Subject: [gentoo-amd64] Re: Re: Wow! KDE 3.5.1 & Xorg 7.0 w/ Composite
From: Duncan <1i5t5.duncan@cox.net>
Date: Fri, 03 Feb 2006 09:28:29 -0700
To: gentoo-amd64@lists.gentoo.org

Mike Owen posted
<8f5ca2210602021712s53d33de5w6794fa384bbf93a5@mail.gmail.com>, excerpted
below, on Thu, 02 Feb 2006 17:12:04 -0800:

> On 2/2/06, Duncan <1i5t5.duncan@cox.net> wrote:
>
> > http://members.cox.net/pu61ic.1inux.dunc4n/
>
> Nice. Now let us know your CFLAGS, and what toolchain versions you're
> running :D

You probably didn't notice, as I had it commented out on the main index page as I don't have the page created to actually list them yet, but if you viewed source, you'd have seen I have a techspecs page link commented out, that'll get that sort of info, when/if I actually get it created.

However, since you asked, your answer, and a bit more, by way of explanation...

I should really create a page listing all the little Gentoo admin scripts I've come up with and how I use them. I'm sure a few folks anyway would likely find them useful.

The idea behind most of them is to create shortcuts to having to type in long emerge lines, with all sorts of arbitrary command line parameters. The majority of these fall into two categories, ea* and ep*, short for emerge --ask <additional parameters> and emerge --pretend ... . Thus, I have epworld and eaworld, the pretend and ask versions of emerge -NuDv world, epsys and easys, the same for system, eplog <package>, emerge --pretend --log --verbose (package name to be added to the command line so eplog gcc, for instance, to see the changes between my current and the new version of gcc), eptree <package>, to use the tree output, etc.

One thing I've found is that I'll often epworld or eptreeworld, then emerge the individual packages, rather than use eaworld to do it. That way, I can do them in the order I want or do several at a time if I want to make use of both CPUs. Because I always use --deep, as I want to keep my dependencies updated as well, I'm very often merging specific dependencies. There's a small problem with that, however --oneshot, which I'll always want to use with dependencies to help keep my world file uncluttered, has no short form, but I use it as the default! OTOH, the normal portage mode of adding stuff listed on the command line to the world file, I don't want very often, as most of the time I'm simply updating what I have, so it's all in the world file if it needs to be there already anyway. Not a problem! All my regular ea* scriptlets use --oneshot, so it */is/* my default. If I ***AM*** merging something new that I want added to my world file, I have another family of ea* scriptlets that do that -- all ending in "2", as in, "NOT --oneshot". Thus, I have a family of ea*2 scriptlets.

The regulars here already know one of my favorite portage features is FEATURES=buildpkg, which I have set in make.conf. That of course gives me a collection of binary versions of packages I've already emerged, so I can quickly revert to an old version for testing something, if I want, then remerge the new version once I've tested the old version to see if it has the same bug I'm working on or not. To aid in this, I have a collection of eppak and eapak scriptlets. Again, the portage default of --usepackage (-k) doesn't fit my default needs, as if I'm using a binpkg, I usually want to ONLY use a binpkg, NOT merge from source if the package isn't available. That happens to be -K in short-form. However, it's my default, so eapak invokes the -K version. I therefore have eapaK to invoke the -k version if I don't really care whether it goes from binpkg or source.

Of course, there are various permutations of the above as well, so I have eapak2 and eapaK2, as well as eapak and eapaK. For the ep* versions, of course the --oneshot doesn't make a difference, so I only have eppak and eppaK, no eppa?2 scriptlets.

... Deep breath... <g>

All that as a preliminary explanation to this: Along with the above, I

have a set of efetch functions, that invoke the -f form, so just do the fetch, not the actual compile and merge, and esyn (there's already an esync function in something or other I have merged so I just call it esyn), which does emerge sync, then updates the esearch db, then automatically fetches all the packages that an eaworld would want to update, so they are ready for me to merge at my leisure.

Likewise, and the real reason for this whole explanation, I */had/* an "einfo" scriptlet that simply ran "emerge info". This can be very handy to run, if like me, you have several slotted versions of gcc merged, and you sometimes forget which one you have eselected or gcc-configed as the one portage will use. Likewise, it's useful for checking on CFLAGS (or CXXFLAGS OR LDFLAGS or...), if you modified them from the normal ones because a particular package wasn't cooperating, and you want to see if you remembered to switch them back or not.

However, I ran into a problem. The output of einfo was too long to quickly find the most useful info -- the stuff I most often change and therefore most often am looking for.

No sweat! I shortened my original "einfo" to simply "ei", and added a second script, "eis" (for einfo short), that simply piped the output of the usual emerge info into a grep that only returned the lines I most often need -- the big title one with gcc and similar info, CFLAGS, CXXFLAGS, LDFLAGS, and FEATURES. USE would also be useful, but it's too long even by itself to be searched at a glance, so if I want it, I simply run ei and look for what I want in the longer output.

... Another deep breath... <g>

OK, with that as a preliminary, you should be able to understand the following:

```
$eis

Portage 2.1_pre4-r1 (default-linux/amd64/2006.0, gcc-4.1.0-beta20060127,
glibc-2.3.6-r2, 2.6.15 x86_64)

CFLAGS="-march=k8 -Os -pipe -fomit-frame-pointer -frename-registers
-funit-at-a-time -fweb -freorder-blocks-and-partition
-fmerge-all-constants"

CXXFLAGS="-march=k8 -Os -pipe -fomit-frame-pointer -frename-registers
-funit-at-a-time -fweb -freorder-blocks-and-partition
-fmerge-all-constants"

FEATURES="autoconfig buildpkg candy ccache confcache distlocks
multilib-strict parallel-fetch sandbox sfperms strict userfetch"

LDFLAGS="-Wl,-z,now"

MAKEOPTS="-j4"
```

To make sense of that...

* The portage and glibc versions are ~amd64, as set in make.conf for the system in general.

* CFLAGS:

I choose -Os, optimize for size, because a modern CPU and the various cache levels are FAR faster than main memory. This difference is frequently severe enough that it's actually more efficient to optimize for size than for CPU performance, because the result is smaller code that maintains cache locality (stays in fast cache) far better, and the CPU saves more time that it would otherwise be spending idle, waiting for data to come in from slower more distant memory, than the actual cost of the loss of cycle efficiency that's often the tradeoff for small code.

-O3, and to a lessor extent, -O2, do things like turn a loop that executes a fixed number of say 3 times, into "faster" code, by avoiding the jump at the end of each loop back to the top of the loop by writing it out as inline code, copying the loop instructions three times. This process would in our example of a 3-time fixed execution loop, save the expensive jump back to the top of the loop two times -- but at the SAME time would expand that section of code to three times its looped size.

Back when memory operated at or near the speed of the CPU, avoiding the loop, even at the expense of three-times the code, was often faster. Today, where CPUs do several calculations in the time it takes to fetch

data from main memory, it's generally faster to go for the smaller code, as it will be far more likely to still be in fast cache, avoiding that long wait for main memory, even if it */does/* mean wasting a couple additional cycles doing the expensive jump back to the top of the loop.

Of course, this is theory, and the practical case can and will differ depending on the instructions actually being compiled. In particular, streaming media apps and media encoding/decoding are likely to still benefit from the traditional loop elimination style optimizations, because they run thru so much data already, that cache is routinely trashed anyway, regardless of the size of your instructions. As well, that type of application tends to have a LOT of looping instructions to optimize!

By contrast, something like the kernel will benefit more than usual from size optimization. First, it's always memory locked and as such can't be swapped, and even "slow" main memory is still **MANY** **MANY** times faster than swap, so a smaller kernel means more other stuff fits into main memory with it, and isn't swapped as much. Second, parts of the kernel such as task scheduling are executed VERY often, either because they are frequently executed by most processes, or because they */control/* those processes. The smaller these are, the more likely they are to still be in cache when next used. Likewise, the smaller they are, the less potentially still useful other data gets flushed out of cache to make room for the kernel code executing at the moment. Third, while there's a lot of kernel code that will loop, and a lot that's essentially streaming, the kernel as a whole is a pretty good mix of code and thus won't benefit as much from loop optimizations and the like, as compared to special purpose code like the media codec and streaming applications above.

The differences are marked enough and now demonstrated enough that a kernel config option to optimize for size was added I believe about a year ago. Evidently, that lead to even MORE demonstration, as the option was originally in the obscure embedded optimizations corner of the config, where few would notice or use it, and they upgraded it into a main option. In fact, where a year or two ago, the option didn't even exist, now I believe it defaults to yes/on/do-optimize-for-size (altho it's possible I'm incorrect on the last and it's not yet the default).

According to the gcc manpage, -frename-registers causes gcc to attempt to make use of registers left over after normal register allocation. This is particularly beneficial on archs that have many registers (keeping in mind that "registers" are what amounts to L0 cache, the fastest possible memory because the CPU accesses registers directly and they operate at full CPU speed. Unfortunately, registers are also very limited, making them an EXCEEDINGLY valuable resource! Note that while x86-32 is noted for its relative */lack/* of registers, AMD basically doubled the number of registers available to 64-bit code in its x86-64 aka AMD64 spec. Thus, while this option wouldn't be of particular benefit on x86, on amd64, it can, depending on the code of course, provide some rather serious optimization!

-fweb is a register use optimizer function as well. It tells gcc to create a */web/* of dependencies and assign each individual dependency web to its own pseudo-register. Thus, when it comes time for gcc to allocate registers, it already has a list of the best candidates lined up and ready to go. Combined with -frename register to tell gcc to efficiently make use of any registers left over after the the first pass, and due to the number of registers available in 64-bit mode on our arch, this can allow some seriously powerful optimizations. Still, a couple of things to note about it. One, -fweb (and -frename-registers as well) can cause data to move out of its "home" register, which seriously complicates debugging, if you are a programmer or power-user enough to worry about such things. Two, the rewrite for gcc 4.0 significantly modified the functionality of -fweb, and it wasn't recommended for 4.0 as it didn't yet work as well as expected or as it did with gcc 3.x. For gcc 4.1, -fweb is apparently back to its traditional strength. Those Gentoo users having gcc 3.4, 4.0, and 4.1, all three in separate slots, will want to note this as they change gcc-configuratiions, and modify it accordingly. Yes, this ***IS*** one of the reasons my CFLAGS change so frequently!

-funit-at-a-time tells gcc to consider a full logical unit, perhaps consisting of several source files rather than just one, as a whole, when it does its compiling. Of course, this allows gcc to make optimizations it couldn't see if it wasn't looking at the larger picture as a whole, but it requires rather more memory, to hold the entire unit so it can consider it at once. This is a fairly new flag, introduced with gcc 3.3 IIRC. While the idea is simple enough and shouldn't lead to any bugs on its own, there WERE a number of initially never encountered bugs in various code that this flag exposed, when GCC made optimizations on the entire unit that it wouldn't otherwise make, thereby triggering bugs that

had never been triggered before. I */believe/* this was the root reason why the Gentoo amd64 technotes originally discouraged use of -Os, back with the first introduction of this flag in gcc 3.2 hammer (amd64) edition, as -funit-at-a-time was activated by -Os at that time, and -Os was known to produce bad code at the time, on amd64, with packages like portions of KDE. The gcc 4.1.0 manpage now says it's enabled by default at -O2 and -O3, but doesn't mention -Os. Whether that's an omission, or whether they decided it shouldn't be enabled by -Os for some reason, I'm not sure, but I use them both to be sure and haven't had any issues I can trace to this (not even back when the technotes recommended against -Os, and said KDE was supposed to have trouble with it -- maybe it was parts of KDE I never merged, or maybe I was just lucky, but I've simply never had an issue with it).

-freorder-blocks-and-partition is new for gcc 4.0, I believe, alto I didn't discover it until I was reading the 4.1-beta manpage. I KNOW gcc 3.4.4 fails out with it, saying unrecognized flag or some such, so it's another of those flags that cause my CFLAGS to be constantly changing, as I switch between gcc versions. This flag won't work under all conditions, according to the manpage, so is automatically disabled in the presence of exception handling, and a few other situations named in the manpage. It causes a lot of warnings too, to the effect that it's being disabled due to X reason. There's a similar -freorder-blocks flag, which optimizes by reordering blocks in a function to "reduce number of taken branches and improve code locality." In English, what that means is that it breaks caching less often. Again, caching is ***EXTREMELY*** performance critical, so anything that breaks it less often is CERTAINLY welcome! The -and-partition increases the effect, by separating the code into frequently used and less frequently used partitions. This keeps the most frequently used code all together, therefore keeping it in cache far more efficiently, since the less used code won't be constantly pulled in, forcing out frequently used code in the process.

Hmm... As I'm writing and thinking about this, the probability that sticking the regular -freorder-blocks option in CFLAGS as well would be a wise thing, occurs to me. The non-partition version isn't as efficient as the partition version, and would be redundant if the partitioned version is in effect. However, the non-partitioned version doesn't have the same sorts of no-exceptions-handler and similar restrictions, so having it in the list, first, so the partitioned version overrides it where it can be used, should be a good idea. That way, where the partitioned version can be used, it will be, but where it can't, gcc will still use the non-partitioned version of the option, so I'll still get */some/* of the optimizations! I (re)compiled major portions of xorg (modular), qt, and the new kde 3.5.1 with the partitioned option, however, and it works, and I haven't tested having both options in there yet, so I'm not sure it'll work as the theory suggests it should, so some caution might be advised.

-fmerge-all-constants COULD be dangerous with SOME code, as it breaks part of the C/C++ specification. However, it should be fine for most code written to be compiled with gcc, and I've seen no problems */yet/* tho both this and the reorder-and-partition flag above are fairly new to my CFLAGS, so haven't been as extensively personally tested as the others have been. If something seems to be breaking when this is in your CFLAGS, certainly it's the first thing I'd try pulling out. What it actually does is merge all constants with the same value into the same one. gcc has a weaker -fmerge-constants version that's enabled with any -O option at all (thus at -O, -O2, -O3, AND -Os), that merges all declared constants of the same value, which is safe and doesn't conflict with the C/C++ spec. What the */all/* specifier in there does, however, is cause gcc to merge declared variables where the value actually never changes, so they are in effect constants, altho they are declared as variables, with other constants of the same value. This */should/* be safe, */provided/* gcc isn't failing to detect a variable chance somewhere, but it conflicts with the C/C++ spec, according to the gcc manpage, and thus */could/* cause issues, if the developer pulls certain tricks that gcc wouldn't detect, or possibly more likely, if used with code compiled by a different compiler (say binary-only applications you may run, which may not have been compiled with gcc). There are two reasons why I choose to use it despite the possible risks. One, I want */small/* code, again, because small code fits in that all-important cache better and therefore runs faster, and obviously, two or more merged constants aren't going to take the space they would if gcc stored them separately. Two, the risks aren't as bad if you aren't running non-gcc compiled code anyway, and since I'm a strong believer in Software Libre, if it's binary-only, there's very little chance I'll want or risk it on my box, and everything I do run is gcc compiled anyway, so should be generally safe. Still, I know there may be instances where I'll have to recompile with the flag turned off, and am prepared to deal with them when they happen, or I'd not have the flag in my CFLAGS.

And, here's some selected output from ei, interspersed with explanations, since I'm editing the output anyway:

```
$ei
!!! Failed to change nice value to '-2'
!!! [Errno 13] Permission denied
```

This is stderr output. It's not in the eis output above because I redirect stderr to /dev/null for it, as I know the reason for the error and am trying to be brief.

The warning is because I'm using PORTAGE_NICENESS=-2 in make.conf. It has a negative nice set there to encourage portage to make fuller use of the dual CPUs under-X/from-a-konsole-session, as X and the kernel do some dynamic scheduling magic to keep X more responsive without having to up */its/* priority. The practical effect of that "magic" is to lower the priorities of everything besides X slightly, when X is running. This */does/* have the intended effect of keeping X more responsive, but the cost as observed here is that emerges take longer than they should when X is running, because the scheduler is leaving a bit of extra idle CPU time to keep X responsive. In many cases, I'd rather be using maximum CPU and get the merges done faster, even if X drags a bit in the mean time, and the slightly negative niceness for portage accomplishes exactly that.

It's reporting a warning (to stderr) here, as I ran the command as a regular non-root user, and non-root can't set negative priorities for obvious system security reasons. I get the same warning with my ep* commands, which I normally run as a regular user, as well. The ea* commands which actually do the merging get run as root, naturally, so the niceness */can/* be set negative when it counts, during a real emerge.

So... nothing of any real matter, then.

```
!!! Relying on the shell to locate gcc, this may break
!!! DISTCC, installing gcc-config and setting your current gcc
!!! profile will fix this
```

Another warning, likewise to stderr and thus not in the eis output. This one is due to the fact that eselect, the eventual systemwide replacement for gcc-config and a number of other commands, uses a different method to set the compiler than gcc-config did, and portage hasn't been adjusted to full compatibility just yet. Portage finds the proper gcc just fine for itself, but there'd be problems if distcc was involved, thus the warning.

Again, I'm aware of the situation and the cause, but don't use distcc, so it's nothing I have to worry about, and I can safely ignore the warning.

I kept the warnings here, as I find them and the explanation behind them interesting elements of my Gentoo environment, thus worth posting, for others who seem interested in my Gentoo environment as well. If nothing else, the explanations should help some in my audience understand that bit more about how their system operates, even if they don't get these warnings.

```
Portage 2.1_pre4-r1 (default-linux/amd64/2006.0, gcc-4.1.0-beta20060127,
glibc-2.3.6-r2, 2.6.15 x86_64)
=================================================================
System uname: 2.6.15 x86_64 AMD Opteron(tm) Processor 242
Gentoo Base System version 1.12.0_pre15
```

Those of you running stable amd64, but wondering where baselayout is for unstable, there you have it!

```
ccache version 2.4 [enabled]
dev-lang/python:   2.4.2
sys-apps/sandbox:    1.2.17
sys-devel/autoconf:  2.13, 2.59-r7
sys-devel/automake:  1.4_p6, 1.5, 1.6.3, 1.7.9-r1, 1.8.5-r3, 1.9.6-r1
sys-devel/binutils:  2.16.91.0.1
sys-devel/libtool:   1.5.22
virtual/os-headers:  2.6.11-r3

ACCEPT_KEYWORDS="amd64 ~amd64"
```

Same for the above portions of my toolchain. AFAIR, it's all ~amd64, altho I was running a still-masked binutils for awhile shortly after

gcc-4.0 was released (still-masked on Gentoo as well), as it required the newer binutils.

LANG="en_US"
LDFLAGS="-Wl,-z,now"

Some of you may have noticed the occasional Portage warning about a SETUID executables using lazy bindings, and the potential security issue that causes. This setting for LDFLAGS forces early bindings with all dynamically linked libraries. Normally it'd only be necessary or recommended for SETUID executables, and set in the ebuild where it's safe to do so, but I use it by default, for several reasons. The effect is that a program takes a bit longer to load initially, but won't have to pause to resolve late bindings as they are needed. You're trading waiting at executable initialization for waiting at some other point. With a gig of memory, I find most stuff I run more than once is at least partially still in cache on the second and later launches, and with my system, I don't normally find the initial wait irritating, and sometimes find a pause after I'm working with a program especially so, so I prefer to have everything resolved and loaded at executable launch. Additionally, with lazy bindings, I've had programs start just fine, then fail later when they need to resolve some function that for some reason won't resolve in whatever library it's supposed to be coming from. I don't like have the thing fail and interrupt me in the middle of a task, and find it far less frustrating, if it's going to fail when it tries to load something, to have it do so at launch. Because early bindings forces resolution of functions at launch, if it's going to fail loading one, it'll fail at launch, rather than after I've started working with the program. That's */exactly/* how I want it, so that's why I run the above LDFLAGS setting. It's nice not to have to worry about the security issue, but SETUID type security isn't as critical on my single-human-user system, where that single-user-is me and I already have root when I want it anyway, as it'd be in a multi-user system, particularly a public server, so the other reasons are more important than security, for me, on this. They just happen to coincide, so I'm a happy camper. =8^)

The caveat with these LDFLAGS, however, is the rare case where there's a circular functional dependency that's normally self-resolving, Modular xorg triggers one such case, where the monolithic xorg didn't. There are three individual ebuilds related to modular xorg that I have to remove these LDFLAGS for or they won't work. xorg-server is one. xf86-vidio-ati, my video driver, is another. libdri was the third, IIRC. There's a specific order they have to be compiled in, as well. If they are compiled with this enabled, they, and consequently X, refuses to load (tho X will load without DRI, if that's the only one, it'll just protest in the log and DRI and glx aren't available). Evidently there's a non-critical fourth module somewhere, that still won't load properly due to an unresolved symbol, that I need to track down and remerge without these LDFLAGS, and that's what's keeping GLX from loading on my current system, as mentioned in an earlier post.

LINGUAS="en"
MAKEOPTS="-j4"

The four jobs is nice for a dual-CPU system -- when it works. Unfortunately, the unpack and configure steps are serialized, so the jobs option does little good, there. To make most efficient use of the available cycles when I have a lot to merge, therefore, I'll run as many as five merges in parallel. I do this quite regularly with KDE upgrades like the one to 3.5.1, where I use the split KDE ebuilds and have something north of 100 packages to merge before KDE is fully upgraded.

I mentioned above that I often run eptree, then ea individual packages from the list. This is how I accomplish the five merges in parallel. I'll take a look at the tree output to check the dependencies, and merge the packages first that have several dependencies, but only where those dependencies aren't stepping on each other, thus keeping the parallel emerges from interfering with each other, because each one is doing its own dependencies, that aren't dependencies of any of the others. After I get as many of those going as I can, I'll start listing 3-5 individual packages without deps on the same ea command line. By the time I've gotten the fifth one started, one of the other sessions has usually finished or is close to it, so I can start it merging the next set of packages. With five merge sessions in parallel, I'm normally running an average load of 5 to 9, meaning that many applications are ready for CPU scheduling time at any instant, on average. If the load drops below four, there's proobably idle CPU cycles being wasted that could otherwise be compiling stuff, as each CPU needs at least one load-point to stay busy, plus usually can schedule a second one for some cycles as well, while the first is waiting for the hard drive or whatever.

(Note that I'm running a four-drive RAID, RAID-6, so two-way striped, for my main system, Raid-0, so 4-way striped, for $PORTAGE_TMPDIR, so hard drive latency isn't */nearly/* as high as it would be on a single-hard-drive system. Of course, running five merges in parallel */does/* increase disk latency some as well, but it */does/* seem to keep my load-average in the target zone and my idle cycles to a minimum, during the merge period. Also note that I've only recently added the PORTAGE_NICENESS value above, and haven't gotten it fully tweaked to the best balance between interactivity and emerge speed just yet, but from observations so far, with the niceness value set, I'll be able to keep the system busy with "only" 3-4 parallel merges, rather than the 5 I had been having to run to keep the system most efficiently occupied when I had a lot to merge.)

PKGDIR="/pkg"
PORTAGE_TMPDIR="/tmp"
PORTDIR="/p"
PORTDIR_OVERLAY="/l/p"

Here you can see some of my path customization.

USE="amd64 7zip X a52
aac acpi alsa apm arts asf audiofile avi bash-completion berkdb
bitmap-fonts bzip2 caps cdparanoia cdr crypt css cups curl dga divx4linux
dlloader dri dts dv dvd dvdr dvdread eds emboss encode extrafilters fam
fame ffmpeg flac font-server foomaticdb gdbm gif glibc-omitfp gpm
gstreamer gtk2 idn imagemagick imlib ithreads jp2 jpeg jpeg2k kde
kdeenablefinal lcms libwww linuxthreads-tls lm_sensors logitech-mouse
logrotate lzo lzw lzw-tiff mad maildir mikmod mjpeg mng motif mozilla mp3
mpeg ncurses network no-old-linux nolvm1 nomirrors nptl nptlonly offensive
ogg opengl oss pam pcre pdflib perl pic png ppds python qt quicktime
radeon readline scanner slang speex spell ssl tcltk theora threads tiff
truetype truetype-fonts type1 type1-fonts usb userlocales vcd vorbis
xcomposite xine xinerama xml2 xmms xosd xpm xrandr xv xvid yv12 zlib
elibc_glibc input_devices_keyboard input_devices_mouse kernel_linux
linguas_en userland_GNU video_cards_ati"

My USE flags, FWTAR (for what they are worth). Of particular interest are the input_devices_mouse and keyboard, and video_cards_ati. These come from variables (INPUT_DEVICES and VIDEO_CARDS) set in make.conf, and used in the new xorg-modular ebuilds. These and the others listed after zlib are referred to by Gentoo devs as USE_EXPAND. Effectively, they are USE flags in the form of variables, setup that way because there are rather many possible values for those variables, too many to work as USE flags. The LINGUAS and LANG USE_EXPAND variables are prime examples. Consider how many different languages there are and that were used and documented as regular USE flags, it would have to be in use.local.desc, because few supporting packages would offer the same choices, so each would have to be listed separately for each package. Talk about the number of USE flags quickly getting out of control!

Unset: ASFLAGS, CTARGET, EMERGE_DEFAULT_OPTS, LC_ALL

OK, some loose ends to wrapup, and I'm done.

re: gcc versions: The plan is for gcc-4.0 to go ~arch fairly soon, now. The devs are actively asking for bug reports involving it, now, so as many as possible can be resolved before it goes ~arch. (Formerly, they were recommending that bugs be filed upstream, and not with Gentoo unless there was a patch attached, as it was considered entirely unsupported, just there for those that wanted it anyway.) At this point, nearly everything should compile just fine with 4.0.

That said, Gentoo has slotted gcc for a reason. It's possible to have multiple minor versions (3.3, 3.4, 4.0, 4.1) merged at the same time. With USE=multislot, that's actually microversion (4.0.0, 4.0.1, 4.0.2...). Using either gcc-config or eselect compiler, and discounting any CFLAG switching you may have to do, it's a simple matter to switch between merged versions. This made it easy to experiment with gcc-4.0 even tho Gentoo wasn't supporting it and certain packages wouldn't compile with 4.x, because it was always possible to switch to a 3.x version if necessary, and compile the package there. I did this quite regularly, using gcc-4.0 as my normal version, but reverting for individual packages as necessary, when they wouldn't compile with 4.0.

The same now applies to the 4.1.0-beta-snapshot series. Other than the compile time necessary to compile a new gcc when the snapshot comes out each week, it's easy to run the 4.1-beta as the main system compiler for as wide testing as possible, while reverting to 4.0 or 3.4 (I don't have a 3.3 slot merged) if needed.

re: the performance improvements I saw that started this whole thing: These trace to several things, I believe. #1, with gcc-4.0, there's now support for -fvisibility -- setting certain functions as exported and visible externally, others not. That can easily cut exported symbols by a factor of 10. Exported symbols of course affect dynamic load-time, which of course gets magnified dramatically by my LDFLAGS early binding settings. When I first compiled KDE with that (there were several missteps early on in terms of KDE and Gentoo's support, but that aside), KDE appload times went down VERY NOTICEABLY! Again, due to my LDFLAGS, the effect was multiplied dramatically, but the effect is VERY real!

Of course, that's mainly load-time performance. The run-time performance that we are actually talking here has other explanations. A big one is that gcc-4 was a HUGE rewrite, with a BIG potential to DRAMATICALLY improve gcc's performance. With 4.0, the theory is there, but in practice, it wasn't all that optimized just yet. In some ways it reverted behavior below that of the fairly mature 3.x series, altho the rewrite made things much simpler and less prone to error given its maturity. 4.1, however, is the first 4.x release to REALLY be hitting the potential of the 4.x series, and it appears the difference is very noticeable. Of course, there's a reason 4.1.0 is still in beta upstream and not supported by Gentoo either, as there are still known regressions. However, where it works, which it seems to do */most/* of the time, it **REALLY** works, or at least that's been my observation. 3.3 was a MAJOR improvement in gcc for amd64 users, because it was the first version where amd64 wasn't simply an add-on hack, as it had been with 3.2. The 3.4 upgrade was minor in comparison, and 4.0 while it's going ~arch shortly, and sets the stage for a lot of future improvement, will be pretty minor in terms of actual improved performance as well. 4.1, however, when it is finally fully released, has the potential to be as big an improvement as 3.3 was -- that is, a HUGE one. I'm certainly looking forward to it, and meanwhile, running the snapshots, because Gentoo makes it easy to do so while maintaining the ability to switch very simply between multiiple versions on the system.

Both -freorder-blocks-and-partition and -fmerge-all-constants are new to me within a few days, now, and new to me with kde 3.5.1. Normally, individual flags won't make */that/* much of a difference, but it's possible I hit it lucky, with these. Actually, because they both match very well with and reinforce my strategy of targeting size, it's possible I'm only now unlocking the real potential behind size optimization. -- I **KNOW** there's a **HUGE** difference in sizes between resulting file-sizes. I compared 4.0.2 and 4.1.0-beta-snapshot file sizes for several modular-X files in the course of researching the missing symbols problem, and the difference was often a shrinkage of near 33 percent with 4.1 and my current CFLAGS as opposed to 4.0.1 without the new ones. Going the other way, that's a 50% larger file with 4.0.2 as compared to 4.1, 100KB vs 150KB, by way of example. That's a ***HUGE*** difference, one big enough to initially think I'd found the reason for the missing symbols right there, as the new files were simply too much smaller to look workable! Still, I traced the problem too LDFLAGS, so that wasn't it, and the files DO work, confirming things. I'm guessing -fmerge-all-constants plays a significant part in that. In any case, with that difference in size, and knowing how */much/* cache hit vs. miss affects performance, it's quite possible the size is the big performance factor. Of course, even if that's so, I'm not sure whether it is the CFLAGS or the 4.0 vs 4.1 that should get the credit.

In any case, I'm a happy camper right now! =8^)

--
Duncan - List replies preferred. No HTML msgs.
"Every nonfree program has a lord, a master --
and if you use the program, he is your master." Richard Stallman in
http://www.linuxdevcenter.com/pub/a/linux/2004/12/22/rms_interview.html

--
gentoo-amd64@gentoo.org mailing list