Recent changes in the CCL compiler on X8664

These changes were introduced into the CCL trunk in version “1.12-dev” in September 2015.

X86 processors have historically been somewhat register-starved. This was especially true of x8632, but (to a lesser degree) is still true of modern 64-bit processors as well when compared to (e.g) ARMv8. Presumably for this reason – and because it is a CISC architecture (where many instructions can have one or more operands in memory – X86 implementations use a variety of techniques (large and sophisticated memory caches and other things) to reduce the cost of memory access (in terms of execution time). A function that can keep frequently-accessed values in machine registers will generally run faster than an equivalent function that keeps some or all of those values in memory (usually on the stack), but the differences in execution time between these cases is often surprisingly small on X86.

CCL has historically done a few simple things to keep frequently-accessed things in registers -
- relatively small number of registers (8 on PPC, 3 or 4 on X8864) were reserved as “non-volatile” with respect to function calls – the compiler could keep some values in a non-volatile register (nvr) within a function as long as it ensured that function saved the register on entry and restored on exit. This seemed to have more of a positive effect on the PPC than it does on X8664, where the results are mixed.
- on all architectures, if the compiler happens to notice that a stack location contains the value of some register and it is about to generate code to access that stack location again (and the register's value hasn't changed, it will generate code to re-use the register and avoid the redundant LOAD from the stack. This was introduced in CCL on the ARM a few years ago and seemed to reduce execution time. It is not clear that it has had a similar effect on x86.

Even if it is consistently true that memory access does not significantly impact execution time in many cases, some users of CCL (and their users) are concerned that excessive memory accesses (stack accesses) can negatively impact power consumption. To try to address these concerns, a new register allocator has been implemented in X8664 versions of CCL available in the CCL trunk.

X8664 CCL no longer considers any general-purpose registers (GPRs) as being non-volatile. This makes a few more GPRs available to the compiler to use as (volatile) temporaries. Register values are never implicitly preserved across function calls

To enable the new allocator globally and pervasively, one can do

? (declaim (optimize (stack-access <value>))) where <value> is an integer between 1 and 3 and the symbol STACK-ACCESS is exported from the CCL package.

That will affect all code compiled with that setting in effect; to change the default, you can do:

? (declaim (optimize (stack-access 0)) ; revert to the traditional allocator, for the most part.

Within a function definition, a declaration can be used to override the default

```
? (declaim (optimize (stack-access 0))  ; turn off the new allocator
?  (defun foo (x)
     (declare (optimize (stack-access 3))) ; try to use the new allocator for FOO and for any functions defined within FOO

   …)
```

The declaration only has this effect when it appears in a LAMBDA form. The declaration is syntactically legal in other places where a declaration can appear, but has no effect in those other contexts.

For the most part, the effect of this declaration is orthogonal to the effects of other OPTIMIZE declarations. Some exceptions.

In the traditional allocator, the compiler generated debugging info which associated variable names with stack frame offsets, and BACKTRACE and some parts of the error system could use this information, which many people (including me) find helpful for debugging. The new allocator tries to avoid using the stack and debugging code generated by the new allocator can be equivalent to debugging code compiled with (OPTIMIZE (DEBUG 0)) in effect in implementations where that has an effect.

Since the goal of the new allocator is to keep things in registers – and since function calls don't generally preserve the values of registers – code compiled with the new allocator is generally more willing to do more things inline than was traditionally the case, and this is no longer purely a speed/space trade off.

When the new allocator is is enabled, the compiler will try to use it unless the compared notices that the target function does one of several things that would prevent it from working well (or. in some cases, from working at all). In general, those things cause the compiler “bail out” and try to compile the function again with the traditional allocator. This happens quietly and transparently and fairly quickly, but it does involve running the compiler frontend twice in some cases. Things that happen in the frontend – notably, macro expansion – may happen more often than expected.

The set of things that can force a reversion to the old allocator has changed over time and may continue to do so. At the moment, it includes

- function calls -either calls to other Lisp functions or calls to “subprimitives” defined in the CCL kernel – made when some registers are “alive” at (have been used before and will be used after) the call. Such registers must be saved to (“spilled”) the stack, and there is often as much stack-access involved when registers are spilled in the new allocator as there was in the old allocator Note that no registers are alive after a tail call.
- functions that receive some of their arguments on the stack. In x8664 CCL, the last 3 arguments to a function are passed in registers and all preceding arguments must be pushed by the caller. This limit is based on empirical studies that indicate that “most” Lisp functions receive around 2 arguments. It would have been possible to increase this limit (and it has been increased a little, in one special case) but changing the calling conventions would have been a

very ambitious (and likely time-consuming) project.
Functions that are defined to receive 4 or fewer required arguments – or 2 required arguments and a “simple” &optional argument whose value defaults to NIL and do not involve 'supplied-p' – do not force the compiler to revert to using the old allocator and all other cases do.

- Functions that receive exactly 4 arguments effectively pop the first of those arguments into a register on entry and can (tail) call themselves with all arguments passed in registers. At least in theory, this could be extended a little to handle a few more arguments.
- For fairly obscure reasons, DYNAMIC-EXTENT declarations force a bailout
- There are a few other things. In general, the list of things is getting shorter.
- A function compiled by the new allocator may not need to establish a frame pointer at all. I believe that it is generally “safe” for the compiler to remove instructions that maintain the frame pointer in functions that don't need to use it, that is only done when generating unsafe code

Code produced by the new backend seems to be correct – it passes CCL's test suite and REBUILD-CCL can produce a new image when the new allocator is enabled during compilation. The generated code seems to do a good job of avoiding unnecessary stack-access, but that code often contains more copies between registers than are necessary. As far as I know, these instructions do not affect power consumption and their impact on execution time is very small.

Common Lisp is a fairly large language, and everyone seems to use a different subset of it. In the subset of CL that I have tested and used, enabling the new allocator seems to work as expected (it either produces correct code with substantially less stack traffic or revert to the old allocator when necessary), but it is certainly possible that I have missed things that only arise in other cases. It is probably wise to view the new allocator as being of approximately “alpha” quality – it is essentially feature-complete but needs further testing.