inheritance) directly from these N upper classes.

Slot Type Although we restricted to homogeneous classes (all slots are always of the same type), we tested both `int` / `float` slots in the C++ case, and `fixnum` / `single-float` ones in the LISP case.

Slot Allocation The experiments have been conducted with both kinds of slot allocation: local (instance-wide) and shared (class-wide) slots, rather called "static members" in C++.

Slot Initialization The configurations above are further decoupled according to whether we provide an initialization value for the slots, or whether we leave them uninitialized. See also section 2.1.2 for additional information.

Optimization Level Depending on the situation, our experiments are conducted in 2 or 3 possible optimization modes, called "Safe", "Optimized" and "Inline" respectively. The precise meaning of these modes will be described later.

Some additional parameters are involved in the LISP case only. We will describe them in the corresponding section. The combination of all these parameters amounts to more than 1300 individual test cases. The main difficulty of this work is not in generating the tests (as this can largely be automated), but in finding the interesting cross-comparison configurations. In this paper, we present only comparative results where we think there is a point to be made.

2.1.2 Notes on Slots Initialization

Slot initialization in CLOS can occur in different ways, including via `:initform` expressions (given at class definition time) and `:initarg` keyword arguments passed to `make-instance`. Keyword arguments processing is not specific to CLOS: it is a general feature of COMMON-LISP that can be very convenient but also quite time-consuming, especially in the case of `make-instance`. Since we are not interested in benchmarking this process, we chose to use only initforms for slot initialization.

The COMMON-LISP standard (ANSI, 1994) requires that an initform be evaluated every time it is used to initialize a slot, so the semantic equivalent in C++ is to provide a default constructor (with no argument) performing the initialization, which we did.

As far as shared slots are concerned, a C++ compiler is able to handle initialization at compile-time (provided, of course, that the value is known at compile-time) because static members are implemented as variables static to a compilation unit. In LISP, the situation is somewhat more complex: initforms are handled at run-time, but an efficient implementation may handle shared slots at class finalization time, or when the first instance of the class is created. Consequently, this semantic difference between the two languages shouldn't have any impact on the resulting measurements.

In our experiments, initializing a shared slot is done with a constant known at compile-time, although this constant is different for every slot. Until now, we did not experiment with initializing shared slots dynamically (at object instantiation time) and we currently don't know if there would be something to learn from that situation.

2.1.3 Note on Memory Management

Benchmarking millions of instantiations involves a great deal of memory management. It makes little sense, however, to include memory management timings in our benchmarks, because:

- this process is manual in C++ and automatic in LISP, thanks to garbage collection (GC),
- different LISP implementations have different garbage collectors, and thus can behave quite differently.

So again, we want to focus our study exclusively on the object-oriented layer. To this aim, the LISP timings reported in this paper include user and system run-time, but do not include GC time. All tested LISP implementations provide more or less convenient way to get that information via the standard `time` function. Some LISP implementations (ACL notably) have a very verbose garbage collector by default. Since we weren't sure of the impact of this on the benchmarks, care has also been taken to turn off GC output.

Note that excluding GC time from the benchmarks still leaves memory *allocation* in. Along with the same lines, our C++ code instantiates objects with the `new` operator (instead of putting them on the stack), and never `free`'s them. Our experiments on the C++ side have been preliminary calibrated to avoid filling up the whole memory, hence avoiding any swapping side-effect.

2.2 Experimental Conditions

The benchmarks have been generated on a Debian GNU/Linux system running a packaged `2.6.26-1-686` kernel version on an i686 Dual Core CPU, at 2.13GHz, with 2GB RAM and 2MB level 2 cache.

In order to avoid non deterministic operating system, hardware or program initialization side-effects as much as possible, the following precautions have been taken.

- The PC was rebooted in single-user mode, right before benchmarking.
- Each experiment was calibrated to take at least one or two seconds, and the results were normalized afterwards, for comparison purposes.

Given the inherent fuzzy and non-deterministic nature of the art of benchmarking, we are reluctant to provide precise numerical values. However such values are not really needed since the global shape of the comparative charts presented in this paper are usually more than sufficient to make some behavior perfectly clear. Nevertheless, people interested in the precise benchmarks we obtained can find the complete source code and results of our experiments at the author's website[1]. The reader should be aware that during the development phase of our experiments, several consecutive trial runs have demonstrated that timing differences of less than 10% are not really significant of anything.

3. C++ Object Instantiation

In this section, we establish the ground for comparison by studying the behavior and performance of object instantiation in C++.

3.1 C++ Specific Details

Some additional information on the C++ experiments are given below.

3.1.1 Compiler and Optimization Modes

For benchmarking the C++ programs, we used the GNU C++ compiler, GCC version 4.3.2 (Debian package version 4.3.2-1). Optimized mode is obtained with the `-O3` and `-DNDEBUG` flags.

3.1.2 `struct` *vs.* `class`

C++ provides both structures and classes. However, the only difference between the two lies in the member access policies: slots are public by default in structures and private in classes *etc.* Since these issues are resolved at compile-time, there is no point in studying both structures and classes. Consequently, the benchmarks presented below apply to both.

[1] `http://www.lrde.epita.fr/~didier/research/publis.php`

3.2 Experiments

3.2.1 Local Slots

Chart 1 presents the timing for the instantiation of 5,000,000 objects composed of local slots only. Timings are grouped by class or object size, as will be the case in all subsequents charts in this paper. Each group is further split by class hierarchy (plain, vertical and horizontal), and then by slot type (uninitialized, initialized `int` and `float` slots). This decomposition is recalled in the chart, on top of the first group for convenience. Timings in safe and optimized mode are superimposed.

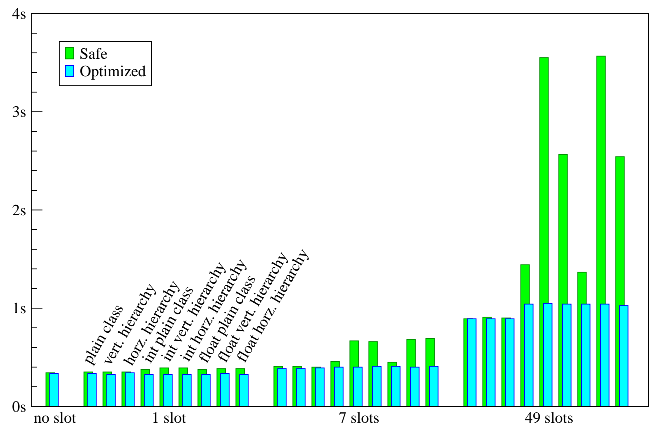

Chart 1. C++ class instantiation / 5,000,000 objects / local slots

Impact of slot type The first thing to notice is that regardless of the optimization mode, the performance seems to be immune to the slot type. This is not surprising from a weakly typed language in which native representations are always used. However, it is reasonable to expect something different from the Lisp side, where tagging / boxing might interfere.

Safe mode In safe mode, a remark must be made in relation to the size of the objects and the slot initialization policy. Starting at the 7 slots case, we begin to notice the impact of slot initialization on performance: horizontal and vertical hierarchies take about 70% more time to perform when initialized. In the 49 slots case, and by order of degradation, the cost of initialization is of 60% for plain classes, 180% for horizontal hierarchies and 290% for vertical ones.

Not only initialization has a cost, but this cost is different between the hierarchies themselves: in the 49 slots case, initializing an horizontal hierarchy takes about 180% of the time it takes to initialize a plain class, and this ratio amounts to 250% for vertical hierarchies.

A possible explanation for this behavior is as follows. Initializing slots in a plain class requires calling the only con-

structor which will initialize the slots sequentially, without consuming much stack. In the case of multiple inheritance (the horizontal hierarchy), the constructor has to call all the superclasses constructors, so this leads to more calls, but this still happens sequentially. In the case of a vertical hierarchy however, we end up with a recursive chain of constructors, each one of them calling the upper one, from the direct superclass. This is a stack consuming process and is likely to involve more work from the MMU.

At the time of this writing, we did not objectively confirmed this hypothesis, although it is the most likely explanation.

Optimized mode In optimized mode, we get a much more uniform shape. The cost of instantiation increases with the object size (it roughly doubles between the 7 and the 49 slots cases), but this cost depends neither on the slot type, nor on the class hierarchy. Besides, even for big objects (the 49 slots case), the cost of initialization is just about 10%, which is not very important.

In all, this demonstrates that when the code is optimized, the cost of instantiation depends almost only on the object size, hence on memory management issues. In particular, the chain of constructor calls involved in the slots initialization process is likely to be completely flattened, explaining why there is no more difference across different hierarchies.

Cross-comparison The superimposition of safe and optimized charts demonstrates that there is little to be gained by optimizing on small classes, and that, as a matter of fact, optimization deals mostly with initialization. Even for big objects, the gain is null for uninitialized slots. On the other hand, it can reach 70% in the case of an initialized vertical hierarchy.

The fact that there is little to be gained by optimizing on small classes is further proof that memory management is the most time-consuming aspect of instantiation, and at the same time, that it is the least subject to optimization. When there are only a few slots to initialize, optimization still occurs, but the initialization time is negligible in front of memory allocation, so the effect of that optimization is completely invisible.

3.2.2 Shared Slots

Chart 2 presents the timings for the instantiation of 5,000,000 objects composed of shared slots only this time. This chart is organized in the same way as the previous one.

This chart has very little to tell, if anything at all. All our experimental parameters typically boil down to the same performance, optimization still having a very small impact (too small to be really conclusive). This result comes as no surprise, because the previous chart demonstrated that optimization occurs mostly on slot initialization, and shared slots are initialized at compile-time in our experiments.

Perhaps the only merit of this chart is to confirm that objects in C++ are represented “sensibly”, meaning that the

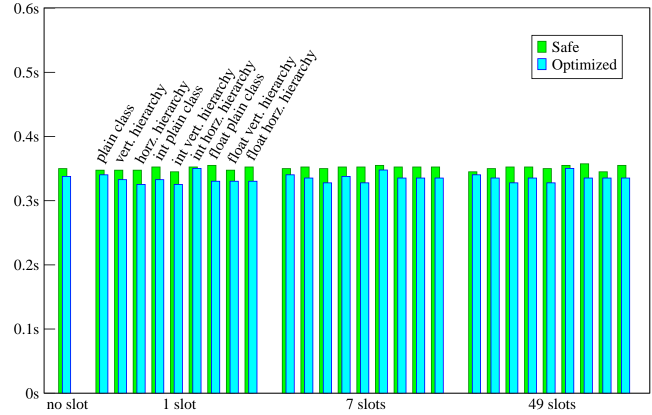

Chart 2. C++ class instantiation / 5,000,000 objects / shared slots

amount of shared information does not affect their size: the instantiation time is here a constant, regardless of the number of shared slots in the class.

3.2.3 Optimized Mode

One final cross-comparison for C++ is presented in chart 3. This time, benchmarks for both local and shared slots in optimized mode are superimposed. A similar chart exists for the safe mode but it is not presented here.

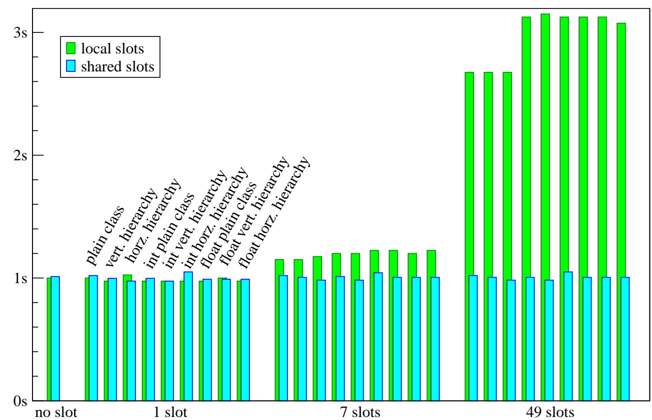

Chart 3. C++ class instantiation / 15,000,000 objects / optimized mode

This chart does not contain any new information in itself, but instead gives us a clearer view on the overall cost of instantiation: it shows that in our experiments, we could expect the creation of at most 15,000,000 objects per second, and around 5,000,000 for our bigger classes.

3.3 Intermediate Conclusion

We can summarize what we have learned so far as follows:

- The slot type has no impact on the performance of object instantiation.
- In safe mode, both initialization and class hierarchy can have an important cost.

- In optimized mode, slot initialization still has a cost (although greatly reduced), but the class hierarchy does not count anymore.
- Optimization deals mainly with slot initialization and can have a non negligible impact on performance.
- Not much can be done in terms of optimization on the memory allocation part of the instantiation process.

Given all these observations, from now on we will drop the use of different slot types when referring to the C++ benchmarks, and only use `float` slots. In addition, we will also drop the use of different class hierarchies in the context of optimization, and stick to plain classes.

4. LISP

In this section, we study the behavior and performance of object instantiation in LISP, and establish a number of comparisons with the C++ experiments described in the previous section.

4.1 LISP Specific Details

Some additional information on the LISP experiments are given below.

4.1.1 Compilers

When benchmarking the LISP programs, we took the opportunity to try several different compilers. We tested CMU-CL (version 19d as packaged in the Debian unstable distribution), SBCL (version 1.0.22.17) and ACL (Allegro 8.1 express edition). Our desire to add LispWorks to the benchmarks was foiled by the fact that the personal edition lacks the usual `load` and `eval` command line options used with the other compilers to automate the tests.

4.1.2 Optimization Modes

For the reader unfamiliar with LISP, it should be mentioned that requesting optimization is not achieved by passing flags to the compiler as in C++, but by "annotating" the source code directly with so-called *declarations*, specifying the expected type of variables, the required level optimization, and even compiler-specific information.

Contrary to our previous experiments (Verna, 2006), type declarations are localized around slot definitions in the code because all we do is typically benchmark calls to `make-instance`. The `optimize` declaration has also its importance: safe mode is obtained with `(safety 3)` and all other qualities set to 0, while optimized mode is obtained with `(speed 3)`.

In addition to the safe and optimized modes (that we also used in the C++ experiments), there is the opportunity for another one in LISP. While instantiation is a *syntactic* construct in C++, it involves a *function call* in LISP, in which the class to instantiate can be parametrized or not. As a consequence, in optimized mode, we use parametrized calls to `make-instance`, as in:

```
(make-instance var)
```

where `var` in turn contains a symbol naming the class to instantiate. We also define a new optimization mode, called "Inline", which still has a maximum `speed` quality, and in which the class to instantiate is directly inlined into the call to make-instance, like this:

```
(make-instance 'myclass)
```

Until now, we did not experiment with the "hybrid" mode consisting of safe quality settings *and* inline calls to `make-instance`, and we currently don't know if there would be something to learn from that situation.

Note that the names we use when referring to our different optimization levels are directly inspired from the LISP optimization qualities. They should not be taken too seriously however. As we will see later on, when it comes to type checking, the "safe" mode is not particularly safe...

4.1.3 Additional Parameters

In the LISP case, our experiments have been further combined with the following 2 additional parameters.

Structure vs. class Contrary to the C++ case, it *is* interesting to benchmark both structure and class instantiation. COMMON-LISP structures provide a restricted form of object orientation (limited inheritance capabilities *etc.*) and because of these limitations, can usually be implemented more efficiently than CLOS objects (typically as vectors). If one does not need the full capabilities of CLOS, it is a good idea to use structures instead, and consequently, it is interesting to know exactly what to expect in terms of performance.

Our LISP experiments will thus start with structures, and will then continue with classes.

Meta-class vs. standard class Another difference with the case of C++ is that CLOS itself may be architectured according to a *Meta Object Protocol*, simply known as the CLOS MOP Paepcke (1993); Kiczales et al. (1991). Although not part of the ANSI specification, the CLOS MOP is a *de facto* standard, supported to various degrees by many COMMON-LISP implementations. Through the MOP, CLOS elements are themselves modeled in an object-oriented fashion (one begins to perceive here the reflexive nature of CLOS). For instance, classes (the result of calling `defclass`) are CLOS (meta-)objects that are instances of other (meta-)classes. By default, new classes are instances of the class `standard-class`, but the programmer can change that.

Since it might be interesting to study the impact of the class definition itself on the performance of instantiation, we provide benchmarks for instantiating standard classes as well as classes based on a user-defined meta-class.

4.2 Structures

In this section, we report the results of instantiating COMMON-LISP structures. The number of experimental parameters is

less important on structures than on classes for the following 3 reasons.

- Firstly, structures support only a limited form of *single* inheritance (they can inherit from only one other superstructure at a time), so our experiments involve only plain structures and vertical hierarchies.
- Secondly, structures do not support the notion of shared slot. All structure slots are hence local to each instance.
- Thirdly, defining a structure in COMMON-LISP involves the creation of a constructor function make-<struct> which is already specific to the structure in question. In other words, there is no equivalent of make-instance for structures, in which the structure name would be parametrized. Consequently, we only have 2 optimization modes: safe and inline.

4.2.1 SBCL

Chart 4 presents the timing for the instantiation of 10,000,000 objects in SBCL. The organization of this chart is the same as before, and is recalled directly on top of the first group of benchmarks for convenience.

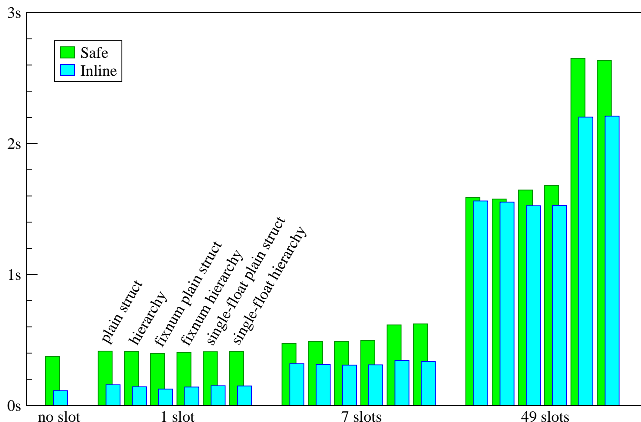

Chart 4. SBCL structure instantiation / 10,000,000 objects

Impact of slot type The first thing to remark here is that contrary to the C++ case (chart 1 on page 3), the performance depends on the slot type, which is particularly explicit in the 49 slots case: instantiating single-float rather than fixnum structures takes between 40 and 60% more time according to the optimization mode. A possible explanation for this behavior is that SBCL stores fixnums as-is in structures, but uses a specialized representation for single floats. Fixnums are already tagged, immediate objects so storing them in structures does not require any internal representation format modification. On the other, initializing a slot with a single-float would require unboxing it. This hypothesis has no been verified yet.

Impact of slot initialization Other than the case of single floats, another difference with the C++ case is striking: there is no difference between initializing fixnum slots and leaving the slots uninitialized. This is even true in both safe and inline mode. The explanation for this lies in the fact that in all tested LISP implementations, an uninitialized slot is actually initialized to nil, which has the same cost as using a constant fixnum value. Technically, this behavior is not required by the standard, which specifies that accessing an otherwise uninitialized slot has undefined consequences. However, the practical consequence here is that structure slots are *always* initialized to something.

Impact of structure hierarchy Either in optimized or inline mode, it is clear on chart 4 that using a plain structure or a vertical hierarchy has no impact on the performance whatsoever. This is not surprising because under the general assumption that structures behave like vectors, the initial structure definition(s) should not have an impact on their final representation: once a structure is defined, there is no trace left of the inheritance property used in its definition. As we saw in section 3.2.1 on page 3, this result is again different from the case of C++.

Impact of optimization One last difference with the C++ case is the impact of optimization on small objects. Whereas this impact was null in the C++ case, even on LISP structures as small as with only one slot, optimization more than doubles the performance.

As usual in LISP, a difference of performance can come from both directions: either something is done in optimized mode to improve the performance, or something is done in safe mode that actually slows things down. In that case, a probable contributor to the timing difference is that SBCL performs type checking on slot initial values in safe mode. Again, note that this is not required by the standard, but this is a general policy of this compiler. On the other hand, in inline mode, SBCL will trust the type declarations, effectively bypassing type checking.

Impact of structure size A final remark, more in line with the behavior of C++ this time, is that the impact of optimization decreases as the size of the instantiated objects increases. While this gain is around 200% for small structures, it hardly reaches 20% in the 49 single-float slots case (it is practically null for fixnum slots). Again, this demonstrates that memory management is the most time-consuming aspect of instantiation, and at the same time, that it is the least subject to optimization.

4.2.2 CMU-CL

Chart 5 on the facing page is the equivalent of chart 4 for CMU-CL. As in the case of SBCL, we see that there is not much difference between uninitialized slots and initialized fixnum ones, and that using single-float slots gives a different shape to the timings.

The behavior of CMU-CL differs from that of SBCL in two important ways however:

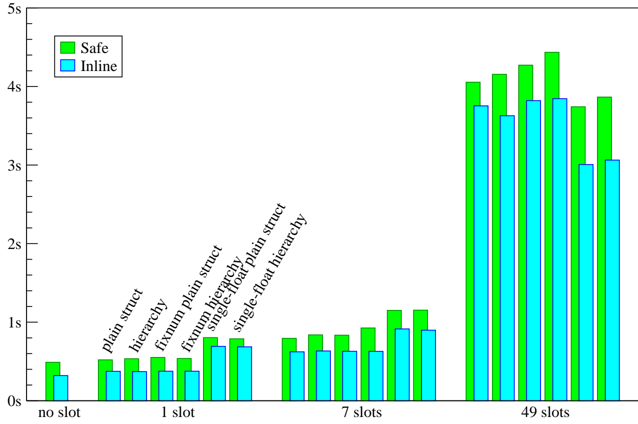

Chart 5. CMU-CL structure instantiation / 10,000,000 objects

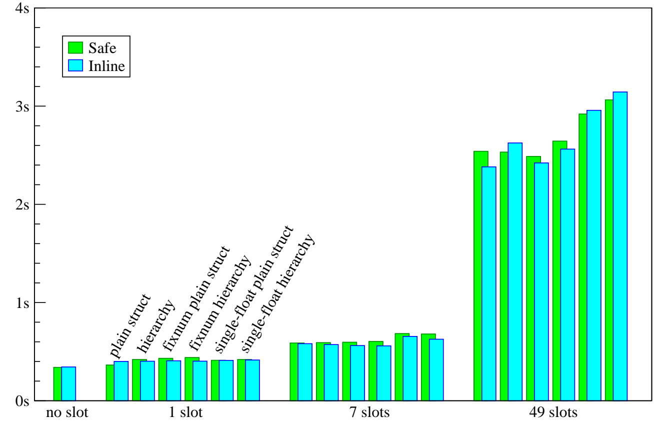

Chart 6. ACL structure instantiation / 10,000,000 objects

- Particularly for small structures and `single-float` slots, it is visible that the impact of optimization is less important in CMU-CL than in SBCL. This can be explained by the fact that CMU-CL happens to perform type checking on the slot initialization values *all the time*, regardless of the optimization mode. So in inline mode, it does *more* than SBCL.
- In the 49 slots case however, we see that the instantiation of `single-float` structures is actually *more* efficient, in both optimization modes, than instantiating `fixnum` ones. This behavior, opposite to that of SBCL, is very surprising and currently unexplained. In fact, it is all the more surprising that we later discovered that CMU-CL actually does *not* perform type checking on `fixnum` slots. We consider this as an inconsistency in CMU-CL's behavior, and we have reported it to the development team (a fix is underway). The most probable explanation for this is that contrary to `single-float` slots, `fixnum` slots do not have a specialized representation in structures, so the need for type checking may not be as critical as for objects requiring (un)boxing.

4.2.3 ACL

Chart 6 is the equivalent of charts 4 on the preceding page and 5 for ACL.

There is not much news to deduce from this chart, apart from a couple of remarks:

- For big structures, ACL (as CMU-CL) seems to be slightly sensitive to the hierarchy: plain structures perform a bit faster. However, the difference in performance is definitely too small to be conclusive.
- Contrary to both SBCL and CMU-CL, there is absolutely no gain in using optimized compilation settings. As a matter of fact, it turns out that ACL does not perform any type checking on the slot initialization values *at all* (a behavior that is perfectly conformant to the standard).

  Consequently, and this is confirmed by one of the ACL maintainers, it is not the case that nothing is done to optimize structures. It is rather the case that even in safe mode, structures are already considerably optimized.

4.2.4 Cross-Implementation Comparisons

We end our study on structures by providing a cross-comparison of the 3 tested COMMON-LISP implementations. Charts 7 and 8 on the following page regroup the timings of all 3 compilers respectively for safe and inline mode.

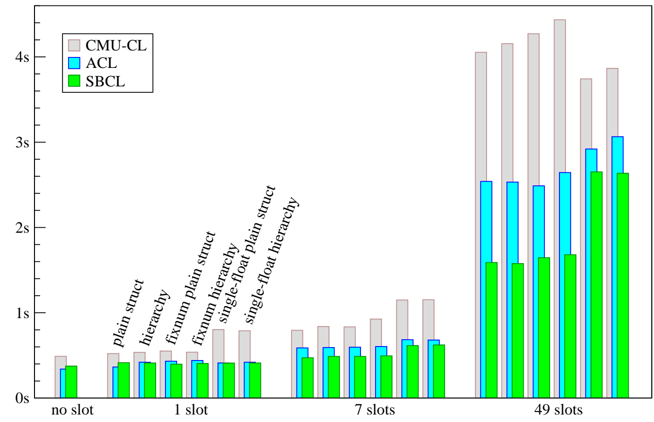

Chart 7. LISP structure instantiation / 10,000,000 objects, safe mode

In safe mode, the performance differs more and more as the size of the instantiated objects increases: for small structures, all 3 implementations are relatively close to each other, apart maybe from the handling of `single-float` slots by CMU-CL. For large structures, SBCL runs noticeably faster than the 2 other ones, especially in the cases of uninitialized and initialized `fixnum` slots: the gain is around 30% compared to ACL and 70% compared to CMU-CL. This performance is all the more commendable that SBCL is the only one to perform type checking in that situation.

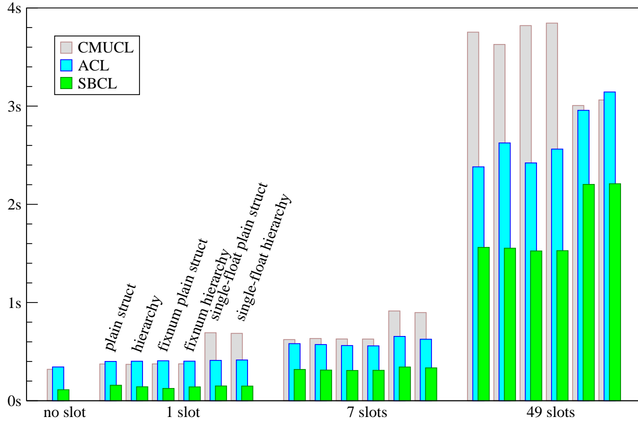

Chart 8. LISP structure instantiation / 10,000,000 objects, inline mode

In inline mode, we get roughly the same shape and so the same comments still apply, with the additional remark that the performance of SBCL becomes noticeably better *even* on smaller structures: it runs roughly twice as fast than both ACL and CMU-CL.

4.2.5 Intermediate Conclusion

We can summarize what we have learned so far as follows:

- Contrary to the C++ case, the slot type has an impact on the performance, both in safe and inline mode, and this impact becomes more and more important as the number of slots in the objects increases. This result is not surprising, although CMU-CL displays a counter-intuitive behavior that remains to be explained.
- Contrary to the C++ case, all tested implementations *do* initialize structure slots to something (be it `nil` or a user provided value), something that is not required by the standard. The consequence is that there is no timing difference between uninitialized slots and initialized `fixnum` slots.
- Contrary to the C++ case, the structure hierarchy has little or no impact on performance.
- Our experiments also lead us to discover that these 3 LISP implementations exhibit very different behaviors with respect to type checking of slot initialization values: SBCL honors his traditional philosophy of performing type checks in safe mode and treating type declarations as assertions in optimized mode; CMU-CL always perform type checking, with the notable and somewhat inconsistent exception of `fixnum` slots; ACL never performs any form of type checking on structure slots.

Given all these observations, from now on we will refer to LISP structure instantiation benchmarks as restricted to plain ones and drop the use of a vertical hierarchy. We will on the other hand continue to present both `fixnum` and `single-float` slot types.

4.3 Classes

In this section, we report the results of instantiating COMMON-LISP classes with the full set of parameters described in sections 2 on page 1 and 4.1 on page 5.

4.3.1 SBCL

Chart 9 presents the timings for the instantiation of 5,000,000 objects in SBCL. These timings correspond to `standard-class` based classes containing local slots only. The organization of the other parameters on this chart is the same as before, and is recalled directly on top of the first group of benchmarks for convenience. This time, the 3 optimization modes do exist and are superimposed. Beware that we use a logarithmic scale on this chart.

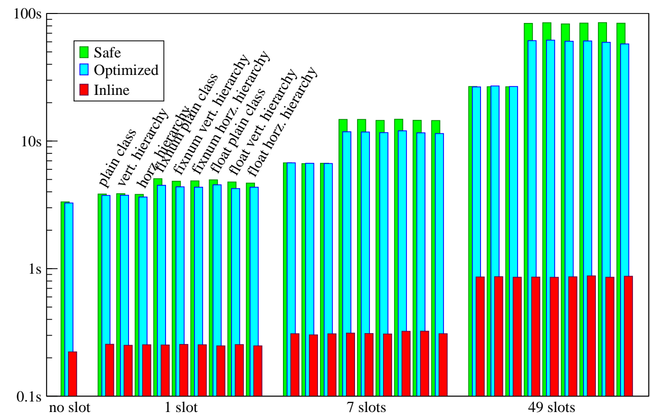

Chart 9. SBCL class instantiation / 5,000,000 objects, standard class, local slots

Impact of slot type and class hierarchy The first result deducible from this chart is that neither the slot type nor the class hierarchy have an impact on the performance. This can be explained by the facts that

- given the dynamic capabilities of CLOS, slots are unlikely to benefit from a specialized representation (they are typically stored in a simple vector),
- and similarly, accessing a slot from an instance vector completely blurs the original class hierarchy.

Impact of slot initialization It should be noted that as for structures (but this time this is a requirement), a slot left uninitialized by the programmer will actually be initialized to the “unbound value” by a conformant LISP implementation. So slot initialization *does* occur every time. Despite this fact, we can see that in both safe and optimized mode, user-level slot initialization has a considerable impact on performance. In the 7 slots case, initializing the slots doubles the instantiation time, and in the 49 slots case, instantiation is typically 3 times slower. We saw earlier that type checking can amount to a certain part of instantiation time, but that this time becomes negligible compared to memory allocation and reference on big objects. Besides, we know that

SBCL does not perform type checking on class slots in optimized mode. So type checking cannot explain our current observation. The real explanation is that in order to initialize a slot, one has to access it; and it happens that slot access is not optimized at all in either safe or optimized mode. This explanation is further confirmed by the next observation below.

Impact of optimization modes Going from safe to optimized mode does bring a bit to the performance of instantiation, but does not change the overall shape of our diagrams. Although the gain in question is null when slots are left uninitialized, we save 20% of the time in the 7 slots case, and up to 30% in the 49 slots case. An important part of this gain can be attributed to the fact that slot initform type checking is turned off by SBCL in optimized mode. In itself, this is already a non negligible achievement, but nothing compared to what we gain in inline mode.

There, the situation is completely different. First of all, the improvement is extremely important. Even on an empty class, instantiation already performs about 15 times faster in inline mode than in safe mode. In the 49 slots case, instantiation performs almost 100 times faster.

The second remark with inline mode is that there is no more difference of performance according to whether the slots are initialized or not. The cost of initialization is completely gone. This comes as a confirmation to what we said in the previous paragraph: as soon as the class to instantiate is known, or at least can be inferred at compile-time, a great deal of optimization becomes possible, in particular with respect to slot access, and this in turn renders the cost of initialization negligible.

Shared slots Chart 10 presents timings similar to those in chart 9 on the preceding page, but for shared slots this time. As we can see, going from safe to optimized mode has practically no impact on performance. Compared to the equivalent C++ chart (chart 2 on page 4) however, this chart presents two surprising differences.

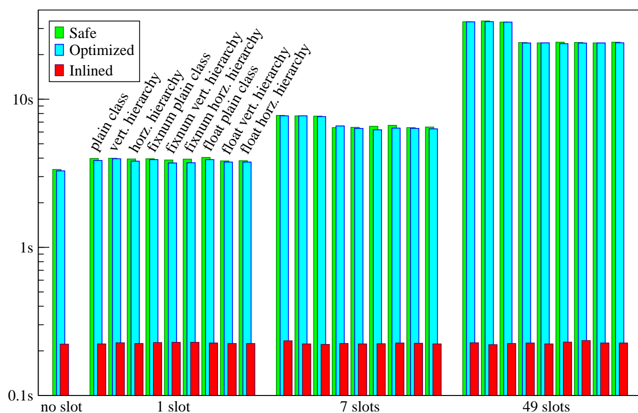

Chart 10. SBCL class instantiation / 5,000,000 objects, standard class, shared slots

- Firstly, the number of slots has a noticeable impact on performance, even though they are all shared. Instantiating a 49 shared slots class can be up to 10 times slower than instantiating an empty or 1 shared slot class.
- Secondly, it is visible that initializing shared slots actually *improves* the performance of instantiation. In the 49 slots case, leaving the slots uninitialized leads of a performance loss of 30%.

These results are both counter-intuitive and a sign that something is wrong in the instantiation process. A possible explanation, suggested by one of the SBCL maintainers, is an incorrect behavior of `shared-initialize`, attempting to pointlessly initialize unbound shared slots from the initform. This issue is normally fixed in version 1.0.23.36 of SBCL, although we didn't have time to confirm the hypothesis yet. Our attempt to analyze this odd behavior also lead us to discover that SBCL did not perform type checking on shared slots initforms, even in safe mode. Although this is not technically required by the standard, this is definitely an inconsistency with respect to SBCL's own policy with respect to type checking, and this has also been fixed recently.

As far as inline mode is concerned, we get a much more coherent and expected result: as with the C++ case, the performance of instantiation basically remains the same whatever the configuration, and the inlined version can run up to 150 times faster than the safe one in the 49 slots case.

Impact of a meta-class We have generated diagrams similar to those presented in charts 9 on the preceding page and 10 in the case where the instantiated classes are provided with a user-defined meta-class through the `:metaclass` option to `defclass`. The timings are rigorously equivalent so they are not presented here. We were (maybe a bit naively) expecting a different outcome, so it is now envisioned that our specific meta-class (an empty class inheriting from `standard-class`) was perhaps too simple to demonstrate anything. On the other hand, such a simple setting might still have a noticeable impact on slot access, if not on class instantiation. This will be investigated in future work.

Cross-comparison So far, we are able to drop the use of 3 parameters previously taken into account in the case of SBCL, because we have seen that they have no impact on performance. These parameters are: slot type, class hierarchy and the use of a user-defined meta-class. From now on, we will only present plain classes with `single-float` slots, either initialized or not. A final cross-comparison of all these SBCL cases is provided in chart 11 on the next page.

This chart has the merit of letting us visually summarize what we have seen so far as having an impact on performance:

- User-level slot initialization has a considerable impact on performance. In the 7 slots case, initializing the slots doubles the instantiation time, and in the 49 slots case, instantiation is typically 3 times slower.

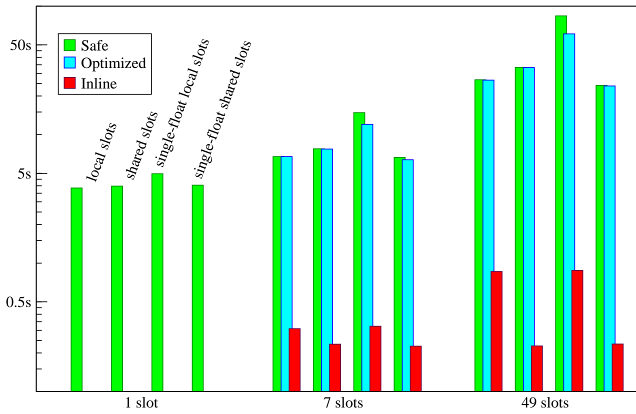

Chart 11. SBCL class instantiation / 5,000,000 objects

- Going from safe to optimized mode has a reasonable impact on performance when the slots are local, and none when they are shared. On local slots, we save 20% of the time in the 7 slots case, and up to 30% in the 49 slots case. This gain can be attributed to the fact that slot initform type checking is turned off by SBCL in optimized mode.
- Going from optimized to inline mode has a tremendous impact on performance for local slots as well as for shared ones. An empty class will be instantiated 15 times faster, and in the 49 slots cases, this can be a hundred times faster. Besides, the impact of slot initialization completely disappears.

4.3.2 CMU-CL

Chart 12 presents the timings for the instantiation of 5,000,000 objects in CMU-CL. These timings correspond to `standard-class` based classes containing local slots only.

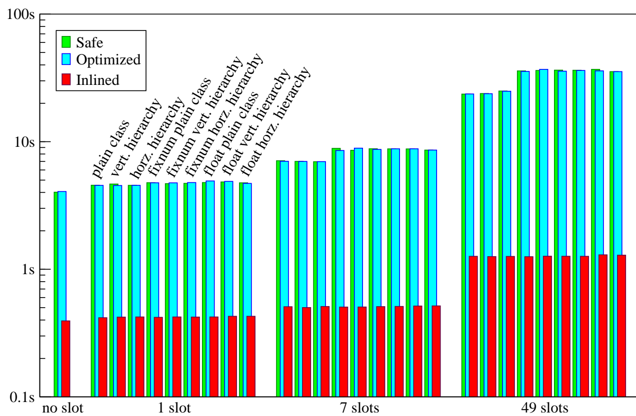

Chart 12. CMU-CL class instantiation / 5,000,000 objects, standard class, local slots

This chart is almost exactly identical to that of SBCL (chart 9 on page 8) so the same remarks apply and we will not reiterate them here. Only one small difference should be emphasized however: contrary to the case of SBCL going from safe to optimized mode does not bring any improvement to the performance. Remember that for SBCL, we observed a gain ranging from 20 to 30% when instantiating classes for slot initforms. This gain was attributed to SBCL turning off initform slot type checking in optimized mode, which the case of CMU-CL helps confirming. Indeed, our investigation lead us to discover that contrary to what the manual says, CMU-CL does *not* perform initform slot type checking in safe mode. As a result, there is nothing different in its behavior from safe to optimized mode. Again, this behavior is authorized by the standard, but still considered at least as an inconsistency, and a fix is currently underway.

The other diagrams for CMU-CL are so close to the SBCL ones that it is not worth putting them here. All our previous observations still apply. We will instead directly jump to the cross-comparison (chart 13), which this time has something new to offer.

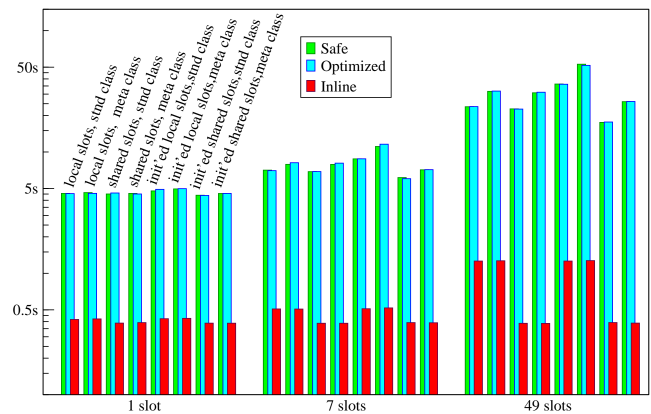

Chart 13. CMU-CL class instantiation / 5,000,000 objects

This chart shows that, contrary to the case of SBCL the use of a custom meta-class (be it an empty one) has some impact on the performance in both safe and optimized mode (but remember that there is no real difference in CMU-CL between these two modes). For instance, in the 49 slots cases, using a custom meta-class leads to a degradation in performance varying between 30 and 50%. The reason for this behavior is currently unknown.

In further charts, CMU-CL timings will be filtered as SBCL ones, except for the fact that we will retain the cases with and without a custom meta-class.

4.3.3 ACL

Chart 14 on the next page presents the timings for the instantiation of 5,000,000 objects in ACL. These timings correspond to `standard-class` based classes containing local slots only.

This chart looks pretty similar to that of SBCL (chart 9 on page 8) and CMU-CL (chart 12). In particular, as for CMU-CL, we can observe that there is no difference between the performance of safe and optimized modes. It turns out that ACL *never* performs slot initform type checking, even in safe mode. Although in CMU-CL, that was considered a bug or at

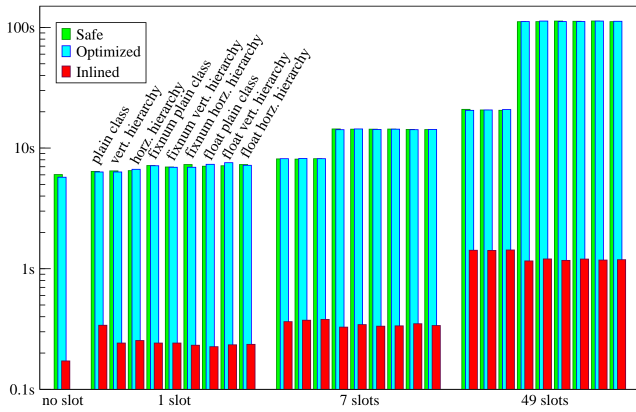

Chart 14. ACL class instantiation / 5,000,000 objects, standard class, local slots

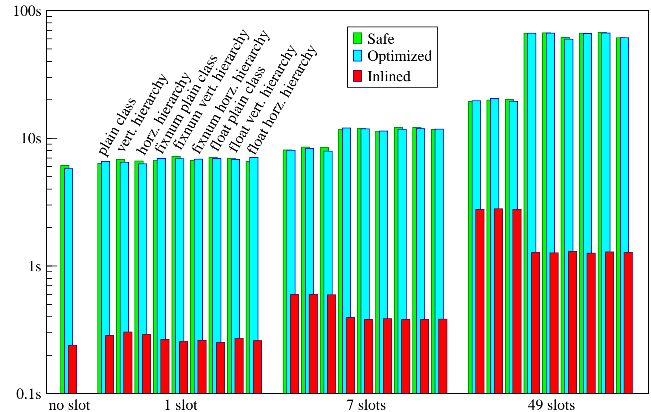

Chart 15. ACL class instantiation / 5,000,000 objects, standard class, shared slots

least an inconsistency, let us repeat that the standard leaves much freedom to the implementation in that matter.

Some differences with the other implementations are worth mentioning however.

- In safe and/or optimized mode, the cost of initialization seems to be higher than in SBCL or CMU-CL. In the 49 slots case, instantiation was 3 times slower with SBCL, 1.5 times slower with CMU-CL, but 5 times slower here. Given the fact that neither CMU-CL nor ACL perform initform slot type checking, the difference is striking and might be grounded into very different schemes for slot access.
- In inline mode, there is something specific to ACL to observe: slot initialization seems to actually *improve* the performance by a small factor: 10% in the 7 slots case and 20% in the 49 slots one. It is possible that ACL has a better optimized way to assign an explicit value from an initform than from the secret "unbound" value, although that hypothesis remains to be confirmed.
- On last remark, of less importance, is that the timings for inline instantiation look a bit fluctuating, especially for smaller classes. This might suggest some impact of the class hierarchy on performance, but the actual numbers are too small to be conclusive.

Shared slots Chart 15 presents timings similar to those in chart 14, but for shared slots this time. This chart is the ACL counterpart of char 10 on page 9 for SBCL, and exhibits a quite different behavior.

- In safe and optimized mode (which unsurprisingly continue to perform equally), initializing the slots degrades the performance by an important factor (3.5 in the 49 slots case).
- In inline mode, initializing the slots now improves the performance by a non negligible factor as well (more than 2 in the 49 slots case).
- Finally, and contrary to the 2 other implementations that we tested, the number of slots has a very important impact on the performance of instantiation, even in inline mode. From the no slot to the 49 slots case, the performance degrades by more than a factor of 10.

Given the fact that shared slots are (or at least can be) normally handled only once (at class finalization time or when the first instance of the class is created), these observations are quite surprising. A possibly explanation will arise from the next chart.

Cross-comparison Just before introducing this new chart, let us mention that the other diagrams for ACL, not displayed in this paper, exhibit a behavior very similar to the other 2 compilers. Most notably, ACL seems immune to the slot type, the class hierarchy or the presence of a custom metaclass. From now on, we will thus drop these parameters and keep plain, standard classes with `single-float` slots.

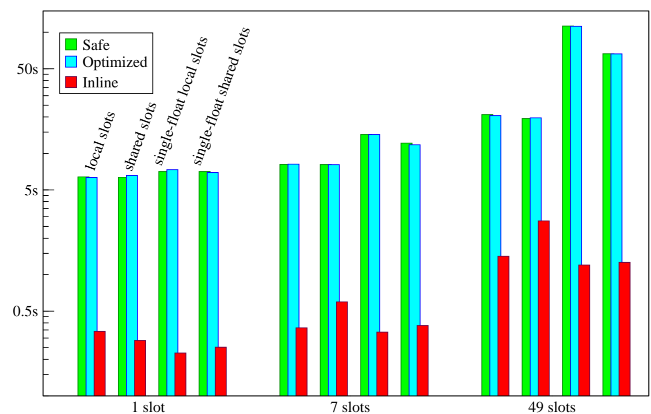

Chart 16. ACL class instantiation / 5,000,000 objects

A final cross-comparison of all the interesting ACL cases is provided in chart 16. This chart, although similar to the SBCL equivalent (chart 11 on the facing page) exhibits one striking difference. In inline mode, instantiating shared slots

do not perform faster (and at a constant speed) than instantiating local slots. It even performs noticeably slower with uninitialized slots in the 7 and 49 slots cases.

One strong hypothesis arises from this, and might as well explain the oddities we mentioned in the previous section. It is highly probable that nothing special is done in ACL to optimize the handling of shared slots at all. If confirmed, this would actually not be very surprising because the use of shared slots in real applications is probably quite marginal, so nobody really cares about their performance. In addition, it is our experience with ACL that improvements to the compiler are mostly driven by customer demands.

4.4 Cross-Implementation Comparison

In this section, we present a cross-comparison of the performance of instantiation with the 3 tested implementations of COMMON-LISP. Because of the combinatorial explosion of the parameters involved, a complete diagram would be unreadable. Instead, we recall here the parameters that we were able to drop because their influence on the performance is either null, or negligible, and we present "filtered" charts only.

For structures:

- The slot type has some influence, so we keep both the `fixnum` and `single-float` cases.
- However, leaving structure slots uninitialized takes the same time as initializing them with `fixnum` values, so we will drop the case of uninitialized structures.
- The structure hierarchy never has any impact, so we will stick to plain structures and drop the vertical hierarchy.

For classes:

- Performance is immune to the slot type, so we will stick to `single-float` ones.
- Performance is immune to the class hierarchy and to the use of a custom meta-class (still, with small exceptions in the cases of CMU-CL and ACL), so we will stick to standard, plain classes.
- On the other hand, we preserve both the cases of initialized / uninitialized slots, and local / shared slot storage.

4.4.1 Safe Mode

Chart 17 presents the filtered representation described above in safe mode. We see that all tested implementations present a similar shape, although some performance variations are to be expected. Structure instantiation performs faster than class instantiation. According to the object size, the gain may vary from a factor of 1.7 up to a factor of 30. Note that the performance of SBCL is commendable because it is the only one (at least in the tested versions) to perform type checking on `fixnum` local slot initforms.

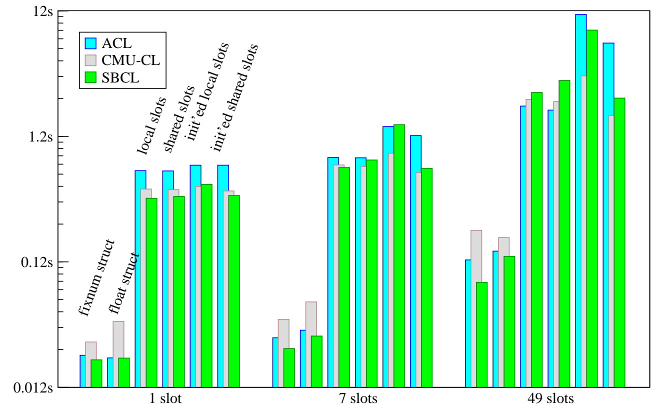

Chart 17. LISP instantiation / 500,000 objects / safe mode

4.4.2 Inline Mode

As we saw at different places in this paper, optimized mode does not bring much to performance, so we will skip the corresponding chart and directly switch to the inline mode, as displayed by chart 18.

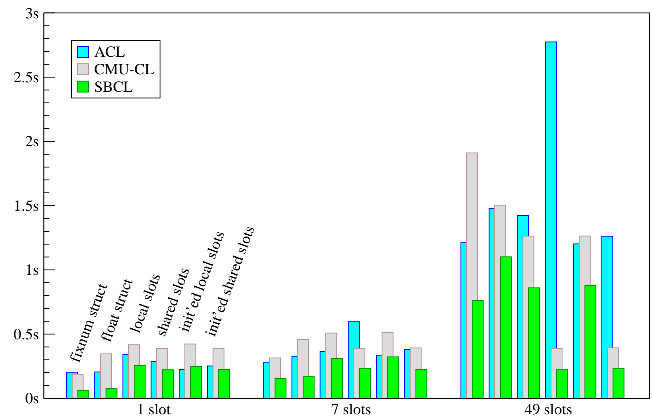

Chart 18. LISP instantiation / 5,000,000 objects / inline mode

With the notable exception of one-slot structures, SBCL and CMU-CL exhibit pretty similar shapes, which is perhaps not surprising given the fact that they both come from the same original compiler (MacLachlan, 1992). As already observed in (Verna, 2006), SBCL is still the best candidate in terms of performance. Compared to CMU-CL and depending on the actual parameters configuration, SBCL can perform between 1.6 and 2.6 times faster. Apart from the notable exception of shared slots handling, ACL is usually but not always faster than CMU-CL.

Perhaps the most interesting remark that can be made out of this chart is the comparative performance of structure *vs.* class instantiation. If we consider the case of SBCL, we can observe that the timing difference decreases when the object grows in size. In the 7 slots case, instantiating a `single-float` class takes twice the time needed for an equivalent structure. In the 49 slots case, instantiating the

structure becomes *slower*, by a factor of 1.2. As for the 2 other compilers, which perform a bit less efficiently, these differences are even more flattened. With CMU-CL for instance, the timing difference between structures and classes instantiation do not differ by more than a factor of 1.2.

Given the additional expressiveness that CLOS has to offer compared to structures, we consider it a considerable achievement from all tested compilers that the timings for class instantiation can be so close to the ones for structures.

5. Cross-Language Comparison

We now end our study by providing a cross-language comparison of the instantiation process in all interesting cases. Given the fact that we are targeting our experiments on performance issues, we skip the cross-comparisons of safe and optimized mode, and switch directly to inline mode. It would not make any sense to present comparisons of timings in safe mode anyway, since the behavior in both languages is so different in that case.

In the C++ case, we use plain classes with `float` slots. This choice is legitimate because we know that the slot type and class hierarchy does not influence the performances in optimized mode.

In the LISP case, we use the same filters as before, that is, we retain `fixnum` and `single-float` plain structures, and `single-float` plain classes. The LISP timings are those of SBCL because it is the most efficient. Since `single-float` handling has been observed to be slower in some cases, one can consider that we are using the "worst of the best" scenario from the LISP side.

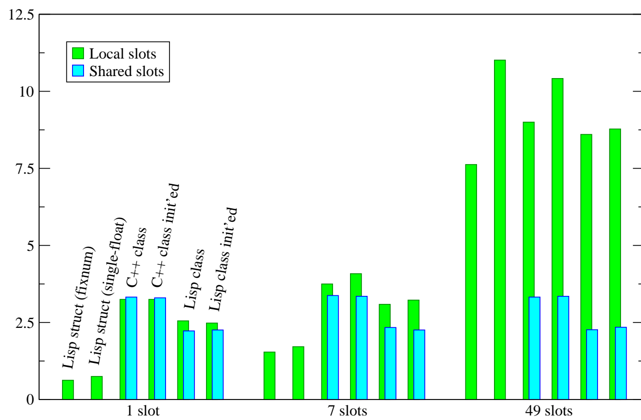

Chart 19. Object Instantiation / 5,000,000 objects / inline mode

Chart 19 regroups all the timings described above and comes with good news for the LISP community. In all considered cases, object instantiation in LISP is *faster* than in C++.

The comparison of LISP structures with C++ is admittedly not a fair one, because LISP structures typically behave as vectors. However, since they provide a restricted form of object orientation, it *is* legitimate to consider them as an additional tool, some sort of "lightweight" object system, that C++ does not provide. For small objects, we see that structure instantiation performs between 4 and 5 times faster than class instantiation (both languages altogether). In the 7 slots case, this ratio downgrades to 2, and in the 49 slots case, to 1.2, with the additional divergence according to the slot type.

When it comes to classes, we see that the difference between LISP and C++ is negligible for small objects, and amounts to a factor of 1.2 in all other situations. LISP outperforms C++ even more when shared slots are involved. In such a case, the gain amount to 30%. These results are clearly more than we originally expected.

6. Conclusion

In this paper, we reported the results of an ongoing experimental research on the behavior and performance of CLOS, the COMMON-LISP object system. We tested the efficiency of the instantiation process in both C++ and LISP, and we also took the opportunity to examine 3 different LISP compilers. With this study, we demonstrated the following points.

- When safety is privileged over speed, the behavior of instantiation is very different from one language to another. C++ is very sensitive to the inheritance hierarchy and not at all to the slot type. LISP on the other hand, is practically immune to the inheritance hierarchy (with very small exceptions in the case of CMU-CL and ACL), but in the case of structures, sensitive to the slot type.
- When optimization is turned on, the effect of our experimental parameters tend to dissipate and both languages come closer to each other, both in terms of behavior and in terms of performance. While turning on optimization in C++ leads to a reasonable improvement, the effect is tremendous in LISP. As soon as the class to instantiate is known or can be inferred at compile-time, we have seen that the instantiation time can be divided by a factor up to one hundred in some cases, and to the point that instantiating in LISP becomes actually *faster* than in C++.
- We should also emphasize on the fact that these performance results are obtained without intimate knowledge of either the languages or the compilers: only standard and / or portable optimization settings were used.

To be fair, we should mention that with micro-benchmarks consisting simply in instantiating objects, we are comparing *compilers* performance as well as *languages* performance, but compiler performance is a critical part of the production chain anyway. Moreover, to be even fairer, we should also mention that when we speak of "equivalent C++ and LISP" code, this is actually quite inaccurate. For instance, we are comparing a C++ *operator* (`new`) with a LISP *function* (`make-instance`). We are comparing the instantiation of classes which are compile-time constructs in C++ but are

dynamic (meta-)objects created at run-time in LISP. This means that it is actually impossible to compare exclusively either language, or compiler performance. This also means that given the inherent expressiveness of LISP and CLOS in particular, compilers have to be even smarter to reach the efficiency level of C++, and this is really good news for the LISP community.

One dark point remains on the LISP side though. Our study exhibited extremely divergent behaviors with respect to slot initform type checking (and probably slot type checking in general), in particular when safety is (supposed to be) preferred over speed. Some compilers never perform any type checking, some always do (even in optimized mode), but then, on structures and not on classes, some others adhere to a consistent principle, except when bugs come and contradict those principles. We think it is a pity that the COMMON-LISP standard leaves so much freedom to the implementation, as it leads to very different behaviors across implementations, and even inconsistencies within a particular implementation. Admittedly, if one considers a class mainly as a storage facility (which is reasonable, given the fact that behavior is not encapsulated in classes in CLOS), type checking is not a critical feature because type errors *will* be caught later on, at the time a slot's value is actually used. Nevertheless, it would not hurt to be informed sooner of a typing error, and would make debugging easier.

7. Perspectives

The perspectives of this work are numerous. We would like to emphasize on the ones we think are the most important.

7.1 Further Investigation

The research described in this paper is still ongoing. In particular, we have outlined several oddities or surprising behaviors in some particular aspects of the instantiation process. Some of these oddities have been reported to the concerned maintainers; some of them have even already been fixed or are being worked on right now. Some others should be further analyzed, as they probably would reveal room for improvement in the compilers.

7.2 Structures as classes

Although not part of the ANSI COMMON-LISP standard, some implementations provide a certain degree of unification between structures and CLOS (for instance, accessing structure slots with `slot-value`). These features have not been tested at all, but it should be interesting to see how they behave.

7.3 Memory Management

GC timings were intentionally left out of this study because our focus was on the object-oriented layer. When comparing so different languages however, it is difficult to avoid taking into account the differences of expressiveness, and in that particular matter, the fact that memory management is automatic on one side, and manual on the other side. Even within the LISP side, it would be interesting to include GC timings in the benchmarks, simply because GC is part of the language design, and also because different compilers use different GC techniques which adds even more parameters to the overall efficiency of one's application.

7.4 Benchmarking other compilers / architectures

The benchmarks presented in this paper were obtained on a specific platform, with specific compilers. It would be interesting to measure the behavior and performance of the same code on other platforms, and also with other compilers. The automated benchmarking infrastructure provided in the source code should make this process easier for people willing to do so.

This is also the place where it would be interesting to measure the impact of compiler-specific optimization capabilities, including architecture-aware ones like the presence of SIMD (Single Instruction, Multiple Data) instruction sets. One should note however that this leads to comparing compilers more than languages, and that the performance gain from such optimizations would be very dependent on the algorithms under experimentation (thus, it would be difficult to draw a general conclusion).

7.5 From dedication to genericity

In Verna (2006), we provided a series of micro-benchmarks on several simple image processing algorithms, in order to evaluate the performance of pixel access and arithmetic operations in both C and LISP. We demonstrated that the behavior of equivalent LISP and C code is similar with respect to the choice of data structures and types, and also to external parameters such as hardware optimization. We further demonstrated that properly typed and optimized LISP code runs as fast as the equivalent C code, or even faster in some cases. That work was the first step in order to get C or C++ people's attention. However, most image treaters want some degree of genericity: genericity on the image types (RGB, Gray level *etc.*), image representation (integers, unsigned, floats, 16bits, 32bits *etc.*), and why not on the algorithms themselves. To this aim, the object oriented approach is a natural way to go. Image processing libraries with a variable degree of genericity exist both in C++ and in LISP, using CLOS. The research presented in this paper is the first step in evaluating the efficiency of the COMMON-LISP object-oriented layer. The next two steps will be to respectively evaluate the performance of slot access, and then generic dispatch in general. Given the optimistic note on which this paper ends, we are eager to discover how LISP will compare to C++ in those matters.

7.6 From dynamic to static genericity

Even in the C++ community, some people feel that the cost of dynamic genericity is too high. Provided with enough ex-

pertise on the template system and on meta-programming, it is now possible to write image processing algorithms in an object-oriented fashion, but in such a way that all generic dispatches are resolved at compile time (Burrus et al., 2003). Reaching this level of expressiveness in C++ is a very costly task, however, because template programming is cumbersome (awkward syntax, obfuscated compiler error messages *etc.*). There, the situation is expected to turn dramatically in favor of LISP. Indeed, given the power of the LISP macro system (the whole language is available in macros), the ability to generate function code *and* compile it on-the-fly, we should be able to automatically produce dedicated hence optimal code in a much easier way. There, the level of expressiveness of each language becomes of a capital importance.

Finally, it should be noted that one important aspect of static generic programming is that the cost of abstraction resides in the compilation process; not in the execution anymore. In other words, it will be interesting to compare the performances of LISP and C++ not in terms of execution times, but compilation times this time.

Acknowledgments

The author would like to thank Roland Levillain, and several LISP implementers for their useful feedback on their respective LISP compilers, most notably Nikodemus Siivola, Duane Rettig and Raymond Toy.

References

Boreczky, J. and Rowe, L. A. (1994). Building COMMON-LISP applications with reasonable performance. `http://bmrc.berkeley.edu/research/publications/1993/125/Lisp.html`.

Burrus, N., Duret-Lutz, A., Géraud, T., Lesage, D., and Poss, R. (2003). A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL)*, Anaheim, CA, USA.

Duret-Lutz, A. (2000). Olena: a component-based platform for image processing, mixing generic, generative and OO, programming. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—-Young Researchers Workshop; published in "Net.ObjectDays2000"*, pages 653–659, Erfurt, Germany. `http://olena.lrde.epita.fr`.

Fateman, R. J., Broughan, K. A., Willcock, D. K., and Rettig, D. (1995). Fast floating-point processing in COMMON-LISP. *ACM Transactions on Mathematical Software*, 21(1):26–62. Downloadable version at `http://openmap.bbn.com/~kanderso/performance/postscript/lispfloat.ps`.

Froment, J. (2000). *MegaWave2 System Library*. CMLA, École Normale Supérieure de Cachan, Cachan, France. `http://www.cmla.ens-cachan.fr/Cmla/Megawave`.

Gabriel, R. P. (1985). *Performance and Evaluation of* LISP *Systems*. MIT Press.

Keene, S. E. (1989). *Object-Oriented Programming in* COMMON-LISP*: a Programmer's Guide to* CLOS. Addison-Wesley.

Kiczales, G. J., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.

MacLachlan, R. A. (1992). The python compiler for CMU-CL. In *ACM Conference on* LISP *and Functional Programming*, pages 235–246. Downloadable version at `http://www-2.cs.cmu.edu/~ram/pub/lfp.ps`.

Neuss, N. (2003). On using COMMON-LISP for scientific computing. In *CISC Conference, LNCSE*. Springer-Verlag. Downloadable version at `http://www.iwr.uni-heidelberg.de/groups/techsim/people/neuss/publications.html`.

Paepcke, A. (1993). User-level language crafting – introducing the CLOS metaobject protocol. In Paepcke, A., editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press. Downloadable version at `http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps`.

Quam, L. H. (2005). Performance beyond expectations. In *International* LISP *Conference*, pages 305–315, Stanford University, Stanford, CA. The Association of LISP Users. Downloadable version at `http://www.ai.sri.com/~quam/Public/papers/ILC2005/`.

Reid, J. (1996). Remark on "fast floating-point processing in COMMON-LISP". In *ACM Transactions on Mathematical Software*, volume 22, pages 496–497. ACM Press.

Steele, G. L. (1990). COMMON-LISP *the Language, 2nd edition*. Digital Press. Online and downloadable version at `http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`.

ANSI (1994). American National Standard: Programming Language – COMMON-LISP. ANSI X3.226:1994 (R1999).

Verna, D. (2006). Beating C in scientific computing applications. In *Third European* LISP *Workshop at* ECOOP, Nantes, France.