Custom Product Integration and the Cray Programming Environment

Sean Byland, Ryan Ward
Cray Inc.
Saint Paul, MN USA
seanb@cray.com, ward@cray.com

***Abstract*—With Cray's increasing customer base and product portfolio, a faster, more scalable, and more flexible software access solution for the Cray Programming Environment was required. The xt-asyncpe product-offering required manual updates to add new product and platform support, took a significant amount of time to evaluate the environment when building applications, and didn't harness useful standards used by the Linux community. CrayPE 2, by incorporating the flexibility of modules, the power of *pkg-config*, and a programmatic design, offers a stronger solution going forward. It provides not only a simplified extensibility and more robust solution for adding products to a system, but also yields a significant reduction in application build time for users. This paper discusses the issues addressed and the improved functionality available to support Cray, customers, and third-party software access.**

***Keywords*-CrayPE; CrayPE 2; Programming Environment; Integration;**

I. Introduction

The number of software products and HPC libraries that may be potentially be useful for Cray customers has grown at a massive rate, as has the number and variety of Cray customers. When required, Cray has provided programming environment products, often built for different programming environments, compilers, accelerators, or CPUs. Cray has chosen to support products, programming models, and systems configurations that will help the greatest number of customers. This is required since it is nearly impossible to support all the possible configurations that customers may desire. This axiom led us to the conclusion that we needed to redesign CrayPE from the ground up, to make product integration as simple and seamless as possible while opening up any internal CrayPE capabilities to all Cray users. Standardizing product integration not only allows Cray users to develop and integrate products that match their own unique needs, but eases the long-term support cost for Cray, allowing Cray to support a larger number of products and system configurations. This redesign and implementation was first released as CrayPE 2.0.

During CrayPE 2's design evaluation, we determined that while it was essential to retain the environment modules and front-end drivers functionality that made CrayPE 2's predecessor (xt-asyncpe) unique and useful for compiling HPC applications, the older driver solution was no longer viable. Retaining products' compiling/linking information in the front-end scripts, while readable and convenient for a small number of products, has the implicit disadvantage of requiring engineer intervention to maintain dependencies, manage version control, or add new products. In order to mitigate these disadvantages, CrayPE 2 moved all product-specific information for a given product into a file-format that would be familiar to users.

II. CrayPE 2 Design Objectives

While the primary design goal of CrayPE 2 was to lower the cost of integration, a number of supporting goals needed to be included to make this possible. In addition, other desirable improvements could be implemented with a redesign:

- No engineer intervention needed to update the compiling front-end for the purpose of adding product support. The front-end needs to be product- and version-agnostic and to require no explicit version tracking.
- A system that could be learned and used easily.
- A system that would easily integrate into both Cray and standard GNU/Linux systems.
- A compiling front-end that works programmatically.
- Increased performance.
- A system that would allow the library stack to be used independently and treated as a separate abstraction layer.
- Improvements in error detecting and reporting.

III. Functionality of CrayPE 2

To provide compilation and linking support for user-level software, standard Linux distributions use a utility, *pkg-config*. For Cray, the advantages of *pkg-config* were clearly desirable: it uses a standardized file format, has compatibility checking, and handles dependencies and linking order well. However, *pkg-config* is designed for standard desktop or server Linux distribution and typically supports only compiling with one compiler, for one hardware configuration, and for a single version of product's libraries. Due to these limitations, a stand-alone *pkg-config* is not a viable solution for Cray systems.

While *pkg-config* doesn't have functionality for selecting single combinations from a large number of heterogeneous configurations, CrayPE environment modules do, through the use of loading, unloading, and swapping desired product

or targeting modules. Using a hybrid approach – loading modules to restrict *pkg-config* to a single configuration and then calling *pkg-config* for linking/compiling flags – CrayPE 2 is able to get the most use of the robust and effective elements of both systems. This allows any HPC user to compile and link (in the same way all Cray products do) with a custom-built version of software, making use of both library dependency resolution and diverse system targeting. Only adding *pkg-config* (*.pc) files and a modulefile are required, both of which can easily be updated and controlled by users.

A. pkg-config Fundamentals

For people who are unfamiliar with *pkg-config*, *pkg-config* is a utility that was first developed by James Henstridge in 2009[1]. It introduced a standard file format (.pc extension) for storing all metadata associated with a library. *pkg-config* files have tags that store: name, version, description, URL, compiling-specific flags, linking-specific flags, any required libraries (static or dynamic), and any potential conflicting libraries. *pkg-config* files can use variables that, with a default value, can be overridden at execution time.

```
prefix=/usr/local
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${exec_prefix}/include

Name: libpng
Description: Loads and saves PNG files
Version: 1.2.8
Libs: -L${libdir} -lpng12 -lz
Cflags: -I${includedir}/libpng12
```

pkg-config can be called with a "--cflags" option to return compiling flags and "--libs" to return linking flags, for any listed or required products:

```
~> pkg-config --libs --cflags
--static netcdf_c++4
-I/opt/cray/netcdf/4.3.2/CRAY/83/include
-I/opt/cray/hdf5/1.8.13/CRAY/83/include
-L/opt/cray/netcdf/4.3.2/CRAY/83/lib
-L/opt/cray/hdf5/1.8.13/CRAY/83/lib
-lnetcdf_c++4 -lrt -lnetcdf -lrt
-lhdf5_hl  -lz  -ldl -lm -lrt -lhdf5
-lz -ldl -lm -lrt
```

pkg-config will search for pc files that the user requests (and required pc files) in both standard system locations and locations specified by the user with **PKG_CONFIG_PATH**.

B. Modulefile Basics and Standard Conventions

Modulefiles use a fully functional scripting language, Tcl, but interact with the user's environment through three primary module-specific commands: setenv, prepend-path, and append-path. Setenv will set an environment variable, replacing the value if it is already set; prepend-path and append-path will add the value to the beginning or end of an environment variable that may already contain a value. In both CrayPE 2 and xt-asyncpe there exist common environment variables that are manipulated for both compile and run-time usability:

```
PATH - List of directories that contain
user-level binaries
LD_LIBRARY_PATH - List of directories
that contain shared or static libraries
needed for link-time or run-time
MANPATH - List of directories that contain
man pages
MODULEPATH - List of directories that
could contain additional modulefiles
```

C. CrayPE 2 Basics

CrayPE 2 reads a number of environment variables to build both a **PKG_CONFIG_PATH** and *pkg-config* command, to be executed to get all compiler/linker flags. The most basic environment variable is **PE_PKGCONFIG_PRODUCTS**, which contains a list of products that are loaded and may have more product-specific environment variables dictating the system attributes for which the product has been compiled. In addition, there is also the variable **$*product_name*_[$*language*_]PKGCONFIG_LIBS**, (where $language can be 'C', 'CXX', 'FORTRAN', or omitted, to be inserted for all programming languages) that contains a list of *.pc files. A modulefile could append product_name to **PE_PKGCONFIG_PRODUCTS** and hpc_library to ***product_name*_C_PKGCONFIG_LIBS**, as well as append a path to **PKG_CONFIG_PATH** containing its pc files. Then, when the C-language drivers are executed, they will pass hpc_library to *pkg-config*, which will search **PKG_CONFIG_PATH** for the required pc file that contains hpc_library compiling and linking information. There are also basic environment variables that modulefiles can set giving CrayPE additional information such as, **$*product_name*_REQUIRED_PRODUCTS**, for information about what other products have required libraries (that need to be staged for a heterogeneous system) or **$*product_name*_MODULE_NAME**, so CrayPE error messages can print the problematic module.

D. CrayPE 2 Support for Single Build per Programming Environment

By default, *pkg-config* assumes that any required pc files that have a matching name and version are applicable and will return the first match found. When libraries have Fortran modules or C++ (name mangling), they typically will be compatible only with binaries that are compiled with the same programming environment: CCE, PGI, GNU or Intel. In order to insure that CrayPE finds only libraries that are compatible with the user's programming

environment, CrayPE has two different schemes that are available to product integrators: fixed path variables and volatile path variables. Modulefiles that follow the fixed path scheme can set variables with the following format: **PE_$*prgenv*_FIXED_PKGCONFIG_PATH**, with an absolute path to the pc files that are applicable to the "prgenv" programming environment. CrayPE 2 will add all fixed paths that are specific to the programming environment to **PKG_CONFIG_PATH** before executing *pkg-config*. Fixed paths are ideal when there is only one build per programming environment, because they are set at module-load time and don't require much processing. As a pass-through environment variable, CrayPE doesn't evaluate the paths in this list. If a product has been built for more than one but not all programming environments, the modulefile can set **$*product_name*_VOLATILE_PRGENV** to the programming environments that the module does support, allowing CrayPE 2 to detect if the product hasn't been built for the programming environment that the user has loaded.

E. CrayPE 2 Support for Multiple Builds per Programming Environment

Fixed paths are easy to understand and cover most custom integration requirements, but there are some situations where more builds are required, special selection behavior needs to be defined, or CrayPE run-time decisions are required. In these cases, modulefiles can set a dynamic path called **$*product_name*_VOLATILE_PKGCONFIG_PATH**. These paths leave an open-ended keyword everywhere the *pkg-config* path can vary and then CrayPE will substitute an absolute directory for each keyword based on other information provided by the module. For example:

```
/opt/cray/product_name/1.0.0/@PRGENV@/
@$product_name_GENCOMPILERS@/
@$product_name_TARGET@/lib/pkgconfig
```

All keywords denoted by "@" will be replaced with an applicable location and inserted into the **PKG_CONFIG_PATH** before *pkg-config* executing, ensuring only a single set of compatible libraries is found and returned by *pkg-config*.

1) Programming Environment Volatile Directories: The volatile path keyword @PRGENV@ is automatically replaced by the programming environment directory matching the user's loaded PrgEnv-* module, which can be CCE, GNU, PGI or INTEL. For improved error handling, the module can also set **$*product_name*_VOLATILE_PRGENV** to a list of programming environments that the product supports.

2) Generation Compiler Volatile Directories: When compiler vendors release new major compiler versions, library compatibility can be broken. GNU, with the gcc compiler, breaks Fortran module compatibility with almost every non-minor release. This means that if a library has been compiled with gcc/4.6.0 and a user is compiling an application with gcc/4.7.0, the libraries may not work with the user's application. In this example, the integrator could use the keyword, **@$*product_name*_GENCOMPILERS@** in the volatile path where each of the libraries' compiler build directories exist (the directories must match the name of the first two compiler digits, "4.6" and "4.7"), and set the variable **$*product_name*_GENCOMPILERS_GNU** with the directory names "4.6"and "4.7". Some compiler vendors typically attempt to maintain backward compatibility, so CrayPE has defined generation compiler fallback behavior. If the libraries have not been built with the user's compiler, CrayPE will substitute a version of the libraries that have been built with the closest older compiler. Here, if the user had gcc/4.8 loaded, CrayPE would substitute the 4.7 directory, but if the user had gcc/4.5 loaded, CrayPE would issue an error indicating that the user's compiler is too old to work with the libraries that exist on the system.

3) CPU Volatile Directories: In order to achieve optimal performance, some libraries are compiled multiple times with different CPU-specific optimizations enabled. If this is required, the integrator can install the various CPU-specific builds into directories that match the CPU name, then use the keyword **@$*product_name*_TARGET@** and set an environment variable that indicates both valid compilers and CPU targets. Libraries that have been compiled with optimizations for older CPUs will work when used in conjunction with an application that has newer compiler CPU optimizations. CrayPE 2 maintains a list of compatible CPU targets and will substitute the library directory that's been built with CPU optimizations that mostly match the user's targeted CPU. For example, if a library has been built only with Intel Ivybridge optimizations and the user is targeting the Haswell CPU, CrayPE 2 will substitute the Ivybridge directory into the VOLATILE path. If the user has opted to build an executable with no optimizations, x86_64 (for cross-CPU compatibility), then the user would receive an error stating that the only existing library build contains incompatible CPU optimizations. Cray products that provide a CPU-specific optimized build also include a build without optimizations so CrayPE will always find a compatible target.

F. Arbitrary Package Configuration pkg-config Variables

pkg-config allow the user to have variables in the *pkg-config* file that will have a value defined in the *pkg-config* file but can be overridden on the command line at *pkg-config* execution time. In order to give CrayPE 2 greater flexibility with potential future integration requirements, CrayPE 2 includes a method of reading module-set environment variables to conditionally supply *pkg-config* with variable definitions. Here the module has to define a variable name, which contains two parts: a key-word that dictates conditional behavior, and the variable name as it appears in the pc file. The modulefile must also define

variables containing definitions for any potential conditional evaluation. If *product_name* has libraries compiled both with and without OpenMP (an API that supports multi-platform shared memory multiprocessing programming) in the same directory with one archive called lib*product_name*.a and another called lib*product_name*_mp.a, the pc file libs section could contain -l*product_name***$OMP_SUFFIX**. A module file could define **$*product_name*_PKGCONFIG_VARIABLES** as OMP_SUFFIX_@openmp@,
$*product_name*_OMP_SUFFIX_openmp as _mp which would trigger CrayPE to apply a
"--define-variable=$*product_name*_OMP_REQUIRES=_mp" to *pkg-config*, when OpenMP is enabled, linking in the OpenMP specific library.

G. Default Product Dependency Resolution

When the CrayPE 2 module is loaded, it sources a number of modulefile segments that contain information pertaining to all default products. This allows CrayPE 2 and therefore *pkg-config* to resolve dependencies without loading the dependent module. Functionally this reduces over-linking by allowing *pkg-config* to only return libraries that satisfy dependencies, not all libraries that are provided with a module. Viewing variables set by the CrayPE module allows the user to gather information about the current state of all default products.

H. CrayPE 2 Integration Tools and Debugging

1) craypkg-gen: The heart of CrayPE product integration requires nothing more than adding applicable .pc files and modulefiles to a product's directories, but many software developers aren't familiar with either tclsh/modulefiles or .pc files. To provide a good starting point to people who wish to integrate a product into the CrayPE software stack, and to lower the amount of time required to integrate a product, Cray developed and released a tool called *craypkg-gen*[3], which generates both modulefiles and pc files.

2) CrayPE 2 Debugging Options: In order to decrease the complexity in CrayPE2's infrastructure, we wanted all product-specific integration steps to be abstracted to their own layer, accessible and able to be bypassed. For this we added a number of CrayPE2-specific options that enable it to print the flags returned by *pkg-config* without compiling, the **PKG_CONFIG_PATH** that is generated and used by *pkg-config*, the .pc files passed to *pkg-config*, and custom *pkg-config* variable declarations. This enables users, if desired, to use CrayPE to see what linking/compiling options are being generated or to stage their environment for *pkg-config* to be used independently of CrayPE 2's front-end drivers.

Some third-party build tools aren't engineered to handle extra options being passed to the compiler, and some users want to be able to print the options, manipulate them, and pass the updated options directly to the compiling driver without the front-end adding more options. To support this, CrayPE 2 has an option to bypass *pkg-config* execution.

IV. Conclusion

CrayPE 2's primary objective was to allow HPC users to integrate their own products within the Cray Programming Environment, with the same capabilities and features that are required to support Cray-released products. Through the creation of an open API this has been made a reality – users can add simple products quickly and use whatever functionality they need for varying language, dependencies, compiler, or accelerator support. Because CrayPE 2 doesn't need to be updated on a monthly basis to add product support, more time can be devoted to feature development. As with most major redesigns, the bug rate initially went up, but has since fallen to levels below CrayPE 2's initial release. Because CrayPE 2 has well-defined behavior, uses *pkg-config*, and is compiled, it now takes one one-hundredth of the time to generate the PE library stack before compilation. Cray looks forward to hearing what HPC sites and users choose to do with their additional capabilities.

Acknowledgment

CrayPE 2 heavily leverages the well-designed and well-executed software, *pkg-config*, currently maintained by *pkg-config* developers, Tollef Fog Heen and Dan Nicholson, and hosted by freedesktop.org.

References

[1] F. Heen and D. Nicholson "pkg-config" freedesktop.org 2 July 2013. Web. 06 Apr. 2015

[2] Cray Inc. Man page: CrayPE_API April 6, 2015

[3] Cray Inc. Man page: craypkg-gen April 6, 2015