opensolaris

CZOSUG & GUUG

developer conference

June 25 - 27, 2008, Prague

PROCEEDINGS

June 25 - 27

Prague, Czech Republic

CONTENT

Title	Author	Page
Governing OpenSolaris: Get Involved!	Grisanzio Jim	3
"The Package Man Always Builds Twice" From SysV Packaging To The Image Packaging Systém	Brandt Volker A.	5
OpenSolaris Testing	Walker Jim	9
The Cryptographic Framework in OpenSolaris	Ley Wolfgang	10
New Security Features in OpenSolaris and Beyond	Pechanec Jan	24
ZFS On-Disk Data Walk (or: Where's my Data?)	Bruning Max	36
Empowering OpenSolaris* Developers on Intel Hardware	Stewart David C.	58
Using DTrace API to write my own consumer	Škovroň Petr	70
Experiences with DTrace development: stddev() and brendan()	Mynhier Chad	77
OpenSolarisTM and NUMA Architectures	Polanczyk Rafael V.	86
Translation of OpenSolaris	Tomášek Petr	96
Presto - Printing Makes Easy	Teo Ghee S.	104

AUTHORS AND CO-AUTHORS INDEX

Authors and co-authors	Page
Brandt Volker A.	5
Bruning Max	36
Černošek Aleš	96
Grisanzio Jim	3
Ley Wolfgang	10
Malovec Róbert	96
Mynhier Chad	77
Pechanec Jan	24
Phalan Mark	24
Polanczyk Rafael V.	86
Schuba Christoph	24
Stewart David C.	58
Škovroň Petr	70
Teo Ghee S.	104
Tomášek Petr	96
Walker Jim	9

Governing OpenSolaris: Get Involved!

Jim Grisanzio
Sun Microsystems
Tokyo, Japan
jim.grisanzio@sun.com

Overview

This is ultimately a discussion about getting involved in governance on the OpenSolaris project -- either in governance itself or in any number of roles that will help you earn Membership and Core Contributor status in the community. In many ways, governance is just another form of community development [1], and there are many social, strategic, and technical factors involved. The governmental systems on OpenSolaris are still evolving, though, and some of the issues have been controversial. This is normal. It's simply the evolution of a complex and unique engineering project, one in which a multi-billion dollar corporation has opened its core product and is building a global community while still maintaining business operations.

We will dig into the evolution of governance from the early Community Advisory Board (CAB) during the Pilot Program right up through today's OpenSolaris Governing Board (OGB) . Topics will include how the Charter and Constitution [2] were developed, the theory and structure behind the OpenSolaris community, the involvement and reactions of the community at various points, what has worked well and not so well, how the election mechanism works, what we've learned in three years, and most importantly, where we are going with the election of the new OGB. The context for the conversation centers around how to participate in the OpenSolaris project and how we can all build a global community of developers, administrators, and users around a large base of source code, binaries, and tools.

Three Phases of Governance

There have been three clearly definable phases of governance on the OpenSolaris project:

(1) Sun's role in creating the CAB from within the Pilot Program, the development of (and confusion about) the Charter and Constitution, the redefinition of the CAB to OGB and the expansion of its mandate and extension of its term, and the ratification of the Constitution and election of the first OGB. This period of time ranges from 2005 to March 2007.

(2) The first elected OGB begins normal operations with a Constitution that doesn't necessarily reflect reality, but many people on the board and in the community make a good faith attempt to make things work. The OGB controls no resources and key parts of the project are still internal to Sun. A community reorganization is specified and attempted but stalls due to lack of interest, disagreements, and no resources to develop the infrastructure to reflect the reorganization. Trademark disputes over a new Sun distro lead to more arguments about the project's lack of openness in many areas. This second phase was March 2007 to March 2008.

(3) The second OGB takes office after a significant community argument with Sun. Most members ran on a platform to reform the governance and reorganize [3] the community for two key reasons: the OpenSolaris community and Sun need to be in sync about the project, and the structure of the OpenSolaris community needs to reflect the reality of how the community actually functions in real life. The re-org can now take place because the OpenSolaris engineering infrastructure team has resources to update the website, move the gates external, and finish the work necessary to make OpenSolaris an open development project -- which was the goal all along. The reorganization is not fully specified yet, but discussions have begun, and Sun's executive engineering management has engaged in the process. This current phase started from March 2008.

The Reorganization

Currently the OpenSolaris community is structured around Members, Community Groups, and Projects. Community Groups sponsor Projects and grant Membership status to Core Contributors. There are some odd groupings. as well, such as user groups, which we have as Projects due to site constraints, and we

have many Community Groups that were crafted back in the Pilot Program that really ought to be Projects today. And Sun has yet another grouping called Consolidations, which doesn't fit the Constitution. The website also doesn't reflect the Constitution, since the site pre-dated the Constitution and its evolution stalled due to resource constraints. So, the OGB's community reorganization has to address all of these issues.

To begin the process of discussing the issue, the OGB recently proposed interpreting the term Community Group to mean a class of groupings in the community, not a group itself. In other words, Projects, Consolidations, Special Interest Groups, and User Groups could potentially make up the new groupings and they could have relationships with each other in a web-like structure instead of the current hierarchical structure. To establish some consistently across the community under this proposed system, a new OGB committee would be formed to create standards for granting Membership status.

The reorganization idea has already generated several counter proposals, but the general concept is moving in the direction of offering more flexibility for different types of groups and crafting a system that reflects how OpenSolaris developers work rather than imposing an artificial structure on top. The reorganization and the infrastructure work necessary to support the changes will probably take a year, so there is plenty of time to get involved and contribute. The site can not be changed rapidly, and neither can a governance decision. The site's current monolithic architecture is being updated in stages to a new modular architecture, but it still must support current operations. And the community will have to participate in and finally approve any new governance structure.

Finally

The OpenSolaris governance process should define how the community operates. Therefor, it's at the core of how people participate in the project. It should not get in the way of participation, but it should offer opportunities for many people to contribute in many ways. All of these issues will be flushed out in the talk, along with a community update and many stories about how the community had come into being.

1 Literature

[1] Building Communities: http://blogs.sun.com/jimgris/tags/building

[2] OpenSolaris Charter, Constitution, and Governing Board: http://opensolaris.org/os/community/ogb/

[3] Proposed Reorganiztion: http://mail.opensolaris.org/pipermail/ogb-discuss/2008-May/005572.html

“The Package Man Always Builds Twice”
From SysV Packaging To The Image Packaging System

Volker A. Brandt
Brandt & Brandt Computer GmbH
Am Wiesenpfad 6
53340 Meckenheim
Germany
vab@bb-c.de

1 Summary

For over 20 years, the familiar *System V Packaging Tools* have brought software to the Solaris environment in an orderly fashion. The tools have worked well and reliably on many operating systems, but show their age in a few places. They are still recovering from the many changes and additions that the introduction of zones into Solaris has caused. Several key features needed in a modern network-based environment are either missing or have been more or less crudely bolted on, such as package encryption and authentication, dependency management, or network package repositories.

All of this is about to change with the introduction of IPS, the new *Image Packaging System* for OpenSolaris. This new style of building packages tries to fix the problems with the System V tools, and bring Solaris packages and software deployment into the 21st century. The IPS project wants to provide a solid software delivery process with explicit dependencies based on network repositories. It also caters to the needs of distribution builders, who want to assemble their custom OpenSolaris distribution from many package building blocks.

However, the IPS is still very much a moving target. Many features are still in the design phase or available more or less as a proof of concept.

This presentation attempts to describe the main features of the Image Packaging System. We also try to give the prospective OpenSolaris developer some guidance on how to design and build software packages in the OpenSolaris environment.

2 Introduction

For many years, the *System V Packaging Tools* (SysV pkgs) were the preferred method of delivering software to Solaris operating system environments. A standard defined by AT&T, the package format has remained largely unchanged and is used in a number of other System V Unix-derived operating systems from other vendors.

Times have changed since the design and first implementation of the SysV pkg environment, and many features now necessary to keep the pace in OS development are either completely impossible or added only with high implementation and maintenance efforts.

3 The Good, the Bad, and the Ugly

3.1 Just the Facts, Ma'am

The main metadata files are the package info and the package map file (in this century, we would call it the "package manifest"). Both of these files are in ASCII and do not contain any checksum or other means of content protection or verification.

Sys V packages are managed with a canonical command set, `pkgadd` and `patchadd` to add them (remember, patches are sets of packages, too). The corresponding removal tools are `pkgrm` and `patchrm`, and there are several informational tools as well.

Sys V packages use a flat namespace. There is only one level of packages, no groupings or metapackages. A package can be partially installed, but usually this indicates some level of brokenness. Depending on the definition of the package (the MAXINST variable in the package info file), it may or may not be installed on a system more than once. Multiple instances are indicated by a counter suffix (e.g. `BBCgreatstuff` and `BBCgreatstuff.1`).

There are several types of deliverable package objects (such as hard or symbolic links, regular files, directories, etc.). A certain level of dynamic content selection is achieved by substituting path name components by variables in the package map, and by dividing package objects in *classes*, and then selecting only certain classes to install. Classes can also be installed by custom scripts provided alongside in the package (again, these scripts run as root).

3.2 Wear and Tear

The SysV pkg tools show their age. They have been hacked on and extended by a cast of many contributors. Focusing on Solaris, several important features have been added that the original designers never dreamt of. These include "compressed packages" (implemented by internal class action scripts), signed packages, the ability of adding packages across a network ("WAN boot"), the management of closed off containers ("Solaris zones"), and many more.

Nobody really likes hacking on the tools themselves much any more. Several vendors even use their own tools to create and add packages into a Solaris instance, because the SysV pkg tools are widely perceived to be too complicated. This of course creates new problems...

3.3 Ten Things We don't Like About You

Many shortcomings of the traditional packaging system are more and more felt as Solaris evolves:

- It is difficult to build a "good" package; there are best practises and examples but it is not very accessible for the "average" developer.
- Parameter provisioning for packages is difficult; either packages ask for them interactively, or the operator has to provide so-called response files. Both methods do not lend themselves well to hands-off provisioning or deployment of virtual servers.
- The packaging tools do not scale well, it is common to have 1000+ packages with 1000000+ objects on an enterprise-level system. Package and patching operations take a long time and are inflexible. Sun tried a database as the central package information source, but performance did not really improve.
- Dependency management is on a package-to-package level and depends on the package builder to explicitly specify such dependencies. There is no way to specify dependency trees, let alone a method to automatically install all other packages a given package depends on.
- The data and metadata within a package are only protected against accidental or intentional modification or damage very primitively. There is a checksum for the package objects maintained in

the package map, but the map itself is completely unprotected. There are no signatures etc.

- Package grouping and package sets do not exist, except in a very limited sense, and only for the Sun-provided "core Solaris" packages within the Solaris installation context.
- There is no way to version control packages, and have concurrent installation of differing versions in a consistent fashion. Change management can only be done manually or in some separate tool; the system does not know about possible predecessor-successor relations among packages.
- Too many things need to be done by root; arbitrary things can be done inside the package scripts; there is no obligation to record the actions of a package script in the package database.
- The packaging system is file-based only. Packages are well-formed trees in some corner of the file system (or a `cpio` stream that gets unpacked in said corner). Then they are copied to some other corner of the file system. They are not aware of any of the nice new features Solaris 10+ offer, like ZFS snapshots and rollback, SMF services to denote service dependencies and perform startup tasks, Solaris Containers to compartmentalize and virtualize servers, etc.
- The packages were originally intended to be used locally. All network-related features (like Solaris installation across NFS, network URLs in the jumpstart profile, etc.) were added later, and these continued add-ons and extensions make life complicated in a network-centric environment. There is no explicit support for a package repository of any kind.

Several of these problems have been overcome individually by third-party tools, integration efforts (like custom jumpstart), or additions made by Sun to the package tools themselves. But taken together, they show the gap between the traditional Solaris tools and modern distribution and installation systems used by such systems as Debian and Red Hat. Clearly, something needed to be done.

4 Mission Possible

4.1 Image Packaging What?

The Image Packaging System is Sun's answer to the problems described above. Continuing the new method of throwing away all old assumptions and re-thinking everything from scratch, a new system was designed. Basic premises were:

- The new system must be network-centric, providing one or more package repositories, a client to insert packages into the repository, and another client to retrieve them on an installation target system.
- The new system must leverage ZFS, SMF/FMRI, and all the other nice new technology.
- Dependency handling must be "as automatic as possible". Specifically, all dependencies of a given installation package must be loaded without the need for manual intervention.
- The set of installation targets must not be restricted to the current Solaris instance the package installation tools are being run on. A normal system user must be able to prepare installation images for use in virtualization or provisioning environments.
- Ease of use for both package builders and package consumers must be increased.

4.2 Bricks and Stones

The key concept of the IPS is the image. An image is just a location where packages and their member objects (such as files) can be installed. An example for an image is the current Solaris instance. Other examples are trees on an install server, CD-ROM images for operating system distributions, and multiple alternative boot environments on the same physical server. An image presents an environment containing a set of packages which can be installed, searched, updated, removed, and generally managed in a reproducible fashion.

The packages are provided by a network service, usually called depot or repository. In the repository, the packages are available for download, together with their metadata ("catalogs"). This data can be retrieved on-demand, or cached on the local system.

A package will usually have attributes. These can describe aspects of the package, or contain "actions". The actions are a powerful concept to describe what a package should "do" with a given object. Most of the time, an action will be simply a file creation, or the creation of a directory path. But it can also specify the addition of a driver object to the Solaris device driver set, an SMF service manifest, and many other types of objects.

A given repository in a depot server together with an access method (such as SSL with certificates) makes up an "authority". It is possible to specify several of these authorities, and to dynamically select one when issuing the command to add one or more packages to a target image.

4.3 All Is Not Lost

Since there is such a huge installed base of SysV packages, the current version of OpenSolaris still support the traditional commands to install them. However, the mechanism is quite different from the IPS. This in effect means that there are two only loosely coupled package management systems active on the same OpenSolaris server.

An alternative to simply deploying old-style SysV packages on the system is to import them into the IPS depot server. This is infinitely preferable. Unfortunately, this process strips away all pre- and post-installation and class action scripts, all parameterized paths, etc.; in short, a SysV package that uses any of these features is broken after it is stuffed into the IPS repository.

5 Back to the Future

The main goal of this presentation is to give some advice to the OpenSolaris developer willing to package software for deployment on OpenSolaris. This is not so easy, since the Image Packaging System is still very much a moving target, and not all features are cast in stone yet, let alone implemented. The strategy we suggest is to isolate the packaging system specific tasks as much as possible, and then to implement these tasks both for the traditional System V packaging environment, and the new Image Packaging System.

In short, "The Package Man Always Builds Twice".

6 Literature and Links

The best places to start learning more about the Image Packaging System are the OpenSolaris Installation and Packaging community web site at

```
http://opensolaris.org/os/community/install/
```

and the IPS project web pages at

```
http://opensolaris.org/os/project/pkg/
```

Also, a much more extensive list of links for this presentation is maintained at the author's site:

```
http://www.bb-c.de/osdevcon2008/
```

OpenSolaris Testing

Jim Walker
Sun Microsystems, Inc.
500 Eldorado Blvd.
Broomfield, CO 80021
USA
james.walker@sun.com

1 Overview

Testing is an important part of software development. In order to maintain the quality of the OpenSolaris code base, we need to perform extensive testing before new code can be integrated. The OpenSolaris Testing presentation [1] discusses the current testing resources available on opensolaris.org that help ensure code quality. The presentation is divided into the following three sections.

OpenSolaris Test Suites and Tools

This section reviews the current OpenSolaris test suites [2] and test tools available on opensolaris.org and what is planned in the future. In addition, it will discuss how to download and run the test suites and describe the various communities and projects doing test development, and how developers and test engineers can contribute.

OpenSolaris Self-Service Testing

This section describes the fully automated OpenSolaris Self-Service Testing [3] application where functional and performance tests can be requested. It will detail how OpenSolaris Contributors can upload their code, submit test requests and review test results online. This section includes an online demo.

OpenSolaris Test Farm

This section describes the OpenSolaris Test Farm [4] where OpenSolaris Contributors can access high end server hardware to test their code. Developers and test engineers can reserve and login to test machines remotely and test their code before it is submitted for integration approval. It will cover how to get access to the test farm, how to reserve test machines and how to use the test machines. Currently the test farm has the following machine types: T2000, X4200M2, X4600M2, X4150, T5120. This section includes an online demo.

2 Literature

[1] OpenSolaris Test Suite webpage: http://www.opensolaris.org/os/community/testing/testsuites/

[2] OpenSolaris Self-Service Testing webpage: http://www.opensolaris.org/os/community/testing/selftest/

[3] OpenSolaris Test Farm webpage: http://www.opensolaris.org/os/community/testing/testfarm/

[4] Current OpenSolaris Testing presentation: http://www.opensolaris.org/os/community/testing/files/opensolaris_testing.pdf/

The Cryptographic Framework in OpenSolaris

Wolfgang Ley
Sun Microsystems
Nagelsweg 55
20097 Hamburg
Germany
wolfgang.ley@sun.com

1 Overview of the Crypto Framework

The Cryptographic Framework was first introduced in Solaris 10 and is still under development with new features being introduced in current Nevada builds.

The framework consists of a user-level interface, a kernel-level interface and supporting administrative commands to control the framework. For each part (userland or kernel) the framework offers APIs for *consumers* (someone who is using the cryptographic functions in his own project) and for *providers* (someone who is offering a certain implementation of a cryptographic service).

Using the cryptographic framework has many benefits. Applications and kernel services do not need to re-implement complete cryptographic functions but can use existing implementations. This avoids duplicate code and therefore reduces the number of bugs. In addition, the cryptographic implementations can be optimized for the available hardware (including hardware accelerators) and the resulting greater performance is directly available to the userland and kernel services.

This paper provides an overview of the entire framework, the APIs and administrative interfaces. We will also show some example consumers and provide code examples of how to use the various available APIs.

2 Glossary

Documentation, source code, comments and standards do use various specific terms and it is important to understand these terms before trying to understand (or use) the cryptographic framework. You might have heard these terms before (e.g. if you have read a standard) but for those new to cryptographic work (and/or the Cryptographic Framework in OpenSolaris) it is crucial to have a common understanding of certain terms.

2.1 RSA PKCS#11

RSA Laboratories (the research center of RSA, now part of EMC [9]) have developed a family of standards called Public-Key Cryptography Standards (PKCS). The first standard PKCS#1 only specified the RSA algorithm together with the key formats but today there are 15 RSA standard documents. These standards define several aspects of applied cryptography. Some of the more commonly used standards are

- PKCS#7 (Cryptographic Message Syntax, also used by RFC 2315)
- PKCS#8 (Private-Key Information Syntax, e.g. used by Apache)
- PKCS#11 (Cryptographic Token Interface, an API also used by the Cryptographic Framework in OpenSolaris)
- PKCS#12 (Personal Information Exchange Syntax, e.g. used by Java key store encrypted private keys together with public key certificates).

Talking about the Cryptographic Framework in OpenSolaris the PKCS#11 is particularly important. This standard defines an API (*Cryptoki*, pronounced crypto-key) and the Cryptographic Framework offers this API in its current version 2.20. If you are already familiar with the PKCS#11 API then you can directly use this API in your applications by simply linking against `libpkcs11.so`. PKCS#11 is an important API but depending on your needs, other simpler interfaces are also available (we will discuss those later and show a comparison in the example section).

2.2 Consumer

An application, library or kernel module that is requesting a certain service from the cryptographic framework (e.g. to encrypt or decrypt some data) is called a *consumer* if it just obtains these services. For example, the `digest` command of OpenSolaris is a consumer of the framework (via libraries). This application does not include the code to perform MD5, SHA1 etc. directly but calls the cryptographic framework to do these calculations instead.

2.3 Provider / Plug-in

A userland library or kernel module which offers cryptographic services to consumers is called a plug-in. The service is plugged into the cryptographic framework (to be visible and usable by the consumers) and therefore a provider is sometimes also just called a plug-in. Example plug-ins in OpenSolaris to offer hardware accelerated crypto services are the `ncp` kernel hardware provider to use the Mathematical Arithmetic Unit (MAU) of the UltraSPARC T1 CPUs or the `n2cp` kernel hardware provider to utilize the Control Word Queue (CWQ) of the UltraSPARC T2 CPU.

2.4 Mechanism

One possible assumption could be that a provider implements one or more cryptographic *algorithms* (such as DES, AES, RSA, SHA1 etc.). This however is only partially true. Each algorithm may be used in multiple ways. A *mechanism* is the combination of one specific algorithm (e.g. DES3 in CBC mode) and the way it is being used (e.g. using DES for authentication is different from using it for encryption).

An example mechanism in OpenSolaris is SHA512_HMAC where the SHA512 algorithm is being used as as a Hashed Message Authentication Code (HMAC) and the implementation is a keyed hash function to digitally sign something (which can be used for authentication).

The term mechanisms is very specific. It does define the algorithm together with the way it is being used and therefore also specifies the kind of arguments required for this usage (e.g. compared to a regular hash function a keyed hash function obviously needs an additional argument: the key).

2.5 Token

A *token* is not a key, but rather the abstraction of a device that can perform cryptography. A token implements one or more mechanisms and additionally has the ability to store information for use in these mechanisms (e.g. a key, initialization vector, state etc.). A token might be a hardware device (e.g. an accelerator board) or a piece of software (which is called a soft token). An example token in OpenSolaris is the `pkcs11_softtoken(5)` token which implements the RSA PKCS#11 v2.20 specification in software.

2.6 Slot and Metaslot

Within the PKCS#11 standard the application is using a *slot* to actually connect to a certain cryptographic service. Inside the Cryptographic Framework the previously described tokens can therefore be plugged into such slots to make them available to consumers. Having multiple available slots adds some complexity to the consumer who must select a particular slot before using the service. In addition the consumer might not know which slot is the "best" (e.g. which slot is using hardware acceleration on a given platform).

The Cryptographic Framework offers a simple solution for multiple providers: a *metaslot*. The *metaslot* is a virtual slot using a superset of all available mechanisms. Mechanisms with hardware acceleration are chosen before software software implementations. The cryptographic library (`libpkcs11.so`) uses this metaslot as default (unless an application selects another specific slot).

Occasionally consumers might want to select a specific slot instead of the default metaslot. An example would be an application which uses key generation and/or key storage on a certain hardware device like a smartcard.

2.7 Session

A *session* is an active connection between the application and a token. Consumers can open multiple sessions with one or more tokens (and of course providers of tokens can have multiple sessions with different applications). If you use multiple sessions with the same token inside one application then some of the PKCS#11 functions may have side effects (e.g. if a token requires a login like a PIN then the `C_Login()` function will affect all current sessions of the application with this token).

2.8 Objects

Objects store various information (e.g. keys). The PKCS#11 standard defines two different types of objects. *Session objects* are only valid while the session itself is active while *token objects* persist beyond the length of a session. Token objects must therefore be stored somewhere.

The default location for OpenSolaris soft token objects is `$HOME/.sunw/pkcs11_softtoken` but the default of `$HOME/.sunw` can be changed by using a different path in the `$SOFTTOKEN_DIR` environment variable. Other tokens may store their token objects somewhere else, including on hardware.

3 User-Level Cryptographic Framework (uCF)

The User-Level Cryptographic Framework contains various components which will be described in this section. After explaining these components and the kernel level framework we will show how these components work together to form the entire Cryptographic Framework in OpenSolaris.

The main parts of the user-level framework are:

- Libraries to be used by consumers (`libpkcs11.so` and others)
- A software plug-in (`pkcs11_softtoken.so`) which is one default provider
- A plug-in to access cryptographic hardware devices via the kernel (`pkcs11_kernel.so`)
- An administrative interface to control providers and mechanisms (enable, disable etc.)
- A subsystem to sign and verify cryptographic modules (providers)
- Example consumers (applications) which are using the framework

This is obviously not a complete list as the framework can be extended by additional pluggable tokens to support more features.

3.1 Libraries

The main library of the user-level framework is `libpkcs11.so` which implements the RSA PKCS#11 API in version 2.20. The specification can be found in [1].

The `libpkcs11.so` library has two convenience functions, `SUNW_C_GetMechSession()` and `SUNW_C_KeyToObject()`, that are not part of the PKCS#11 standard. These are to simplify the initialization of sessions and generate key objects, and are documented in the libpkcs11 manpage..

The OpenSSL Library (`libssl.so` which is installed in `/usr/sfw/lib/`) offers an engine plug-in API to add additional features to this library. OpenSolaris includes a special `pkcs11` engine for OpenSSL and once this engine has been activated all OpenSSL calls will go into `libcrypto.so` (to compute the crypto operations) and will therefore benefit from the Cryptographic Framework. Note that this addition is not yet in the freeware version bundled (but available via the contrib section of the OpenSSL site. See [11] for details about this PKCS#11 engine patch for OpenSSL.

In addition to the general purpose PKCS#11 standard library and the OpenSSL library, OpenSolaris contains a message digest library to provide simple access to hashing routines (MD4, MD5, SHA1, SHA256, SHA384 and SHA5112). The `libmd.so` has a simple interface (e.g. `SHA2Init()`, `SHA2Update()` and `SHA2Final()`) and is using the same software implementation that pkcs11_softoken uses.

Apart from these public interfaces OpenSolaris contains two additional private libraries. These libraries should not be used by your own applications but you might stumble across them when browsing the OpenSolaris source code. The two private helper libraries are `libcryptoutil.so` and `libelfsign.so`. The `libcryptoutil.so` includes a few simple wrapper functions around the PKCS#11 functions and also offers some debugging of those libraries (just set the environment variable `SUNW_CRYPTO_DEBUG` to either "syslog" or "stderr" to activate this debugging). The `libelfsign.so` library handles ELF signatures and is used by utilities like `elfsign(1)` but also by the `kcfd(1M)` which is responsible for verification of the signatures of cryptographic providers during runtime.

Section 3.5 contains more information about signature management and verification.

3.2 Software plug-in

OpenSolaris is shipped with one software token provider (`pkcs11_softtoken`) which includes various PKCS#11 mechanisms (including hashes, encryption, decryption, key generation etc.). The complete list of available mechanisms is available in the `pkcs11_softtoken(5)` manual page but is also visible through the administrative commands discussed in section 4 (see example output there).

Previous Solaris releases had an optional provider `pkcs11_softtoken_extra.so` to include longer keylengths than the `pkcs11_softtoken.so` provider supported. Starting with OpenSolaris this provider is obsolete and the `pkcs11_softtoken.so` contains the strong crypto. Note that the older packages `SUNWcry` and `SUNWcryr` have also been obsoleted and removed from OpenSolaris.

3.3 Plug-in to access hardware via the kernel

The provider to access hardware devices from the user-land (via the kernel) is always installed and active. If you do not have any hardware acceleration in the system then this provider will not offer any slots and all consumers will need to use the soft-token implementation instead. Depending on the installed hardware, additional mechanisms will show up in this slot. External hardware accelerator cards are available but not widely used. Some platforms however now already include hardware crypto acceleration as part of their design. Examples are the UltraSPARC T1 CPUs (e.g. T1000, T2000 etc.) which offer asymmetric functions (DSA, RSA, DH) and UltraSPARC T2 CPUs (e.g. T5120, T5140, T5220, T5240 etc.) which additionally support DES, DES3, AES, RC4, MD5, SHA1 and SHA256. External cards such as the PCI Express Sun Crypto Accelerator 6000 card offer even more mechanisms including key generation and storage.

Work is ongoing to implement other hardware providers (e.g. the VIA PadLock support).

3.4 Administration

We have already learned that the OpenSolaris framework is highly flexible but have not discussed how to show or modify its configuration. OpenSolaris has one single administrative command to control the entire framework: `cryptoadm(1M)`. Note that this command is also used to control the kernel part of the framework --- which will be discussed in section 4.

It is important to understand the terminology (as explained in section 2) before using this command. The most important subcommands are `list` (to show the current configuration), `enable` or `disable` (to activate or deactivate providers or mechanisms) and `install` or `uninstall` (to add or remove additional providers). The details are explained in the manual page of `cryptoadm(1M)` but I do want to show a few examples to provide a better understanding of the framework. The following examples are from a T1000 system.

```
t1000# cryptoadm list

User-level providers:
Provider: /usr/lib/security/$ISA/pkcs11_kernel.so
Provider: /usr/lib/security/$ISA/pkcs11_softtoken.so
```

```
Kernel software providers:
        des
        aes
        arcfour
        blowfish
        ecc
        sha1
        sha2
        md4
        md5
        rsa
        swrand

Kernel hardware providers:
        ncp/0


t1000# cryptoadm list -p

User-level providers:
=====================
/usr/lib/security/$ISA/pkcs11_kernel.so: all mechanisms are enabled.
/usr/lib/security/$ISA/pkcs11_softtoken.so: all mechanisms are enabled.
random is enabled.

Kernel software providers:
==========================
des: all mechanisms are enabled.
aes: all mechanisms are enabled.
arcfour: all mechanisms are enabled.
blowfish: all mechanisms are enabled.
ecc: all mechanisms are enabled.
sha1: all mechanisms are enabled.
sha2: all mechanisms are enabled.
md4: all mechanisms are enabled.
md5: all mechanisms are enabled.
rsa: all mechanisms are enabled.
swrand: random is enabled.

Kernel hardware providers:
==========================
ncp/0: all mechanisms are enabled.


t1000# cryptoadm list -mv 'provider=/usr/lib/security/$ISA/pkcs11_kernel.so'
Provider: /usr/lib/security/$ISA/pkcs11_kernel.so
Number of slots: 1

Slot #1
Description: ncp/0 Crypto Accel Asym 1.0
Manufacturer: SUNWncp
PKCS#11 Version: 2.20
Hardware Version: 0.0
Firmware Version: 0.0
Token Present: True
Slot Flags: CKF_TOKEN_PRESENT CKF_HW_SLOT
Token Label: ncp/0 Crypto Accel Asym 1.0
Manufacturer ID: SUNWncp
Model: NCP1
Serial Number:
Hardware Version: 0.0
```

```
Firmware Version: 0.0
UTC Time:
PIN Length: -1--1
Flags: CKF_WRITE_PROTECTED
Mechanisms:
                                                         P
                                                 S   V K a   U D
                                                 i   e e i   n e
                                               S g V r y r W w r
                                         E D D i n e i G G r r i
                                       H n e i g + r + e e a a v E
mechanism name               min  max  W c c g n R i R n n p p e C
---------------------------- ---- ---- - - - - - - - - - - - - - -
CKM_DSA                      512  1024 X . . . X . X . . . . . . .
CKM_RSA_X_509                256  2048 X X X . X X X X . . . X . .
CKM_RSA_PKCS                 256  2048 X X X . X X X X . . . X . .
CKM_RSA_PKCS_KEY_PAIR_GEN    256  2048 X . . . . . . . . X . . . .
CKM_DH_PKCS_KEY_PAIR_GEN     512  2048 X . . . . . . . . X . . . .
CKM_DH_PKCS_DERIVE           512  2048 X . . . . . . . . . . . X .
CKM_EC_KEY_PAIR_GEN          163  571  X . . . . . . . . X . . . .
CKM_ECDH1_DERIVE             163  571  X . . . . . . . . . . . X .
CKM_ECDSA                    163  571  X . . . X . X . . . . . . .
```

```
t1000# cryptoadm disable provider=ncp/0 mechanism=CKM_RSA_X_509
t1000# cryptoadm list -p

User-level providers:
=====================
/usr/lib/security/$ISA/pkcs11_kernel.so: all mechanisms are enabled.
/usr/lib/security/$ISA/pkcs11_softtoken.so: all mechanisms are enabled.
random is enabled.

Kernel software providers:
==========================
des: all mechanisms are enabled.
aes: all mechanisms are enabled.
arcfour: all mechanisms are enabled.
blowfish: all mechanisms are enabled.
ecc: all mechanisms are enabled.
sha1: all mechanisms are enabled.
sha2: all mechanisms are enabled.
md4: all mechanisms are enabled.
md5: all mechanisms are enabled.
rsa: all mechanisms are enabled.
swrand: random is enabled.

Kernel hardware providers:
==========================
ncp/0: all mechanisms are enabled, except CKM_RSA_X_509.
```

3.5 Subsystem for verification of providers

All providers which are plugged into the framework must have a valid ELF signature. Signatures can be managed by the `elfsign(1)` command. Cryptographic providers must be signed with a certificate that has been signed by Sun Microsystems Inc. This is required to comply with US export restrictions regarding pluggable cryptography.

The procedure and requirements to apply for a certificate are explained in appendix F of [2].

During runtime the kernel-level cryptographic framework daemon (`kcfd`) is responsible for verifying the digital signature of the provider. This daemon is automatically invoked in run level 1 after /usr is mounted. This is needed because all other cryptographic services rely on this daemon. You will not be able to use the

Cryptographic Framework in OpenSolaris at all if you disable this daemon..

The functions for signature creation and validation can be found in the `libelfsign.so` library which is not a public interface though.

3.6 Example consumers

Various parts of OpenSolaris use the Cryptographic Framework. If you want to play around with simple applications (or read the source code to have examples for your own code) then check the commands

- `digest(1)` (to calculate message digests like SHA512)
- `mac(1)` (to use a hash function together with a key as a message authentication code)
- `encrypt(1)` and `decrypt(1)` to do actual encryption.

All of these commands have an `-l` option to display the available mechanisms

Another notable consumer is the `pktool(1)` utility which can be used to manage keystores for various other users, such as Mozilla NSS, OpenSSL or PKCS#11 directly.

```
% cat /etc/motd
Sun Microsystems Inc.   SunOS 5.11      snv_90  January 2008
% digest -a md5 /etc/motd
1b46878661817aa1510a42871110c00d
```

4 Kernel-Level Cryptographic Framework (kCF)

The Kernel-Level Cryptographic Framework consists of components similar to the user-level framework but of course the implementation differs. Within this section we will present the following main parts:

- Kernel programmer interface (kernel consumer)
- Service provider interface (kernel provider)
- Interaction with the User-Level Cryptographic Framework

4.1 Kernel programmer interface

Unfortunately the kernel programmer interface does not yet have complete public documentation (unlike the user-level interfaces). Answerbook [2] even states that this interface is only available through a special contract with Sun Microsystems, Inc. and you should send your questions to the email address given in [2].

The OpenSolaris project page on the Cryptographic Framework [3] has some parts of the functional specification (but not all links to the real documentation are active as of June 2008). The page however does include the draft kCF manual pages. If you want to use the Cryptographic Framework from your own kernel module, then you will need these pages as they are not yet included in the current OpenSolaris distributions.

The interface used for consumers differs slightly from the official PKCS#11 API. This is due to special additional requirements inside the kernel (e.g. to be able to use asynchronous scheduling where the caller provides a callback function). The overall usage is very similar to the PKCS#11 though: a consumer looks up a mechanism, creates and initializes a context, uses the context and then finalizes the work including collecting the result and destroying the context.

Inside the ON source tree (see the source code browser at the OpenSolaris webpage) you will find several kernel consumers like IPsec or the WiFi drivers. More work is in progress (ZFS Crypto or more generally a crypto support for all backends via `lofi`). If you are interested in some examples then the net80211 code might be a good start.

Also check the OpenSolaris kCF specification page for further updates as the complete material (incl. much simpler examples) is expected to be available there in the near future (the links are already there but the material itself is still missing): http://www.opensolaris.org/os/project/crypto/Documentation/kef_design_spec/

4.2 Cryptographic Service Provider Interface (Kernel Provider)

Adding your own providers to the kernel-level framework is much better documented (and stable) than the kernel consumer interface.

Kernel providers may import up to eight functions from the framework and export another set of up to 67 routines. The exact number of imported and exported interfaces depends on the provider model (software or hardware) and on the capabilities offered by this provider.

The next two subsections provide an overview of the different provider models. The complete specification (including the documentation of the called functions, defines etc.) is available in [4].

4.2.1 Software Provider Model

A kernel software provider is a monolithic loadable kernel module which offers cryptographic services for a single algorithm. The module may offer various implementations and/or different modes of this algorithm and therefore actually provides one or more mechanisms. These mechanisms can be compared to the PKCS#11 mechanisms (and in fact it is recommended to use the same numbers to identify them – even though certain vendor specific number ranges are available).

The programmer interface is kept simple so that the provider does not need to care about scheduling, queueing etc. The provider is always operating in synchronous mode: the call to the provider will not return until the request is completed. The framework will take care of the rest and maintain queues, callbacks etc. if required.

Note: A software provider might be called in interrupt context (by other drivers) and you need to be careful about locking and sleeping. In particular: software providers should import and use the `crypto_kmflag()` function to determine the flag to be used by kernel memory allocations. This is important because the provider might be called in interrupt context (or similar critical paths where the caller is holding locks) and using `KM_SLEEP` might lead to deadlocks.

4.2.2 Hardware Provider Model

A kernel hardware provider is a device which offers cryptographic services. Such hardware providers are visible to the rest of the operating system (including the framework) via a device driver and a node in the device tree.

A single hardware provider can provide many mechanisms (but of course might also choose to only provide one single mechanism like a random number generator).

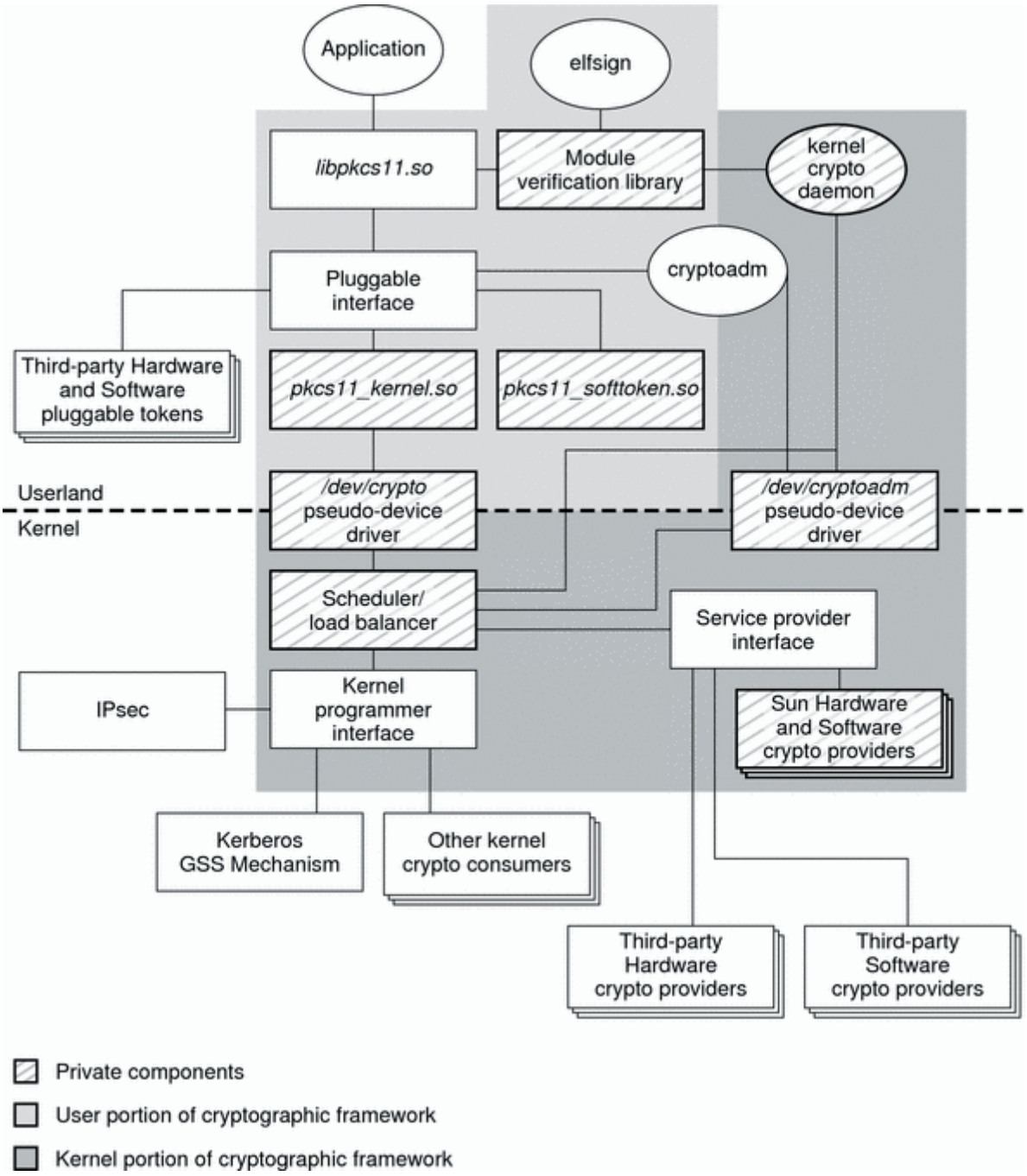
The hardware provider usually operates in asynchronous mode (which is quite different compared to the software provider) and a call might return before being completed. The interface model provides the required framework to keep the state and do the callbacks.

Depending on the implemented mechanisms, the driver might require the setup of a session before serving any cryptographic functions to the caller. Such a session management is optional though.

Hardware providers do not need to import/use the `crypto_kmflag()` function as they can always use `KM_SLEEP`.

5 Gluing it all together

We have talked about the various parts that make up the Cryptographic Framework in OpenSolaris. Now it is time to show the overall picture to get a better understanding of how the components work together:



At the top you see an application which is linked against `libpkcs11.so`. This provides the PKCS#11 API to the application. The library is uses various providers (plug-ins) which can be managed via the `cryptoadm` command. The provider might be a software provider which implements the mechanisms, but might also use kernel implementations. If using kernel implementations, it will use `/dev/crypto` via `ioctl()` to get in contact with the kernel framework which then dispatches the request to the service providers shown on the right. In addition the kernel framework is also accessible via the kernel programmer interface which is used by other kernel modules like IPsec.

Execution is controlled/verified by the kernel crypto daemon `kcfd` in the upper right which verifies the integrity (via digital signatures) of the userland libraries as well as the kernel providers. The `kcfd` and the `cryptoadm` utility also use the `/dev/cryptoadm` device (again via `ioctl()` calls) to talk to the kernel implementations.

6 Debugging

In section 4.3 we already introduced the `cryptoadm` command for inspection and administration. This command should be your starting point to check whether a certain mechanism is available and active on your system.

Section 3.1 introduced the `libcryptoutil.so` library with the `SUNW_CRYPTO_DEBUG` debugging variable. This might be helpful if the framework does not start up at all.

Section 3.5 explained `kcfd`, the crypto verification daemon. This daemon must be running to actually start and manage the entire framework. Use the SMF commands to verify that the service is online and the `kcfd` is running:

```
t1000# svcs -p cryptosvc
STATE          STIME    FMRI
online         Jun_02   svc:/system/cryptosvc:default
               Jun_02        155 kcfd
```

If the framework is not running, then all consumers will be unable to use the services and a general error will be reported in the syslog output.

```
t1000# svcadm disable -t cryptosvc
t1000# digest -a md5 /etc/motd
digest: failed to initialize PKCS #11 framework: CKR_GENERAL_ERROR
t1000# tail /var/adm/messages
Jun  4 14:36:27 v4v-t1000e-muc07 digest[1371]: [ID 137798 user.error]
digest: libpkcs11: Unable to contact kcfd: Bad file number
Jun  4 14:36:27 v4v-t1000e-muc07 digest[1371]: [ID 727670 user.error]
digest: libpkcs11: /usr/lib/security/64/pkcs11_kernel.so unexpected
failure in ELF signature verification. System may have been tampered
with. Cannot continue parsing /etc/crypto/pkcs11.conf
```

If some providers fail to work, then you might want to enable debugging in the `kcfd`. Setting the environment variable `SUNW_CRYPTO_DEBUG` to `syslog` will cause all debugging to be logged via syslog at level `user.debug`. The output is too large to be included here and only the steps to obtain it will be shown here:

```
t1000# svcadm disable -t cryptosvc
t1000# env SUNW_CRYPTO_DEBUG=syslog /usr/sbin/cryptoadm start
t1000# digest -a md5 /etc/motd
b81516e203d89d25f0ed9f7dd6eb307f
```

Now you will see a lot of debugging messages showing all the details of the framework (in `user.debug` via syslog, so you might need to tune your `/etc/syslog.conf` to capture the output).

7 Future and current projects

There are a couple of active (or future) projects related to the OpenSolaris Cryptographic Framework . The list can be found at the web page [3]. The two biggest projects are “Highlander” which is a project about the complete unification of the remaining (duplicate) implementations of crypto and the project “libkcf” to combine the userland and kernel interface into one library (which might be part of project “Highlander” or independent of this). This would help code which has parts in userland and kernel (such as Kerberos or ZFS).

There are other related projects which are not directly changes at the OpenSolaris Cryptographic Framework but will benefit from this framework. In the paper we already mentioned `pktool` as one of the consumers and you might want to have a look at the related project about Key Managerment which is described in [5].

The list of consumers will also grow which includes ZFS crypto [7] or general encryption via lofi [6].

8 References

[1]PKCS#11 Cryptoki Standard http://www.rsa.com/rsalabs/node.asp?id=2133

[2]Solaris 10 Security for Developers Guide http://docs.sun.com/app/docs/doc/816-4863

[3]OpenSolaris Project Cryptographic Framework http://www.opensolaris.org/os/project/crypto/

[4]Crypto SPI for kernel providers http://opensolaris.org/os/project/crypto/Documentation/kef_design_spec/cspi/

[5]Key Management Framework http://opensolaris.org/os/project/kmf/

[6]Lofi compression and crypto support http://opensolaris.org/os/project/loficc/

[7]ZFS on disk encryption support http://opensolaris.org/os/project/zfs-crypto/

[8]Alias crypto-discuss@opensolaris.org archived at http://www.opensolaris.org/jive/forum.jspa?forumID=179

[9]RSA Security Devision of EMC http://www.rsasecurity.com

[10]White Paper avaliable from BigAdmin http://www.sun.com/bigadmin/features/articles/crypt_framework.jsp

[11]PKCS#11 engine patch for OpenSSL http://blogs.sun.com/janp/entry/pkcs_11_engine_patch_for

9 Appendix

Here is an example program to show the use of the different APIs for a user-level consumer of the Cryptographic Framework. This example is using mmap() to get the contents of /etc/motd and then calculates the MD5 hash function for these data. Note: for real applications you should utilise the more secure SHA256 or SHA512 algorithms where possible.

9.1 Using PKCS#11 interface only

```
 1  #include <stdio.h>
 2  #include <sys/types.h>
 3  #include <sys/stat.h>
 4  #include <fcntl.h>
 5  #include <errno.h>
 6  #include <stdlib.h>
 7  #include <unistd.h>
 8  #include <sys/mman.h>
 9  #include <security/cryptoki.h>
10  #include <security/pkcs11.h>
11
12  #define MD5LEN  16
13
14  int
15  main(int argc, char **argv)
16  {
17          CK_MECHANISM            mechanism;
18          CK_SESSION_HANDLE       hSession;
19          CK_BYTE                 digest[MD5LEN];
20          CK_ULONG                digestlen=MD5LEN;
21          CK_ULONG                slotcount;
22          CK_SLOT_ID_PTR          slot_list;
23          CK_MECHANISM_INFO       mech_info;
24          CK_RV                   rv;
25          int                     fd, i;
26          struct stat             filestat;
27          void                    *input_data;
28
29          /* open /etc/motd and mmap it into memory */
30          if ( (fd=open("/etc/motd", O_RDONLY)) == -1 ) {
31                  perror("unable to open input file /etc/motd");
32                  exit(errno);
33          }
34          if ( fstat(fd, &filestat) != 0 ) {
35                  perror("unable to get stat() infos for /etc/motd");
36                  exit(errno);
37          }
38          input_data = mmap((caddr_t) 0, filestat.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
39          if ( input_data == MAP_FAILED ) {
```

```
40              perror("unable to get mmap() /etc/motd");
41              exit(errno);
42      }
43
44      /* init PKCS#11 framework */
45      if ( (rv = C_Initialize(NULL) ) != CKR_OK ) {
46              fprintf(stderr, "C_Initialize: rv = 0x%.8X\n", rv);
47              exit(1);
48      }
49      /* load the available slots */
50      if ( (rv = C_GetSlotList(0, NULL, &slotcount) ) != CKR_OK ) {
51              fprintf(stderr, "C_GetSlotList (count): rv = 0x%.8X\n", rv);
52              exit(1);
53      }
54      if ( slotcount == 0 ) {
55              fprintf(stderr, "found no PKCS#11 slots\n");
56              exit(1);
57      }
58      if ( ( slot_list = malloc(slotcount * sizeof (CK_SLOT_ID)) ) == NULL ) {
59              perror("unable to allocate memory for slots");
60              exit(errno);
61      }
62      if ( (rv = C_GetSlotList(0, slot_list, &slotcount) ) != CKR_OK) {
63              fprintf(stderr, "C_GetSlotList (load): rv = 0x%.8X\n", rv);
64              exit(1);
65      }
66      /* find the MD5 slot */
67      mechanism.mechanism = CKM_MD5;
68      mechanism.pParameter = NULL_PTR;
69      mechanism.ulParameterLen = 0;
70      for ( i = 0; i < slotcount ; i++ ) {
71              if (C_GetMechanismInfo(slot_list[i], mechanism.mechanism,
72                                      &mech_info) == CKR_OK) {
73                      /* found the MD5 slot */
74                      break;
75              }
76      }
77      if ( i == slotcount ) {
78              fprintf(stderr, "could not find the MD5 slot.\n");
79              exit(1);
80      }
81      /* open the PKCS#11 session for this MD5 slot */
82      if ( ( rv = C_OpenSession(slot_list[i], CKF_SERIAL_SESSION, NULL,
83                              NULL, &hSession) ) != CKR_OK ) {
84              fprintf(stderr, "C_OpenSession: rv = 0x%.8X\n", rv);
85              exit(1);
86      }
87
88      /* init the digest session */
89      if ( (rv = C_DigestInit(hSession, &mechanism)) != CKR_OK ) {
90              fprintf(stderr, "C_DigestInit: rv = 0x%.8X\n", rv);
91              exit(1);
92      }
93      /* feed the data into the digest */
94      if ( (rv = C_DigestUpdate(hSession, (CK_BYTE_PTR)input_data,
95                              (CK_ULONG)filestat.st_size)) != CKR_OK ) {
96              fprintf(stderr, "C_DigestUpdate: rv = 0x%.8X\n", rv);
97              exit(1);
98      }
99      /* get the digest as the result */
100     if ( (rv = C_DigestFinal(hSession, (CK_BYTE_PTR)digest,
101                             &digestlen)) != CKR_OK ) {
102             fprintf(stderr, "C_DigestFinal: rv = 0x%.8X\n", rv);
103             exit(1);
104     }
105
106     /* print the calculated MD5 digest */
107     printf("MD5(/etc/motd) = ");
108     for ( i = 0; i < digestlen; i++ )
109             printf("%.2x", digest[i]);
110     printf("\n");
111
112     /* cleanup and exit */
113     (void) munmap(input_data, filestat.st_size);
114     (void) close(fd);
115     (void) C_CloseSession(hSession);
116     (void) C_Finalize(NULL_PTR);
```

```
117
118         return(0);
119 }
120
```

9.2 Using PKCS#11 interface with Sun functions

```
 1 #include <stdio.h>
 2 #include <sys/types.h>
 3 #include <sys/stat.h>
 4 #include <fcntl.h>
 5 #include <errno.h>
 6 #include <stdlib.h>
 7 #include <unistd.h>
 8 #include <sys/mman.h>
 9 #include <security/cryptoki.h>
10 #include <security/pkcs11.h>
11
12 #define MD5LEN  16
13
14 int
15 main(int argc, char **argv)
16 {
17         CK_MECHANISM          mechanism;
18         CK_SESSION_HANDLE     hSession;
19         CK_BYTE               digest[MD5LEN];
20         CK_ULONG              digestlen=MD5LEN;
21         CK_RV                 rv;
22         int                   fd, i;
23         struct stat           filestat;
24         void                  *input_data;
25
26         /* open /etc/motd and mmap it into memory */
27         if ( (fd=open("/etc/motd", O_RDONLY)) == -1 ) {
28                 perror("unable to open input file /etc/motd");
29                 exit(errno);
30         }
31         if ( fstat(fd, &filestat) != 0 ) {
32                 perror("unable to get stat() infos for /etc/motd");
33                 exit(errno);
34         }
35         input_data = mmap((caddr_t) 0, filestat.st_size, PROT_READ,
36                                 MAP_PRIVATE, fd, 0);
37         if ( input_data == MAP_FAILED ) {
38                 perror("unable to get mmap() /etc/motd");
39                 exit(errno);
40         }
41
42         /* init the framework and select PKCS#11 mechanism type for MD5 */
43         mechanism.mechanism = CKM_MD5;
44         mechanism.pParameter = NULL_PTR;
45         mechanism.ulParameterLen = 0;
46         if ( (rv = SUNW_C_GetMechSession(mechanism.mechanism,
47                                 &hSession)) != CKR_OK ) {
48                 fprintf(stderr, "SUNW_C_GetMechSession: rv = 0x%.8X\n", rv);
49                 exit(1);
50         }
51
52         /* init the digest session */
53         if ( (rv = C_DigestInit(hSession, &mechanism)) != CKR_OK ) {
54                 fprintf(stderr, "C_DigestInit: rv = 0x%.8X\n", rv);
55                 exit(1);
56         }
57         /* feed the data into the digest */
58         if ( (rv = C_DigestUpdate(hSession, (CK_BYTE_PTR)input_data,
59                                 (CK_ULONG)filestat.st_size)) != CKR_OK ) {
60                 fprintf(stderr, "C_DigestUpdate: rv = 0x%.8X\n", rv);
61                 exit(1);
62         }
63         /* get the digest as the result */
64         if ( (rv = C_DigestFinal(hSession, (CK_BYTE_PTR)digest,
65                                 &digestlen)) != CKR_OK ) {
66                 fprintf(stderr, "C_DigestFinal: rv = 0x%.8X\n", rv);
67                 exit(1);
68         }
```

```
69
70          /* print the calculated MD5 digest */
71          printf("MD5(/etc/motd) = ");
72          for ( i = 0; i < digestlen; i++ )
73                  printf("%.2x", digest[i]);
74          printf("\n");
75
76          /* cleanup and exit */
77          (void) munmap(input_data, filestat.st_size);
78          (void) close(fd);
79          (void) C_CloseSession(hSession);
80          (void) C_Finalize(NULL_PTR);
81
82          return(0);
83  }
84
```

9.3 Using message digest library (libmd)

```
 1  #include <stdio.h>
 2  #include <sys/types.h>
 3  #include <sys/stat.h>
 4  #include <fcntl.h>
 5  #include <errno.h>
 6  #include <stdlib.h>
 7  #include <unistd.h>
 8  #include <sys/mman.h>
 9  #include <md5.h>
10
11  #define MD5LEN  16
12
13  int
14  main(int argc, char **argv)
15  {
16          unsigned char           digest[MD5LEN];
17          int                     fd, i;
18          struct stat             filestat;
19          void                    *input_data;
20
21          /* open /etc/motd and mmap it into memory */
22          if ( (fd=open("/etc/motd", O_RDONLY)) == -1 ) {
23                  perror("unable to open input file /etc/motd");
24                  exit(errno);
25          }
26          if ( fstat(fd, &filestat) != 0 ) {
27                  perror("unable to get stat() infos for /etc/motd");
28                  exit(errno);
29          }
30          input_data = mmap((caddr_t) 0, filestat.st_size, PROT_READ,
31                                  MAP_PRIVATE, fd, 0);
32          if ( input_data == MAP_FAILED ) {
33                  perror("unable to get mmap() /etc/motd");
34                  exit(errno);
35          }
36
37          /*
38           * calculate the MD5 digest
39           * use MD5Init(), MD5Update() and MD5Final() if the data
40           * is not available in one chunk
41           */
42
43          md5_calc(&digest, input_data, filestat.st_size);
44
45          /* print the calculated MD5 digest */
46          printf("MD5(/etc/motd) = ");
47          for ( i = 0; i < MD5LEN; i++ )
48                  printf("%.2x", digest[i]);
49          printf("\n");
50
51          /* cleanup and exit */
52          (void) munmap(input_data, filestat.st_size);
53          (void) close(fd);
54
55          return(0);
56  }
57
```

New Security Features in OpenSolaris and Beyond

Jan Pechanec
Sun Microsystems, Inc.
Jan.Pechanec@Sun.COM

Christoph Schuba
Sun Microsystems, Inc.
Christoph.Schuba@Sun.COM

Mark Phalan
Sun Microsystems, Inc.
Mark.Phalan@Sun.COM

1

Abstract

This paper examines several new security features and enhancements to existing security features that were introduced into the OpenSolaris Operating Environment in the time period of approximately mid 2006 through mid 2008. We focus on the following contributions, rather than present an exhaustive list: Solaris Trusted Extensions (the multi-level security features that is now an integral part of the Solaris architecture), the Key Management Framework (KMF - a unified set of interfaces for managing PKI objects), the OpenSSL PKCS#11 engine, and a number of functional enhancements to our Kerberos system.

Furthermore, we present work in progress on filesystem encryption (most notably ZFS encryption and the loopback file system encryption), PKCS#11 engine, SunSSH, and Kerberos, new security features that, as of mid 2008, are being actively developed and are scheduled to become part of future OpenSolaris versions and distributions

2 Overview of the OpenSolaris project

The OpenSolaris project[1] is an open source project sponsored by Sun Microsystems, Inc, that was initially based on a subset of the source code for the Solaris Operating System. It is a nexus for a community development effort where developers from Sun and elsewhere can collaborate on developing and improving operating system technology. The OpenSolaris source code will find a variety of uses, including being the basis for future versions of the Solaris OS product, other operating system projects, and third-party products and distributions.

Since its opening day launch in June 14, 2005, in which the bulk of the Solaris system code was released, a growing community of OpenSolaris users and contributors has emerged. There are already several distributions based on OpenSolaris code, including but not limited to SchilliX, Belenix, Nexenta, and Sun's official distribution called Opensolaris.

3 Overview

All the security features and enhancements mentioned in this paper were or are being developed internally at Sun. They will be or have been first introduced as a part of the Nevada distribution (also known as Solaris Express Community Edition or SXCE[2]). The code is available externally very shortly after it has been committed to the Nevada code base. Also, it is released as part of Sun's official OpenSolaris distribution, also known as project Indiana[3].

In this paper, we will examine two filesystem encryption options which are under development – lofi and ZFS crypto. Next the Trusted Extensions project will be introduced, which is a reimplementation of Trusted Solaris 8 based on new security features in Solaris 10. The Key Management Framework (KMF) is a unified set of interfaces (both programming API and administrative tools) for managing Public Key Infrastructure (PKI) objects in Solaris, and SunSSH is an example consumer that uses this API for implementation of X.509 authentication in the SSH2 protocol. The OpenSSL PKCS#11 engine is a way to access the Solaris Cryptographic Framework from OpenSSL. There is a project in progress to add new mechanisms to the engine, and the SunSSH project uses those enhancements to off-load cryptographic operations to cryptographic hardware providers, if available, and thus significantly speed up any bulk transfer of data. Validated execution is a way to verify the integrity of program and library objects at the time of execution. Finally, we examine

some of the new features recently added to Kerberos or in development as of June 2008. **For features under development as of June 2008, the details are subject to change as work progresses. This document is not a feature commitment from Sun Microsystems for a future releases of Solaris.**

4 Solaris Trusted Extensions

4.1 Overview

The Solaris Trusted Extensions project[6] has moved security technologies that constituted the difference between Solaris 8 and Trusted Solaris 8 into the base of Solaris 10 and OpenSolaris. In the process, most of these technologies have been expanded significantly in functionality, usability, and robustness. The Trusted Solaris products has been renamed because it is delivered as software that is part of the regular Solaris distributions that can be optionally enabled and configured.

4.2 How it works

Solaris Trusted Extensions is an optionally-enabled layer of secure labeling technology that allows data security policies to be separated from data ownership. Integration to OpenSolaris (and Solaris 10) allows the system to support both traditional Discretionary Access Control (DAC) policies based on ownership, as well as label-based Mandatory Access Control (MAC) policies.

Different objects of the system, including files, packets, devices, window management services or printers are assigned labels or range of labels. Those labels establish explicit relationships between objects. Authorization allows applications and users to read or write to the objects. When two labels are compared, the first label can be greater than, less than, equal to, or disjoint from the second label. Labels consist of hierarchical components called classifications (or levels) and a non-hierarchical components called compartments (or categories). Classifications are compared as integers, and compartments are compared as bit masks. Labels are disjoint when each contains at least one compartment bit which is not present in the other.

Users interact with labels as strings. Graphical user interfaces and command line interfaces present these strings. Human readable labels are classified at the label that they represent. Thus the string for a label A is only readable (viewable, translatable to or from human readable to an opaque label data type) by a subject whose label allows read (view) access to that label.

Applications do not need to be modified nor profiled to bring them into conformance with the MAC policy. Instead, the entire application environment is virtualized for each label through the use of Solaris Containers (zones), Solaris primary OS-level virtualization technology. The Solaris Containers facility provides an isolated environment for running applications. Processes running in a zone are prevented from monitoring or interfering with other activity in the system. Access to other processes, network interfaces, file systems, devices, and inter-process communication facilities are restricted to prevent interaction between processes in different zones. At the same time, each zone has access to its own network stack and name space, enabling per-zone network security enforcement, such as firewalling or IPsec.

All the zones are centrally administered from a special, protected global zone which manages the Trusted Computing Base (TCB) known as the Trusted Path. The zones share a single Lightweight Directory Access Protocol (LDAP) directory in which network-wide policy is defined, as well as a single name service cache daemon for synchronizing local databases. All labeling policy and account management is done from within the Trusted Path. MAC policy enforcement is automatic in labeled zones and applies to all their processes, even those running as root. Access to the Global Zone (and hence Trusted Path applications) is restricted to administrative roles.

Each zone is assigned a unique sensitivity label and can be customized with its own set of file systems and network resources. Each mounted file system is automatically labeled by the kernel when it is mounted. The file system label is derived from the label of the zone or host which is sharing it. All files and directories within the mounted file system have the same label as their mount point. No extensions to the file system structure is required. Processes are uniquely labeled according to the zone in which they are executing. All processes within a zone (and their descendants) must have the same label, and are completely isolated from processes in other zones.

4.3 Multilevel Desktop Sessions

Users can log in via the Trusted Path and can be authorized to select their multilevel desktop preference (Common Desktop Environment or Java Desktop System). Once authenticated they are presented with an option to select an explicit label or a range of labels within their clearance and the label range of their workstation or desktop unit. The window system initiates a user session in the zone whose label corresponds to the user's default or minimum label.

Attempts to cut and paste data, or drag and drop files between clients running in different zones are mediated by the Trusted Path. Specific authorizations are required for upgrading or downgrading selections and files, and are prohibited by default.

5 Filesystem Encryption

5.1 lofi driver encryption

5.1.1 Overview

The lofi file driver[4] exports a file as a block device. Reads and writes to the block device are translated to reads and writes on the underlying file. This is useful when the file contains a file system image. Exporting it as a block device through the lofi file driver allows normal system utilities to operate on the image through the block device (like fstyp1M), fsck(1M), and mount(1M). This functionality was originally created for mounting ISO CD images.

5.1.2 How it works

The lofi driver will gain the ability to encrypt/decrypt the raw blocks. The administration command `lofiadm` is extended to support requesting encryption and setting the key.

It is not desirable for lofi to have the encrypted data take up more space than the clear text data would. For this reason it is not possible to use a separate digest and HMAC algorithm as is commonly done in network protocols, for example AES for encryption and HMAC-SHA1 for integrity protection of the cipher text. Encrypted block device support on other platforms does not provide any integrity protection support. To ensure we have integrity protection in lofi a future case will introduce a cipher suite that provides a non-expanding ciphertext output that has integrity protection built in.

No algorithm or key data is written to disk, however some metadata space is reserved at the start of the file. That space is currently used to store a version number only. However, in future versions this space will most likely be used to store other types of data. If an incorrect key value is given then the lofiadm mapping will still succeed but any filesystem layered on it will fail to mount (since the data on the device will not appear to be a filesystem).

5.1.3 Example

```
# mkfile 35m /export/home/test
# lofiadm -a -c aes-256-cbc /home/secrets
Enter key: xxx
Re-enter key: xxx
/dev/lofi/1
# newfs /dev/rlofi/1
...
```

```
# lofiadm
Block Device     File                Options
/dev/lofi/1      /home/secrets       Encrypted
```

5.2 ZFS encryption

5.2.1 Overview

The first phase of the ZFS encryption project[5] covers the addition of encryption and decryption to the ZFS IO pipeline and key management for ZFS datasets and ZFS pools.

As of June 2008 the phase one code development is complete and under code review. While code integrations into OpenSolaris is planned for later in the year, we cannot give any committed dates at this time. This is the second filesystem encryption project for OpenSolaris under development as of June 2008, see Subsection 5.1 on lofi driver encryption above. However, the lofi project has important limitations -- no data integrity protection, for example. The ZFS encryption project aims at seamless integration of dataset encryption/decryption and key management in the frame of ZFS technology[13].

5.2.2 Goals of the project

The project has several phases, with the first one, as of June 2008, is development complete. The project will provide per file system encryption with randomly generated keys. Those keys are stored on disk with the dataset in wrapped form. It is wrapped by a master per pool key or per dataset key that the user or the administrator provides. While other filesystem encryption technologies often offer per file or per directory encryption, ZFS datasets (datasets are either file systems or ZVOLs) are relatively very cheap compared with traditional filesystems so using the dataset in ZFS provides equivalent functionality. All data and file system metadata, such as file owner Access Control List (ACL) size etc, must be encrypted when on disk. For ZVOLs all data is encrypted on disk; encryption support for ZVOLs allows, for example, encrypted swap areas and databases which directly use raw devices. For encryption, CCM mode of operation was chosen because it provides data integrity, too. The algorithm mode and key length are fixed at dataset creation time. There is no direct use of asymmetric cryptography in the file system. A software only solution will be provided but whenever appropriately supported hardware is present, this hardware can be used, for example, to store the master key or to accelerate cryptographic operations. The SCA-6000 is an example of a device where the master key can be secured in a hardware token. The features we have just mentioned are all part of the first phase of the project.

Future project phases are likely to include features such as support for an encrypted root filesystem, remote key manager (here, asymmetric cryptography might be used in remote key manager protocols), and secure deletion.

5.2.3 Example usage

The following example shows how to set the master per pool key via a passphrase for a ZFS pool. An encryption key is generated from the user supplied passphrase. Raw key data can also be supplied via a file, smart card, or from standard input.

```
# zpool set -o keysource=passphrase,prompt encryption=aes-256-ccm tank
Enter passphrase for 'tank': ****
Enter again: ****
```

After creation before any encrypted dataset can be mounted, the pool key must be loaded:

```
# zpool key -l tank
```

```
Enter passphrase for 'tank': ****
Enter again: ****
```

The key status of the pool can be displayed as follows:

```
# zpool get keystatus tank
NAME   PROPERTY   VALUE        SOURCE
tank   keystatus  available    local
```

6 Key Management Framework (KMF)

6.1 Overview

The goal of the project[7] was to provide a unified set of interfaces (both programming APIs and administrative tools) for managing PKI objects in Solaris. Before KMF, there were several different "keystore systems" that developers and administrators could chose from when designing systems that employed PKI technologies – Network Security Services (NSS), OpenSSL, and, PKCS#11 were the three main choices for Solaris users. Each of these systems presents very different programming APIs and administrative tools and none of them has any sort of concept of a PKI policy enforcement system.

Not having a unified infrastructure for PKI-enabled applications means that every application must choose between the existing systems, which makes interoperability more difficult and limits the use of the application. KMF provides utilities and an API for managing public key objects in a format-neutral manner and bridges the gap between the existing PKI technologies currently used in Solaris. It also introduces new features not available in existing technologies, for example policy enforcement.

6.2 How it works

KMF provides a consistent management interface through the `pktool` utility. `pktool` was originally developed to manage PKCS#11 keystores, the KMF project enhanced it so that it could serve as a generic PKI tool. This allows an administrator to use `pktool` to administer all three keystore systems, whereas previously, he would have had to use `openssl` command to work with OpenSSL key files, `certutil` to work with NSS databases, and `pktool` to work with PKCS#11 keystores.

From the developer's perspective, the KMF API hides the details of the underlying keystores, and provides information necessary to identify those keystores and stored objects. For example, for OpenSSL files, a filename must be provided, but the developer can use a single KMF API call to access the keystore in a format-neutral manner. The KMF API is also pluggable so that third party vendors can introduce, for example, proprietary or legacy implementations while benefiting from KMF independence. For certificate validation[14], Online Certificate Status Protocol (OCSP), and Certificate Revocation List (CRL) checking is provided, with Public Key Infrastructure (PKIX) path validation planned for future version of OpenSolaris.

KMF API operations include all the commonly needed routines to work with keys, certificates, and Certificate Signing Requests (CSR), including but not limited to creating, deleting, searching, importing, and exporting such objects. All common cryptographic operations are provided, including signing, verifying, encrypting, and decrypting using keys and certificates. The KMF API also provides access to complex objects like X.509 attributes and extensions in human readable formats.

Security policy is defined in a system wide configuration file in XML format, and a new `kmfcfg` tool was introduced to administer that file. Any application can use a different policy file if it chooses to. KMF policy is a set of parameters that controls the use of X.509 certificates by an application. The application must use the KMF API to make use of the policy system.

Any applications that work with certificates are potential consumers of KMF, and we will examine SunSSH as one of them in the next paragraph. One enhancement of KMF in the design phase as of June 2008, is certificate to name mapping. This pluggable mapping system implements a policy which controls which username or hostname the particular certificate should be mapped to.

6.3 X.509 support for SunSSH

6.3.1 Overview

Using X.509 certificates[14] for SSH authentication is already included in the SSH2 protocol as part of `pubkey` authentication but not fully defined. Having the option to use X.509 certificates means here that there is no need to verify public keys using a secondary information channel or blindly accepting those keys during the first session within which a man-in-the-middle attack can be mounted. There are several ways of implementing it – for example, using NSS, OpenSSL or the PKCS#11 API. The last one would also allow us to use hardware keystores, and sign data without exposing private keys from the tokens. Another approach, as already suggested, would be to use the public KMF API[1].

There are two different IETF drafts on X.509 authentication in the SSH2 protocol. One documents a desirable way of using such an authentication method[15] while the second, which is an informational draft only, documents the way it is implemented in existing SSH implementations. The informational one is the draft that we must follow for compatibility with SSH implementations that already supports X.509 authentication. To our best knowleadge, no SSH implementation in production supports [15].

KMF is used for X.509 authentication only. The existing manner of working with public and private keys in files is unchanged. The existing HostKey and IdentityFile options keywords are overloaded so that a PKCS#11 URI[2] can be used. The URI specifies a private key in the token. A Solaris soft token keystore can be used as well, making easy to try it out. An example of this URI when used with the `IdentityFile` option follows:

```
-o IdentityFile="pkcs11:token=Sun Software PKCS#11 softtoken;object=user"
```

6.3.2 The use of KMF policy in SunSSH

One goal when adding KMF support to SunSSH[17] was to leverage the existing policy file configuration method rather than adding extra options to SunSSH. Note that a user can always use a specific policy file which will define the policy on the user's side of the connection. Policy files should never be edited manually, `kmfcfg` tool should be always used. The main thing we need the policy file for is to define a Trusted Anchor (TA), i.e. the certificate of a certification authority. There is no separate option to define a different TA that SunSSH should use. So, the only two new options SunSSH needs is the policy file name and a policy name within that file.

7 OpenSSL PKCS#11 engine

7.1 Overview

OpenSSL provides an ENGINE API[10], which allows the creation, manipulation, and use of cryptographic modules in the form of ENGINE objects. These objects act as containers for implementations of cryptographic algorithms. With the standard OpenSSL distribution, users can write their own engine implementations and use the dynamic loading capability of OpenSSL to load and use those engines on the fly. Depending on engine support, various operations including but not limited to RSA, DSA, DH, symmetric ciphers, digests, and random number generation can be offloaded to the engine. For operations that are not available in the engine, native OpenSSL code is used[3].

OpenSSL in OpenSolaris is shipped with an internally (in Sun) developed PKCS#11 engine which enables cryptographic operations to be offloaded to the Solaris Cryptographic Framework (CF)[18]. Due to import restrictions in some coutries, OpenSSL in Solaris doesn't support dynamic engine loading, and no other engines are shipped as any hardware cryptographic provider should be plugged into the CF and used through the PKCS#11 engine. The main reason the PKCS#11 engine was developed was to offload RSA/

DSA operations to the ncp cryptographic provider on the Niagara-1 platform when using the Apache web server. However, with introduction of the n2cp driver for Niagara-2 which is capable of hardware accelerated symmetric cryptographic and digest mechanisms, there are many more applications now that could make use of the PKCS#11 engine.

7.2 The current version of the engine

As of June 2008, the version of the engine consumes the PKCS#11 API from the `libpkcs11` library. This library provides a special slot[4] called the metaslot. The metaslot provides a virtual union of capabilities of all other slots. When available, the metaslot is always the first slot provided by `libpkcs11`. The problem with this approach is that all mechanisms for which hardware acceleration is not present on the machine used are offloaded to the softtoken, which is a software implementation of a PKCS#11 token. The inherent overhead of the CF then causes operations performed by softtoken to be performed more slowly than the native OpenSSL code. One possible solution may be to use the `pkcs11_kernel` library directly[5]. This library provides the same PKCS#11 API but offers only those slots that have hardware capabilities. Such a solution is needed if we want applications to use the engine by default where no regression is acceptable on platforms without hardware acceleration for cryptographic operations. Using the `pkcs11_kernel` library directly would mean no speed regression on machines without supported cryptographic hardware accelerators as the native OpenSSL cryptographic operations will be used.

Another limitation of the current engine implementation is that it doesn't offer many mechanisms:

```
$ openssl engine -vvv -t -c
(pkcs11) PKCS #11 engine support
 [RSA, DSA, DH, RAND, DES-CBC, DES-EDE3-CBC, AES-128-CBC, RC4, MD5, SHA1]
```

7.3 New code in progress in the engine

Given the current cryptographic accelerator hardware that is available for and integrated into Sun machines, the decision was made to operate with two provider slots. The first slot is used for the most appropriate RSA/DSA/DH/RAND instance, and the second slot is used for symmetric cryptographic algorithms and digests. The issue that needed to be solved to implement this change in the engine code was that the ncp hardware provider presents key usage limitations for RSA (2048 bits), DSA (1024) and DH (2048). For all operations that use longer keys we must call back into the native OpenSSL code. Note that with the `libpkcs11` library and the metaslot no code changes were required since operations using such keys were automatically offloaded to the `pkcs11_softtoken` library. When the project integrates to OpenSolaris, applications using OpenSSL will be able to load the PKCS#11 engine and transparently experience significant performance improvements when an appropriate hardware accelerators is present in the system, while avoiding the overhead of the Cryptographic Framework for mechanims for which no hardware accelerator is present.

The current engine prototype on Niagara2 now shows these mechanisms:

```
$ openssl engine -vvv -t -c
 (pkcs11) PKCS #11 engine support
  [RSA, DSA, DH, DES-CBC, DES-EDE3-CBC,DES-ECB, DES-EDE3, RC4, AES-128-CBC,
  AES-192-CBC, AES-256-CBC, AES-128-ECB, AES-192-ECB, AES-256-ECB,
  AES-128-CTR, AES-192-CTR, AES-256-CTR, MD5, SHA1, SHA256]
```

7.4 SunSSH with OpenSSL PKCS#11 engine support

7.4.1 Defining the problem

SunSSH is a good example of an application that could make use of hardware acceleration for symmetric

cryptographic algorithms. Recently, we have been receiving many requests for this enhancement which reflects the number of Niagara T2 machines sold. Sun servers that are based on the Niagara T2 CPU chipsets and configured with many virtual CPU's present an ideal computing platform for multithreaded applications or workloads where many requests need to be served in parallel. However, applications that are CPU intensive and run in a single thread exhibits apparent slow-down in comparison to common 1-2 CPU machines. The problem is that all the CPU intensive part is now done on one virtual CPU which is slower in comparison to CPU's shipped with other workstations with one or a few processors.

7.4.2 Hardware cryptographic support

To remedy the situation, the n2cp hardware provider together with SunSSH using the PKCS#11 engine presents an attractive combination with one condition for successful use of the engine. The data blocks to process must be of a reasonable size. There is a significant overhead present when data is offloaded to the hardware, so offloading data blocks of, say 128 bytes, would actually slow down the transfer significantly. However, with bulk data transfer[6], that's not the case and with current version of SunSSH blocks of around 8KB are processed. Very small SSH packets during interactive sessions are not of concern here, because human beings are not capable of typing fast enough to notice the slow down caused by offloading small data blocks to the hardware.

The changes needed for SunSSH to use the PKCS#11 engine are not small. This is in part due to how privilege separation is done in SunSSH. SunSSH forks after user authentication, with the parent becoming a privileged monitor responsible for allocating pty's, auditing logins and logouts, working with utmpx/wtmpx databases, and using the host's private keys and Generic Security Services API (GSS-API) credentials. The child drops privileges and continues to process the packets, including encryption and decryption. The problem with using the engine is that the child process needs to use the cryptographic contexts after the parent process has forked.

7.4.3 Using the PKCS#11 engine from SunSSH

The PKCS#11 standard[12] forbids the use of existing PKCS#11 sessions after a fork which means that all contexts must be cleaned up, the engine closed and reinitialized. However, encrypted data sent in one direction is considered a single data stream, with initialization vectors passed from the end of one packet to the beginning of the next packet. Reinitializing the engine without other actions means that we loose the context and can't properly process the packets any more.

There are several solutions to the situation:

- the monitor process could be pre-forked before setting up the first cryptographic contexts
- a new key re-exchange process could be initialized, finish the engine and clear all contexts, fork, initialize the engine again and set up all contexts with the new keys after the fork,
- the child could assume a role of the monitor while the parent would continue using the current engine session.

From those three options, the third option seems to be the simplest to implement. The problem with it is that we would be changing the way an unprivileged child and monitor currently work together. While this is private to SunSSH, it is assumed that there are users that might depend on the current behavior. Debugging is an example of this. Clearly, this isn't the best way to go.

For the server, the first option was chosen, to avoid the second rekeying shortly after the first one for every SSH connection. Also, let's not forget that the SSH1 protocol doesn't support rekeying at all. Given the fact that the monitor then doesn't know all the needed information, for example that the user authenticated. A new privilege separation message was introduced and the privilege separation code was rearranged.

For the client, usage of `-f` option and `~&` escape sequence must have been resolved since forking of the ssh client was involved. To solve that, the second key re-exchange right before daemonizing was used. Since the SSH protocol 1 is used mostly by old implementations and probably older appliances, we think that there is no problem not to offer hardware acceleration on client side for SSH protocol 1.

7.4.4 Results using the SunSSH prototype

Every SSH packet is processed twice. Slightly simplified, a HMAC is computed over the plain data, the data is encrypted, and the HMAC is appended. We got significant gains from offloading symmetric cryptographic, and not much for offloading digest operations. With the prototype we have (subject to change) with the default AES-128-CTR encryption mode, we see roughly a 2.5x speedup on the Niagara T2 platform when the engine is used. We also provide a new `UseOpenSSLEngine` option that might be used to switch off the default loading of the engine, in case of a faulty driver of the 3rd party accelerator, for example.

Note that we do not get much speed up from offloading RSA/DSA to the engine. This is a direct consequence of how SSH implementations are commonly used. In contrast to web servers, the speed up we ask for is for bulk data transfer, i.e. for sessions that last some time. Usually, machines do not accept hundreds of SSH connections in parallel, and RSA/DSA operations are used only during the initialization and key re-exchange (which is every 1-4GB of data by default, depending on the algorithm used).

8 Validated Execution

The Validated Execution project[9] provides a way to verify the integrity of program and library objects at the time of execution. Such verification provides assurance that the executable has not been altered, either accidentally or deliberately, since it was released by its publisher (which may be Sun, a third-party ISV, or the end customer).

There have been other attempts to solve this problem, such as Tripwire[22] or Basic Audit Reporting Tool (BART)[23], by comparing files against a manifest of known valid files. However, there is no protection against programs that are modified after the comparison but before they are executed. Performing the verification at the time of use avoids the race condition inherent in solutions that periodically compare files against a known reference.

Each executable file object is protected by a digital signature, which is constructed from a one-way hash of the file contents encrypted with a private RSA key known only to the publisher of the software. The corresponding public key can be used to decrypt the hash value and verify that it matches the file contents. Most ELF files in Solaris already contain an embedded digital signature; for those that do not and for non-ELF objects such as shell scripts, we provide a separate manifest listing the signatures for each file. This mechanism extends easily to allow third-party software vendors and end customers to create their own manifests of file signatures.

The general strategy employed is for entities that cause code to be loaded and executed, such as the kernel and run-time loader, to verify such code before execution. This approach establishes a chain of trust where each code component is validated by a previously validated component. To validate the earliest executing code in the system and initialize this process, we rely on a hardware component called a Trusted Platform Module (TPM). For systems without a TPM, all of the functionality described herein is still available, but the user must implicitly trust that the earliest code has not been modified.

This feature provides some degree of protection even against an attacker who is able to run processes with all privileges. We cannot protect in general against such a process, because it is able to alter any part of the file system or memory. However, the privileged process cannot alter the information stored in the TPM without knowledge of the TPM's owner password. Even if the running system is compromised, any alteration to the file system will be detected the next time the system is booted. Any executable files modified by the attacker will fail to start when the system reboots. Therefore, if an attacker gains privilege by exploiting a vulnerability, he must do so every time the system is booted rather than leaving behind a back door that can be used in the future to gain privileged access to the system.

9 Kerberos

Kerberos[19] is a computer network authentication protocol, and the Solaris implementation[21] is based on the MIT Kerberos implementation. We tightly cooperate with MIT on the development.

9.1 Pluggable Kerberos DB backends and LDAP

The Kerberos database of principal and policy information is traditionally stored in a db2 file-based database. LDAP support was recently added allowing for the principal and policy records to be stored in a directory server instead of the traditional db2 store. A new plugin interface was created to allow future backends to be implemented with little or no impact on the existing code. Both the new LDAP support and the older db2 support are now implemented as plugins. The db2 plugin is still the default. There are a number of advantages to using the LDAP plugin including simplified administration. The LDAP plugin opens the possibility of having multi-master Key Distribution Centers (KDC).

9.2 Zero-Conf Kerberos clients

In older versions of Kerberos, clients had to be configured in order to operate in a Kerberos realm. At a minimum a default realm, host-to-realm mapping, and realm-to-kdc mapping had to be provided. The default configuration file was shipped in an essentially misconfigured state so it could not be used without modification. A number of small changes were made to remedy this situation:

- by default Kerberos clients will look up DNS to locate a KDC for a given realm. This is equivalent to having the "dns_lookup_kdc" option set to "true".
- the Kerberos realm for a given host (or the local hostname for the default realm) is determined by looking at the DNS domain name. If a KDC can't be found for a realm corresponding to the DNS domain name (in caps) then a component of the domain name is dropped and the search begins again. This procedure is repeated until there are only two components left in the potential realm name or a KDC is found.
- krb5.conf has been modified so that it isn't mis-configured out of the box. If certain conditions are true then krb5.conf may need no modifications for a working client.
- Limited client-side referral support. By default now all Ticket Granting Service (TGS) requests ask for tickets with the referral bit turned on. This allows for the KDC to inform the client what realm the host is in. Microsoft's Active Directory product supports referrals.

The result of these changes is that in some suitably configured environments (KDC information in DNS, realm names matching DNS names, DNS configured on the client) no further configuration is necessary.

9.3 Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)

Traditionally a KDC will supply a Ticket Granting Ticket (TGT) to anybody who asks for one. The TGT is encrypted with the principal's long term key. As only the KDC and the user know the user's secret key only that user can use the TGT. The key used to encrypt the TGT is a symmetric key. In order to reduce the possibility of an off-line brute-force attack on the user's long term key a pre-authentication scheme is used. The default pre-authentication scheme uses a timestamp. For PKINIT[20] a new pre-authentication scheme is introduced whereby the client sends the KDC pre-authentication information containing the public key of the client. The KDC then encrypts the reply with either a symmetric key (signed by KDC, encrypted with public key of client) or performs a Diffie-Hellman exchange to establish a shared key which is then used to encrypt the reply. The MIT Kerberos implementation relies on OpenSSL for certificate processing. The plan for OpenSolaris is that KMF will be used. The PKINIT functionality is implemented via a plugin. New infrastructure was added so that pre-authentication methods may be implemented as plugins. A pam module using PKINIT to authenticate to a KDC would allow for smart-card based authentication. There are plans to implement this but as a follow-on project and not part of the initial integration.

9.4 Kerberos master-key encryption type migration

The Kerberos principal and policy information is encrypted with a key generated at the same time the Kerberos database is initialized. This key cannot be changed once set. By default a single DES key is used. In order to decrypt the principal or policy information a password is provided to the KDC on startup -

either interactively or via a "stash" file. The project aims to remove this limitation and allow for the key to be changed and stronger encryption types be used (e.g., AES) without re-creating the Kerberos database. Instead of storing the secret information in a stash file a keytab will be used. This approach provides for key versioning among other things. An interesting side-note to this project: This work is being done by Sun but instead of integrating first into Solaris and contributing those changes back to MIT, we're working directly with MIT and the first commit of the code changes or contributions will be to the MIT codebase. Recently, MIT set up a new Kerberos Consortium which allows interested parties to have more influence on the direction that Kerberos is taking. Sun is a part of that consortium.

9.5 Improvements to kclient

kclient is a utility used to simplify Kerberos client configuration. It aims to make client configuration simpler and less error prone. It can be run in interactive or non-interactive modes. Recently kclient was enhanced to support more client configuration scenarios. Significant additions include Microsoft AD support - kclient can now perform a Active Directory join. It also now supports non-Solaris KDCs such as MIT, Heimdal, and Shishi. Unfortunately there is no standardized Kerberos administration protocol - all Kerberos implementations use their own which precludes interoperability. kclient is a part of the solution. Other new features of kclient include better support for dynamic clients and clusters.

10 Conclusions

This paper presented security technologies that are or are going to be part of the Solaris and OpenSolaris operating systems. Some of these technologies are enhancements to existing security features, some are new security features, and some are part of ongoing projects. The paper focuses on these technologies in the timeframe of mid 2006 through mid 2008. With technologies such as multi-level security included in the operating system and future use of validated execution, the OpenSolaris operating system continues to be a leading contributor of technology innovation.

11 References

[1] OpenSolaris project, http://www.opensolaris.org

[2] Solaris Express Community Edition, http://www.opensolaris.org/os/downloads/

[3] Project Indiana, http://opensolaris.org/os/project/indiana/

[4] lofi(7D) driver, http://www.opensolaris.org/os/project/loficc

[5] ZFS Crypto, http://www.opensolaris.org/os/project/zfs-crypto

[6] Trusted Extensions, http://www.opensolaris.org/os/community/security/projects/tx/

[7] Key Management Framework, http://www.opensolaris.org/os/project/kmf/

[8] http://www.opensolaris.org/os/community/security/projects/SSH/ssh-x509v3-design.html

[9] Validated Execution project, http://www.opensolaris.org/os/project/valex

[10] engine(3), manual page for ENGINE cryptographic module support

[11] n2cp(7D), manual page for Ultra-SPARC T2 cryptographic provider device driver

[12] PKCS#11 standard, version 2.20, http://www.rsa.com/rsalabs/pkcs/

[13] ZFS, http://www.opensolaris.org/os/community/zfs/

[14] RFC 3280: Internet X.509 Public Key Infrastructure, Certificate and Certificate Revocation List (CRL) Profile

[15] X.509 authentication in SSH, `draft-ietf-secsh-x509-03.txt`

[16] X.509 authentication in SSH, `draft-saarenmaa-ssh-x509-00.txt`

[17] SunSSH project, http://www.opensolaris.org/os/community/security/projects/SSH

[18] Solaris Cryptographic Framework, http://opensolaris.org/os/project/crypto/

[19] RFC 4120: The Kerberos Network Authentication Service (V5)

[20] RFC 4556: Public Key Cryptography for Initial Authentication in Kerberos

[21] Kerberos project, http://www.opensolaris.org/os/project/kerberos

[22] Tripwire Enterprise, http://www.tripwire.com/products/enterprise/index.cfm

[23] bart(1M), manual page for Basic Audit Reporting Tool

(Footnotes)

1 **(Footnotes)**

[1] as of June 2008, the new KMF API is available in OpenSolaris only. The one present in Solaris 10 is a different private API and should not be used by third party applications.

2 [2] format of the PKCS#11 URI is not standardized yet so the current format used is subject to change

3 [3] during the initialization of the engine, a set of mechanisms that the engine supports is returned to OpenSSL through the engine initialization function. OpenSSL then offloads only mechanisms that are part of the set.

4 [4] for a definition of a *slot,* please see the PKCS#11 standard[12]

5 [5] that is what `libpkcs11` library itself uses

6 [6] the speedup gained covers the SCP and SFTP protocols as well since those protocols work above the SSH protocol and and rely on the SSH protocol to securely transfer the data they need

ZFS On-Disk Data Walk (or: Where‘s my Data?)

DRAFT

Abstract

This paper will examine the on-disk format of ZFS by using modified versions of mdb and zdb to actually look at the data structures of a ZFS file system on disk. The paper will cover the steps taken to convert a pathname in a ZFS file system to get to the actual data of the file at the end of the pathname. At each step, the data structure(s) being examined will be described. You may try to follow along by using a ZFS file system on your machine, but it is recommended that the file system not be very large. For the example that is covered here, the total amount of space taken by the file system is 3.1MB out of a single disk volume that is 37GB large.

The techniques shown should work for larger file systems, and should also work for file systems that span multiple disks in any configuration. The technique has been tried on a much larger file system, and on a 2-way mirror, and works. However, there may be more steps that need to be taken, particularly with respect to indirection. To get the most from this paper, it is recommended you have a copy of the “ZFS On_Disk Specification”, available at:

http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf. and the source code, available at:

The paper is divided into the following sections:

- Overview of On-Disk Data Structures
- On-Disk data structure walk.
- Future Work.
- Conclusions.
- References.

Overview of On-Disk Data Structures

The following is a list of most of the data structures found on the disk(s) making up a ZFS pool. This is not a complete list. Rather, we concentrate on the structures that we’ll be using in the “On-Disk data structure walk” portion of this paper. Header files for these structures can be found at uts/common/fs/zfs/sys/*.h. For each structure, we’ll list the header file where the structure can be found. There are additional structures used to maintain information about a mounted ZFS file system in memory, but these structures will have to be the subject of another paper.

For a more complete description of these structures, please see the “ZFS On-Disk Specification” paper available at http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf.

- uberblock_t - This is the starting point for locating any structure within a ZFS file system. These are organized in an array, starting at physical location 0x20000 bytes within a vdev label. There are 4 copies of the vdev label on a given physical disk in use with ZFS. These are at the beginning of the disk at physical location 0x0 and at 0x40000 (256k), and at the end of the disk, N-512k and N-256K (where "N" is the size of the disk). Each uberblock_t in the 128k array of uberblocks is 1k large (making 128

uberblock_t in the array. At any point in time, only 1 uberblock_t in the array is "active". The uberblock_t with the highest transaction group id and valid checksum is the active uberblock. You can use "zdb -uuu zpool_name" to display the active uberblock_t for a "zpool_name". The uberblock_t is defined in uts/common/fs/zfs/sys/uberblock_impl.h.

- blkptr_t - Structure used to locate, describe, and verify blocks on disk. These are embedded within other structures (most commonly, dnode_phys_t structs), and also exist in arrays as "indirect" blocks. They are defined in uts/common/fs/zfs/sys/spa.h.

- dnode_phys_t - Almost everything in ZFS exists as an object. The dnode_phys_t is a 512-byte structure that describes an object. It is defined in uts/common/fs/zfs/sys/dnode.h. The dnode_phys_t contains a blkptr_t and may also contain data in a "bonus buffer" (dn_bonusbuffer). The types of objects a dnode_phys_t describes are defined in uts/common/fs/zfs/sys/dmu.h, and the type of object for a given dnode_phys_t is in the dn_type structure member.

- objset_phys_t - The objset_phys_t describes a group of objects. There will be one (at least) of these for the set of all objects (the "meta object set", or MOS), and one for ZFS file system objects (files and directories). This is defined in usr/common/fs/zfs/sys/dmu_objset.h.

- ZAP Objects - These are block(s) containing name/value pair attributes. (ZAP is "ZFS Attribute Processor"). There are two basic types of ZAP objects, "microzap" and "fatzap". Microzap objects are used when the attributes can fit in one block (there are other restrictions, not important here). Fatzap objects allow for more attributes than fit in one block. The first 64 bits of a ZAP object define whether it is a microzap object or a fatzap object. Microzap objects use the mzap_phys_t type defined in uts/common/fs/zfs/sys/zap_impl.h. The last field in the mzap_phys_t is an mzap_ent_phys_t. There may be more than one mzap_ent_phys_t in a microzap object. The number of mzap_ent_phys_t depends on the size of the block where the microzap resides. Not all entries need be in use. A fatzap object is made up of two structures. The starting point is a zap_phys_t. This contains a pointer table of blocks containing zap_leaf_chunk_t structs. We won't be looking at fatzaps in this paper. Take a look at uts/common/fs/zfs/sys/zap_leaf.h for more information.

The following objects exist in the bonus buffer of some dnode_phys_t structures.

- dsl_dir_phys_t - This is in the bonus buffer of a DMU_OT_DSL_DIR (DSL Directory) dnode_phys_t for the MOS. It contains the object id of a DMU_OT_DSL_DATASET dnode_phys_t. See uts/common/fs/zfs/sys/dsl_dir.h.

- dsl_dataset_phys_t - This is in the bonus buffer of a DMU_OT_DSL_DATASET dnode_phys_t in the MOS. This contains a blkptr_t which (indirectly) leads to a second array of dnode_phys_t for objects within a ZFS file system.

- znode_phys_t - This is in the bonus buffer of dnode_phys_t structures for plain files and directories. It contains attributes of the file/directory (time stamps, ownership, size, etc.). This structure is the closest equivalent to the disk inode for UFS file system files/directories.

On-Disk Data Structure Walk

In this section of the paper, the data for a file is found on a ZFS file system. To do this, we start at the uberblock_t and follow data structures on disk until we get to the data. If you want to "follow along" on your machine, you'll need a version of mdb that allows you to use the "::print" dcmd to print data structures from a raw file, a version of zdb that allows you to decompress blocks before dumping them out, and an mdb dmod that has the "::blkptr" command. These are available here??? Note: the ZFS file system I am using for this does not span multiple disks. If your file system spans multiple disks, the technique should still work, but there may be more indirect blocks in use, and the vdev used with the zdb command will probably change for different blkptr_t structs. In other words, pay attention to the output you get from each step before proceeding to the next step.

At a high level, the following steps are taken:

1. Copy a file with known contents into the top directory in a mounted ZFS file system. This step isn't completely necessary. It is done here so that when we get to the data, we know we have the correct data.
2. Display the active uberblock_t, along with its blkptr_t.
3. Display the objset_phys_t for the MOS.
4. Display the Object Directory dnode_phys_t, and its ZAP object for the MOS.
5. Display the DSL Directory dnode_phys_t and its bonus buffer for the MOS.
6. Display the DSL Dataset dnode_phys_t and its bonus buffer for the MOS.
7. From the DSL Dataset dnode, display the objset_phys_t for the ZFS file system.
8. Using the objset_phys_t for the ZFS file system, display the Master Node dnode_phys_t, and its ZAP object. This will almost certainly involve walking through indirect blocks.
9. From the Master Node ZAP object, display the root directory dnode_phys_t for the ZFS file system.
10. From the blkptr_t in the root directory dnode_phys_t, find the object id of the file we want to display. (Note, the copy of the file in step 1 placed the file in the root directory of the file system). If you want to find a file in a subdirectory, you will need to take more steps.
11. From the dnode_phys_t of the file we want to examine, display the data via the blkptr_t structure(s) for the file. The following diagram shows the path we are going to take. You might find it useful to refer to this as we go through the steps.

First, we'll copy a file with known contents into the top directory of a zfs file system.

```
# cp /usr/dict/words /laciedisk/words
#
```

Next, we'll use zdb to read the current uberblock for the pool.

```
# zdb -uuu laciedisk
Uberblock

        magic = 0000000000bab10c
        version = 10
        txg = 19148
        guid_sum = 17219723339164464949
        timestamp = 1203801884 UTC = Sat Feb 23 22:24:44 2008
        rootbp = [L0 DMU objset] 400L/200P DVA[0]=<0:4e00:200> DVA[1]=<0:1c0004e00:200>
DVA[2]=<0:380001200:200> fletcher4 lzjb LE contiguous birth=19148 fill=18 cksum=89aca5d29:38d2
71ef883:be5570b26779:1af282de579a51
#
```

The above output shows that the objset_phys_t contains a blkptr_t with 3 copies of the DVA (Data Virtual Address), one at disk block 0x4e00, the second at 0x1c0004e00, and the third at 0x380001200. The three copies of the same data are called "ditto blocks". Ditto blocks are used for most of the metadata in a zfs file system, and provide a redundancy mechanism in case of failures. Each of these is 0x200 bytes on the disk, and is on vdev 0 (the only disk in the pool). The objset_phys_t is compressed with lzjb compression, and after compression it is 0x400 bytes (400L in the above output). Note that locations reported for DVAs (4e00, c004e00, 380001200 above) are byte locations from the end of the 2 zfs labels at the front of the disk. The 2 disk labels occupy the first 0x400000 bytes (4MB) of the disk. So, the byte offsets from the beginning of the disk for the three ditto blocks shown above are 0x404e00, 0x1c0404e00, and 0x380401200. We'll need to find which device(s) make up the pool. We'll use zpool(1) for this.

```
# zpool status laciedisk
  pool: laciedisk
 state: ONLINE
```

```
 scrub: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        laciedisk   ONLINE       0     0     0
          c4t0d0p1  ONLINE       0     0     0

errors: No known data errors
#
```

Now we'll use zdb to dump the raw data referred to in the uberblock. This is an objset_phys_t. The syntax of the "-R" option to read zfs blocks is:

```
# zdb -R pool:vdev_specfier:offset:size:flags
```

where:

- "pool" is the zfs pool.
- "vdev_specifier" is either device name or id. In this case, c4t0d0p1 is specified. Alternatively, one can use "0" for a zfs file system on a single device. In the case of a 2-way miror, you can use the device name as here, or "0.0" to specify the first device, and "0.1" to specify the second device. See the comment at the beginning of the zdb_vdev_lookup() function in usr/src/cmd/zdb/zdb.c for naming conventions.
- "offset" is the byte offset following the disk label (L1 and L2). The L1 and L2 labels (along with boot block info) takes up 4MB (0x400000). So the physical address on the disk is "offset + 0x400000".
- "size" is the physical size in bytes on the disk.
- "flags" are a set of characters specifying options. The "d" flag used here is not supported in the current version of zdb. The version of zdb being used here has been extended to support the "d" option. With the "d" option, one specifies the compression algorithm and decompressed size.

For more information on the "-R" command, see the beginning of the zdb_read_block() routine in the zdb source code (usr/src/uts/cmd/zdb/zdb.c). Note that the zdb(1M) manual page states that the Interface Stability is "Unstable", indicating that the syntax of the zdb(1M) command is subject to change.

The command below reads a 0x200 bytes from the laciedisk zpool and the /dev/dsk/c4t0d0p1 device in the pool. It reads block number 0x4e00 and performs lzjb decompression. The size after decompression is 0x400 bytes. The output is placed in /tmp/metadnode.

```
# zdb -R laciedisk:c4t0d0p1:4e00:200:d,lzjb,400 2> /tmp/metadnode
Found vdev: /dev/dsk/c4t0d0p1
#
```

/tmp/metadnode contains the objset_phys_t data. (The beginning of the objset_phys_t is a dnode_phys_t which describes the meta object set, i.e., the "metadnode"). We use (a modified) mdb to print the structure contents. With the modified mdb, the current kernel CTF information is loaded (::loadctf), and a dmod is loaded. The kernel CTF information allows one to ::print data structures.

```
# mdb /tmp/metadnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::print -a -t zfs`objset_phys_t
{
   0 dnode_phys_t os_meta_dnode = {
      0 uint8_t dn_type = 0xa        <-- DMU_OT_DNODE (see uts/common/fs/zfs/sys/dmu.h)
      1 uint8_t dn_indblkshift = 0xe
      2 uint8_t dn_nlevels = 0x1     <-- no indirect blocks
```

```
      3 uint8_t dn_nblkptr = 0x3     <-- 3 copies in the blkptr_t
      4 uint8_t dn_bonustype = 0     <-- no data in bonus buffer
 ... <--output omitted
      40 blkptr_t [1] dn_blkptr = [    <-- blkptr is at address 0x40 in tmp file
        {
          40 dva_t [3] blk_dva = [
            {
              40 uint64_t [2] dva_word = [ 0x4, 0x28 ]
            }
            {
              50 uint64_t [2] dva_word = [ 0x4, 0xe00028 ]
            }
            {
              60 uint64_t [2] dva_word = [ 0x4, 0x1c0001c ]
            }
          ]
          70 uint64_t blk_prop = 0x800a07030003001f
          78 uint64_t [3] blk_pad = [ 0, 0, 0 ]
          90 uint64_t blk_birth = 0x4acc
          98 uint64_t blk_fill = 0x11
          a0 zio_cksum_t blk_cksum = {
            a0 uint64_t [4] zc_word = [ 0x8348a7aa95, 0x8a9a1c0eb664,
0x5b0e32bd611ac7, 0x2d691acc5e1f456f ]
          }
        }
      ]
... <-- output omitted
>
```

So, now will use the blkptr dcmd to get a “nicer” view of the blkptr_t in the objset_phys_t.

```
> 40::blkptr
DVA[0]: vdev_id 0 / 5000
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 80000000000
DVA[0]: :0:5000:800:d
DVA[1]: vdev_id 0 / 1c0005000
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 80000000000
DVA[1]: :0:1c0005000:800:d
DVA[2]: vdev_id 0 / 380003800
DVA[2]:       GANG: FALSE  GRID:  0000  ASIZE: 80000000000
DVA[2]: :0:380003800:800:d
LSIZE:  4000                    PSIZE: 800
ENDIAN: LITTLE                          TYPE:  DMU dnode
BIRTH:  4acc            LEVEL: 0     FILL:  1100000000
CKFUNC: fletcher4                  COMP:  lzjb
CKSUM:  8348a7aa95:8a9a1c0eb664:5b0e32bd611ac7:2d691acc5e1f456f
> $q
#
```

The objset_phys_t contain block number(s) of either indirect blocks or block numbers of a block containing an array of dnode_phys_t. This is from the dn_type and dn_nlevels fields in the dnode_phys_t, or the “TYPE” information (and LEVEL) information in the blkptr_t. For this objset_phys_t, the blkptr_t is for an array of dnode_phys_t, (no indirection since LEVEL is 0). This array of dnode_phys_t makes up the meta object set, or

"MOS". Now we'll go back to zdb to dump the decompressed dnode_phys_t array, using block number 5000.

```
# zdb -R laciedisk:c4t0d0p1:5000:800:d,lzjb,4000 2> /tmp/mos
Found vdev: /dev/dsk/c4t0d0p1
#
```

The physical size of the block is 0x800 bytes, the logical size (after decompression) is 0x4000 bytes (from the output of ::blkptr above).

```
# mdb /tmp/mos
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> ::sizeof zfs`dnode_phys_t   <-- how large is a dnode_phys_t?
sizeof (zfs`dnode_phys_t) = 0x200
> 4000%200=K       <-- how many dnode_phys_t are there in the block?
              20

> 0,20::print -a -t zfs`dnode_phys_t <-- dump the 32 dnode_phys_t of the objset
{
   0 uint8_t dn_type = 0   <-- DMU_OT_NONE (not in use)
...  <-- output truncated
}
{
   200 uint8_t dn_type = 0x1  <-- DMU_OT_OBJECT_DIRECTORY
... <-- output omitted
   240 blkptr_t [1] dn_blkptr = [
      {
         240 dva_t [3] blk_dva = [
            {
               240 uint64_t [2] dva_word = [ 0x1, 0x1 ]
            }
            {
               250 uint64_t [2] dva_word = [ 0x1, 0xe00001 ]
            }
            {
               260 uint64_t [2] dva_word = [ 0x1, 0x1c00000 ]
            }
         ]
         270 uint64_t blk_prop = 0x8001070200000000
         278 uint64_t [3] blk_pad = [ 0, 0, 0 ]
         290 uint64_t blk_birth = 0x4
         298 uint64_t blk_fill = 0x1
         2a0 zio_cksum_t blk_cksum = {
            2a0 uint64_t [4] zc_word = [ 0x5a6f58679, 0x1cf035fcb09,
0x528ae171d3a8, 0xaabf09f1373ae ]
         }
      }
   ]
   2c0 uint8_t [320] dn_bonus = [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ... ]
}
{
   400 uint8_t dn_type = 0xc   <-- DMU_OT_DSL_DIR
```

```
... <-- output omitted
}
{
   600 uint8_t dn_type = 0xf  <-- DMU_OT_DSL_PROPS
...  <-- output omitted
}
{
   800 uint8_t dn_type = 0xd  <-- DMU_OT_DSL_DIR_CHILD_MAP
... <-- output omitted
}
{
   a00 uint8_t dn_type = 0x10  <-- DMU_OT_DSL_DATASET
... <-- output omitted
}
... <-- output omitted, several other objects, and some unused entries
>
```

In the above, the first dnode_phys_t is not used. The second (object id #1) is for a DMU_OT_OBJECT_DIRECTORY. The object directory is always object id 1 in the objset_phys_t. The blkptr_t for the object directory is a zap object that contains object names and ids to allow one to find all other objects (besides the object directory).

Now, we'll dump the blkptr_t at 0x240 (this is the blkptr_t for the DMU_OT_OBJECT_DIRECTORY).

```
> 240::blkptr
DVA[0]: vdev_id 0 / 200
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[0]: :0:200:200:d
DVA[1]: vdev_id 0 / 1c0000200
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[1]: :0:1c0000200:200:d
DVA[2]: vdev_id 0 / 380000000
DVA[2]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[2]: :0:380000000:200:d
LSIZE:  200                     PSIZE: 200
ENDIAN: LITTLE                          TYPE:  object directory
BIRTH:  4               LEVEL: 0     FILL:  100000000
CKFUNC: fletcher4                   COMP:  uncompressed
CKSUM:  5a6f58679:1cf035fcb09:528ae171d3a8:aabf09f1373ae
> $q
#
```

The blkptr is for an object directory, and is uncompressed. We'll use zdb to get the raw output of the object directory.

```
# zdb -R laciedisk:c4t0d0p1:200:200:r 2> /tmp/objdir
Found vdev: /dev/dsk/c4t0d0p1
#
```

An object directory (dn_type == DMU_OT_OBJECT_DIRECTORY) is a ZAP (ZFS Attribute Processor) object. ZAP objects can be "fatzaps" or "microzaps". ZAP objects contain name-value pairs used to store attributes of an object. The first 64-bit value in a ZAP object is used to identify the type of ZAP contents contained within the ZAP block. A value of ZBT_MICRO (=0x8000000000000003) indicates a microzap. A microzap contains all of the attributes within a single block. A value of ZBT_HEADER (= 0x8000000000000001) is used for the first block of a fatzap. All other blocks for fatzap objects have a value of ZBT_LEAF (=0x800000000000000) in the

first 64-bits of the block.

```
# mdb /tmp/objdir
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0/J
0:              8000000000000003 <-- this is a microzap

> 0::print -a -t zfs`mzap_phys_t  <-- print an mzap_phys_t (see uts/common/fs/zfs/sys/zap_impl.h)
{
    0 uint64_t mz_block_type = 0x8000000000000003
    8 uint64_t mz_salt = 0x41d10a3
    10 uint64_t mz_normflags = 0
    18 uint64_t [5] mz_pad = [ 0, 0, 0, 0, 0 ]
    40 mzap_ent_phys_t [1] mz_chunk = [
        {
            40 uint64_t mze_value = 0x2
            48 uint32_t mze_cd = 0
            4c uint16_t mze_pad = 0
            4e char [50] mze_name = [ “root_dataset” ]
        }
    ]
}
```

The structure used to maintain microzap objects is a mzap_phys_t. The last member of the structure is a variable sized array of mzap_ent_phys_t structures. Each mzap_ent_phys_t contains a name-value pair. Possible attributes for the object directory include: “root_dataset”, “config”, “sync_bplist”, and “deflate”. The root_dataset attribute contains the object number of the root DSL (Dataset and Snapshot Layer) directory for the pool. The root DSL contains information about all top level datasets within the pool. It is an object of type DMU_OT_DSL_DIR. We’ll take a closer look at this object shortly.

Below, we look at the next 2 mzap_ent_phys_t structures in the object directory. One is a “config” object with contains the object number of a DMU_OT_PACKED_NVLIST, the other is a “deflate” object. (For more information on these, refer to the source code).

```
> ::sizeof zfs`mzap_ent_phys_t
sizeof (zfs`mzap_ent_phys_t) = 0x40
> 80::print -a -t zfs`mzap_ent_phys_t
{
    80 uint64_t mze_value = 0xb
    88 uint32_t mze_cd = 0
    8c uint16_t mze_pad = 0
    8e char [50] mze_name = [ “config” ]
}
> c0::print -a -t zfs`mzap_ent_phys_t
{
    c0 uint64_t mze_value = 0x1
    c8 uint32_t mze_cd = 0
    cc uint16_t mze_pad = 0
    ce char [50] mze_name = [ “deflate” ]
}

> $q
#
```

Now, we take a look at object id 2, the "root_dataset". This is the third dnode_phys_t in the array of dnode_phys_t referenced by the objset_phys_t. For this, we go back to the dnode_phys_t array from the objset_phys_t referenced by the uberblock, and look at object id 2. The object starts at 0x400 (2 * sizeof(dnode_phys_t)) from the beginning of the array.

```
# mdb /tmp/mos
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 400::print zfs`dnode_phys_t
{
   400 uint8_t dn_type = 0xc   <-- DMU_OT_DSL_DIR
 ... <-- output omitted
   404 uint8_t dn_bonustype = 0xc
... <-- output omitted
   40a uint16_t dn_bonuslen = 0x100
... <-- output omitted
   440 blkptr_t [1] dn_blkptr = [
      {
         440 dva_t [3] blk_dva = [
            {
               440 uint64_t [2] dva_word = [ 0, 0 ]  <-- blkptr not used
            }
... <-- output omitted
   ]
   4c0 uint8_t [320] dn_bonus = [ 0x34, 0x9c, 0xb9, 0x47, 0, 0, 0, 0, 0x5, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ... ]
}
>
```

This dnode_phys_t has information in the dn_bonus section. The type of this information (dn_bonustype == 0xc) is a DMU_OT_DSL_DIR (see uts/common/fs/zfs/dmu.h). DSL directory objects store a dsl_dir_phys_t in the bonus buffer (dn_bonus) of the dnode_phys_t. Note that the blkptr_t is all 0, so there is no blkptr_t for this dnode_phys_t. All information is in the bonus buffer.

The bonus buffer is at offset 0x4c0.

```
> 4c0::print -a -t zfs`dsl_dir_phys_t
{
   4c0 uint64_t dd_creation_time = 0x47b99c34
   4c8 uint64_t dd_head_dataset_obj = 0x5
 ... <-- output omitted
}
>
```

So, the dd_head_dataset_obj object id is 5. Let's take a look at that dnode_phys_t.

```
> 5*200::print -t -a zfs`dnode_phys_t
{
   a00 uint8_t dn_type = 0x10  <-- DMU_OT_DSL_DATASET
... <-- output omitted
   a04 uint8_t dn_bonustype = 0x10
... <-- output omitted
   a0a uint16_t dn_bonuslen = 0x140
```

```
... <-- output omitted
   a40 blkptr_t [1] dn_blkptr = [
      {
         a40 dva_t [3] blk_dva = [
            {
               a40 uint64_t [2] dva_word = [ 0, 0 ]  <-- blkptr not used
            }
... <-- output omitted
   ]
   ac0 uint8_t [320] dn_bonus = [ 0x2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ... ]
}
>
```

Once again, the blkptr_t is not in use, but the bonus buffer contains data. The type of the bonus buffer data is DMU_OT_DSL_DATASET (dn_bonustype == 0x10). The bonus buffer is a dsl_dataset_phys_t. Let's take a look at this.

```
> ac0::print -a -t zfs`dsl_dataset_phys_t
{
   ac0 uint64_t ds_dir_obj = 0x2
 ... <-- output omitted
   b40 blkptr_t ds_bp = {
      b40 dva_t [3] blk_dva = [
         {
            b40 uint64_t [2] dva_word = [ 0x1, 0x26 ]
         }
         {
            b50 uint64_t [2] dva_word = [ 0x1, 0xe00026 ]
         }
         {
            b60 uint64_t [2] dva_word = [ 0, 0 ]
         }
      ]
... <-- output omitted
>
```

Here, there is a blkptr_t at 0xb40 bytes. We'll take a look at this.

```
> b40::blkptr
DVA[0]: vdev_id 0 / 4c00
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[0]: :0:4c00:200:d
DVA[1]: vdev_id 0 / 1c0004c00
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[1]: :0:1c0004c00:200:d
LSIZE:  400                   PSIZE: 200
ENDIAN: LITTLE                        TYPE:  DMU objset
BIRTH:  4acc            LEVEL: 0     FILL:  1500000000
CKFUNC: fletcher4                COMP:  lzjb
CKSUM:  a9a691571:477e2285748:f44e24c94c3d:23418ff35bb270
> $q
#
```

Note that this is the second DMU objset we have seen. The first was pointed to by the uberblock (see output of zdb -uuu laciedisk above). The first objset is for all objects in the pool. The second object set is for the DSL objects. The metadnode in this object set is for the root dataset of the file system. The blkptr_t specifies physical size 0x200 and logical size 0x400, with lzjb compression. We'll use zdb to dump the block.

```
# zdb -R laciedisk:c4t0d0p1:4c00:200:d,lzjb,400 2> /tmp/root_dataset_metadnode
Found vdev: /dev/dsk/c4t0d0p1
#
```

And now back to mdb to examine the (decompressed) data of the block. This is an objset_phys_t.

```
# mdb /tmp/root_dataset_metadnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::print -a -t zfs`objset_phys_t
{
   0 dnode_phys_t os_meta_dnode = {
      0 uint8_t dn_type = 0xa  <-- DMU_OT_DNODE
      1 uint8_t dn_indblkshift = 0xe
      2 uint8_t dn_nlevels = 0x7  <-- levels of indirection
... <-- output omitted
      40 blkptr_t [1] dn_blkptr = [
         {
            40 dva_t [3] blk_dva = [
               {
                  40 uint64_t [2] dva_word = [ 0x2, 0x24 ]
               }
               {
                  50 uint64_t [2] dva_word = [ 0x2, 0xe00024 ]
               }
               {
                  60 uint64_t [2] dva_word = [ 0, 0 ]
               }
            ]
... <-- output omitted
>
```

Decoding the first blkptr_t gives:

```
> 40::blkptr
DVA[0]: vdev_id 0 / 4800
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[0]: :0:4800:400:id
DVA[1]: vdev_id 0 / 1c0004800
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[1]: :0:1c0004800:400:id
LSIZE:  4000                    PSIZE: 400
ENDIAN: LITTLE                          TYPE:  DMU dnode
BIRTH:  a1d0            LEVEL: 6     FILL:  1400000000
CKFUNC: fletcher4                COMP:  lzjb
CKSUM:  5b1be31814:3f256d9bc248:172d1fb48446d2:5f55736a7cf2c95
> $q
```

```
#
```

To get to the dnode_phys_t structure(s) that are referenced by the blkptr_t, we must go through 6 levels of indirection to get to the blkptr_t's that contain the dnode_phys_t's (7 levels to get to the dnode_phys_t structures themselves).

Note the LEVEL value of 6. This blkptr_t is for a block containing blkptr_t's of blkptr_t's. It is 0x400 bytes on the disk, and, after lzjb decompression is 0x4000 bytes large. We want the dnode_phys_t for the root of the file system. To get this, we need the dnode_phys_t whose dn_type is DMU_OT_MASTER_NODE. This dnode_phys_t is always object id 1. This means that regardless of the levels of indirection, we always want the first indirect block at every level. When we get to the block containing dnode_phys_t, we want the second dnode_phys_t in the array.

```
# zdb -R laciedisk:c4t0d0p1:4800:200:d,lzjb,4000 2> /tmp/blkptr6
Found vdev: /dev/dsk/c4t0d0p1
#
# mdb /tmp/blkptr6
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 4400
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[0]: :0:4400:400:id
DVA[1]: vdev_id 0 / 1c0004400
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[1]: :0:1c0004400:400:id
LSIZE:  4000                    PSIZE: 400
ENDIAN: LITTLE                          TYPE:  DMU dnode
BIRTH:  a1d0            LEVEL: 5     FILL:  1400000000
CKFUNC: fletcher4                COMP:  lzjb
CKSUM:  5b9be7d355:3f9ee0f14633:1766faff22becd:607cb255f0cef14
```

Here is the level 5 indirect block. Let's read it and dump it out.

```
# zdb -R laciedisk:c4t0d0p1:4400:400:d,lzjb,4000 2> /tmp/blkptr5
Found vdev: /dev/dsk/c4t0d0p1
#
```

Now, back to mdb to display the blkptr_t array. Each blkptr_t takes up 0x80 bytes, so in a 0x4000 byte array of blkptr_t, there are 0x80 blkptr_t.

```
# mdb /tmp/blkptr5
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 4000
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[0]: :0:4000:400:id
DVA[1]: vdev_id 0 / 1c0004000
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[1]: :0:1c0004000:400:id
LSIZE:  4000                    PSIZE: 400
ENDIAN: LITTLE                          TYPE:  DMU dnode
BIRTH:  71d2            LEVEL: 4     FILL:  1400000000
CKFUNC: fletcher4                COMP:  lzjb
```

```
CKSUM:  572ea43879:3c2b4d3c6fed:15ebe34a088538:59553188934fe94
> $q
#
```

Note that the LEVEL value is now 4. Also, out of 128 blkptr_t at level 6, only one is being used. When do others get used? When more files are added to the file system. This file system only uses about 3.1MB out of an available 37GB (from "df -h" output). How do we pick the blkptr_t that we need? This depends on the object id (in UFS, the "inumber") in which we are interested. Here, we want to start from the root of the filesystem and traverse pathnames. So we start from the DMU_OT_MASTER_NODE, which is always object id 1. So, we'll use the first blkptr_t as our starting point (which is good in this case, since there is only one valid blkptr_t in the array).

Now, we'll use zdb to dump the blkptr_t array from this level of indirection. The blkptr_t array is at physical block number 0x4000 and the (lzjb compressed) size on disk is 0x400 bytes. The uncompressed size is 0x4000 bytes.

```
# zdb -R laciedisk:c4t0d0p1:4000:400:d,lzjb,4000 2> /tmp/blkptr4
Found vdev: /dev/dsk/c4t0d0p1
#
```

And again mdb to display the blkptr_t array. Again, we'll only look at the first indirect blkptr_t.

```
# mdb /tmp/blkptr4
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 2000
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[0]: :0:2000:400:id
DVA[1]: vdev_id 0 / 1c0002000
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[1]: :0:1c0002000:400:id
LSIZE:  4000                    PSIZE: 400
ENDIAN: LITTLE                          TYPE:  DMU dnode
BIRTH:  4acc            LEVEL: 3     FILL:  1400000000
CKFUNC: fletcher4                COMP:  lzjb
CKSUM:  5b1a31ac6b:3f202983b5d8:1728de4368d22a:5f36cda2d00cc77

$q
#
```

The first blkptr_t in the array is for level 3 indirection. Again, it is lzjb compressed and is 0x400 bytes on the disk. The uncompressed (LSIZE) is 0x4000 bytes. The second blkptr_t is unallocated (as, we'll assume, are the remaining blkptr_t).

Now, back to zdb to get the next level of indirection.

```
# zdb -R laciedisk:c4t0d0p1:2000:400:d,lzjb,4000 2> /tmp/blkptr3
Found vdev: /dev/dsk/c4t0d0p1
#
```

And back to mdb to display the array of 0x80 entries (4000%80, where 0x4000 is the uncompressed size and 0x80 is the size of a blkptr_t). And again, we're only interested in the first entry.

```
# mdb /tmp/blkptr3
> ::loadctf
```

```
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 1800
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[0]: :0:1800:400:id
DVA[1]: vdev_id 0 / 1c0001800
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[1]: :0:1c0001800:400:id
LSIZE:  4000                    PSIZE: 400
ENDIAN: LITTLE                        TYPE:  DMU dnode
BIRTH:  4acc            LEVEL: 2     FILL:  1400000000
CKFUNC: fletcher4                COMP:  lzjb
CKSUM:  5971752f5e:3d8cfbe170db:1668b663cd16fc:5b61debef02b3cd

> $q
#
```

And we'll go back to zdb to get the blkptr_t array specified for level 2 indirection.

```
# zdb/i386/zdb -R laciedisk:c4t0d0p1:1800:400:d,lzjb,4000 2> /tmp/blkptr2
Found vdev: /dev/dsk/c4t0d0p1
#
```

And back to mdb. (It would be nice if one could do all of this within mdb or zdb. Or maybe a new tool, mzdb/zmdb anyone?).

```
# mdb /tmp/blkptr2
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 1400
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[0]: :0:1400:400:id
DVA[1]: vdev_id 0 / 1c0001400
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[1]: :0:1c0001400:400:id
LSIZE:  4000                    PSIZE: 400
ENDIAN: LITTLE                        TYPE:  DMU dnode
BIRTH:  4acc            LEVEL: 1     FILL:  1400000000
CKFUNC: fletcher4                COMP:  lzjb
CKSUM:  5853930aed:3d385d7c3c5f:166a8b5f2e721c:5bd9492aa861e02
> 80::blkptr
LSIZE:  0                       PSIZE: 200
ENDIAN:    BIG                        TYPE:  unallocated
BIRTH:  0               LEVEL: 0     FILL:  0
CKFUNC: inherit                  COMP:  inherit
CKSUM:  0:0:0:0
> $q
#
```

And again to zdb to get the blkptr_t array at level 1.

```
# /zdb -R laciedisk:c4t0d0p1:1400:400:d,lzjb,4000 2> /tmp/blkptr1
```

```
Found vdev: /dev/dsk/c4t0d0p1
#
```

And once more to mdb...

```
# mdb /tmp/blkptr1
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 3600
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: a0000000000
DVA[0]: :0:3600:a00:d
DVA[1]: vdev_id 0 / 1c0003600
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: a0000000000
DVA[1]: :0:1c0003600:a00:d
LSIZE:  4000                    PSIZE: a00
ENDIAN: LITTLE                          TYPE:  DMU dnode
BIRTH:  4acc            LEVEL: 0     FILL:  1400000000
CKFUNC: fletcher4                  COMP:  lzjb
CKSUM:  ad27cf4b34:ecbe6f671f03:c231a96352c039:7726f4dc30e0e44b
> $q
#
```

This is the level 0 blkptr_t. It is 0xa00 bytes on the disk, and logical (i.e., decompressed) size is 0x4000 bytes. We'll go back to zdb once more to get this block. This should be an array of dnode_phys_t structures.

```
# zdb -R laciedisk:c4t0d0p1:3600:a00:d,lzjb,4000 2> /tmp/dnode
Found vdev: /dev/dsk/c4t0d0p1
#
```

Let's use mdb again. This time, instead of indirect blkptr_t, we should have dnode_phys_t.

```
# mdb /tmp/dnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0,20::print -a -t zfs`dnode_phys_t  <-- ",20" is the number of dnode_phys_t in 0x4000
                              <--  bytes (4000 divided by sizeof dnode_phys_t)
{
   0 uint8_t dn_type = 0  <-- first entry not used
... <-- output omitted
{
   200 uint8_t dn_type = 0x15  <-- DMU_OT_MASTER_NODE (from uts/common/fs/zfs/sys/dmu.h)
   201 uint8_t dn_indblkshift = 0xe
... <-- output omitted
   240 blkptr_t [1] dn_blkptr = [
      {
         240 dva_t [3] blk_dva = [
            {
               240 uint64_t [2] dva_word = [ 0x1, 0x63 ]
            }
... <-- output omited
{
   600 uint8_t dn_type = 0x14  <-- DMU_OT_DIRECTORY_CONTENTS (a directory)
```

```
... <-- output omitted
}
{
   800 uint8_t dn_type = 0x13  <-- DMU_OT_PLAIN_FILE_CONTENTS  (a regular file)
... <-- output omitted
}
... <-- output omitted.  The remaining dnode_phys_t are either
   <-- DMU_OT_PLAIN_FILE_CONTENTS or DMU_OT_NONE
>
```

The DMU_OT_MASTER_NODE dnode_phys_t is a ZAP object. It is used to identify the root directory, delete queue, and version information for a file system. Let's look at the blkptr_t.

```
> 240::blkptr
DVA[0]: vdev_id 0 / c600
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[0]: :0:c600:200:d
DVA[1]: vdev_id 0 / 1c000c600
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[1]: :0:1c000c600:200:d
LSIZE:  200                     PSIZE: 200
ENDIAN: LITTLE                          TYPE:  ZFS master node
BIRTH:  6               LEVEL: 0     FILL:  100000000
CKFUNC: fletcher4                   COMP:  uncompressed
CKSUM:  264216abd:f8ce291b17:347052edfaa6:79edede0bbfd6
> $q
#
```

So, this is for a 0x200 byte block and is uncompressed. We'll use zdb to dump out the block.

```
# zdb -R laciedisk:c4t0d0p1:c600:200:r 2> /tmp/zfs_master_node
Found vdev: /dev/dsk/c4t0d0p1
#
```

And back to mdb to display the contents.

```
# mdb /tmp/zfs_master_node
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0/J
0:              8000000000000003 <-- this is a microzap
> 0::print -t -a zfs`mzap_phys_t
{
   0 uint64_t mz_block_type = 0x8000000000000003
... <-- output omitted
   40 mzap_ent_phys_t [1] mz_chunk = [
      {
         40 uint64_t mze_value = 0x3
 ... <-- output omitted
         4e char [50] mze_name = [ "VERSION" ]
      }
   ]
}
```

```
80::print -a -t zfs`mzap_ent_phys_t
{
... <-- output omitted
   8e char [50] mze_name = [ “DELETE_QUEUE” ]
}
c0::print -a -t zfs`mzap_ent_phys_t
{
   c0 uint64_t mze_value = 0x3  <-- the object id for the root directory of the fs
... <-- output omitted
   ce char [50] mze_name = [ “ROOT” ]
}
$q
#
```

Now, back to the dnode_phys_t array to look at object id 3, the object id for the root directory of the ZFS file system.

```
# mdb /tmp/dnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 3*200::print -a -t zfs`dnode_phys_t
{
   600 uint8_t dn_type = 0x14  <-- DMU_OT_DIRECTORY_CONTENTS from before
... <-- output omitted
   604 uint8_t dn_bonustype = 0x11  <-- DMU_OT_ZNODE (from dmu.h)
... <-- output omitted
   640 blkptr_t [1] dn_blkptr = [
      {
         640 dva_t [3] blk_dva = [
            {
               640 uint64_t [2] dva_word = [ 0x1, 0x51088 ]
            }
... <-- output omitted
>
```

This dnode_phys_t has both blkptr_t and bonus buffer. The bonus buffer is a znode_phys_t. Let’s take a look.

```
> 6c0::print -a -t zfs`znode_phys_t
{
   6c0 uint64_t [2] zp_atime = [ 0x47c08f1b, 0x31e41b57 ]
... <-- output omitted, ownership, other time stamps, size, etc.
}
> 6c0/Y
0x6c0:          2008 Feb 23 22:24:43    <-- when the fs was last accessed(?)
```

Now, let’s look at the blkptr_t.

```
> 640::blkptr
DVA[0]: vdev_id 0 / a211000
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[0]: :0:a211000:200:d
DVA[1]: vdev_id 0 / 1c003a400
```

```
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[1]: :0:1c003a400:200:d
LSIZE:  600                   PSIZE: 200
ENDIAN: LITTLE                        TYPE:  ZFS directory
BIRTH:  46b6          LEVEL: 0     FILL:  100000000
CKFUNC: fletcher4                COMP:  lzjb
CKSUM:  172e5bed24:7e94f1c3dba:179eeebee7275:325d97fad3423c
> $q
#
```

This blkptr_t is for a ZFS directory (should be the root directory of the file system). We'll use zdb to dump this. It is 0x200 bytes on the disk, and the decompressed size is 0x600 bytes.

```
# zdb -R laciedisk:c4t0d0p1:a211000:200:d,lzjb,600 2> /tmp/zfs_root_directory
Found vdev: /dev/dsk/c4t0d0p1
#
```

The ZFS directory blkptr_t is a ZAP object. We'll look at the first 64-bits to determine the type of ZAP.

```
# mdb /tmp/zfs_root_directory
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0/J
0:            8000000000000003 <-- this is a microzap
```

Ok. It's a microzap. Let's take a look.

```
> 0::print -a -t zfs`mzap_phys_t
{
   0 uint64_t mz_block_type = 0x8000000000000003
 <-- output omitted
   40 mzap_ent_phys_t [1] mz_chunk = [
      {
         40 uint64_t mze_value = 0x8000000000000004  <-- the object id (inumber equivalent)
         48 uint32_t mze_cd = 0
         4c uint16_t mze_pad = 0
         4e char [50] mze_name = [ "foo" ]  <-- file in the root directory of the file system
      }
   ]
}
```

Let's dump all of the mzap_ent_phys_t in the block. The block is 0x600 bytes large, and the size of an mzap_ent_phys_t is 0x40 bytes. We start 0x40 bytes into the block and dump the remainder of the block as mzap_ent_phys_t structures. Each entry in this microzap is a directory entry.

```
> ::sizeof zfs`mzap_ent_phys_t
sizeof (zfs`mzap_ent_phys_t) = 0x40
> 40,(600-40)%40::print -a -t zfs`mzap_ent_phys_t
{
   40 uint64_t mze_value = 0x8000000000000004  <-- low-order bits are object id numbers
                                  <-- 8 at the high bits is type info
   48 uint32_t mze_cd = 0
   4c uint16_t mze_pad = 0
```

```
   4e char [50] mze_name = [ “foo” ]
}
... <-- output omitted
{
   440 uint64_t mze_value = 0x8000000000000015  <-- ls -i should show 21 for this file
   448 uint32_t mze_cd = 0
   44c uint16_t mze_pad = 0
   44e char [50] mze_name = [ “words” ]  <-- here’s the file we want
}
{
   480 uint64_t mze_value = 0  <-- unused
... <-- output omitted
> $q
#
```

Now, we go back to the dnode_phys_t array for object id 0x15. Note that if the object id is >= 0x20, we would need to look at other (indirect) blkptr_t to get the dnode_phys_t we want. Here, because the file system does not have many files, everything “fits” so that we don’t have to go through too many levels of indirection.

Let’s say, for example, that the object id is 0x99f1d, and the number of levels of indirect blocks for the DMU_OT_DNODE (dn_type = 0xa) for the DSL dataset is 7, as in the above. To get the index into the indirect blkptr_t array at each level requires a little arithmetic. Assuming each block of dnode_phys_t contains 0x20 dnodes, the index into the array is simply 0x99f1d&0x1f (= 0x1d). Also assuming that a block of indirect blkptr_t’s contains 0x80 entries (as in the above case), the index into the level 1 indirect blkptr_t is (0x99f1d>>5)&0x7f (= 0x78). The second level index is (0x99f1d>>0xc)&0x7f (= 0x19). The third level is (0x99f1d>>0x13)&0x7f (= 0x1). The fourth level is (0x99f1d>>1a)&0x7f (= 0x0). The fifth level is (0x99f1d>>21)&0x7f (= 0x0). The sixth level index is (0x99f1d>>28)&7f (= 0x0). So, at Level 6, 5, and 4, you use the 0th blkptr_t. At Level 3, the 1th (second) blkptr_t. At level 2, the 0x78th blkptr_t. And at level 0 (the array of dnode_phys_t), the 0x1d entry. (I have done this with the modified mdb/zdb, but for this paper, it is left as an exercise for the reader. For the data walk we do here, we’ll use the nice small object id because it avoids the arithmetic. We basically take the 0th entry of all the indirect blocks and the 0x15 entry from the dnode_phys_t array at the end).

The following diagram shows the traversal of indirect blocks for object id 0x99f1d.

```
# mdb /tmp/dnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 15*200::print -a -t zfs`dnode_phys_t
{
   2a00 uint8_t dn_type = 0x13  <-- DMU_OT_PLAIN_FILE_CONTENTS
   2a01 uint8_t dn_indblkshift = 0xe
   2a02 uint8_t dn_nlevels = 0x2  <-- one layer of indirect blocks
   2a03 uint8_t dn_nblkptr = 0x1
   2a04 uint8_t dn_bonustype = 0x11  <-- DMU_OT_ZNODE (for “words” file)
... <- output omitted
   2a40 blkptr_t [1] dn_blkptr = [
      {
         2a40 dva_t [3] blk_dva = [
```

```
            {
                2a40 uint64_t [2] dva_word = [ 0x2, 0x51054 ]
            }
... <-- output omitted
}
```

The dn_bonus buffer contains file system attributes for the file in a znode_phys_t (time stamps, ownership, size, etc.). We just want the data, so let's dump the first level indirect blkptr_t.

```
> 2a40::blkptr
DVA[0]: vdev_id 0 / a20a800
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[0]: :0:a20a800:400:id
DVA[1]: vdev_id 0 / 1c003a800
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 40000000000
DVA[1]: :0:1c003a800:400:id
LSIZE:  4000                    PSIZE: 400
ENDIAN: LITTLE                          TYPE:  ZFS plain file
BIRTH:  46b6            LEVEL: 1     FILL:  200000000
CKFUNC: fletcher4                COMP:  lzjb
CKSUM:  5d33a847e1:3e4f850ab0b0:16415a51e8464d:59b54146c060c42
> $q
#
```

And we'll use zdb to get the block.

```
# zdb -R laciedisk:c4t0d0p1:a20a800:400:d,lzjb,4000 2> /tmp/words_indirect
Found vdev: /dev/dsk/c4t0d0p1
#
```

And back into mdb to display it.

```
# mdb /tmp/words_indirect
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 4000%80=K
            80 <-- There is space for 0x80 blkptr_t in this indirect block.
> 0::blkptr
DVA[0]: vdev_id 0 / a220000
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 2000000000000
DVA[0]: :0:a220000:20000:d
LSIZE:  20000                   PSIZE: 20000
ENDIAN: LITTLE                          TYPE:  ZFS plain file
BIRTH:  46b6            LEVEL: 0     FILL:  100000000
CKFUNC: fletcher2                COMP:  uncompressed
CKSUM:  281ad9d864b9dc57:79fe4143faf3e2b7:5e064a117c12a92e:5b788125e084c6b2
>
> $q
#
```

This is the blkptr_t for a 0x20000 byte (128KB) block of uncompressed file data. The block should contain the first 128KB of the copy of /usr/dict/words. We'll use zdb to dump it out. Note that here, there is only one entry in

the DVA in use, I.e., only 1 ditto block. This is the default for file data. In SXCE and Solaris 10 Update 6(?), the "zfs set copies" command will allow the number of ditto blocks to be set for regular file data.

```
# zdb -R laciedisk:c4t0d0p1:a220000:20000:r
Found vdev: /dev/dsk/c4t0d0p1
10th
1st
2nd
3rd
4th
5th
6th
7th
8th
9th
a
AAA
AAAS
Aarhus
Aaron
AAU
ABA
Ababa
aback
... <-- output omitted
```

And here is the first 128KB of the copy of the /usr/dict/words file.

Future Work

It would be very nice to have a version of mdb with zdb integrated into it. I have tried various combinations (for instance, including a decompress dcmd within mdb), but I took the easiest route.

Examination of snapshots, ZIL (ZFS Intent Log), and various other features of ZFS is possible using the same techniques.

A similar walk through of the data structures in kernel memory would be nice. This is simpler than the walk-thru here (in-memory, the meta-data is not compressed, and one can use mdb to do the walk).

Conclusions

I have found that in writing this paper, access to the source code, and to the ZFS On-Disk Specification paper have proven to be invaluable in understanding the disk layout of ZFS. The ZFS On-Disk Specification paper really does cover everything about the on-disk layout. I found that the ability to "walk through" the data helped me to understand what the on-disk specification paper is saying. The layout is complex, but understandable.

Examination of the block numbers used to contain the meta-data shows that the metadata for everything except the indirect blocks for the file itself resides in the first 128K bytes following the disk labels. It is possible that this metadata is retrieved by a single 128K read when the file system is first accessed. The following table shows the disk locations, sizes, and data for all of the data structures examined in the disk walk. Physical size is size on disk, logical size is size after decompression. Blocks with matching physical and logical sizes are not compressed.

On-Disk Location	Physical Size	Logical Size	Description
0x4e00	0x200	0x400	MOS objset_phys_t
0x5000	0x800	0x4000	MOS dnode_phys_t array
0x200	0x200	0x200	ZAP Object Directory
0x4c00	0x200	0x400	Root Dataset objset_phys_t
0x4800	0x200	0x4000	Indirect block Level 6 for File System Objects
0x4400	0x400	0x4000	" " " 5 " " " "
0x4000	0x400	0x4000	" " " 4 " " " "
0x2000	0x400	0x4000	" " " 3 " " " "
0x1800	0x400	0x4000	" " " 2 " " " "
0x1400	0x400	0x4000	" " " 1 " " " "
0x3600	0xa00	0x4000	dnode_phys_t array for File System Objects
0xc600	0x200	0x200	ZAP Master Node Object
0x1000	0x200	0x600	ZFS Root Directory ZAP Object
0xa20a800	0x400	0x4000	Indirect Blocks for File Data
0xa220000	0x20000	0x20000	File Data

On-Disk Locations Examined During Disk Data Walk

Note that except for 2 ZAP objects and the actual file data, everything is compressed.

References

Empowering OpenSolaris* Developers on Intel Hardware

David C. Stewart
Intel Corporation
2111 NE 25th Ave
MS: JF1-255
Hillsboro, OR
USA
david.c.stewart@intel.com

1 Abstract

Abstract: Open source is about more than being able to get software for free. It is about empowering developers to innovate, contribute, and build community for everyone's benefit. It has been over a year since we announced the historic Intel/Sun collaboration agreement, and there have been some notable accomplishments and a few frustrations. Intel is contributing development work upstream in areas such as performance improvement, developer tools, power use, new technologies, and drivers. There are also some exciting developments for Intel's next generation processors, code named Nehalem. Although there is good progress in this development work, there are also some challenges in development in the current community framework. Intel is happy to contribute to the project to address these issues.

2 Introduction

After a history of having a somewhat poor working relationship, Sun and Intel surprised the technology world by announcing a major partnership and collaboration in January of 2007. Sun would become a customer of Intel's microprocessors and platforms, and Intel would work collaboratively with Sun on porting and optimization of Solaris and Java for Intel hardware. For the Solaris operating system work, Intel chose to work in the open source OpenSolaris* project. Selected optimizations would also be back-ported to Solaris 10 and provided in updates. This enables users of both Solaris 10 and OpenSolaris to benefit from the collaborative effort to optimize Solaris for users of Intel hardware.

Intel chose to do its work through the open source project because of a strong belief at Intel that important forces of innovation and progress are unleashed when a vibrant community gathers around any given software project. In effect, Intel is in the role of empowering those forces of development progress. This empowerment stems from three major themes:

1. **Intel empowers developers and users through hardware and software innovation**. Intel is driving a relentless pace of hardware evolution that provides great value for users of OpenSolaris. The “Tick Tock” model[1][1], described below, is providing a predictable roadmap for technology consumers for the future. Perhaps less well known are the contributions that Intel’s Software and Solutions Group is making in all layers of the software ecosystem, to make sure that developers and users can take advantage of this hardware innovation.
2. **Intel empowers developers and users of OpenSolaris* through our targeted development work**. Intel engineers are working in a variety of areas in OpenSolaris to optimize the experience of using Intel processors in servers, workstations, desktops, and mobile computers. This work spans the areas of performance optimization, power optimization, I/O acceleration, virtualization, predictive self healing, and driver support for Intel silicon.

[1] http://www.intel.com/technology/magazine/computing/cadence-1006.htm

3. **The OpenSolaris community itself is an empowering force to unleash innovation**. By working in a collaborative community, there is an opportunity for synergy between developers across barriers of language and organizational/cultural lines. Though the OpenSolaris community is relatively young as an open source effort, there has already been considerable progress to engage developers in the project. However, there continue to be barriers to further growth of the community that we can and should address.

3 Intel empowers developers and users through hardware and software innovation

In the technology industry, things are in constant motion. This requires nimble execution to achieve and retain leadership. To frame our technology roadmap and to drive that constant forward motion, Intel began describing our roadmap as the "Tick Tock" model.

3.1 The "Tick Tock" Hardware Innovation Model

- **Cadence** – Intel has been operating on a two-year cadence for the complete tick tock cycle. The cycle alternates with first a "tick" with a process technology shift, followed by a "tock", which is a microarchitecture shift. This is like the ticking of a great clock, and thus the name "tick tock".
- **Tick** – One half of the model is moving our manufacturing efforts to the next silicon process technology. Process technology advances typically allow semiconductor features on a silicon die to be reliably manufactured at a smaller scale or size than the previous generation. Thus, a tick usually results in a reduction of the processor "line size" or the distance between conductive wires on the silicon die. This reduction in line size usually results in reduced processor power use. To ensure a smooth transition to the new process technology, we keep the lead processor microarchitecture for this technology roughly the same as on the previous process technology. Often this "shrink" of the design is also accompanied by some additional features and new machine instructions.
- **Tock** – Once the new process technology is in place, the other half of the model is to transition to a new microarchitecture. This usually includes many new features, new instructions, and technologies. Transition to the new microarchitecture in the "tock" phase is simplified because the process technology is maintained as the previous "tick" phase.

3.2 The "Tick Tock" Model in Action

In 2005, Intel was selling the Pentium® D processor for desktops, the Intel® Xeon® processor architecture for servers, and the Intel® Core™ architecture for mobile computers. These products were built using the 65nm process technology, which means that the line size, or distance between conductive wires, on the silicon die were 65 nanometers apart.

Then in 2006, we introduced the Intel® Core™2 Duo processor, which was based on a microarchitecture with the codename "Merom". This was a radically new microarchitecture but was delivered on the same 65nm process technology as the previous generation. This represents the "Tock".

Then in 2007, Intel launched leadership 45nm process technology, which featured some unique advances in silicon manufacturing. To fill this new 45nm process capability, we launched our Core 2 products based on the codename "Penryn". The Penryn processor was essentially a shrink of the Merome processor design, with added instructions, such as SSE 4.1 (Streaming SIMD Extensions). This was the most recent "Tick."

In 2008, Intel anticipates that we will launch a new processor which is code named “Nehalem”.This is again a totally new microarchitecture, and will introduce both new instructions (SSE4.2), new platform technologies (extensions to Virtualization Technology and I/O Acceleration Technology) and a new platform architecture (use of Quick Path Interconnect and embedded memory controllers, versus a front-side bus architecture). All of this will be introduced on the now ramped-up 45nm process technology. This is the “tock” to complete this two-year cycle.

In future, we anticipate another tick-tock cycle like the previous two. In this case, 32nm process technology transition “tick” is being prepared with the codename “Westmere” processor, and a “tock” is planned with a processor codenamed “Sandy Bridge.” To the best of our ability, we will continue in this pattern.

This pattern of innovation delivery provides the industry with a sense of predictability that helps in technology and business planning. It also ensures a relentless pace of innovation for developers and users alike.

3.3 Software Innovation in the Software and Solutions Group

In addition to the tick tock model for hardware, Intel’s Software and Solutions Group (SSG) works throughout the software stack to match this hardware innovation pace with software innovation.

Here are some examples of the software efforts Intel is making at these various levels:

- **Platform firmware –** Intel has developed a replacement for the traditional PC BIOS called the Extensible Firmware Interface, or EFI Framework. This is an open source specification and implementation of a complete system BIOS, written predominantly in the C programming language rather than assembly language, increasing the modularity and portability of system firmware, and reducing cost of development.
- **Virtualization Software –** Intel has taken a strong contribution role in both open source (Xen, xVM, KVM) and traditional model virtualization (VMWare, Microsoft). This work ensures that this virtualization layer is tuned optimally for Intel platforms and ensures that features, such as Virtualization Technology (VT), are supported.
- **Operating Systems –** Intel has taken a leadership role in Linux and OpenSolaris, as well as operating systems from Microsoft and Apple. This ensures that these operating systems make best use of the many features of our new processors and that issues are addressed quickly.
- **Development tools –** Intel’s performance leading compiler and developer tools suite for C, C++, and FORTRAN make the developer’s job easier by analyzing performance, identifying performance bottlenecks, and generating optimal code for our processors. To this suite we add the open source Threading Building Blocks, which enable users to select from a suite of predesigned C++ components to make best use of multi-core systems. We also contribute to development environments, such as Eclipse and NetBeans*. Finally, we are contributing to Sun’s Java* and Sun Studio, to ensure that these products are optimized for Intel processors.
- **Middleware and applications –** Intel makes performance contributions to many of the top middleware and application products in use. Examples span the range of databases, such as Oracle and MySQL*, application suites from vendors, such as SAP and BEA, and applications suites, such as OpenOffice.

These are just a few examples of the impact that Intel is making, across the full spectrum of software that developers touch, to make use of our systems and ensure that the experience is optimal.

Open source software is an important element to much of the software work that Intel does. As a result, Intel is considered a leader in open source contribution. Some examples of this include:

- **Linux** – Intel is the number 4 largest contributor to the Linux kernel. Through projects, such as lesswatts.org and intellinuxwireless.org, Intel takes a leadership role to help the Linux community make best use of Intel's products.
- **Moblin,org** – Intel is also providing optimizations to Linux specifically for the new class of Intel® Atom™ processor based products.
- **Xen** – Open source virtualization has taken on a surprisingly rapid trend of being used in data centers. Intel is the number 2 largest contributor to the Xen project (after XenSource).
- **Harmony** – Intel has made considerable contributions to the open source Harmony JVM, which is part of the Apache project.
- **EFI Framework** – The Extensible Firmware Interface (EFI) is the open source replacement for traditional system BIOS which has been broadly adopted by system manufacturers. Intel created and nurtures the EFI open source project.
- **Threaded Building Blocks (TBB) –** Intel has released its TBB suite under open sources, and collaborated with Sun to provide a port to Solaris* for TBB.
- **OpenSolaris –** Intel is the number 2 largest contributor to the OpenSolaris kernel. Some of the contributions are listed below.

This commitment to open source innovation has unleashed developers to give their best on the open source platform, and ensured that their solutions will be optimized for Intel products.

4 Intel Empowers Developers and Users of OpenSolaris through our Targeted Development Work

Intel's contribution to OpenSolaris is designed to make the Intel Xeon processor and Intel Architecture based systems the best choice to run OpenSolaris, and thus Solaris. Some recent examples of code, which has been contributed by Intel or included Intel consulting, include:

- **Microcode update** – enhancements to OpenSolaris and Solaris to accept and load Intel processor microcode field updates.
- **CPUID for Intel Core2 and Nehalem** – properly decoding the CPUID instruction allows all of the unique processor features to be interpreted by the OS to enable the proper optimizations of memory placement and thread scheduling.
- **Libc optimizations for Intel Core2** – using SSE 2 and 3 instructions when the system is running on a processor that will correctly interpret them for memory operations in libc.
- **NUMA support (SLIT/SRAT) for Nehalem** – proper interpretation of ACPI data which lays out the NUMA configuration of a system.
- **Performance counter support for Core2** – ensure that performance events can be interpreted correctly for performance tools such as Sun Studio
- **SSE 2, 3, Supplemental SSE3, SSE 4.1, 4.2** – compiler and assembler support for new Streaming SIMD Extensions.
- **P-state power optimizations** – Support for Demand-Based Switching (DBS) in Intel's platforms ensures good active power management
- **4965 Wireless LAN support** – a new driver for current Intel wireless components
- **Zoar and Oplin NIC support** – drivers for Intel wired NICs
- **Graphics drivers for Intel® Centrino® Pro, Intel Centrino with vPro™ technology** – desktop and mobile graphics drivers
- **IO Acceleration Technology** – IOAT support reduces CPU overhead as network traffic continues to speed up.
- **Virtualization Technology in xVM** – various extensions to VT have been contributed to Xen and xVM Server.

This collection of contributions span the whole range of technologies that Intel is building into server, desktop, and mobile systems.

Performance optimization – Performance is really at the core of Intel's design thinking. When planning a new generation of processors, the processor architects will examine instruction traces for various classes of applications and identify common programming operations which can be consolidated into single instructions. These enhancements speed up applications, which make use of these new instructions, and the compilers and assemblers in Sun Studio and GCC are enhanced to support these new instructions. However, it typically requires a programmer to analyze their code to identify places where these new instructions will benefit them. To make it simpler to take advantage of these new instructions in limited cases, Intel has optimized various libc functions, such as memset() and memcpy(), to make use of SSE2 and SSE3 instructions.

Figure 1 is an example of the speedups experienced when measuring the new code with micro benchmarks.

This is an example of some in-process work with the memcpy() routine, showing a comparison of the old code (which is the baseline) versus using SSE2 and SSE3 instructions. The chart is a comparison of large unaligned memory copies. The X axis shows the size of the copies in bytes and the Y axis is a percentage speedup. As can be seen, the SSE3 version looks very attractive, with large unaligned copies versus the SSE2 version. In fact, one size shows a 350% speedup in one case.

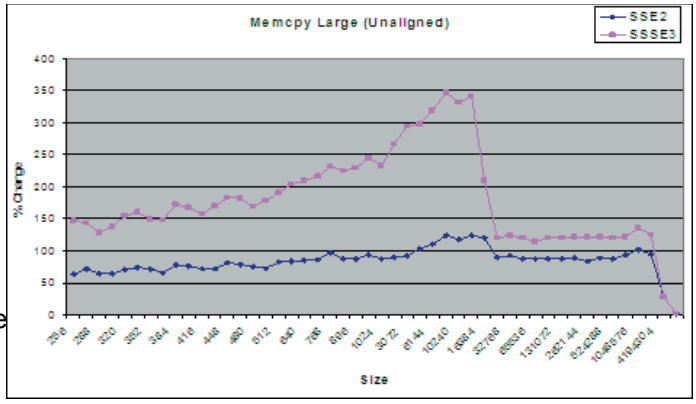


Figure 1: Example of performance speed up for memcpy() using SSE instructions

Of course, most applications don't spend all of their time in memcpy(), so one should not expect applications to speed up three times. But some applications do spend significant time in these routines. These are available in Nevada, build 87 and later for 64-bit applications.

In addition to memcpy() and memset(), there is work in progress to optimize string functions in libc, such as strcmp() and strlen(). Figure 2 is another example of in-progress work, not the final version, but an intermediate result on strlen().

Again, one shouldn't expect these kinds of speedups on micro benchmarks to be realized in full on all applications, but some improvement is likely, depending on the application. Fortunately, these libc speedups do not require an analysis or recoding of the application or even a recompile. If the application loads libc as a shared object, it will take advantage of these new routines at run time. Note again that these improvements are available for 64 bit applications only.

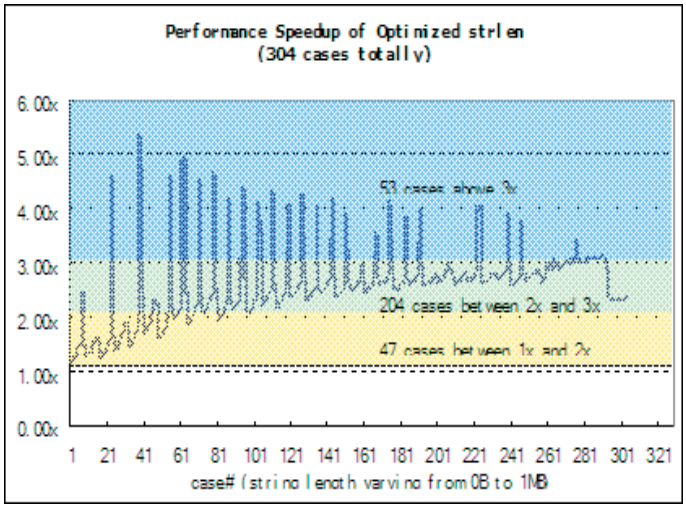


Figure 2: Example of strlen() speed up

Those who develop in Java or Ruby or other web stack applications might not make use of libc routines. To address this, Intel is also working on recoding kernel functions, such as bcopy(), kcopy(), bzero() and kzero(). This work is preliminary at this time but may also yield impressive speedups.

Power savings – There is considerable focus at Intel to design products which can save as much power as possible. This is because of the growing cost of energy in the data center and a growing

awareness of the connection between carbon emissions and global climate change. The upward trend of power consumption in data centers has been documented by the US Environmental Protection Agency in a recent study. (http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final1.pdf). In 2006, power consumed by data centers in the US exceeded the power used by color television sets, and represented 1.5% of power consumed in the country (61B kWh), which represents a doubling over the past 5 years. The report projected that this consumption will double again by 2011.

Intel's processors do active power management using ACPI P-states when the system's CPU is non-idle, and do idle power management using ACPI C-states when the CPU is unoccupied.

Active power management using P-states. The goal of active power management is to ensure that the full power and performance of the system is available when the CPU use is high, but when the CPU is less busy and utse is lower, the power is lowered to enable power savings. Intel processors can change P-states very quickly, resulting in the maximum power savings with the maximum responsiveness to usage spikes.

Idle power management using C-states. Under normal running conditions, the CPU is in C0 state. When the CPU is idle, the maximum power can be saved by putting the CPU into a lower C-state. When the C-state is lowered, various parts of the CPU are turned off, saving power. Lower C-states typically take longer to transition back into C0 state.

The current P-state and C-state are under the control of the operating system, so the OS needs to be modified to ensure that the maximum power is saved.

Currently, OpenSolaris and Solaris have initial support for P-states and C-states. There is development under way to implement support for deeper C-states and to link the P-state management framework with the dispatcher, to ensure that scheduling is synchronized with power management.

Note that when the CPU is idle, the most power is saved when the CPU is put into a low C-state. However, software which polls for work periodically will tend to wake up the CPU, preventing it from being idle as long. There are many examples of polling behavior, in the operating system, which must be eliminated. But many software applications need to have polling behavior addressed as well.

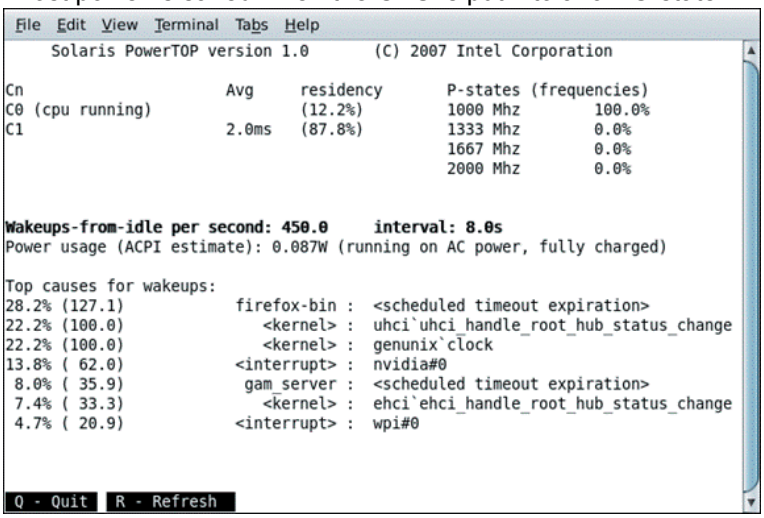


Figure 3: Screen shot of PowerTOP application for OpenSolaris

To assist with determining which software is guilty of polling behavior and of waking up the CPU when it doesn't need to, Intel developed a tool called PowerTOP. The tool was originally developed for Linux, was ported to Solaris, and is available for download. (http://www.opensolaris.org/os/project/tesla/work/powertop). Figure 3 shows a screen shot of the 1.0 version of PowerTOP.

Note that the current version of PowerTOP will show the C-states and P-states and how much time is being spent in each state, and will show the activity of what software is causing the CPU to wake up. In this example, on an otherwise idle laptop, the CPU is being woken up 450 times per second, and

the top culprits are Firefox*, the USB driver polling for a device insert (uhci), the system clock, and various driver interrupts. These are all candidates for moving to event-driven behavior.

But once we have good P-state and C-state support in the OS, and software polling behavior is eliminated, there are more opportunities to eliminate power wasting behavior. Intel's labs monitor power use in a variety of areas, to benchmark where power is being used and which software needs to be modified, to address power wasting.

Virtualization – Another area of Intel innovation is supporting virtualization. Current data centers often have many systems running far below their maximum capacity. This is often due to enterprise buying patterns which encourage buying sufficient server capacity for multiple years of forecasted growth. In fact, on average, it is likely that many servers will be used no more than 20%. Thus, many data center managers are looking for ways to reduce cost and power/cooling expense by using virtualization to consolidate many workloads onto a single server. Virtualization has also been used to implement fail-over scenarios or isolate development environments from production environments on the same server.

Intel has been shipping processors which have Virtualization Technology (VT-x), which supports better virtualization. VT-x introduces a new operating mode for the processor, by providing the VMENTER and VMEXIT instructions. This mode allows a virtualized guest operating system to run its privileged mode code (also called ring 0 code) in the ring 0, rather than run it at a non-privileged mode, such as ring 3, which is required without technology like VT.

Open source virtualization, based on the Xen project, has been a focus of Intel's effort to apply the benefits of VT. In fact, Intel is the number 2 largest contributor to Xen, adding VT support, interoperability, and performance optimizations. Also, Intel is the number 2 contributor to the OpenSolaris project. Intel has been collaborating with the community to optimize OpenSolaris for the Intel Xeon processor. By bringing these two together in the xVM Server project, the result is a virtualization product solution that is ideal for the Intel Xeon processor.

Predictive Self Healing – A key enterprise differentiator for Solaris is the Predictive Self Healing feature, provided by the Fault Management Architecture. This feature reduces costs for businesses running Solaris by clearly identifying failed components and giving administrators direction on which field replaceable unit should be replaced.

One of the advances in OpenSolaris and in Solaris 10 updates has been the addition of support for the Intel Xeon 5000 and 7000 series processor-based systems. These are the volume 1, 2, and 4 socket servers. The generic support for these platforms ensures that failures will be correctly identified, reported, and managed, within the FMA framework.

In addition, Intel has been doing fault injection in our lab, to exercise the FMA code under accurate failure conditions. Fault injection is a technique Intel uses to simulate a hardware failure by stimulating a special system BIOS in a particular way, to report a hardware failure to the operating system. Since FMA provides code which improves the reliability of the system, fault injection is a necessary validation step, to ensure that actual failures will be reported correctly.

Memory failures are often a source of frustration. As rare as they might be, an uncorrectable error in a memory DIMM would create a problem in a large data center, unless the physical location of the failed component can be known right away. For failing memory DIMMs, Intel has added code to correctly identify which physical memory slot in a system maps to the address of the failed memory. This lowers the cost and frustration of locating the failed memory by trial and error or through diagnostics.

Developer Tools – The Sun Studio team has done very impressive work to optimize the C/C++ compiler to generate code optimized for the Intel Core microarchitecture. This includes, for example, instruction selection, code scheduling, alignment, and prefetch tuning. In addition, programmers can now make use of Intel's latest Streaming SIMD Extensions (SSE) instructions to optimize vector and

floating point intensive applications. Finally, Sun Studio's Performance Profiler/Analyzer has been enhanced with support for the unique performance event counters in the Intel Core2 processors. The result of these enhancements is significant improvement in standard integer and floating point code benchmarks.

Recently announced at JavaOne, Intel's Threading Building Blocks (TBB) is now available with Solaris support. (threadingbuildingblocks.org) TBB is an open-source C++ runtime library from Intel, that abstracts the low-level threading details necessary for optimal for multi-core performance. TBB makes it easier to develop threaded applications for newer multi-core processors, including Quad-Core Intel Xeon processors.

Another tool available for developers, moving from a SPARC-based deployment to a Solaris and Intel Xeon processor-based system, is QuickTransit from Transitive Corporation (www.transitive.com). QuickTransit enables a SPARC-based binary to be run unchanged on an Intel Xeon processor-based system, often with better performance than was experienced on the legacy hardware. This is ideal for situations in which the source code for an application is perhaps not available or the resources for a porting project are not available.

Device Support – If OpenSolaris is to be adopted on Intel hardware, it is ideal if developers can have good support on desktop computers. Outside of the processor itself, Intel provides market-leading components for laptop and desktop computers, in areas such as wireless LAN, wired LAN, and graphics. Intel is working to enhance all of these areas, and work with the community to bring drivers for these products into Solaris and OpenSolaris.

For example, in the wireless LAN product space, Intel has worked with the community to bring a new 4965 wireless driver for Solaris to developers. This driver provides 802.11bg support today. For the future, 802.11n support is being developed actively, which should provide accessibility to key wireless performance, and distance features from this draft standard.

For the breadth of Intel hardware at the server, mobile, and desktop levels, Intel is fully engaged to enable and optimize Solaris and OpenSolaris, and make sure these become the best platforms for running Solaris.

5 The OpenSolaris Community is a Force for Unleashing Innovation

As a relatively new open source community, OpenSolaris has made great strides. The UNIX source code, on which Solaris was originally based, dates back to the 1970s, and has been enhanced greatly over the years by Sun and partners. The very act of providing all of this vast quantity of software as open source is a remarkable accomplishment.

Intel is a relatively new participant in the OpenSolaris community, dating 16 to 17 months as of the date of this conference. The experience of working with the OpenSolaris community has been very positive overall. Sun in particular has some very experienced OS experts working on OpenSolaris, and Intel has assigned some strong experts as well, some of whom have 20+ years of OS development experience. The community as a whole have received and welcomed Intel's contributions to OpenSolaris. The author has extensive experience working with a variety of communities and companies, and the OpenSolaris experience has been some of the best the author has experienced, both in terms of technical excellence and positive personal regard.

Of course, any community development experience has its share of frustrations and learning, particularly with integrating new contributors. Intel developers have experienced some of the challenges of getting a case through the Architecture Review Committee (ARC) process. The result of this learning process has been improved contributions from Intel.

There are really three main advantages for open source:

- **The ability for programmers to see clearly how functions work.** The experience of programming on an operating system or application stack where the underlying source code is available, is much more productive than if the source is not available. Without source code, developers must depend on manual pages, documents, and sample code, all of which may be out of date or harder to understand than the source itself. The author's own experience as a manager helping engineers program on a UNIX-like system, without the source of system calls or library calls being available, shows this major advantage of open source.
- **Beyond making programmers more productive, some open source projects enable developers to enhance the code that they are using.** If a function is not available, for example the support for a particular hardware device or optimized alignment of memory, the system can be changed through the use of open source. Another example would be to make a boot sequence less "noisy" in terms of informative messages or to fix bugs. The ability to use open source to build an improved product with new or changed features is a major value to some open source projects.
- **Finally, some open source projects allow innovations developed to be fed back to maintainers of the project upstream, to raise the overall value of the project as it is experienced by all users.** Clearly work done on a bug fix or a new feature will have greater impact if it is broadly available. Of course, this now introduces the added burden of assuring that the overall architecture, quality, reliability, and compatibility of the change matches the standards set in the project. When there are differences of opinion between innovators and maintainers, there is often the pressure to "fork" projects or create derivative distributions.

The act of making the OpenSolaris code freely available certainly accomplishes the first advantage. Programming on OpenSolaris or Solaris should now be considerably easier than before the code was released as open source.

The ability to rebuild a functional OpenSolaris provides a second major advantage as well. Enhancements and bug fixes are now available to users of OpenSolaris and are now instantly available for programmers who are able to rebuild the system, assuming that the procedures for building the operating system reliably produce a working binary.

The final advantage, the ability to drive innovations upstream in OpenSolaris has had a reputation for being more challenging in OpenSolaris than in other open source projects. This deserves some analysis, since it is a major criticism of the project.

- OpenSolaris is made up of many "consolidations" which include the Gnome desktop, the Operating System and Networking (ON), and the xVM program, to name a few. Some of these consolidations or sub-projects operate under rules which allow people in the broader community to do code put backs into their gates. So, in this sense, parts of OpenSolaris operate with the ability to have maintainers in the broader community.
- The ON consolidation, which includes the kernel, still requires a Sun employee to do code put backs. This is because, at least at the time of this writing, there is a mechanism in place enforcing this requirement, but it is being revised.
- Due to the high compatibility and reliability standards set for Solaris, there is a strong desire in the community to balance change more strongly in favor of review versus pure innovation. This means that enhancements which require a developer- or user-visible change must be approved by an ARC (Architecture Review Committee). This makes it appear that innovation and change are less welcome in OpenSolaris. However, it would seem that any open source project must and does maintain review and approval standards unique to the project to enable the project to remain viable.
- The general style of kernel development in the ON consolidation is different from the style currently practiced in Linux. This is perhaps the most significant difference between these projects, and deserves some additional scrutiny.

For the Linux kernel project, the general flow works as follows:

- **First thing delivered: code.** When a new project or subsystem is proposed, the most success comes from a mail message which begins with the word “diff”. Code is given attention first and foremost. In addition, this code is delivered as a collection of self-consistent patches. A larger set of changes may be delivered as a collection of patches, each of which provides a piece of functionality, and may be understood on its own.
- **Design documents or architecture documents are delivered in code comments or email discussions**. Often these are ignored or may not be consistent with the code, since the code itself has the most importance.
- **Email feedback, discussion, argument.** Once code appears in an email message, there ensues review discussion, clarification, and changes. This gives the sense that the Linux community has a vibrant and active community of contributors, some of whom may be contributing to significant architecture questions or perhaps only to the coding style used.
- **Maintainer either approves or not.** Ultimately, the change is either accepted or rejected by a maintainer. This step could be preceded by many other approval steps including Linus himself and the Operating Systems Vendors.

The Linux kernel development model can be thought of as an *incremental release* model, where functionality is developed and delivered as a set of mutually exclusive and independently functional patches. This is often referred to as a "release early and release often" model, and the code is often not very well tested when it is first made visible.

For ON kernel development, the flow is quite different:

- **First thing delivered: Product requirements, architecture specs, and design documents.** These documents are sometimes freely available on the opensolaris.org website, although for ON they are rarely available. When the documents are released, it ensures that there is opportunity for review in the community. However, if these designs were done by Sun engineers, the documents have received considerable review by Sun, prior to being released openly.
- **Code is developed and delivered, after being debugged, unit tested, and performance tested.** The usual notification in the community that code has been developed is that notice goes out that a code put back has occurred. Developers in the broader community are invited to review the code, but the onus is on them to check the code out of the repository and hope they can identify the exact changes.
- **ARC case.** If a change may have user-visible impact, then an ARC case will require review by the Architecture Review Committee.
- **Mail sent out giving notification of the put back.**

In contrast to the Linux model, this is more of a *big bang release*, where the design, code, tests, and documents are all available and at high quality. This model is due to the pedigree of Solaris as an operating system product, rather than a project. Code innovations which are done in the community, are assumed to be part of this product, rather than community innovations which may someday make it out into a product.

Given that OpenSolaris kernel development flow is in fact the opposite of what is experienced in the Linux community, there will likely be a barrier to experienced Linux kernel developers transitioning to being OpenSolaris kernel developers.

However, even though the OpenSolaris development model seems to be less prone to vibrant online discussion and innovation, in fact the delivery of finished and supported functionality to the market is

often more rapid in OpenSolaris, because there are fewer decision points and decision makers than in the Linux model.

The best hope for a significant growth in the number of OpenSolaris kernel contributors would be a fundamental change in the development process to mirror the Linux process, or a realization that new kernel developers will grow from university and other settings, without the earlier experience with and preference for the Linux model. Of course, it may not be the right goal for OpenSolaris to have a large number of kernel contributors

There are some other barriers for attracting developers accustomed to the Linux model, like the different license model for OpenSolaris. It is also sometimes hard to know who to talk with when you are looking for an expert in a particular area. This is because the maintainers are not well known or as public as they are in other open source projects. These factors will also tend to keep the community of ON developers small.

One possible vision going forward would be to address these barriers to entry:

- make it easier to for developers to figure out how to contribute
- make the identity of the contributor much more obvious when change sets are announced
- identify the maintainers better

The OpenSolaris community can be quite welcoming and has a lot of very smart people in it. We can make the community better and more inclusive by dealing with some of these barriers.

6 Conclusion

OpenSolaris is a wonderful option as an operating system. Intel, for its part, is driving forward with innovation in hardware and software advances. Intel is fully engaged in the OpenSolaris enhancement work, and has delivered compelling results with more to come. The OpenSolaris community has made a wonderful start, and, with additional work, will achieve much more.

7 Legal Content

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site, http://www.intel.com**.**

This document contains information on products in the design phase of development.

Using DTrace API to write my own consumer

Petr Škovroň
Sun Microsystems Czech, s.r.o.
V parku 2308/8, Praha 4
Czech Republic
petr.skovron@sun.com

1 Introduction

DTrace is a framework for observing internals of a live Solaris system. It allows to insert programmable probes into the kernel or another program so that a specified action is performed when the execution hits the place where the probe is inserted. DTrace can be very useful for debugging or performance tuning, and is a very strong argument for Solaris when comparing to other operating systems. To learn more about DTrace itself, see the documentation [DTG].

I work as a maintainer of TSlvm, a test suite for Solaris Volume Manager. One of the tests consists of performing various actions with disks and metadevices and checking that the number of open calls on the disks matches the number of close calls. To do so, I needed a mechanism to perform the counting. This looks as a problem well-suited for using DTrace. However, when I tried to use the public documented interface `dtrace`(1M), I met certain issues I was not able to avoid. Therefore I decided to learn about the yet undocumented C API provided by `libdtrace`(3LIB) to see whether this could help resolve my issues. So far it seems that I succeeded, and this paper describes my experience, the API, and the code I wrote.

The main problem with the `dtrace`(1M) interface is that its output is designed to be interpreted chiefly by a human reader, while I needed to have the intermediate data accessible from a shell script. I could implement the counters either by plain numeric variables or by aggregations, but neither way seemed usable. Using the plain variables would suffer with dangers of accessing them from several concurrent threads, since the D language does not offer any locking facilities. Trying to use aggregations, I did not find how to access the intermediate results. Well, maybe I could print them to the output of the DTrace script, wait for some time till the output appears on `dtrace`'s stdout, and parse it by some nasty piece of Perl, but this seemed to be an overkill.

One might propose other, completely non-DTrace solutions to my problem. For example, in previous versions of the test suite the counting has been done by a kernel driver called `dsm`. When `dsm` was asked to watch open/close activity on a device *D*, it overwrote *D*'s operations table (*cb_ops*) to redirect opens and closes to *dsm_open* and *dsm_close*, which incremented a counter for *D* and called the original *D*'s operation. Unfortunately, there was an inherent race condition leading to system crashes when we deactivate the `dsm` driver and unload it from memory while another thread is in the middle of an open or close call, and I believe that it can not be avoided reliably.

The final code is a part of the TSlvm system, which is not open-sourced. To accompany this paper with a working code, I made a version simplified to the marrow and I made it available at `http://blogs.sun.com/xofon/entry/sample_dtrace_consumer`. The excerpts presented in the paper itself are rearranged and the declarations are moved towards the place where the variables are used so that the flow of the explanation is uninterrupted.

This paper is written from an uncalled-for interloper's point of view, since the `libdtrace`(3LIB) interface is officially private to the implementation and subject to change without notice. However, in an OpenSolaris discussion list one of DTrace authors Bryan Cantrill [Can07] (presumably unofficially) states: "[libdtrace]'s a Private interface, so it can change at any time. But as a practical matter, I don't think it's going to change much, at least for the foreseeable future—and almost certainly not in a way that breaks existing consumers". Still, nobody does guarantee anything. You have been warned.

Apart from this paper, another piece of unofficial documentation for `libdtrace`(3LIB) is Tom Erickson's blog entry [Eri05].

2 The DTrace script

As indicated in the Introduction, I am going to present a C code for a DTrace consumer that is able to communicate values of variables of a D script to the outside world. However, before I present the C code let us look at the D script `dsmtrace`.

My task was to count the number of open and close calls invoked on disk devices so that the test suite can check whether the counts match when the disk device is expected to be closed. Thus we want to set a probe on some functions that are called during every device open and close.

The selection of suitable functions to probe is influenced by a subtle point. One of the functions in the call chain counts how many clients use individual devices, and if a close is called on a device that is still held open by another client, the close routine of the device driver is not called. In other words, the choice of the probed function may generally influence the results.

Furthermore, for my purposes I needed to divide opens and closes into several categories:

- USR—successful ones originating from the userland,
- LYR (layered)—successful ones originating from the kernel,
- ERR—unsuccessful ones,
- ALL—the total.

So I required that the probed function would allow to determine the originator of the call.

I decided that the best probe points for my purposes are *spec_open* and *spec_close*. At entry to *spec_open* we determine the major and minor number of the device and origin of the open call (setting the thread variable *layered* to a nonzero value if the call is initiated by the kernel). We have to remember these values until return from *spec_open*, since we do not know yet whether the open will succeed. The values are remembered in a stack fashion so that we do not get confused in a case that a virtual device opens a physical device in its own open handler. Then we increment the global counter of opens.

```
fbt::spec_open:entry {
      self->depth += 1;

      self->dev = (*(struct vnode **)arg0) -> v_rdev;
      self->major[self->depth] = self->dev >> MINOR_BITS;
      self->minor[self->depth] = self->dev & ((1 << MINOR_BITS) - 1);
      self->dev = 0;
      self->layered[self->depth] = (arg1 & FKLYR) || (arg2 == OTYP_LYR);

      @ctr[DOPOPENALL,self->major[self->depth],
         self->minor[self->depth]] = count();
}
```

The identifier MINOR_BITS specifies the number of bits occupied by the minor number in the device number and is determined from the architecture of the kernel:

```
#if defined(ARCH64)
#define MINOR_BITS 32
#else
#define MINOR_BITS 18
#endif
```

If appropriate, we define ARCH64 in the moment we compile the D script in the consumer program.

The identifier DOPOPENALL determines the category of counted events and is defined by the preprocessor similarly to identifiers for other categories:

```
#define DOPOPENALL 0
#define DOPOPENLYR 1
...
```

The values FKLYR and OTYP_LYR are defined in system header files.

On return from *spec_open* we increment the counter for the appropriate category based on the return value of *spec_open* and on the originator of the open:

```
fbt::spec_open:return
   /(arg1==0) && (self->layered[self->depth])/ {
      @ctr[DOPOPENLYR,self->major[self->depth],
         self->minor[self->depth]] = count();
}

...
```

Then we release the variables from the stack:

```
fbt::spec_open:return {
      self->major[self->depth] = 0;
      self->minor[self->depth] = 0;
      self->layered[self->depth] = 0;
      self->depth -= 1;
}
```

For *spec_close*, the procedure is analogous.

3 Compiling and executing the DTrace script

Now we are getting to the C program called `dsmdriver` using the `libdtrace`(3LIB) interface.

To run the DTrace script we have to do a few preliminary tasks. First we have to connect to the DTrace framework using *dtrace_open*:

```
int err;
dtrace_hdl_t *dh;

dh = dtrace_open(DTRACE_VERSION, 0, &err);
```

The arguments are *version*, *flags*, and *errp*. For *version* we should specify DTRACE_VERSION; when running our program, the value compiled in the binary is cross-checked by the system to detect an API incompatibility. For *flags*, 0 is the safe choice. If the call succeeds, we get a pointer to an opaque DTrace handle that we will use when calling all further DTrace library functions. If we are unlucky, NULL is returned and **errp* is set to an error code, which can be translated into a meaningful string using *dtrace_errmsg*:

```
if (dh == NULL) {
      ERROR("Cannot open dtrace library: %s\n",
         dtrace_errmsg(NULL, err));
}
```

If *dh* is available when *dtrace_errmsg* is called, we shall provide it as the first argument so that we can get more specific information.

The next step is to compile the DTrace script. There are two possibilities of feeding the script into DTrace: a string or a *FILE* stream. The former is great for one-liners since it keeps the things compact, but we want to use the C preprocessor, so we need the latter. Speaking about the C preprocessor, our DTrace script assumes that we *#define* some symbols, and now is the right time to do that. Depending on the result of the *sysinfo* call we *#define* ARCH32 or ARCH64:

```
archsym = (sysinfo(SI_ARCHITECTURE_64, buf, 128) < 0) ?
   "ARCH32" : "ARCH64";
dtrace_setopt(dh, "define", archsym);
/* Check that the return value is not -1 */
```

We set various DTrace options by calling *dtrace_setopt*; the preprocessor *#define*s are just one class among them (the `-D` option to `dtrace`(1M)). To set a particular value for the symbol, e.g., to define ARCH to stand for 32, just specify "ARCH=32".

Now we can call *dtrace_program_fcompile*:

```
FILE *pfile;
dtrace_prog_t *prog;

pfile = fopen(argv[1], "r");
/* Check that fopen succeeded */

prog = dtrace_program_fcompile(dh, pfile, DTRACE_C_CPP, 0, NULL);
/* Check that the return value is not NULL */

fclose(pfile);
/* Check that fclose succeeded */
```

The first two parameters to *dtrace_program_fcompile* are pointers to the DTrace handle and the FILE with the script. We continue with *cflags*: the most notable are DTRACE_C_CPP which puts the stream through the C preprocessor, and DTRACE_C_ZDEFS which suppresses error if some probe in the script does not match any actual known probe; i.e., essentially the `-C` and `-Z` command-line options of `dtrace`(1M). Finally, *argc* and *argv* specify the macro arguments available to the script as `$1` etc. A pointer to the compiled program (in an opaque form) is returned.

If we wanted to pass the script as a string kept in *char *str*, we would probably want to call *dtrace_program_strcompile*(*dh*, *str*, DTRACE_PROBESPEC_NAME, *cflags*, *argc*, *argv*). By DTRACE_PROBESPEC_NAME we mean that the probe descriptions are to be interpreted as probe names, like using `-n` option to `dtrace`(1M) as opposed to `-f` etc.

We are ready to instrument the DTrace probes and start collecting the data:

```
dtrace_proginfo_t info;

dtrace_program_exec(dh, prog, &info);
/* Check that the return value is not -1 */

dtrace_go(dh);
/* Check that the return value is 0 */
```

Now DTrace is watching over our system and does what we want it to. However, even if we believe that the things go well, we have to ask DTrace about its status periodically, thus notifying it that we are alive. If we don't, the deadman mechanism deduces that we brought the system into an unresponsive state and that we have to be punished by disabling our probes. For more details search for *dtrace_deadman_timeout* in the DTrace source code [Dtr] or see Jon Haslam's blog entry [Has07].

```
while (1) {
      (void) dtrace_status(dh);
      sleep(1);
}
```

4 Accessing current data in the aggregations

Now when the DTrace script runs and watches the system, we want to read the data collected in the aggregations. To do this, we first take a snapshot of the aggregated data (which is then guaranteed to be stable and consistent):

```
dtrace_aggregate_snap(dh);
/* Check that the return value is 0 */
```

Access to the data is provided by functions of *dtrace_aggregate_walk* family; they are passed a callback function which is successively invoked on the individual tuples of all defined aggregations. The order of the tuples may be specified by choosing a suitable function from the family: we have *dtrace_aggregate_walk_keysorted*, *dtrace_aggregate_walk_valrevsorted*, etc.

The callback function is of type *dtrace_aggregate_f* which means that its prototype is something like

```
int walk(const dtrace_aggdata_t *data, void *arg)
```

When *walk* is called, the keys and the value of the aggregation tuple are provided in *data*, and *arg* is a pointer passed to *dtrace_aggregate_walk* that can be used to carry consumer's internal information. The interesting parts of *data* are *dtada_data* which is a raw byte array, and *dtada_desc->dtagd_rec* which is an array containing descriptors of individual elements of the tuple. From a descriptor we want to know *dtrd_offset*, which is the offset of the element's actual data in *dtada_data*. For numeric types, the entry is stored in the common way in the machine endian. The first descriptor in the array determines the aggregation variable (note that all *dtrace_aggregate_walk* functions walk over *all* aggregate variables); we do not need to look at it, since we have only one aggregation in our script. The further descriptors correspond to the keys, and the value assigned to the key tuple is given by just another record. The return value of the callback function usually should be DTRACE_AGGWALK_NEXT (to continue the walk with more tuples) or DTRACE_AGGWALK_ABORT (to stop the walk), however, special values for clearing, truncating, and normalizing the aggregation are defined too.

In our program, the function *walk* at first determines the offsets of individual keys in the data buffer:

```
{
      dtrace_aggdesc_t *aggdesc = data->dtada_desc;
      dtrace_recdesc_t *catrec, *majrec, *minrec, *datarec;

      catrec = &aggdesc->dtagd_rec[1];
      majrec = &aggdesc->dtagd_rec[2];
      minrec = &aggdesc->dtagd_rec[3];
      datarec = &aggdesc->dtagd_rec[4];
```

then retrieves the data:

```
      int cat, maj, min;
      int count;

      cat = *(int *)(data->dtada_data + catrec->dtrd_offset);
      maj = *(uint64_t *)(data->dtada_data + majrec->dtrd_offset);
      min = *(uint64_t *)(data->dtada_data + minrec->dtrd_offset);
      count = *(uint64_t *)(data->dtada_data + datarec->dtrd_offset);
```

and if *maj* and *min* correspond to the device we are interested in, we record the *count*:

```
typedef struct {
  int maj, min;
  int data[DOPMAX];
} dsm_buf_t;

  dsm_buf_t *pbuf = (dsm_buf_t *)arg;

  if (maj == pbuf->maj && min == pbuf->min) {
        pbuf->data[cat] = count;
        }
```

We continue with the remaining tuples:

```
      return (DTRACE_AGGWALK_NEXT);
}
```

To initiate the walk we run *dtrace_aggregate_walk*:

```
dsm_buf_t buf;

/* Set buf.maj and buf.min to major and minor number of the
```

```
 * queried device */
dtrace_aggregate_walk(dh, walk, (void *)&buf);
/* Check that the return value is 0 */
```

After returning from *dtrace_aggregate_walk*, we should have *buf.data* filled with the data from the aggregation.

5 Setting buffer sizes

If we expect that our script will generate lot of data, we should increase sizes of the DTrace internal buffers so that the collected data do not get lost. This can be done by *dtrace_setopt* in the same manner as when we defined the symbols for the preprocessor:

```
dtrace_setopt(dh, "aggsize", "128k");
/* Check that the return value is not -1 */
```

The `aggrate` option limits how often we can ask DTrace for the up-to-date aggregation data. Snapping the aggregation takes time, so by default we are limited to take 1 snapshot per second, and if we try to do so more often, we get the last snapshot taken instead of a brand new one. If there is a chance that we will need the current data more often, we can increase `aggrate`.

```
dtrace_setopt(dh, "aggrate", "10hz");
/* Check that the return value is not -1 */
```

Note that some options can be changed dynamically at any time (like `aggrate`), some are effective only when set before we start collecting the data by *dtrace_go* (like `aggsize`), and some have to be set before compilation (like `define`). For more details on the division, see *_dtrace_ctoptions* and below in the DTrace sources.

6 Providing an interface to make the gathered data available to shell

Since the test suite TSlvm is mostly written in shell, I needed a command-line interface to access the data. For that purpose I equipped `dsmdriver` with a Doors interface and wrote a dedicated utility `dsmstat` employing this interface. In `dsmstat` I determine the major and minor number of the examined device, allocate the *dsm_buf_t* structure *buf* and fill *buf.maj* and *buf.min*, call the exported Doors function in `dsmdriver`, and format *buf.data* for output. For convenient usage, the exit status determines whether the number of kernel-initiated opens matches the number of kernel-initiated closes.

7 Compilation and running

To compile the program, we should include the `libdtrace` function prototypes:

```
#include <dtrace.h>
```

We have to link with the `libdtrace` library, and since we used Doors for interprocess communication, we link with `libdoors` too:

```
cc -ldoor -ldtrace -o dsmdriver dsmdriver.c
```

After compiling the client program `dsmstat` the system is ready for use:

```
root@eschaton:/tmp/src# ./dsmdriver &
[1] 12501
root@eschaton:/tmp/src# ./dsmstat /dev/null
```

```
/dev/null: open 0:lyr 0,usr 0,err 0; close 0:lyr 0,usr 0,err 0
root@eschaton:/tmp/src# dd if=/bin/ls of=/dev/null
64+1 records in
64+1 records out
root@eschaton:/tmp/src# dd if=/bin/ls of=/dev/null
64+1 records in
64+1 records out
root@eschaton:/tmp/src# ./dsmstat /dev/null
/dev/null: open 2:lyr 0,usr 2,err 0; close 2:lyr 0,usr 2,err 0
root@eschaton:/tmp/src# kill %1
root@eschaton:/tmp/src#
```

8 Conclusion

We have covered only a small part of the DTrace API, just enough to solve my original problem. A reader interested in more information can check [Eri05], or even see the DTrace sources [Dtr], which are actually very enjoyable to read.

I want to thank Milan Čermák for reading the preliminary version of this paper and for providing helpful comments.

9 References

[Can07] Bryan Cantrill: dtrace-discuss@opensolaris.org, `http://opensolaris.org/jive/thread.jspa?threadID=40526&tstart=390`, 2007

[DTG] Sun Microsystems, Inc.: *Solaris Dynamic Tracing Guide*, `http://wikis.sun.com/display/DTrace/Documentation`, 2007

[Dtr] Sun Microsystems, Inc.: DTrace kernel and library source code, `http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/dtrace/`, `http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libdtrace/`

[Eri05] Tom Erickson: *Using libdtrace*, `http://blogs.sun.com/tomee/entry/tom_erickson_s_weblog`, 2005

[Has07] Jon Haslam: *The DTrace deadman mechanism*, `http://blogs.sun.com/jonh/entry/the_dtrace_deadman_mechanism`, 2007

Experiences with DTrace development: `stddev()` and `brendan()`

Chad Mynhier
New York, NY
USA
cmynhier@gmail.com

1 Introduction

DTrace[1] is a recent addition to Solaris that provides a framework for dynamic instrumentation of systems. DTrace has a number of features that make it ideal for use on production systems, among them that it has zero disabled probe effect and that it was designed with safety as an overriding goal. DTrace provides a large number of probes on stock Solaris systems, including system call boundary tracing, kernel function boundary tracing, and probes at various points of interest in the kernel related to I/O, networking, NFS, etc. DTrace provides a simple scripting language, similar to awk and C, that makes it easy to do very powerful things with a small amount of code.

In this paper, the author presents two pieces of DTrace development that he has completed. `stddev()` is an aggregating action that calculates the standard deviation of a set of values. The work for `stddev()` has been put back into OpenSolaris. `brendan()` is an action that prints a message to standard output to inform a user that DTrace is collecting data and that the user should press `ctrl-C` to stop the data collection and print the results. The work for `brendan()` was intended as an April Fool's joke directed at the author of the DTrace Toolkit. Although intended in humor, the work stands as a complete tutorial on adding an action to DTrace. This paper will also briefly discuss the DTrace test suite and give some representative examples of test suite scripts.

2 `stddev()`

When DTrace was released to the public in 2003, it included a number of aggregating actions to perform basic statistical functions: `count()`, `sum()`, `min()`, `max()`, and `avg()`. An aggregating action represents a function that can be calculated by combining the results of that individual calculation on each of an arbitrary number of CPUs. For example, the average of a set of data related to a probe can be calculated in this fashion by calculating the average of the data generated on each CPU and then calculating the average of these values across CPUs. On the other hand, the median of a set of numbers cannot be calculated this way, as one needs the entire data set for the calculation. Aggregating actions are useful in DTrace because they minimize the amount of data that need to be stored in the kernel and copied out to userspace. For the above aggregating actions, a constant amount of data is stored, and for `quantize()` and `lquantize()`, a constant amount of data is stored for each of a small number of buckets.

The creators of DTrace did not originally include an aggregating action for standard deviation, although there has been a long-standing RFE[2] to correct this oversight. The author decided to implement this RFE as an introduction to DTrace development. On initial inspection, the implementation seemed simple, as it could be modeled after the existing implementation of one of the existing aggregations, e.g., `avg()`. Aside from the DTrace particulars, which will be discussed later in this paper, the `avg()` action is implemented in three functions. The first of these is in the kernel, in which a count and running sum of the data points is stored in a buffer. The other two are in userland, one the computation of average values for display and the other a comparison function used to sort the data. Two of these functions are listed below (we exclude the comparison function):

```
static void
```

```
dtrace_aggregate_avg(uint64_t *data, uint64_t nval, uint64_t arg)
{
        data[0]++;
        data[1] += nval;
}

static int
dt_print_average(dtrace_hdl_t *dtp, FILE *fp, caddr_t addr,
    size_t size, uint64_t normal)
{
        uint64_t *data = (uint64_t *)addr;
        return (dt_printf(dtp, fp, " %16lld", data[0] ?
                (long long)(data[1] / normal / data[0]) : 0));
}
```

The straightforward implementation of `stddev()` would involve three similar functions. Whereas `avg()` is computed by dividing the sum of the items in the data set by the count, `stddev()` is slightly more complicated. Standard deviation is calculated as the square root of the difference of the average of x squared and the average squared of x, i.e., `sqrt(avg(x^2) - avg(x)^2)`. Using `avg()` as a model, the functions to implement `stddev()` would be implemented as follows:

```
static void
dtrace_aggregate_stddev(uint64_t *data, uint64_t nval, uint64_t arg)
{
        data[0]++;
        data[1] += nval;
        data[2] += nval * nval;
}

static int
dt_print_stddev(dtrace_hdl_t *dtp, FILE *fp, caddr_t addr,
    size_t size, uint64_t normal)
{
        uint64_t *data = (uint64_t *)addr;
        uint64_t avg = data[0] ? (data[1] / data[0]) : 0;
        uint64_t avgsq = data[0] ? (data[2] / data[0]) : 0;

        return (dt_printf(dtp, fp, " %16lld",
            (long long) sqrt(avgsq - avg * avg)));
}
```

Having initially implemented `stddev()` this way, the author noted that there was a possibility for integer overflow. Indeed, this possibility exists in the implementation of `avg()`, but in that case, the possibility of overflowing a 64-bit unsigned integer is very unlikely. The implementation of `stddev()` involves squaring a value, however, which means that a 33-bit `nval` would cause overflow. This would still likely be a rare event, but it is far more likely than overflowing a 64-bit sum. As implemented above, the `stddev()` action introduces the possibility of silently returning invalid data to the user, which is unacceptable.

There are two courses of action available. The first is to note the overflow, report it to the user, and discard the data. This is certainly a valid approach in some cases: when exceeding a hard physical constraint such as an internal buffer, one has no choice but to discard the data, and one has an obligation to the user to report this. This is not one of those cases, however, as no hard constraint is involved. A second approach is to implement arithmetic to a precision sufficient to avoid this problem, in this case 128-bit arithmetic, and it is this approach that the author chose.

It should be noted that the truly general solution to this problem is to implement arbitrary-precision arithmetic. The author chose not to do so, as 128 bits should be sufficient for quite a number of years to come. One might also question the necessity of implementing this from scratch as opposed to using an existing arbitrary-precision library. Given that some part of this code executes in the kernel, the necessary functions would need to be copied into the kernel source, which might have involved license incompatibilities. This also would have introduced a dependency in the kernel on the data structures used by that library to represent arbitrary-precision numbers. Given the relative simplicity of implementing fixed 128-bit arithmetic, the author chose to do so rather than introduce that dependency.

3 DTrace internals

The above discussion of `stddev()` only briefly touches on DTrace internals. Before discussing the `brendan()` action, we present certain aspects of DTrace internals that will be helpful in understanding that implementation. We will touch on the target language for the compilation of D scripts, the virtual machine that interprets that language, the object that represents an expression in the target language, and the object that contains the predicate and list of actions associated with a probe.

D is the scripting language used by DTrace. Each clause in a D script consists of one or more probe specifiers, an optional predicate, and a set of actions. The following example counts the number of system calls being made by the StarOffice executable:

```
syscall:::entry
/ execname == "soffice.bin" /
{
        @[probefunc] = count();
}
```

3.1 DIF

The predicate `(execname == "soffice.bin")` and actions `(@[probefunc] = count();)` are compiled into DIF, the DTrace Intermediate Format, a RISC-like instruction set. This instruction set is defined in `/usr/include/sys/dtrace.h`, as below:

```
#define DIF_DIR_NREGS   8               /* number of DIF integer registers */
#define DIF_DTR_NREGS   8               /* number of DIF tuple registers */

#define DIF_OP_OR       1               /* or   r1, r2, rd */
#define DIF_OP_XOR      2               /* xor  r1, r2, rd */
#define DIF_OP_AND      3               /* and  r1, r2, rd */
[ ... ]
#define DIF_OP_TST      16              /* tst  r1 */
#define DIF_OP_BA       17              /* ba   label */
#define DIF_OP_BE       18              /* be   label */
[ ... ]
#define DIF_OP_LDSB     28              /* ldsb [r1], rd */
#define DIF_OP_LDSH     29              /* ldsh [r1], rd */
#define DIF_OP_LDSW     30              /* ldsw [r1], rd */
```

This instruction set is interpreted by a virtual machine inside the kernel. The virtual machine is defined in `usr/src/uts/common/dtrace/dtrace.c` in the function `dtrace_dif_emulate()`. This virtual machine emulates the simplest functions of a CPU: fetch the next instruction, decode that instruction, execute it, repeat. This is demonstrated in the following snippet of code from the function, including selected instructions:

```
        while (pc < textlen && !(*flags & CPU_DTRACE_FAULT)) {
                opc = pc;

                instr = text[pc++];
                r1 = DIF_INSTR_R1(instr);
                r2 = DIF_INSTR_R2(instr);
                rd = DIF_INSTR_RD(instr);

                switch (DIF_INSTR_OP(instr)) {
                case DIF_OP_OR:
                        regs[rd] = regs[r1] | regs[r2];
                        break;
                case DIF_OP_TST:
                        cc_n = cc_v = cc_c = 0;
                        cc_z = regs[r1] == 0;
                        break;
```

```
                case DIF_OP_BE:
                        if (cc_z)
                                pc = DIF_INSTR_LABEL(instr);
                        break;
```

We get the next instruction, incrementing the program counter as a side effect. We determine the two source instructions `r1` and `r2` and the destination register `rd`. We then perform certain actions based on the opcode of the instruction. For the `DIF_OP_OR` opcode, we perform a bitwise OR of the two source registers and place the result into the destination register. For the `DIF_OP_TST` opcode, we first clear other condition codes and then set the condition code `cc_z` to reflect whether the value in source register `r1` is zero. For the `DIF_OP_BE` opcode ("branch if equal to zero"), we check the condition code `cc_z` and set the program counter to the target address.

3.2 DIFO

Each expression in this target language is represented by a DIFO, a DTrace Intermediate Format Object. This object contains everything necessary to evaluate a D expression: the compiled DIF, the return type of the expression, and certain tables. The relevant parts of this data structure (defined in `/usr/include/sys/dtrace.h`) are shown below:

```
typedef struct dtrace_difo {
        dif_instr_t *dtdo_buf;          /* instruction buffer */
        uint64_t *dtdo_inttab;          /* integer table (optional) */
        char *dtdo_strtab;              /* string table (optional) */
        dtrace_difv_t *dtdo_vartab;     /* variable table (optional) */
        uint_t dtdo_len;                /* length of instruction buffer */
        uint_t dtdo_intlen;             /* length of integer table */
        uint_t dtdo_strlen;             /* length of string table */
        uint_t dtdo_varlen;             /* length of variable table */
        dtrace_diftype_t dtdo_rtype;    /* return type */
```

To understand the use of these tables, consider the following D expression:

```
printf("%d\n", timestamp - this->ts)
```

The string `"%d\n"` is retrieved from the string table so that the resultant string can be constructed, and the location of the variable `this->ts` is obtained from the variable table.

3.3 ECB

An enabling control block (ECB) is the data structure that associates a predicate DIFO and a set of action DIFOs with a specific probe. Each probe has an associated linked list of ECBs, as we can specify a probe multiple times in a D script (for example, to perform different actions based on conditions specified in different predicates associated with that probe.) When a probe fires, DTrace traverses the list of ECBs attached to the probe. For each ECB, the predicate DIFO is first evaluated. If it evaluates to a non-zero value, the list of actions is traversed, and each action DIFO is run.

The relevant fields of this data structure (defined in `/usr/include/sys/dtrace_impl.h`) are shown below:

```
struct dtrace_ecb {
     dtrace_predicate_t *dte_predicate;      /* predicate, if any */
     dtrace_action_t *dte_action;            /* actions, if any */
     dtrace_ecb_t *dte_next;                 /* next ECB on probe */
```

4 brendan()

The author attended dtrace.conf(08) in March, 2008. During the conference, Bryan Cantrill directed some good-natured ribbing at Brendan Gregg for this probe that is included in most of the scripts in the DTrace Toolkit[3]:

```
BEGIN
{
     printf("Tracing... hit ctrl-C to end.\n");
}
```

It was suggested that a `brendan()` action be implemented to replace this particular print statement. The author decided to do so as an exercise and, given the timing, decided to present it as an April Fool's joke. He submitted the patch to the dtrace-discuss mailing list[4] and received responses that this would make a good tutorial for adding an action to DTrace.

4.1 Userland code changes

The bulk of the code changes for `brendan()` were in userland. These components can be classified as follows: reserve the keyword `brendan` in the D scripting language, and associate certain information with that keyword; inform the D compiler how to handle this new action; define an action to perform when consuming the `brendan()` action; and handle the bookkeeping for the above.

4.1.1 dt_open.c

As the first step in adding the action, we declare the new `brendan` keyword and associate certain information with that keyword. In the file `usr/src/lib/libdtrace/common/dt_open.c`, we see the `_dtrace_globals[]` array, which is the table of global identifiers for the D scripting language. Some example entries in this table are as follows:

```
{ "avg", DT_IDENT_AGGFUNC, 0, DTRACEAGG_AVG, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "void(@)" },
{ "basename", DT_IDENT_FUNC, 0, DIF_SUBR_BASENAME, DT_ATTR_STABCMN,
        DT_VERS_1_0, &dt_idops_func, "string(const char *)" },
{ "breakpoint", DT_IDENT_ACTFUNC, 0, DT_ACT_BREAKPOINT,
        DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_func, "void()" },
{ "caller", DT_IDENT_SCALAR, 0, DIF_VAR_CALLER, DT_ATTR_STABCMN,
        DT_VERS_1_0, &dt_idops_type, "uintptr_t" },
```

with the corresponding data structure as follows:

```
typedef struct dt_ident {
        char *di_name;                  /* identifier name */
        ushort_t di_kind;               /* identifier kind (see below) */
        ushort_t di_flags;              /* identifier flags (see below) */
        uint_t di_id;                   /* variable or subr id (see <sys/dtrace.h>)*/
        dtrace_attribute_t di_attr;     /* identifier stability attributes */
        uint_t di_vers;                 /* identifier version number (dt_version_t)*/
        const dt_idops_t *di_ops;       /* identifier's class-specific ops vector*/
        void *di_iarg;                  /* initial argument pointer for ops vector */
```

The interesting fields in this context are the name, the kind, the identifier, and the prototype. For example, `avg` is an aggregating action (`DT_IDENT_AGGFUNC`) with identifier `DTRACEAGG_AVG` and function prototype `void(@)`, indicating that it's a function taking a single argument of arbitrary type and returning no value.

Based on the above, we add the `brendan()` action as follows:

```
{ "brendan", DT_IDENT_ACTFUNC, 0, DT_ACT_BRENDAN, DT_ATTR_STABCMN,
        DT_VERS_1_7, &dt_idops_func, "void()" },
```

We specify that it's an action, give it an identifier (`DT_ACT_BRENDAN`), and declare its type to be a function taking no arguments and returning no value.

The other notable difference in the declaration of `brendan` here is the identifier version, `DT_VERS_1_7`.

The prior version of this value had been DT_VERS_1_6, where the minor number had been changed to reflect the addition of the stddev() action. The minor number is increased for any change that could break scripts that had previously worked. For stddev() or brendan(), one could imagine existing scripts using one of those strings as a variable. The addition of these actions would break those scripts during the compilation phase, as these would now be reserved words.

4.1.2 dt_cc.c

Having reserved the keyword brendan for our use, we must now tell the compiler how to process it. The changes to accomplish this are located in usr/src/lib/libdtrace/common/dt_cc.c. The first change is very simple, as it is merely adding a case to the switch statement that comprises the dt_compile_fun() function:

```
static void
dt_compile_fun(dtrace_hdl_t *dtp, dt_node_t *dnp, dtrace_stmtdesc_t *sdp)
{
        switch (dnp->dn_expr->dn_ident->di_id) {
        case DT_ACT_BREAKPOINT:
                dt_action_breakpoint(dtp, dnp->dn_expr, sdp);
                break;
        case DT_ACT_BRENDAN:
                dt_action_brendan(dtp, dnp->dn_expr, sdp);
                break;
[ ... ]
```

Put simply, when compiling an action, DTrace will perform certain actions based on the identifier for that action. dt_action_exit() is a good example to consider before looking at the implementation of the brendan() action:

```
static void
dt_action_exit(dtrace_hdl_t *dtp, dt_node_t *dnp, dtrace_stmtdesc_t *sdp)
{
        dtrace_actdesc_t *ap = dt_stmt_action(dtp, sdp);

        dt_cg(yypcb, dnp->dn_args);
        ap->dtad_difo = dt_as(yypcb);
        ap->dtad_kind = DTRACEACT_EXIT;
        ap->dtad_difo->dtdo_rtype.dtdt_size = sizeof (int);
}
```

We first allocate a data structure to contain information about this invocation of the exit() action. (This data structure is linked into the list of actions for this ECB.) We next generate DIF code by calling the code generator function dt_cg(), assemble the DIF by calling dt_as(), and attach that DIFO to this particular exit() action. We declare what kind of action this is and what size the return value is.

The dt_action_brendan() function is much simpler, as the brendan() action takes no arguments and returns no value:

```
static void
dt_action_brendan(dtrace_hdl_t *dtp, dt_node_t *dnp, dtrace_stmtdesc_t *sdp)
{
        dtrace_actdesc_t *ap = dt_stmt_action(dtp, sdp);

        ap->dtad_kind = DTRACEACT_BRENDAN;
        ap->dtad_arg = 0;
}
```

Here we're merely setting the kind for the action and specifying that there is no argument.

4.1.3 dt_consume.c

Having told the the compiler how to compile the `brendan()` action, we need to define the action to take when we consume the action after a probe using it fires. This we do in `usr/src/lib/libdtrace/common/dt_consume.c` in the function `dt_consume_cpu()`. This appropriately-named function consumes data returning from a single CPU. The `brendan()` action itself returns no useful data, but we have set things up so that it reserves space for a payload if it were to do so. As we shall see, consumption is predicated on the fact that there is a returning payload; if `brendan()` did not reserve this space, we would not see the intended output.

The following is a skeletal outline of this function:

```
static int
dt_consume_cpu( ... )
{
        for (offs = start; offs < end; ) {
                dtrace_eprobedesc_t *epd;
                for (i = 0; i < epd->dtepd_nrecs; i++) {
                        dtrace_actkind_t act = rec->dtrd_action;
                        if (act == DTRACEACT_BRENDAN) {
                                if (dt_printf(dtp, fp, "Tracing... Hit ctrl-C "
                                   "to end.\n") < 0)
                                        return (-1);
                                goto nextrec;
                }
```

We have a main loop that consumes the buffers returning from the kernel. Each buffer contains some number of records, which we consume in the inner loop. For the `brendan()` action, we simply print the message without actually consuming data. We will see later that we need to reserve space for data to be returned from the kernel for this action in order to trigger the above code.

4.1.4 Constants

In the code above, we've used two new constants, which we must define. We define `DT_ACT_BRENDAN` in `usr/src/lib/libdtrace/common/dt_impl.h`. Both actions like `brendan()` and subroutines like `basename()` are represented by the same abstract syntax tree node type, so care is taken to avoid overlapping ranges for these:

```
#define DT_ACT_BASE             DIF_SUBR_MAX + 1
#define DT_ACT(n)                       (DT_ACT_BASE + (n))
[ ... ]
#define DT_ACT_UADDR            DT_ACT(27)      /* uaddr() action */
#define DT_ACT_SETOPT           DT_ACT(28)      /* setopt() action */
#define DT_ACT_BRENDAN          DT_ACT(29)      /* brendan() action */
```

We define `DTRACEACT_BRENDAN` in `usr/src/uts/common/sys/dtrace.h`. The upper byte for the constants for the DTrace actions determines the class of the action, with no specified class (i.e., a zero byte) indicating that the action can record process- or kernel-related data. The `brendan()` action records no data, but we put it in this class regardless:

```
#define  DTRACEACT_PRINTA               4        /* printa() action */
#define  DTRACEACT_LIBACT               5        /* library-controlled action */
#define  DTRACEACT_BRENDAN              6        /* brendan() action */
```

4.2 Kernel code changes

The remainder of the code changes for `brendan()` are in the kernel. These components can be classified as follows: specify how much space to reserve for data generated by the `brendan()` action and specify what to do when the action is encountered in a firing DTrace probe. The code changes necessary for both of these components are in `usr/src/uts/common/dtrace.c`.

The first of these changes can be found in the function `dtrace_ecb_action_add()`. This function is used to create the list of actions contained in an ECB. In addition to the DIFO for the action, we store certain other information. For `brendan()`, we are primarily interested in specifying how much data the action will generate. This particular action generates no data, but we still specify a non-zero size so that the action will be consumed in userland. The relevant parts of `dtrace_ecb_action_add()` are shown below:

```
static int
dtrace_ecb_action_add(dtrace_ecb_t *ecb, dtrace_actdesc_t *desc)
{
        switch (desc->dtad_kind) {
        case DTRACEACT_BRENDAN:
                size = sizeof (uint32_t);
                break;

        action->dta_rec.dtrd_size = size;
```

We arbitrarily choose an unsigned 32-bit integer as the size of the data to be returned, and this size is recorded as the return data size for this particular action.

The second of the kernel changes is in the function `dtrace_probe()`. This function is central to DTrace, as it is the function that is called when any probe fires. Skeletally, this function appears as follows:

```
void
dtrace_probe( ... )
{
        for (ecb = probe->dtpr_ecb; ecb != NULL; ecb = ecb->dte_next) {
                if (pred != NULL) {
                        dtrace_difo_t *dp = pred->dtp_difo;
                        rval = dtrace_dif_emulate(dp, &mstate, vstate, state);
                        if (!(*flags & CPU_DTRACE_ERROR) && !rval) {
                                continue;
                        }
                }
                for (act = ecb->dte_action; !(*flags & CPU_DTRACE_ERROR) &&
                   act != NULL; act = act->dta_next) {
                        switch (act->dta_kind) {
                        case DTRACEACT_BRENDAN:
                                continue;
```

As stated before, `dtrace_probe()` walks the list of ECBs, evaluates the predicate for each, and then performs the associated list of actions if the predicate evaluates to true. In the case of `brendan()`, we're not actually doing anything in the kernel, so we simply continue to the next action.

5 DTrace Test Suite

As with any good software, DTrace has an associated test suite. In contrast to many of the other subsystems in OpenSolaris, the DTrace test suite is included as part of the code base. The intent of the authors[5] was

that every code change to DTrace include an addition to the test suite. To this end, the authors designed the test suite to be as simple as possible, both to run and to update.

The driver for the test suite is very straightforward. It walks the subtree rooted at foo/bar and runs the scripts it finds there. The scripts are either D scripts or ksh scripts, being run under dtrace or ksh, respectively. (Success for a test run is defined in a similarly simple fashion.) Each script is accompanied by an optional output file. For those scripts accompanied by an output file, a test run is successful if the output of the script matches the contents of the output file; for scripts without an output file, a test run is successful if the return value of the script is 0.

To demonstrate the simplicity of the test suite, we present a sample test suite D script, the test for the `brendan()` action, `tst.brendan.d`:

```
#pragma D option quiet

BEGIN
{
        brendan();
        exit(0);
}
```

The output file, tst.brendan.d.out, is simply as follows:

```
Tracing... Hit ctrl-C to end.
```

6 Conclusion

In late 2007, the author decided to implement the `stddev()` aggregating action in DTrace as an introduction to DTrace development. The code for this was put back into the Solaris code base in build 84. In March of 2008, the author implemented the `brendan()` action as an April Fool's joke. Although intended in humor, this work was a complete, working implementation of a new action in DTrace, and as such, proved effective as a tutorial for anyone interested in performing similar work.

7 Acknowledgments

The author would like to thank Jon Haslam for sponsoring the code putback of `stddev()` into Solaris, Jon and Adam Leventhal for the code reviews necessary to transform the code into something worthy of being put back, and Bryan Cantrill for approving the request-to-integrate. He would also like to thank Brendan Gregg for inspiring the `brendan()` action.

8 Literature

[1] http://wikis.sun.com/display/DTrace/DTrace

[2] http://bugs.opensolaris.org/view_bug.do?bug_id=6325485

[3] http://opensolaris.org/os/community/dtrace/dtracetoolkit/

[4] http://www.opensolaris.org/jive/thread.jspa?threadID=56137&tstart=75

[5] http://blogs.sun.com/ahl/entry/on_testing

OpenSolaris™ and NUMA Architectures

Rafael V. Polanczyk
Sun Microsystems

rafael.vanoni@sun.com

1 Introduction

Parallel and multi-threaded software development have become an every day reality. With the advent of multicomputers and multiprocessors, we now have increasing multi-processing power available at a fraction of what it cost a decade ago.

An application can take advantage of parallel processing in different ways. An Internet server will spawn multiple connections to address an intrinsically parallel problem, or achieve higher performance and throughput by decomposing a complex calculation into various parts and executing them concurrently, for example. However, the development of a parallel application must take into account the nature of the machine on which it will be executed. Aspects such as communication, programming paradigm, scalability and throughput must be taken into account.

Multicomputers, or clusters, are a highly scalable and flexible architecture composed of nodes connected through a network mechanism. The main advantage of clusters is the simplicity of adding more nodes and growing the system's processing power. However, it's necessary to use some form of message passing programming, such as MPI, so processes executing in different nodes can communicate with each other. This adds an undesirable level of complexity to application development.

Multiprocessors, such as the Symmetric Multiprocessing (SMP) class, contain one or more processing units connected to memory through a single bus. All processors share the same address space, resulting in an environment for the shared memory programming paradigm. This environment is more natural to developers and simplifies parallel application development, available through programming libraries such as POSIX Threads and OpenMP. However, SMP does not scale well after a certain number of processing units, with communication saturating and turning the single bus model into a performance bottleneck.

A solution that merges the benefits of both architectures while minimizing their disadvantages is the cache coherent Non Uniform Memory Access (ccNUMA, or simply NUMA) model. A NUMA machine is composed of a series of nodes, each containing processors and local memory. The nodes are connected through interconnection hardware, forming a single shared address space. Cache coherence between the different processors is implemented in hardware, transparent to software.

The goal of the NUMA model is to allow a scalable multiprocessor architecture to maintain the shared memory programming paradigm while avoiding the single memory access bus seen on SMP computers. However, this arrangement of processors and nodes at different physical distances from one another causes non-uniform memory access (NUMA) times throughout the system. A processor accessing local memory will get the information in less time than accessing a remote memory position because it won't need to travel across the interconnect. This characteristic is known as the NUMA factor, the ratio between local and remote access times.

The concept of resource affinity plays a major role when optimizing application performance on NUMA systems. It is desirable to avoid remote accesses because they add a considerable amount of latency to an operation, a process will perform optimally if every memory access is local. A relationship of affinity to the node in which its memory resides – usually the node where it was created – is established, and the operating system must understand and explore it.

From the operating system's point of view, NUMA architecture presents new challenges and demands. The system must be aware of the available hardware resources and the distances (and resulting access latency) between such resources. The transfer of data between nodes becomes an important subject since the added overhead can cause a considerable performance drop.

Operating systems must recognize the presence of such architecture and adapt their subsystems, such as the virtual memory and dispatcher/scheduler to optimize application performance. Systems based on the OpenSolaris™ kernel employ the locality group abstraction to identify and properly represent the underlying hardware layout and its different access times. Such identification takes place during system boot and generates a latency topology, consulted by the relevant kernel subsystems to optimize memory placement for user applications. This 'out of the box' support is complemented by a user library and command line tools that fulfill different and complementing roles in system administration and software development. This set of optimizations for NUMA systems is known as the Memory Placement Optimization framework, or MPO.

Such an approach has proven to be successful, with considerable performance gains across different systems and workloads. However, the recent advances in both hardware and software technologies present challenges. The increasing density of processor cores and new NUMA topologies, device locality and power efficiency on systems with various multiprocessor nodes are all characteristics and requirements that the operating systems must address.

This paper presents the design of the OpenSolaris support for NUMA architectures and key points of its implementation, discussing today's challenges and how the operating system is evolving to meet these demands.

2 NUMA Architectures

The need for parallelism extended the fundamental concept of the single flow of execution process, allowing the simultaneous execution of multiple tasks. This parallelism is achieved in different ways, through the execution of single instructions on a set of data, running a task while another is awaiting the completion of an event, or having multiple threads execute concurrently.

To allow this evolution, computer architectures have been providing different alternatives in hardware over the years. These include new instructions that operate on larger sets of data, deeper execution pipelines and even more complex interconnection systems to allow multiple instances of processing units to communicate and cooperate through difference memory sharing paradigms.

Operating systems have followed these developments closely, supporting the evolution of hardware so the end user can take full advantage of new features and fully harvest the processing power of the system.

2.1 Parallel Architectures

There are several ways of designing computers to support the parallel execution of different tasks. To properly classify the different kinds of parallelism, Flynn [1] proposed a taxonomy based on a two-dimensional combination. The first dimension identifies the number of flows of execution that a machine is able to execute at a given time; the second refers to the amount of data that can be processed at the same time (Figure 1).

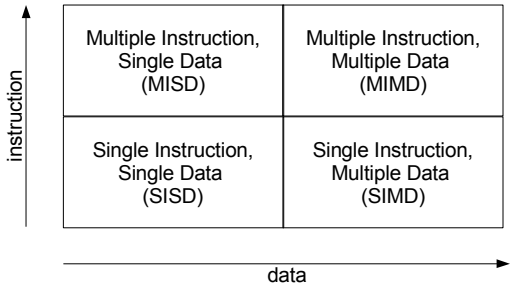


Figure 1: Flynn's taxonomy

Today's systems are, in the majority of cases, part of the *multiple instruction, multiple data* (MIMD) class of parallel computers. At any given time, multiple processing units - independent from each other - are performing calculations over sets of data.

Flynn's classification, however, does not address a key element in parallel software development: memory access. We can further divide these models into *tightly coupled* and *loosely coupled*.

Tightly coupled systems provide a *shared address space*, where all processing units see the same single memory space, and communicate with each other through reading and writing memory positions. Loosely coupled systems are characterized by the fact that each processing node has its own *private address space*. Communication is only possible through message passing (send and receive) primitives. These models are also referred to as multiprocessors (tightly coupled) and multicomputers, or distributed systems (loosely coupled).

An even further categorization of such models reflects the disposition of memory within the system. Tightly coupled architectures can have their memory centralized or distributed. Centralized memory systems present a *uniform memory access* time (UMA, Figure 2a) to its processors, meaning that every CPU will experience the same latency when accessing main memory - a typical characteristic of symmetric multiprocessors (SMPs), while the distributed model enforces *non-uniform memory access* (NUMA, Figure 2b) times throughout the system. The shared address space is composed of physically distant memory banks, each attached to a subset of the total amount of processors.

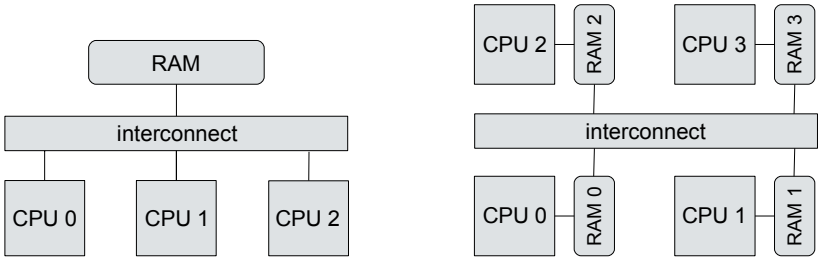


Figure 2: examples of UMA and NUMA models

With NUMA, a processor accessing node local memory is said to be performing a *local access*. While accesses to distant nodes are referred to as *remote accesses*. Local accesses are faster because there's no need to go through the interconnect hardware. Remote accesses have the added penalty of the interconnect, and are subject to the cache coherency mechanism implemented in hardware by the manufacturer. Cache coherency is needed since today's processors have internal, private caches. When two or more processors are accessing the same memory position, both will store a copy of the information in their respective caches. NUMA implements cache coherency to avoid having inconsistent copies of the same data in different parts of the system.

This cache coherency is a key element of NUMA architectures. In fact, current NUMA machines are actually called *ccNUMA* - cache coherent NUMA - as the first generation of NUMA systems did not implement this consistency in hardware. The coherency mechanism itself varies according to the manufacturer and model of the system. The performance of such an algorithm and its implementation is a key component of the overall system throughput.

2.2 NUMA Characteristics

NUMA machines are built by interconnecting different modules composed of processors and memory. From this general description, NUMA might resemble a traditional cluster organization, but there are two important distinctions. First, a NUMA machine has a single shared address space comprised of all existing physical memory, each node being responsible for a range of the total memory. Second, each node of a cluster runs its own instance of an operating system, while NUMA nodes are all managed by a single OS.

As mentioned earlier, this arrangement of processors and nodes within different physical distances of one another causes *non-uniform memory accesses* throughout the system. The ratio between local and remote access times is known as the *NUMA factor*, and is often used by manufacturers as an indication of system performance.

The distances between different nodes and their layout dictates the system's latency topology, that is, the connections and access times between nodes. The operating system needs to identify this topology. For this purpose, the Advanced Configuration and Power Interface (ACPI) specification defines two tables that vendors can use to identify a given system's latency topology. These tables are the *System Resource Affinity Table* (SRAT) and the *System Locality Information Table* (SLIT). The SRAT provides information that

allows the OS to associate processors and memory ranges with system localities [3]. While the SLIT exports a matrix describing the relative distance (memory latency) between different nodes [4]. However, these two ACPI tables are not always supplied by vendors. This can lead to poor performance if the OS does not verify the accuracy of the supplied information and fails over to a coherent topology.

3 OpenSolaris support for NUMA

NUMA systems are a subclass of multiprocessors with specific latency characteristics. An operating system needs to adapt its subsystems to offer the highest possible throughput on such an architecture. Amongst the necessary modifications are extensions to the dispatcher, the virtual memory subsystem and early system identification routines, which will recognize the existence of NUMA characteristics and properly identify the system.

OpenSolaris provides support for NUMA architectures through a feature called *Memory Placement Optimization*, or MPO. This feature is based around three main concepts: affinity, load balancing and dynamic topology support [2].

On OpenSolaris, a process is said to have an affinity with the node in which it was created and/or is running because its resident set is very likely to be allocated at that node's memory region. The kernel knows which physical resources (memory, processors, I/O channels, etc.) are placed in each node and uses this information to optimize performance. The objective is to always allocate resources and/or dispatch processes at its home node, avoiding remote accesses.

Even though affinity is important and extremely relevant in this context, the operating system seeks a balanced distribution of its work load. If the OS excessively enforces locality, it may overload certain nodes while others remain idle. The OpenSolaris kernel is aware of this issue and tries to schedule threads to different nodes, maintaining load balance across the system when necessary.

Finally, OpenSolaris maintains data structures to represent the hardware topology and its latency values. If the physical configuration changes during runtime - a dynamic reconfiguration event - the new topology is identified and incorporated into the system.

3.1 Abstraction

OpenSolaris introduces the concept of *locality groups* - or *lgroups* - an abstraction to represent sets of hardware resources that are next to each other in terms of access latency. Each group has at least one processor and possibly memory pages and devices associated with them [2].

Representing groups of processors and hardware resources grouped together based on different latency values allows a faithful modeling of NUMA architectures. From simple configurations with four nodes (Figure 3a), to more complex systems (Figure 3b), latency is the factor that separates nodes from each other. One can see closer nodes as part of a smaller group, with lower remote latency, and farther nodes as part of a wider group, contemplating more resources but with higher access times.

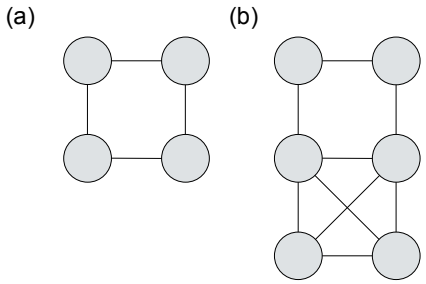


Figure 3: two examples of NUMA configurations

As of build 88 of the OpenSolaris kernel (May 2008), the topology identification uses the ACPI tables discussed earlier. Before that time, the kernel would probe latency values from the *northbridge* (or *memory controller hub*, depending on the manufacturer) registers for x86 systems. These registers have information about where a node's memory range lies within the overall shared address space and, again depending on the manufacturer, can also inform the access latency to each node.

SPARC® architectures are configured so that a node can access every other node's memory with a single remote access[1]. This means that there are only two level of latencies across the system, local and remote. Therefore, the topology is currently much simpler on such systems.

The latency times discovered will form a *latency topology*, a hierarchy represented as a directed acyclic graph similar to a tree. This modeling allows the scheduler to easily navigate the hierarchy when searching for the optimal lgroup in which to place a thread [2].

The first level contains the *leaf lgroups*, representing hardware resources within the same local latency of each other [5]. The topmost level contains all the system's resources, the *root lgroup*, that has the highest latency values and represents the entire system. The intermediate nodes identify the next level of latency access costs, based on hop distances from the current node, that are less expensive than accessing nodes in the root lgroup. Figure 4 exemplifies a four node configuration and its lgroup topology.

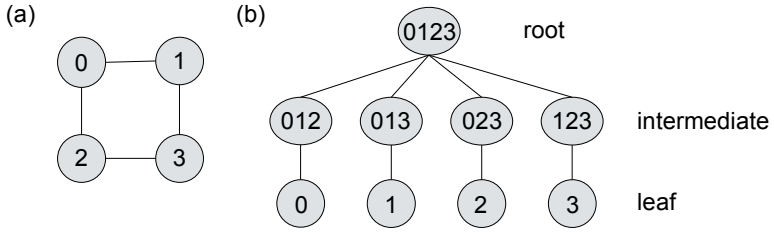


Figure 4: (a) four node NUMA configuration and (b) its respective lgroup hierarchy

Every thread is associated to a home lgroup upon creation, based on the number of threads of the parent process, how many lgroups the process is spread over, and the size and load of each lgroup [2]. The kernel will always try to dispatch and allocate memory for a thread at its home lgroup to avoid remote accesses. If the lgroup is saturated or load balancing becomes an issue, it will travel upwards in the lgroup hierarchy and try at the next level.

3.2 Scheduling

The OpenSolaris kernel relies on the lgroup topology for scheduling decisions, optimizing for locality while seeking an even load distribution. The dispatcher will first try to have a thread execute at its home lgroup, or as close as possible to it when saturated. The saturation threshold is indicated by a user-adjustable system variable. When the lowest load among all the lgroups across which the process is spread exceeds this value, the system will place the next thread at a different lgroup.

The scheduler uses the lgroup hierarchy whenever it needs to search a different lgroup for a thread to run. It starts by looking into that thread's home lgroup, then upwards toward the root lgroup. As it distances itself from the leaf node (the home lgroup), the scheduler considers more and distant resources. Figure 5 exemplifies this behavior [5].

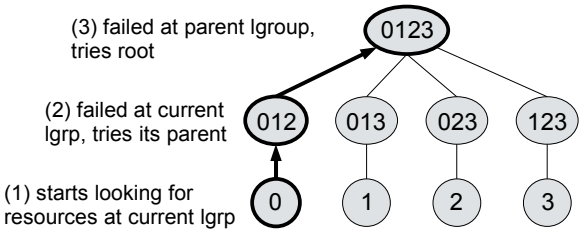


Figure 5: scheduling with the lgroup hierarchy on a four node, ring connect system

The execution of a thread on a different lgroup than its home lgroup is called remote execution, and happens when resources at the home lgroup are limited, load balancing comes into play or when the user explicitly calls for it through a programming interface or command line tool.

[1] This characteristic is likely to change as NUMA topologies become more complex.

A thread's home lgroup will not be modified in the event of a remote execution, causing the scheduler to always try to dispatch its initial lgroup. The thread's home lgroup can only be modified by the user when explicitly changing the thread's affinity.

Dispatching a thread to its home lgroup is the key concept employed by the OpenSolaris scheduler on NUMA machines, it demonstrates locality awareness while keeping a global view by maintaining load balance across nodes.

3.3 Memory management

Much like the scheduler, the OpenSolaris memory management subsystem uses the lgroup topology to optimize performance in a NUMA system. Again, the objective is to avoid remote accesses, and the home lgroup is a key element to that.

When a thread allocates memory, the kernel assigns virtual memory and only maps physical memory when that space is first accessed. Since we're interested in having memory allocated as close to the thread as possible, the system will map physical memory on the lgroup of the thread that first accesses it, taking into account that the dispatcher will always try to run a thread on its home lgroup. This is called *first touch* policy [2].

When a program requests memory space for multiple threads residing in different lgroups, it's interesting to distribute this space among different lgroups. In this case, OpenSolaris employs the *random memory allocation* policy. When dealing with multiple threads, it's better to have most threads accessing local memory than all accessing a centralized remote address space. The random policy spreads memory across nodes, distributing the load across many memory controllers and bus interfaces [2].

The first touch policy is the default policy for private memory allocation in OpenSolaris, whereas random is used for shared memory.

3.4 APIs

The MPO framework exports an API called liblgrp(3LIB) that allows developers to better understand the behavior of an application as well as to optimize it for a NUMA system. The existing calls can be divided into informational routines and routines that advise and explicitly affect scheduling and memory allocating decisions.

The main informational routines report a thread's home lgroup, it's memory allocation map, the number of lgroups and their latency values, as well as an lgroup's affinity. These routines are implemented through cookies, created as a kernel snapshot of the system topology at the time of the call. Most informational routines take a cookie as an argument.

The developer can explicitly change a thread's lgroup affinity, causing it to be dispatched differently and potentially migrating it's data. The developer can also provide hints to the kernel as to how an application is expected to behave. This can be done through a liblgrp(3LIB) routine, a shared object (madv.so.1(1)) or the plgrp(1) command line tool.

There are also a series of system variables that can be tuned to improve performance, either through (k)mdb(1) or the /etc/system file.

3.5 Tools

OpenSolaris provides two command line applications: lgrpinfo(1), that displays information about locality groups and the latency topology, and plgrp(1) to display or set the home lgroup and lgroup affinities for one or more processes, threads, or LWPs [4]. Detailed information for both applications can be found on their respective main pages. Figure 6 displays the output of lgrpinfo(1) for a two node NUMA machine.

```
# lgrpinfo -Ta
0
|-- 1
|   CPU: 0
|   Memory: installed 2.0G, allocated 168M, free 1.8G
|   Load: 0.951
|   Latency: 59
`-- 2
    CPU: 1
    Memory: installed 1.9G, allocated 265M, free 1.7G
    Load:    0
    Latency: 59

Lgroup latencies:

------------
  | 0 1 2
------------
0 | 96 96 96
1 | 96 59 96
2 | 96 96 59
------------
```

Figure 6: output of lgrpinfo(1) for a two node NUMA system

There is also pmadvise(1) that applies advice about memory to a process using madvise(3C) and proc(4). Unlike madv.so.1(1), pmadvise(1) allows the user to apply advice to a specific subrange at a specific instant in time rather than applying advice to the segment(s) during the entire execution of the process.

The MPO framework also provides several statistics about the memory allocation behavior of the system. These can be accessed through the kstat(1M) command. It is also possible to take a look inside the kernel structures and see the values of related lgroup information with the Modular Debugger, or mdb(1). Another resource, dtrace(1M), can be used to track down application behavior and further investigate its performance.

4 Implementation Overview

The MPO framework is implemented in the OpenSolaris kernel through a platform-dependent layer and a common layer, which interfaces with the rest of the kernel subsystems. The platform-dependent layer is divided into x86 and SPARC, each implementing architecture-specific routines that address details of their respective hardware family. This section provides a brief overview of the initialization process and the main modifications to the dispatcher and the virtual memory subsystems.

4.1 Topology identification

The kernel creates the basic lgroup structures early in system boot and, on x86 systems, reads the information contained in the SRAT and SLIT tables. If these are inconsistent or not present, the kernel falls back to probing the hardware for similar information. On SPARC, this procedure varies according to the architecture. For sun4v systems, latency groups are defined by memory-latency-group nodes in the Machine Description [5], which is read during system initialization. Each SPARC architecture implements a subset of the platform dependent code according to its own specifications.

After initializing different kernel subsystems, such as the virtual memory, accounting and interrupts, the system executes platform-independent routines of the MPO framework. This includes adding lgroup related information to the various kernel structures such as CPUs, processor groups and partitions, as well as creating accounting structures for lgroups. It will then call platform dependent code to probe for the latencies of each node.

It is important to note that, in case there are any errors in identifying the configuration of the nodes in a NUMA machine, the kernel will fall back into a UMA configuration. This will be the same topology used for regular, non-UMA machines. It consists of a single root lgroup, under which all resources are located.

4.2 Dispatcher and Virtual Memory

As discussed earlier, the dispatcher and virtual memory subsystems are aware of NUMA systems and take advantage of the latency topology when dispatching/allocating memory for threads. This is accomplished by consulting the lgroup topology and its policies whenever these subsystems make a dispatching/allocation decision.

The lgroup code contains routines that, given information about the thread for which resources are required, will indicate from where to allocate memory or to which processor a thread should be dispatched.. For the VM system, that means consulting the lgroup code when allocating memory and when dispatching a thread for the scheduler. These routines will take into consideration various factors which include the thread's home lgroup, the allocation policy for the process or memory region, and the system's current load indicators.

5 Where NUMA is going

NUMA architectures have been established as a valid alternative to bridge the gap between large clusters and simpler multi-processed machines. Over the years, OpenSolaris has developed a solid framework to support this model and offer the end user the means to optimize performance even further.

However, as the industry and the technologies evolve, we are starting to see some interesting trends both in design and in consumer demands. Among these, we are seeing higher CMT density, the introduction of complex topologies and interconnect systems, the need for more observability, virtualization and power efficiency. This section provides an overview of these new challenges.

5.1 NUMA I/O

NUMA I/O configurations are becoming more and more common. We now see systems where I/O channels are connected only to a subset of the nodes, as exemplified in Figure 7. This introduces access latencies to I/O operations as well and raises interesting questions. For instance, how should the kernel I/O subsystem be modified to explore this characteristic? Should the entire subsystem be moved to these I/O local nodes or just certain parts, such as buffers? Should we schedule I/O intensive applications to these nodes?

We have to pay attention to the fact that the kernel is loaded during system boot and resides mainly in the first node - where the *boot CPU* is. Therefore, the I/O subsystem is located in a node that might not have the fastest I/O connections.

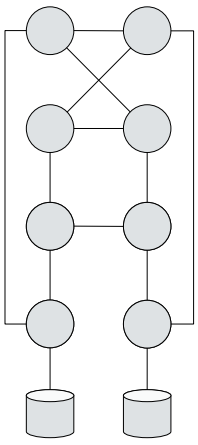


Figure 7: eight node configuration with NUMA I/O

5.2 Observability

Modern operating systems such as OpenSolaris maintain internal data structures for performance and system accountability. In most cases, the OS does not take advantage of hardware performance counters. This includes interconnect performance counters that indicate valuable information such as transfer ratios between nodes and statistics related to the cache coherency mechanism.

Extending the kernel and adding this functionality is not a simple task, as these counters are platform dependent. Creating support for a wide range of hardware manufacturers and models incur a considerable effort. The OpenSolaris kernel is currently not monitoring these registers, although there are projects to incorporate such data.

5.3 Virtualization

Perhaps one of the most important trends seen in the last few years in the IT industry is virtualization. It has presented a solution to many requirements of today's business, such as the consolidation of many physical machines into fewer machines that continue to provide a vast array of services with lower costs by hosting virtual operating systems. However, NUMA architectures have not been properly addressed by today's virtualization systems. The question remains as to which operating system should optimize performance when running on a NUMA machine, the host or the guest?

Most, if not all, host operating systems do not export the latency topology to the guest. But the hosts themselves have not shown optimizations for the underlying NUMA architecture. At the same time, it's unclear what the guest OS will be able to do if the host decides to export such characteristics, since the guest would require interfaces to the host subsystems in order to optimize performance for locality awareness.

5.4 Power management

Today's data center economics are demanding that systems offer not only high performance but also high energy efficiency. Traditional multiprocessors are faced with the dilemma of how much of the system to keep active and how much to power down and still provide a good performance:power efficiency ratio.

NUMA architectures add a new level to the question of what parts of the system to maintain active during idle periods, as the cost of remote accesses is added to the cost of bringing processors back to a fully operational state. For instance, when facing an idle period, should the operating system turn a portion of all CPUs off across the entire machine or should it turn a number of nodes completely off, while keeping the rest on?

6 Conclusions

NUMA architectures have been and continue to be an area of active development, presenting interesting questions and challenges for both operating system and application developers. OpenSolaris has a solid framework to support NUMA systems, and has proven itself to consistently improve performance in such environments. However, the evolution of NUMA machines and the new challenges presented by the industry have to be addressed in order to remain an OS that offers great performance to these systems.

Finding the solution to many of the questions ahead requires not only good designs and projects that have a clear understanding of all the pieces involved, but also a good deal of experimentation and analysis. These difficulties present not only challenges, but a wide space for innovation and creativity.

7 Literature

[1] Michael J. Flynn. Some Computer Organisations and Their Effectiveness. IEEE Transactions Computer, Vol. C21, pp. 948, 1972.

[2] Richard McDougall and Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. 2nd Edition. Prentice Hall, 2006.

[3] Advanced Configuration and Power Specification. Revision 3.0a. 2005

[4] http://www.opensolaris.org/os/community/performance/numa/

[5] http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/sun4v/os/mpo.c#47

Translation of OpenSolaris

Petr Tomášek, Róbert Malovec, Aleš Černošek
Sun Microsystems
V Parku 2308/8
148 00 Praha 4
Czech Republic
Petr.Tomasek@Sun.COM, Robert.Malovec@Sun.COM, Ales.Cernosek@Sun.COM

1 Introduction

1.1 Goal of this Document

The goal of this document is to introduce the globalization issues to the whole OpenSolaris community and describe the tools and services that Sun's Globalization department will offer to the open-source communities sponsored by Sun, such as OpenSolaris.org (OS.o). We would like the community to evaluate the tools available for translation and we will use the feedback to support only the tools and processes that are working and widely accepted by the community. Our long-term goal is to enable community translation of all Sun-sponsored open-source projects.

1.2 Introducing Sun's Globalization Department

The Globalization department of Sun Microsystems is responsible for globalization of selected Sun products. With the introduction of OpenSolaris it became clear that Sun's Globalization department will play an important role of the provider of globalization know-how to the OpenSolaris community.

It is important to note, that although there is only one Globalization department at Sun providing its services to other internal or external partners, the globalization of various Sun products differs substantially and the process of OpenSolaris community involvement outlined here does not apply to OpenOffice.org, NetBeans or other Sun-sponsored open-source communities. The differences find their roots in the way each open-source product has evolved from the beginning and how the developer or user base has been formed. That said, the globalization of OpenSolaris shares a lot of tools and practices with the Solaris operating system as the OpenSolaris source code roughly corresponds to Solaris 10 operating system. One good example is the selection of "core languages" -- a set of languages fully supported by Solaris -- which propagates to OpenSolaris as well. Sun's Globalization department does not specialize in the core languages only, we develop, integrate and test support for as much languages and territories as possible (a pair of language and territory is called a locale, such as en_US). You can get the list of the supported locales and also many other technical documents about globalization of software applications at the Software Globalization Technical Topic page at Sun Developer Network (SDN) that is maintained by the Globalization department [9]. The site is mainly aimed at developers to provide them with information and resources needed for internationalization and localization (i.e. globalization) of their applications. We also provide a technical support by our globalization experts on topics not covered by the articles published at SDN.

1.3 Software Globalization Today

We use the term globalization (a.k.a. g11n) as an umbrella term for both localization (l10n) and internationalization (i18n). Internationalization is usually defined as the process of preparing the software application for use in various locales, i.e. in various territories by various nationalities speaking various languages and using various character encodings for text input and output. For example preparing an

application for use by Brazilians speaking Portuguese requires internationalizing the application into pt_BR locale. Localization is then the actual translation (t9n) of an internationalized application into a specified locale. The scope of internationalization is not limited to language and character encoding, it includes any aspect of cultural difference, but the most common are language, alphabet, calendar and currency.

Internationalized applications which are ready for translation usually use the concept of resource bundles (this term is used in the Java programming language). A resource bundle stores all data to be translated in the original language and it is separated from the application. The application is locale-independent, because it does not contain any locale-specific data. During localization, the original resource bundle is translated to various languages and each resource bundle is uniquely identified by the locale name, such as en_US for US English or de_AT for German in Austria. Based on the user's preference, the application chooses the preferred resource bundle and the user interface (UI) appears to be translated into the preferred language.

As you can see, the actual translation of resource bundles is a separate process, which is nowadays being outsourced by software developers. The main advantage of translation outsourcing is cost reduction, of course. The usual argument for outsourcing applies here as well (cost savings due to concentration on the core business and leaving the scarce resources for tasks with higher revenue generation). But the translation vendor is significantly reducing the cost of translation. The translation industry utilizes economies of scale by using translation memory (TM). Translation memory is more than a dictionary, because it stores not only words, but also groups of words called segments. Segmentation splits the original text into parts, which still possess enough context to translate the segment correctly. A segment can be a sentence or even a paragraph. For structured file formats, such as XML, the translation memory also separates the textual content from the XML markup.

There are many providers of computer-aided translation (CAT), but very few of them are free or open-source, because the concept of using community for translation is still in its beginnings. But the potential of open-source community contribution is promising. What is more important, it allows the software developer to let the user decide what language to support and to what extent, which is an optimal solution.

2 Challenge of OpenSolaris Translation

2.1 Translation Model

One of the big challenges of OpenSolaris translation is the fact that the translation is taken from different sources based on the following major components:

- JDS (Java Desktop System) -- translations are taken primarily from GNOME.org and Mozilla.org (Firefox and Thunderbird) communities.
- OpenOffice -- translations are taken from the OpenOffice.org community.
- Core OpenSolaris components -- contain Operating system and Networking (O/N or ON) messages, e.g. ZFS, Package Manager, Caiman (installer) etc.

Hereafter, the article describes the translation approach of core OpenSolaris components. Translation process of other parts is not covered in this article and it is described and driven by their particular communities.

2.1.1 Translation to Core Languages

Sun supports translation of OpenSolaris into 11 languages: French, Spanish, German, Swedish, Italian, Brazilian Portuguese, Russian, Japanese, Korean, Traditional and Simplified Chinese. All types of content (such as documentation, UI messages or on-line help) are translated into the 11 core languages.

However, this fact does not imply that the system is not ready for translation to other languages. The only requirement for adding a new language is its locale support (or internationalization) -- the system supports more than 50 languages and the number is growing every month.

Besides the supported languages, there are also many partially translated languages, e.g. Polish, Czech, Slovak, Bulgarian, Hungarian. The number of translated languages differs in each open-source community. For the OpenSolaris community, the support is still not very high. Only Czech and Slovak localizations have been delivered recently. Bulgarian and Greek localizations are coming soon.

2.1.2 Translation Scope

From the localization perspective, there are many aspects of project translation. One can translate:

- off-line or on-line documentation
- user interface (a.k.a. SW messages)
- web pages and wikis

We are focusing on translation of two major parts of the project in this article: SW messages and documentation.

2.1.2.1 Software Messages

There are different types of resource files to be translated depending on the programming language used. OpenSolaris uses various formats which are easily identified by the file extension:

- .po (PO stands for Portable Object): files used mostly in C/C++/shell-script programs
- .properties: plain-text resource bundles used in Java applications
- .java: resource bundles to be compiled as Java classes
- .txt: OnlineHelp
- .html: OnlineHelp
- .xml: documentation in SolBook format

2.1.2.2 Documentation

Documentation covers many types of books, from learning materials to user and developer guides. Originally, SolBook 3.5 XML has been used as a source format for documentation, but from G11n perspective this format is converted to standardized XLZ format. Throughout this document we are mainly focusing on translation of system message strings, however our concept can be used for translation of documentation, too. More detailed information about documentation translation can be found in How to Join OpenSolaris Translation Project document [8], provided by the OpenSolaris Documentation Community. Those files can contain translation memory and can be translated in any system with XLZ format support, e.g. Open Language Tools [6].

2.2 Message Workspace

The original (English) and the translated messages are located in the OpenSolaris code workspace. However, the location may be different for specific projects and it is necessary to know it in advance. There are three examples of OpenSolaris workspaces for localized messages:

ON messages (almost all command line tools) and Installer messages are placed in G11n team workspace [10]. You can use the Mercurial source control management system to check it out:

```
hg clone ssh://anon@hg.opensolaris.org/hg/nv-g11n/messages/ messages
```

One can find here for example:

- GUI (Caiman) installer messages [11]
- ZFS GUI and CLI messages [12]
- Command line tools (e.g. cp, rm, date) [13]

On the other hand, desktop tools translation is stored in JDS team workspace. Let's see some examples:

- IPS GUI (Package Manager) [14]
- Trusted Extension for OpenSolaris [15]
- JDS-specific messages (e.g. menu) [16]

Regarding the documentation (on-line helps are part of workspaces together with the original source code), the sources are located in documentation team workspace directly. You can download the books sources in XML format [2] or directly from G11n workspace [17].

From the examples above you can see that the major challenges of OpenSolaris translation are:

- the source files for translation are stored in different locations

- the source files use different file formats, which the translator needs to understand before she begins the translation
- there are different ways of translated content delivery

Therefore manual direct translation by editing the source files off-line seems to be quite difficult.

Next chapter introduces new approach to UI and documentation translation. It overcomes all issues mentioned above and uses the benefits of having rich functions such as translation memory, text segmentation and so on.

3 Community Translation Process

3.1 Current Status

Currently we are using Translate Toolkit & Pootle [3] for community translation. Pootle is a nice and simple web-based tool which can be used for translation of PO and XLIFF message files. As you already know, the repository of OpenSolaris text resources uses various file formats. That's why the files have to be preprocessed first before the integration to Pootle project workspace takes place. Translate Toolkit contains conversion scripts to do that for some file types, but not all of them work well enough. Especially the XLIFF files are difficult to handle and there is no option to work with the XLIFF files with multiple message files inside.

The most common file format in OpenSolaris is the PO [4]. The PO file has a very simple structure that consists of separate pairs of msgid and msgstr variables:

- *msgid* – stands for an original (English) string
- *msgstr* – stands for translated string according to the *msgid* variable

The problem is that two different standards of PO format exist today: GNU and Uniforum format. Pootle can handle only the GNU format. Unfortunately, Sun's PO files are created in the Uniforum standard. The main difference between those two standards is the use of white-spaces, lines and escape sequences [5]. It means that we need to use conversion tools also for the PO files. Translate Toolkit does not include such a conversion routine, so we were forced to make one by ourselves.

Pootle also includes its own implementation of a translation memory. It has an ability to show related translations for a translated string in the translation page. Also the terminology can be suggested from a terminology project in the real time. We also have to mention that Pootle is good for translation of system messages, but processing of the documentation is not very good. There are no routines for splitting a huge text into smaller pieces.

3.2 Translation with WorldServer

Sun's Globalization department and their vendors are internally using Idiom WorldServer for message translation. WorldServer is a commercial CAT web application which is used for translation (automatic and also manual) of messages and documentation. The advantage of using one translation tool is in its translation memory. Translation memory is a set of translated fragments which can be used for automatic translation of new messages. WorldServer right now handles translation memories for core languages only. Translation memory is used for pre-translation of new messages so translators need to localize only those strings which were not leveraged in that process. Our suggestion and plan for the future is that WorldServer will include translation memories for every supported language translators will be both translation vendors and open-source communities. This will rapidly decrease the amount of work required for manual translation.

Community has a big potential which could be also used for localization of Sun's products into many languages and, in the same shot, for filling up the translation memory. Translation Memory is growing automatically with the new translations. We would be very happy if we could enable every community member to work directly in WorldServer. But this is not possible, since WorldServer is used for internal Sun purposes only and cannot be directly accessed by anyone outside Sun Microsystems company.

3.3 Community Involvement

3.3.1 Community Translation Work-flow

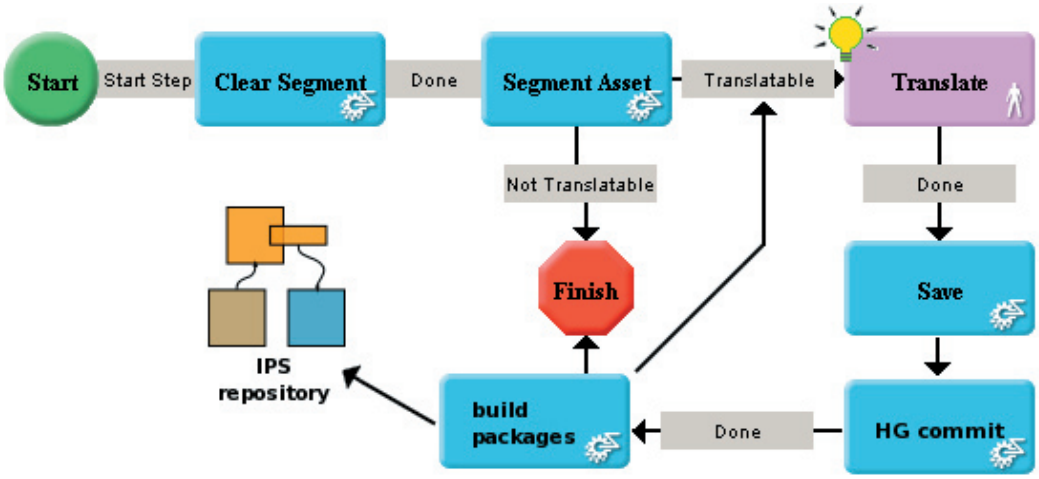


Figure 1: Scheme of Community Translation

As we said before the whole translation process is directed by WorldServer. Here the program manager (PM) launches the project which consists of message files to be localized. To start a WorldServer project we have to define files for localization, the target language of our translation and the project's work-flow. It is good to realize that the source files in the project are never edited by WorldServer which means that we can run multiple projects using the same set of source files in the same time. The Picture 1 illustrates the community work-flow used by WorldServer.

1. Community translation work-flow steps:

Start: At the beginning PM chooses the files for translation and the target language. Optionally the source language can be chosen, but the default one is English.

2. Clear Segment: Project's cache is cleared to force translation server to reload source files. (It is important only when restarting the project to be sure that up-to-date message files will be used instead of pre-cached ones from previous project's work-flow.)
3. Segment Asset: Message strings are segmented and pre-translated by WorldServer using its translation memory.
4. Two options:
    1. Not Translatable: If all the strings were successfully leveraged from translation memory, project's work-flow is automatically finished and jumps to step #10.
    2. Translatable: Otherwise project is shown in the Community Translation Interface and community members can start localizing (Translate).
5. Translation work-flow stays in state #4.2 until program manager decides to terminate it. This can be also done automatically at a specific point of time.
6. Translate-Done: When the translation checkpoint has been passed, messages are physically stored onto the disk (Save).
7. HG commit:
    1. Conversion scripts for filename's renaming are executed. It is needed because some files in the OpenSolaris repository have filenames dependent on the target locale.
    2. Newly created and changed files are committed to the OpenSolaris repository running on Mercurial.
8. Build packages: Routines for package creation are executed.
9. IPS repository: created packages can be easily published into the IPS repository. Our Globalization IPS repository used for the purposes of this paper can be reached at http://www.sunvirtuallab.com:11001.
10. After everything is done, translation work-flow jumps to translation state #4.2, or it can be

terminated.

11. At the very end the project is successfully terminated.

From our perspective the most critical steps from this work-flow are:

- translation
- submission of the translated files into the OpenSolaris repository
- creation of an IPS package

3.3.1.1 The rest of the steps are standard functions which WorldServer takes care of. Let's take a closer look at the critical steps.

Translation with Community Translation Interface

Community Translation Interface (CTI) [18] is a web application developed by Sun's Globalization department to bring up the functionality of WorldServer [1] and its Translation Workbench editor to community members since WorldServer and its partials are not a free software. It lets the community members to select a product (e.g. OpenSolaris), target language (e.g. Czech) and a particular message file or product's component (e.g. ZFS). CTI is currently still under development and the functionality is limited.

The current CTI Alpha allows you to choose a particular WorldServer project and any of the files the project contains. The original text strings are shown in text fields on the left-hand side and the translated text strings are to be inserted to empty text areas on the right-hand side. Translated texts, defined in the leverage process using translation memory, are already filled in.

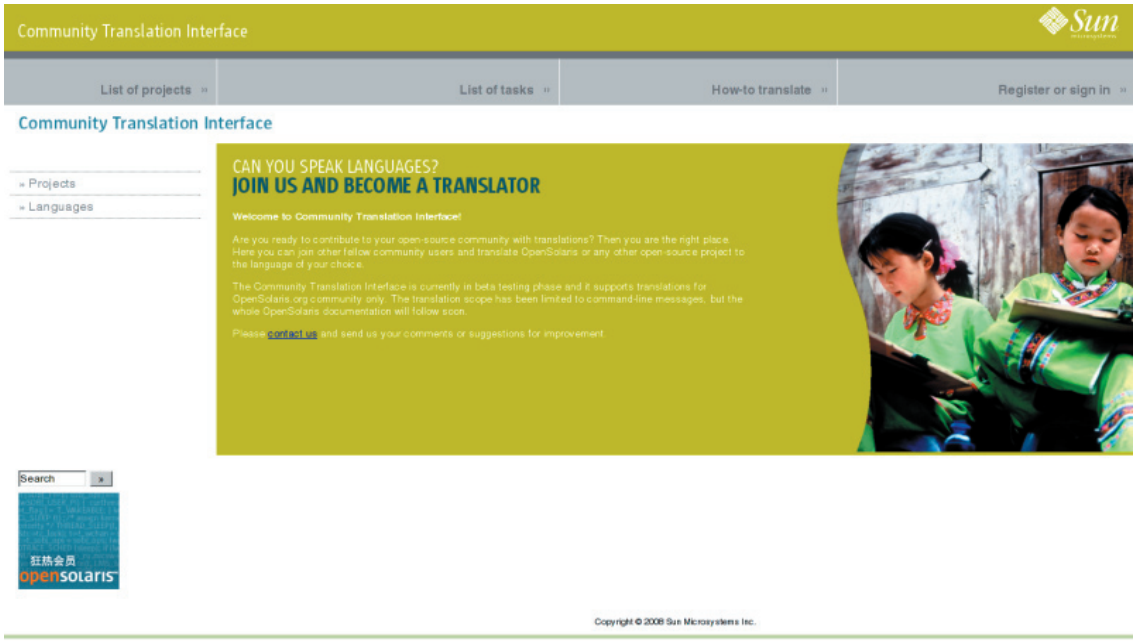


Figure 2: Welcome page of the Community Translation Interface

The final release of CTI will feature authentication routines, roles management for defining project's community leader and team members working on particular project, search engines which would allow users to search in set of translation strings, files and projects and also an ability to create testing package with translated messages instantly for personal testing purposes.

3.3.1.2 Submission to the OpenSolaris Repository

As is mentioned in the Community Translation How-To [7], system message translations are stored in the OpenSolaris repository in a predefined directory structure. Every locale has its own space and file structures are

the same for every language. Submission of new translations is not difficult. The existing directories represent the locales containing some translated files. Translations for new locales can be simply added by creating a new directory structure which is a copy of en_US locale.

The submission process consists of the following two steps:

1. rename the filenames
2. upload the files to the repository

Some resource files have wrong filenames which include a locale name (e.g. server_en_US.properties). It is obvious that when the file gets translated, the filename will not match its locale. All filenames should be renamed first in order to remove any locale references.

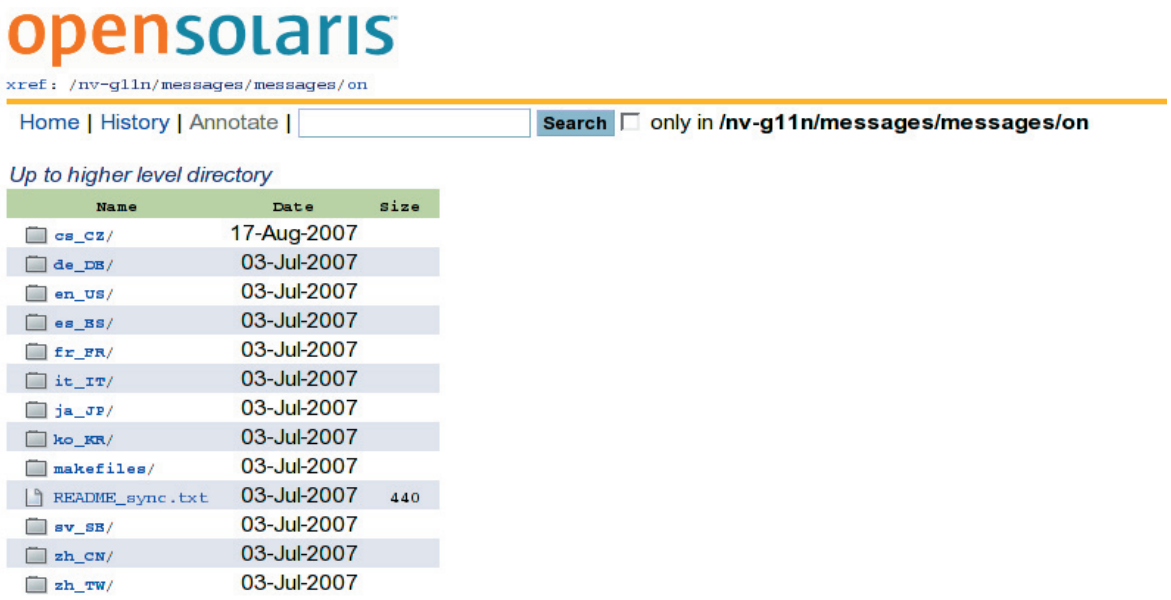


Figure 3: List of present localizations for ON system messages

The submission has to be done manually for now since there is no implementation of Mercurial repository connector for WorldServer to do that automatically. This is solved by simple bash script which takes care of files submission into OpenSolaris repository.

3.3.1.1 Creating and Publishing IPS Packages

OpenSolaris Common Work Space (CWS) is split into small parts. For successful compilation and creation of packages we need to check out at least three of them:

- Globalization workspace
  - accessible at http://src.opensolaris.org/source/xref/nv-g11n/g11n/
  - includes source files and package prototypes which are needed for OpenSolaris g11n workspace compilation
- Messages workspace
  - accessible at http://src.opensolaris.org/source/xref/nv-g11n/messages/
  - includes additional source files and message files with localization; messages translated by community are stored in this workspace
- Globalization-closed workspace
  - this workspace is not accessible to public
  - includes additional Sun's source files needed for the process of package creation

Before you start compiling and creating the packages the workspace has to be compiled first. This process takes quite a long time. The following steps have to be followed:

1. The above three workspaces are linked together. This is done only once, when the workspaces are downloaded for the first time. You do not need to do that again after you update the workspace.
2. The Messages workspace is compiled. Compilation of the Messages workspace has some dependencies in the Globalization workspace and Globalization-closed workspace. This is why we needed to download them as well.

3. Creation of a package with newly translated strings.

After this is done, any partial recompilations of this workspace are much faster. Created packages can be simply downloaded by the user for testing. Also we can simply publish created packages to some IPS network repositories for a better distribution.

4 Conclusion

Although it might be easy to imagine how to handle translation of an open-source software application, when it comes to complex projects such as OpenSolaris, things get very complicated. We have outlined not only the current pain spots of any globalization department all over the world, but also the challenges of OpenSolaris globalization. As the globalization experts of Sun Microsystems have come to a solution, it is now up to the community if it supports it or not. Nevertheless, Sun will always strive to innovate and share the ideas with the OpenSolaris community.

5 Literature

[1]http://www.idiominc.com
[2]http://opensolaris.org/os/community/documentation/feature_docs/
[3]http://translate.sourceforge.net
[4]http://www.gnu.org/software/autoconf/manual/gettext/PO-Files.html
[5]http://wikis.sun.com/display/g11n/Using+Pootle+for+translation
[6]https://open-language-tools.dev.java.net/
[7]http://wikis.sun.com/display/g11n/OpenSolaris+Translation+How-to
[8]http://opensolaris.org/os/community/int_localization/howto-translate.pdf
[9]http://developers.sun.com/global/index.html
[10]http://src.opensolaris.org/source/xref/nv-g11n/messages/messages/
[11]http://src.opensolaris.org/source/xref/nv-g11n/messages/messages/install/
[12]http://src.opensolaris.org/source/xref/nv-g11n/messages/messages/on/en_US/usr/lib/locale/en_US/LC_MESSAGES/ZFS.po
[13]http://src.opensolaris.org/source/xref/nv-g11n/messages/messages/on/
[14]http://src.opensolaris.org/source/xref/jds/ipsgui/po/
[15]http://src.opensolaris.org/source/xref/jds/tsoljdsdevmgr/trunk/po/
[16]http://src.opensolaris.org/source/xref/jds/spec-files/trunk/po-sun/
[17]http://dlc.sun.com/osol/g11n/downloads/docs/current/
[18]http://www.sunvirtuallab.com/cti/

Presto - Printing Makes Easy

Ghee S. Teo[¶]
Sun Microsystems Inc.
Boole House,
East Point Business Park,
Dublin 5,
Ireland
Ghee.Teo@Sun.Com

1 Introduction

Configuring a printer for printing is one of the hardest thing to do on Solaris/ OpenSolaris. In one survey that was conducted shortly after Solaris 10 was shipped, shown that this is the most difficult task for users to accomplish successfully. Prior to this project, there are two ways one can configure print queue on OpenSolaris, lpadmin(1M) which is a command line interface tool and printmgr(1M) which is a Java based GUI tool. Both tools suffered one big short-coming which is to demand user to type in numerous parameters on the command line or text boxes. To address this short coming, Presto tries to pre-fill the configuration data as much as possible as it detects or discover the printer. Once the print queues are added to the system, we provide a Print Manager for the management of these queues as well as print jobs on the system.

A quick survey of various printing management tools is described in the next section, follows by an architectural overview of Presto. Some key features of the OpenSolaris Print Manager are highlighted in section 4 follows by a road map of Presto in section 5. Finally, a conclusion about various things learnt from Presto is included in section 6.

2 A survey of Printing management tools

A quick survey of the printing management tools on Unix/Linux platforms in some ways related to Presto are described in this section. These can be broadly categorized into:

- GUI tools wrapped around command line programs
- GUI tools based on printing libraries

2.1 GUI Tools wrapped around command line programs

The oldest of this is the CDE (Common Desktop Environment) Print Manager (aka. Dtprintinfo(1)). This is a GUI tool which obtains information about print queues and jobs using output of lpstat(1). The use of lpstat(1) was a convenient choice as the tool was developed to worked across the platforms that COSE (Common Open Software Environment) [1] consortium made up of. The main drawback of this tool is that much computation are spent in spawning processes, parsing the data and periodically spawning more commands to refresh the display.

Its close sibling called gnome-printinfo was implemented using GNOME and GTK+ toolkits when Sun integrated GNOME 2.0 into Solaris. This program suffers the same fundamental inefficiency though performance wise is improved slightly as user's configured queues are cached in the user's GCONF database. Both dtprintinfo and gnome-printinfo are user land applications, they are used to monitor the status print queues and print jobs, it has not the ability to create or delete queue.

[¶] This paper will be present by Michal Pryc, Michal.Pryc@Sun.COM.

2.2 GUI Tools based on printing libraries

2.2.1 GNOME CUPS Manager

As GNOME development is picking in momentum and CUPS is becoming more wide-spread on Linux, Dave Camp (Ximian later acquired by Novell) rose to the challenge. He wrote a very nicely designed printer management tools called gnome-cups-manager and gnome-cups-add back in 2003. gnome-cups-add provides a wizard like interface that allows the user to select from a list the type of print queues to create. Though it still required user to type in some options in the wizard like interface. While the hackers are excited about this, Eric Redmond pretty much poured buckets of cold water over it in his article [2]. Many of his critics are valid and some are really hard to make it right still through this day because of the heterogeneous environment printing often has to work. Gnome-cups-manager has been adopted by some distros like SuSE, Ubuntu, JDS Linux. The advantage of gnome-cups-manager over the previous tools are:

- it uses a library, libcups to access printer and job information
- it provides a unified interface to the user and only prompted for root password when privilege operations is being performed.
- It is integrated to the desktop through the notification area.

2.2.2 PrinterDrake

A lesser known Printer configuration tool called PrinterDrake [3] but it is feature rich as described here by Till Kamppeter [4] including:

- Plug and Print for Parallel and USB printerDrake
- Network printer discovery using SNMP
- Fuzzy matching of Manufacturer and model name
- configuration of sharing of CUPS printers
- automatic download of certain firmware if needed

and many more. This is distributed in Mandriva Linux and was proposed to be adopted for Unbuntu, though system-config-printer was adopted eventually. Many of the features are actually implemented by Presto.

2.2.3 System-config-printer

While gnome-cups-manager is getting all the attention, a interesting tool called system-config-printer [5] is being developed by some hackers in Red Hat in Python. Its initial aim is to allow easy configuration of CUPS server both local and remote. It was released first in Fedora Core 6 and has been subsequently adopted by Debian and Ubuntu as of late. It is gradually evolving to auto detect local printer using helper program called hal_lpadmin which utilises hald's capabilities. It has also extended to browse Windows printers using SMB.

In fact, as the tool has gradually developed and maintained as such that it has been adopted by Ubuntu [6] in place of gnome-cups-manager. This is one tool that is actively maintained and developed in the community, for example, a Google Summer of Code project is incorporating the Presto's Printer Group into system-config-printer.

3 Architecture of Presto

Figure 1 shows a high level view of the architecture of Presto. This is a more up to date diagram than the original one [7]. This diagram should help to explain how different bits of the Presto are hang together. The main simplification provided by Presto is removing the needs to manually specifying printers's parameters through auto-detection or auto-discovery. While the auto-detection or discovery should gather as much data for the user, there may be some user intervention to complete information to access the printers correctly occasional.

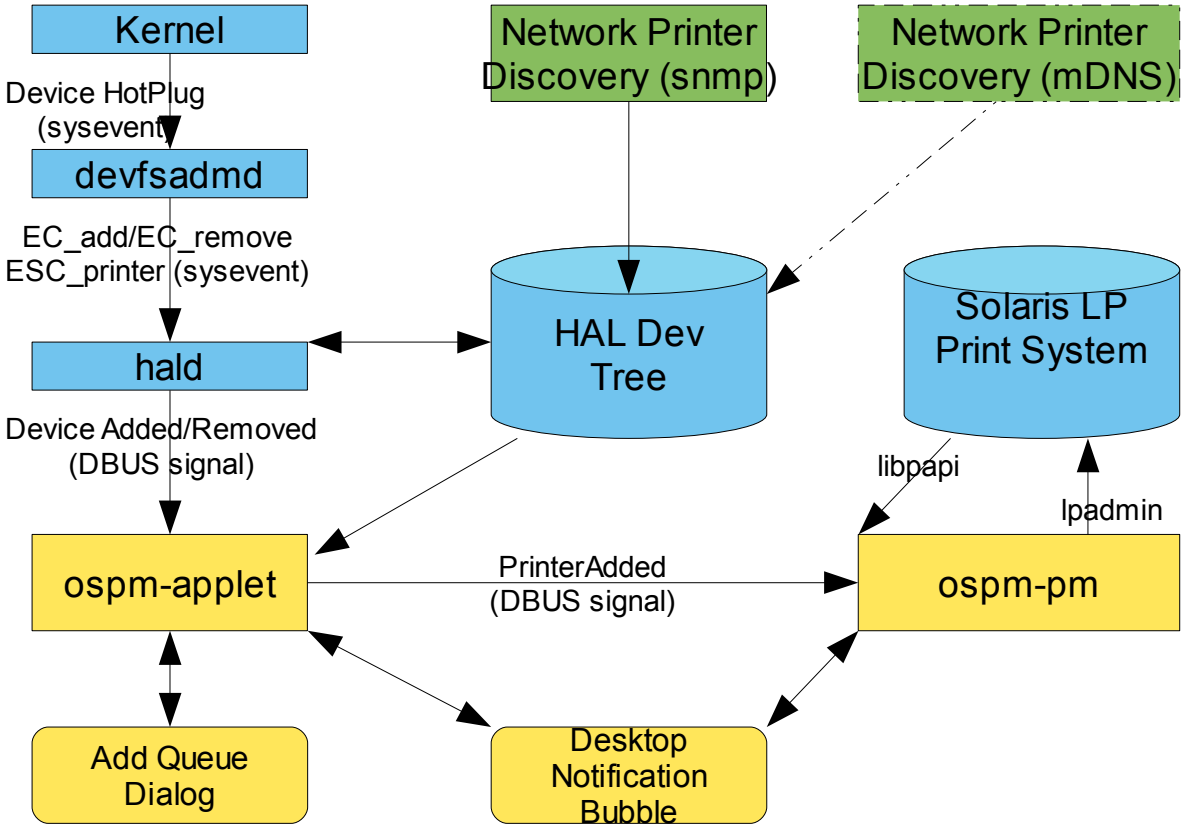


Figure 1: Block Diagram Overview of Presto's Architecture

3.1 Local Printer Auto-Detection

When a user plugs in a brand new printer to his laptop for the first time, the system should find out what make and model of the printer it is and add a queue that contains all the correct information so that the user can submit print job to it. This is simple conceptually and is easier to implement than network printer. In fact, phase I of Presto did basically just that for USB printers.

How it works in Presto is as follows when a USB printer is plugged in:

- the OpenSolaris kernel notices the Device HotPlug event
- the sysevent causes appropriate /dev to be created
- devfsadmd(1M) notices the Printer Add or Remove sysevent, it notifies hald (HAL daemon) [8]
- hald interrogates the printer and populate the HAL device tree with all the relevant data it can get
- hald sends out a Device Added or Removed DBUS signal [9]
- ospm-applet which is a user's session daemon waiting and responding to these signals. Based on the unique udi (Unique Device Identifier) it received from hald, it looks up the rest of the data from the Hal device tree.
- Ospm-applet then displays fully populated Add Queue Dialog for the user to add the queue, the displaying of this dialog is configurable. If it is turned off, the queue will be automatically added.
- Once the queue is added, a notification bubble is shown on the notification area of the panel

Once the above steps are completed, the printer is ready to be used.

3.2 Network Printer Discovery

Most network printers these days are intelligent devices, it often is capable of getting a DHCP address and begin to broadcast itself or listening to requests to print. It varies depending on the manufacturer and model. Presto handles the discovery of network printers by creating discovery agents based on different communication protocols as part of hald add-on.

Broadly speaking there are two communication means that a network printer can be discovered.

1. Printer advertises its presence when plug in. mDNS is an example.
2. Printer responses to a particular broadcast request, these include SNMP and ICMP.

The idea is to have various HAL add-on, each to handle one type of protocol. Given that fact that not all network printers are of equal capabilities, this is a necessary step. This approach though gives the ability of modular development and future extension. Presto has thus far implemented auto-discovery of network printers based on SNMP, a more commonly supported protocol.

These are the steps that Presto goes through when a network printer that supports SNMP is plugged into the network:

- (Assuming the network printer discovery service is enabled, svc:/network/device-discovery/printers:snmp), the hald network printer add-on broadcast a SNMP GET
- Network printer which is SNMP capable would then response to it
- The SNMP agent then populates the HAL Device Tree with the network printer data
- hald detected changes in the HAL device tree and deduces that these are printers, it sends out printer Added DBUS signal to ospm-applet
- ospm-applet added print queues for these printers in the background until these are all added
- ospm-applet pop-ups a generic message as a notification bubble notifying the user that network print queues have been added. Ospm-applet also sends out a DBUS message to the print manager, ospm-pm.
- If the Print Manager is running at the time, it will refresh its view immediately and hence shows the newly added queues. Otherwise, these messages are ignored.

4 Print Manager

As one can see from the architecture diagram many of the enabling technologies of Presto to make printing easy happens behind the scene. Many things that enhance the user's experience though is in what the user sees and interacts with. The real user's experience rests heavily on how the Print Manager is designed and implemented. Some pragmatic rationales that drives the design of the Print Manager are:

- the tool must get away from parsing lpstat's output which is highly inefficient.
- the tool must be able to handle large number of queues reasonably as some environment contains.
- the tool must provide functionalities adequately to replace some existing tools such as gnome-printinfo and Solaris printmgr on the desktop.

Other requirements include:

- seamless integration into the desktop – the tool should have the same look and feel as the rest of the desktop applications. The integration of notification mechanism into the panel's notification area as well as the use of notification bubbles are part and parcel of the seamless integration.

4.1 Some Design decisions and Implementation choices

4.1.1 Implementation Language

When the Desktop was first involved to provide the GUI for Presto back in September 2006, C was chosen as a natural choice because:

- libpapi is the implementation of PAPI on Solaris written in C, with this we got more efficient way to get print queue and job information.
- libpapi is also the client libraries that the OpenSolaris command line utilities based on
- GTK+ toolkit is written in C well suited to implement Desktop GUI
- DBUS and HAL are also available as C libraries.

4.1.2 Print system supported

At the start of the project, Solaris LP is the sole print system that OpenSolaris supports. While CUPS [10] was only an unsupported print system available on the companion CD. So this is another straight forwards decision then. Now that CUPS has been integrated into Nevada build 87, as it is becoming the de facto standard on the Open Source arena, CUPS cannot be ignored except that it comes too late in the implementation of Presto Print Manager. So for the first release of the Print Manager, only Solaris LP is supported.

4.1.3 Linking up of various components of Presto

As the Presto's components spread across the OS starting from kernel to SMF service to user land applications, proper plumbing of these components is crucial. Two of the major plumbing components are DBUS and Sysevent. The Sysevent is one used in the kernel land and DBUS is the one use in user land. DBUS is a light weight inter-application communication mechanism which is ever increasing popular in the Linux Desktop world. It is more widely accepted because it is secure in clearly separating user DBUS daemon and system DBUS daemon, and its API is relatively simple to use.

This the the underlying plumbing that links up between hald and the ospm-applet and notification daemon which eventually pop up the bubble message.

4.1.4 Threads – How many?

The Presto Print Manager is actually a multi-threaded application. As the implementation progresses, we realized that we need more than one thread in the application. The rationale in hind sight is quick simple, we need to look up the state of printers, print queues on print servers, job status and so on. When these numbers are large and the containing bad data, the whole GUI would hang for a long time. The strategy is to make sure that none of the potential lengthy operations are allowed to run in the GUI thread.

After careful analysis, the final conclusion is to have 3 threads. One for the GUI update, one for the updating of the queue states, and the other for the updating of the job states. Mutex locking are used in each of the backend threads to ensure data integrity.

4.2 Why Printer Group Manager?

In order to address the situation when there are lots of printers available to the user, the user could be overwhelmed by the sheer numbers and gives up locating the printer that he had submitted job to. This problem can be resolved by popping up notification on the panel when the job is completed.

Presto Print Manager introduces the Printer group concept by allowing the user to create grouping of queues. The grouping of queues can be carried out by selecting a set of queues from the All Printers group into an arbitrary named group or constructed from a set of search criteria. The selection of queues is known as a Static group, and the latter one is known as Dynamic group.

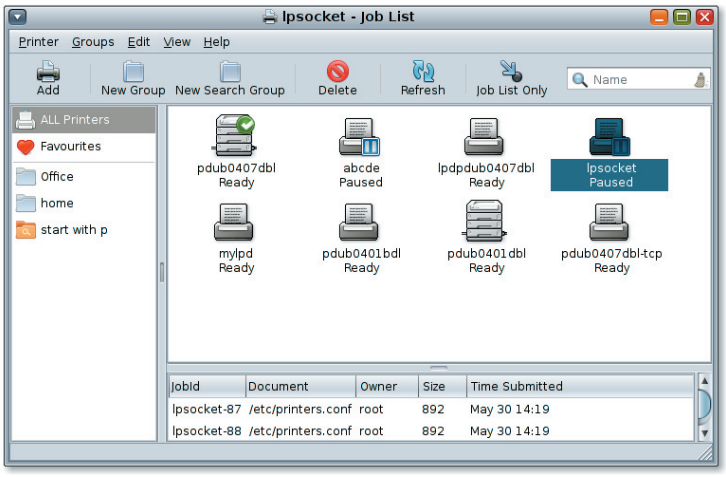


Figure 2: Screen Shot of the Presto Print Manager.

On the left hand pane of the main window in Figure 2 is the Group Manager view. There are 5 printer groups here, namely:

- All Printers – the complete list of printers available on that system
- Favourites – User's favourite list of printer and is the one displayed in the Print Dialog
- Office – A static list of printer named Office
- Home – Another static list of printer named Home
- start with p – A list of printers which is dynamically found based on some search criterion, in this case it contains printer's name start with the letter 'p'

4.2.1 All Printers Group

This is a special group, it contains the complete list of print queues that the user has access to on the system. Since queues are references to real printers, the actual printer may or may not be accessible depending on the network the user's machine is connected to or the type of queue. Local queue will always be visible and available or it can be paused. Remote queue can be inaccessible which is indicated by a red exclamation mark. Default printer is marked with a green tick.

The one thing that is worth noting here, one can select one or more queues here and click on the 'Delete' button on the tool bar, the queues will remove from the system for you if you are on the console. Risky you may said, but if you are proud owner of the laptop, this is most probably what you want. Of course, you will be prompted for confirmation.

4.2.2 Favourites Group

This should contain your favourite list of print queue. This is the list of print queue you will see in your OpenSolaris Desktop print dialog. The goctha is if the list here is empty, you will get the All Printers list in your print dialog which could be a really bad thing if you are using NIS/LDAP printer database. The list could be a ten or hundred long. Of course, this is the time you can stress test the Presto Print Manager.

Sometimes, you may wish to be able to make the Favourites a dynamic group selecting only the colour printers for example. However, this feature is not available because Favourites really mapped to what you have in your $HOME/.printers (printers(4)) and is a feature of OpenSolaris LP.

The print queues shown in this view is also the printer list shown in the GTK+ print dialog that many of the Desktop applications uses including Firefox, Evince, Gedit, Eye of GNOME and so on.

4.2.3 Static Group

This group generally contains a list of printers that one has manually selected from All Printers or other groups. Drag and drop interaction is supported here to move one group to another. A static group can be individually named. A static group should be small in general and ideally created for specific location such as home, client office, special type of printers and so on.

Print queues within a static group or Favourites can be deleted by selecting the queue object and click on Delete button. No confirmation dialog is popped up as the deletion is only from the group within the tool and not from the system.

4.2.4 Dynamic Group

Dynamic printer group meaning the entries within this group is created by some search criteria. These criteria are derived from a combination of the printer's attributes. The simplest form of these is Name of the printer. Search criteria can be a single query or a combination of multiple queries. Due to the limitation of PAPI and as to not impede the user's experience of the tool, we narrowed the single query criteria to:

- Name
- Description
- Make or Model

The primary reasons for not including other type of queries are due to:

- some queries could be extremely slow when the printer set is large
- any query involved individual printer communication will be very time consuming.

- lacking PPD file parsing support in PAPI

The advanced search criteria which allow user to specify multiple queries are listed in Figure 3.

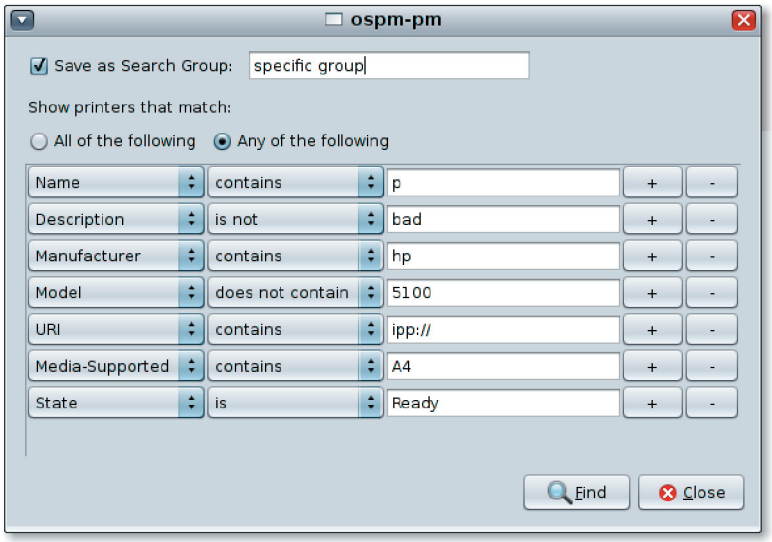


Figure 3: An Example of a dynamic group based on all the possible search criteria for Presto Print Manager

4.2.5 Integration of Groups to GTK+ Print Dialog

In the initial design, the complete Group Manager is purported to be integrated into the GTK+ Print Dialog such that all the different groups created in the Print Manager can be used when user wants to print from the applications. The design is now omitted from the implementation for two reasons:

- For unmanaged network environment, the number of printer may not be big enough to warrant a Printer Group Manager
- The current GTK+ Print Dialog's front end and back end breakdown make it extremely difficult to implement anything graphical intensive outside its existing framework.

However, this is an important part of the Printer Group concept that is worth revisiting in the future.

5 Road Map

5.1 What has been implemented to date?

- USB printer Detection – integrated into Nevada build 69 including desktop integration
- Network Printer discovery service using SNMP – integrated into Nevada build 77, it is turned off by default.
- Presto Print Manager – this is almost feature complete, there are a couple technical issues to resolve in the manual Add queue dialogs and showing the printer's properties. After that, there will be ARC case to fill and integration paper work to be done for Nevada, pre-integration system testing. The target build for integration is Nevada 97.

5.2 What is next?

5.2.1 More Print Services ?

Phase II of Presto should address many of the system wide printer related usability issues. With the integration of the Print Manager, the complete framework is in place to help user to auto detect and discover printers to use using the OpenSolaris LP print system. More network printer discovery services using other protocol such as Multicast DNS (mDNS) or even Windows could be added and fit into the architecture nicely. The addition of these services will be depending on the resources available, demands from the customers and pretty much the management of the Printing group will need to decide upon.

5.2.2 CUPS Support ?

CUPS 1.3.6 was integrated into Nevada build 87 and is now an integral part of OpenSolaris as much as LP is. Since Presto has started long before the decision to integrate CUPS, no support for CUPS is yet available in Phase II of Presto. There is a possibility to extend Presto Print Manager to support CUPS as a backend as well. To do so the following problems need to be tackle:

- Favourites Group support in CUPS – CUPS does not have the notion of user's own list of favourite printers as in $HOME/.printers. This is though no very critical in the Print Manager GUI since it only means one less special group.
- Factoring of code into separate backend libraries – one for PAPI, one for CUPS and so on. This may be harder than it sounds, mainly because PAPI is designed to be synchronous and CUPS is asynchronous. The GUI code may need to be structured differently.

While balancing with tackling these issues, there is the possibility of leveraging Open Source tool such as system-config-printer or gnome-cups-manager. The cost leveraging one of these tools would be just a case of providing a wrapper layer on the desktop. The wrapper layer will simply determine what print system the system is running at the time to launch either Presto Print Manager or one of the CUPS Print Manager. The advantages in adopting an existing Open Source printer management tool is obviously smaller development effort, lower support cost, more mature code base. The disadvantage is lesser control in what to integrate and also having at least 2 very different look and feel of the tools. On the whole, it is possibly a more cost effective approach.

6 Conclusions

The paper provides a favour of what is there on OpenSolaris printing development. It skimmed through Presto and show how it helps to improve the user's experience in an area which is hugely lacking. The technologies used in the improvement may not be rocket science, but the net effect is significant in what the user can now do. Less overhead burden in remembering the make, model, ever more all the strange command line options that one has to supply to create access to the printer. The automated aspect of Presto helps to minimize this.

However, it is not to say this piece of software is done and dusted, there are still many areas of improvement as highlighted in the road map section. After all, the printer management tool is only part of printing related improvement, other areas of work include consistent application print dialogs through out the desktop. This is a hard can of worms to tackle as applications can come from many different sources. The OpenPrinting group though is working hard to address this issue [11] and trying to push the idea through different communities and also having students working on Google Summer of Code to implement the concept.

One bug drawback of Presto is perhaps due to its ability to detect the printer yet not having the appropriate driver and PPD file on the system, the Print Manager would do well to be able to download the driver and PPD files from some online repositories. However, the availability of this feature requires more than just code; it needs communities and OEM efforts to provide drivers and PPD files on-line. This is a harder issue to tackle than programming.

7 Acknowledgements

Presto is the result of hard work of many people, I like to thank them here (named alphabetically): Calum Benson, Evan Yan, Halton Huo, Norm Jacobs and Wendy Phillips (all of Sun). Also want to thank Ken VanDine for his gallant attempt to compile the Print Manager on InsighLinux. Michal Pryc for presenting this paper on my behalf.

8 Literature

[1] CDE, COSE, http://en.wikipedia.org/wiki/Common_Desktop_Environment

[2] Eric Redmond,2006, http://catb.org/~esr/writings/cups-horror.html

[3] PrinterDrake, 2004, http://doc.mandrivalinux.com/MandrivaLinux/100/en/Starter.html/printerdrake.html

[4] Till Kamppeter,2006, https://wiki.ubuntu.com/PrinterDrake/

[5] Tim Waugh, http://fedoraproject.org/wiki/Printing/AdminToolOutline#A_new_printer_administration_tool

[6]Ubuntu, https://wiki.ubuntu.com/SystemConfigPrinter

[7]Norm Jacobs, 2007, http://www.opensolaris.org/os/project/presto/Documents/presto_design.pdf (page 4).

[8]HAL, Hardware Abstraction Layer, 2006, http://www.freedesktop.org/wiki/Software/hal

[9]DBUS, 2008, http://www.freedesktop.org/wiki/Software/dbus

[10]CUPS, 2008, http://www.cups.org

[11] OpenPrinting, 2008, http://portland.freedesktop.org/wiki/PrintDialog