Creating Self-Balancing Solutions with Solaris™ Containers

David Collier-Brown, SunPS Data Center Solutions

Sun BluePrints™ OnLine
June 2005

Part No. 819-2888-10
Revision 1.0
Edition: June 2005

Please

Creating Self-Balancing Solutions with Solaris 10 Containers

Transactions of some kind are an integral part of every organization, and must be completed on time if the business is to operate effectively. Chaos — and damage — can be caused if critical transactions are not handled efficiently. Billing systems, for example, are often under heavy time constraints, and are expected to be able to process the previous day's transactions in an eight hour period. Imagine arriving at work, only to discover the billing application is still processing yesterday's transactions, delaying the sending of invoices to some customers. This behavior often arises when multiple instances of a business-critical program are running — one or more always seem to run longer than the rest.

Performance problems such as these are best handled by multiprocessor systems running multiple instances of the application. Today, IT managers may try to break workloads into chunks and process each chunk with a separate program instance to keep up with demand. They hope to distribute the workload across the instances, to ensure they can keep pace. But what happens when one instance fails to finish in time? What if the business is growing, and every month the number of lagging instances increases? How are system administrators supposed to figure out which instance is going to be late the next time?

Scenarios such as these cause system administrators great pain and give management great cause for concern. If systems fail to process a day's transactions in one day, the business is in trouble. If these transactions are a critical part of the business, like a billing system, accounts recievable falls further behind each day. No matter how management allocates the work, some customers account for more transactions than others, leading to some systems and program instances being overloaded, and others under utilized. If only the instances could balance themselves, and share the work equally, the work would be done in time.

System administrators need to find ways to balance workloads across computing resources. The Solaris™ 10 Operating System includes a new facility, Solaris Zones, that can be used with resource management techniques to create a Container with

which to manage unbalanced load problems. This BluePrint article presents several techniques for dealing with unexpected workload changes, and provides best practices for employing Solaris Containers in this effort.

Changing Workloads Are Inevitable

Balancing the work between multiple instances of a program is a common problem in production data centers, regardless of the platform deployed. In fact, balancing workloads is really part of a larger problem — dealing with change. The most common kind of change in a data center is to have more work to do, to provide more service and better availability while still keeping to cost constraints.

Today, many data centers try to balance workloads much the same way they did during the mainframe era — by allocating a chunk of work to each instance of the program, and hoping the chunks are all the same size and will complete in the same amount of time. Unfortunately, this approach is an invitation to frustration. It rarely works — the inputs are bound to change, perhaps every day.

Recognizing Imbalance Problems

The first step in creating balanced workloads is recognizing this kind of problem when they arise. Often, workload conditions are characterized by the presence of multiple instances, statically allocated work, and dynamic changes to the amount of work needed to be done. While these characterizations describe the situation, they do not describe the symptoms present in the environment. In fact, symptoms are often ambiguous — a program that runs slowly or takes too long to complete — and describe virtually any performance-related problem.

Perhaps the most critical symptom is easiest to spot: one instance of a program runs too long, and at least one other instance of the same program finishes early. Several types of workloads are prone to this behavior, including:

- Billing systems that explicitly configure ranges based on customer names
- Stock trading systems that suffer whenever there is increased activity in one stock, suggesting some form of balancing based on stock ticker symbol
- Geographically distributed organizations trying out a marketing initiative in a single region
- Phone systems that bill based on area code

Any one of these scenarios might indicate an imbalance problem. However, it is up to the system administrator to recognize that multiple instances exist, or that one instance of a program that seems to be running too slowly is, or is not, fast enough to meet demand.

Unfortunately, recognition alone is not enough — the details of the problem must be expressed clearly to management. Management, however, is rarely concerned with underlying technical details. They are concerned with the impact, or cost, of the problem and its solution, including:

- The cost of not meeting defined service levels (SLAs)
- The cost of extra hardware to ensure SLAs are met
- The cost of making any change to the infrastructure
- The total cost of creating a solution, including lead time, manpower and testing

Once an imbalance problem is identified, it is important to:

- Communicate the existence of the workload imbalance
- Explain it is a type of performance problem with potentially good solutions
- Stress that good solutions are those which involve reorganization of the work rather than additional investment.

Your management will prefer solutions which preserve their existing investment, and don't require either rewriting the programs or buying large amounts of additional hardware.

Solutions to Imbalance Problems

Numerous approaches are available to address imbalance problems, some better than others.

- *Throw hardware at the problem*

  The simplest solution for an imbalance problem is no solution at all — throw hardware at the problem. Like many performance issues, imbalance problems can be solved by providing enough additional resources so that the imbalances do not slow down the systems and applications enough to hurt or be noticed. It does cure the imbalance, but it poses a funding problem that management may consider worse that the disease.

- *Tune software until the system runs fast enough*

  Another technique often employed involves tuning applications or databases so the slowest program instance completes in a reasonable time frame. A program that uses a poor algorithm or a subtly incorrect SQL query can be sped up enough

so that most imbalances remain hidden. This approach, however, will fall victim to a heavy burst of correlated transactions, resulting in an instance that runs overtime. Furthermore, constantly tuning the environment through source code changes can be prohibitively expensive.

- *Statically rebalance the load*

  One possible solution — one that has delivered results — is to rebalance the program instances. Typically this task is accomplished manually, based on historical data. For this approach to be effective, it should be performed each week after looking at the load distribution for the previous few weeks. While valid, this is only a probabilistic approach, one that will fail if the workloads vary greatly.

- *Add more program instances*

  Adding more program instances combines the techniques of adding more computing resources with statically rebalancing workloads. Adding one or two program instances reduces the likelihood of an imbalance by a moderate amount, and at the same time, resulting in the programs consuming more machine resources. This works only if those needed resources are available when needed, and are not needed by other programs.

  For example, increasing the number of program instances at a customer site by one or two improved performance overall, while an increase to three program instances slowed down processing. Adding the third instance starved the database the programs were using, slowing the entire system down as a result.

- *Add more instances but manage resources*

  In cases where adding instances starves other programs of resources, we would need to manage the resources the programs could use. This is done with the resource management mechanisms of Containers in Solaris 10, or SRM in Solaris 9.

- *Run many instances and manage resources*

  Building upon the previous solution, system administrators can attempt to run many instances but limit the resources they can consume. This is a time-tested practice: many organizations running mainframes used to assign multiple application instances to a logical partition and managed the number of CPUs in the partition to ensure each instance had sufficient resources.

  For example, a stock trading system could run 26 program instances, one for each letter in the alphabet, and distribute work according to ticker symbol. This would spread out the work well, but 26 instances would try to use a large number of CPUs. This could be done exactly as on the mainframe by creating a domain with a limited number of CPUs for just this program. However, this prevents other

programs from using those CPUs when the billing program is inactive. Using Solaris Containers instead allows the CPUs to be shared while still limiting the amount of CPU resources used by the particular program.

- *Dynamically rebalance workloads and manage resources*

  If it was easy — and cheap — to determine how many transactions exist for each workload group, an equal number of transactions could be assigned to each program instance. If the number of transactions cannot be predicted, it may still be possible to dynamically apportion small chunks of work to several instances of the program and create a means for them to come back for more work when they are finished. This dynamic rebalancing of workloads may be accomplished without modifying applications. Scripts can be written that divide up the work to be done, or a batch queueing system could feed the list of work to the program instance.

  For example, a stock trading system might assign all transactions for stock symbols beginning with the letters A through G to seven instances of the program, one for each first letter. The first instance to complete would then be given transactions for symbols beginning with H, the second transactions beginning with the letter I, and so on. This technique gives system administrators the ability to increase or decrease the number of program instances and the amount of CPU assigned to them. It is possible to find, for example, that the best throughput results when there are twice as many instances as processors, but that no more than eight CPUs worth of power are used.

- *Make the program divide up the work better*

  Some workloads, like those experienced in stock trading systems, have characteristics that make it nearly impossible for all transactions to affect a single entity, such as a stock symbol. This cannot be guaranteed for other types of balancing problems. Cases do exist in which the entire workload can land squarely on a single program instance. When this occurs, the best opportunity for problem resolution lies in changing the algorithm.

  The classical approach is to break the program up into a parent instance that collects the work to be done and a pool of child processes that are handed work to be done in reasonably sized chunks. Changes such as these are not small, nor are they inexpensive. Substantial source code tuning may be required.

  It is often better to dynamically rebalance the workload as previously discussed rather than re-engineer solution unless absolutely needed. If a method can be found to select small enough chunks, staying with dynamic balancing is a reasonable solution.

Selecting the Best Solution

When looking at the solutions outlined above, a pattern emerges. Good solutions provide:

- One mechanism for breaking the work into chunks
- Another mechanism for managing the resources those chunks use to get the job done.

In addition, good solutioins also involve reorganizing the queues of work, without requiring additional hardware or rewriting software. This means that good, inexpensive solutions do exist. The task is selecting a solution that fits *your* particular variant of the problem and minimizes costs, something your management will appreciate.

Introducing Solaris Containers

Containers are a resource management concept that are implemented with Solaris Zones, a general mechanism for simplifying system administration. They provide isolation between software applications or services using flexible, software-defined boundaries. These applications can then be managed independently of each other, even while running in the same instance of the Solaris Operating System. Solaris Containers create an execution environment within a single instance of the Solaris OS and provide:

- *Full resource containment and control* for more predictable service levels
- *Fault isolation* to minimize fault propagation and unplanned downtime
- *Security isolation* to prevent unauthorized access as well as unintentional intrusions

Solaris Containers used Solaris Zones to provide a virtual environment that appears to be a complete, standalone machine or domain with its own hostname, IP address and users. Since each Container acts like a separate operating environment, it provides a familiar environment to system administrators that is easy to manage. Consider the common task of placing two different Web servers on a single server that each want to run on port 80. By placing each instance in a Container, each Web server gets its own port 80 to use. Each Container is also isolated from security problems that may arise in other Containers.

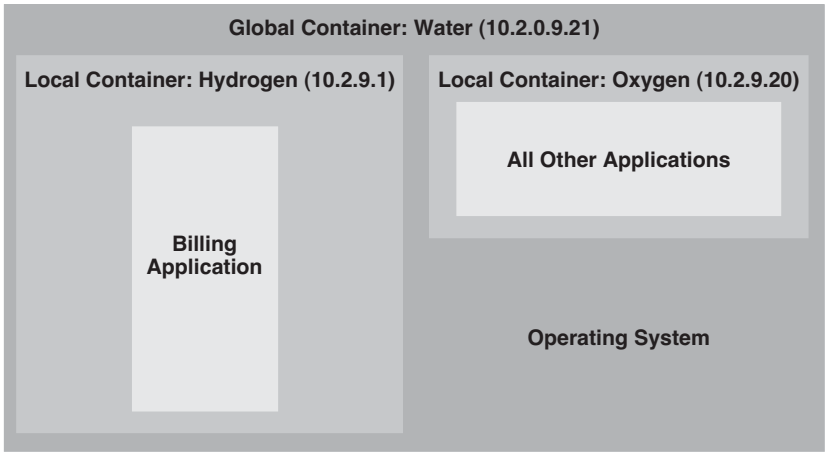

FIGURE 1 A container example,

The resource management capabilities inherent in Solaris Containers provide a unique balance between sharing and resource limitation. It allows a large number of competing programs to share the free CPUs in a multiprocessor machine. As all CPUs in the system are available to each Container, CPU time that remains unused by one Containers is made available to other Containers. Unlike a logical partition or a domain, assigning a seven processors to a Container does not take them away from other containers. Whenever a Container is not using all the CPUs, unused power is shared with the programs in other Containers.

At the same time, each Container is guaranteed a certain share of the total CPU, memory and network resources. If all programs on the machine demand CPU power, it is rationed so that each Container gets a share specified by the system administrator. As a result, system administrators can give programs a guaranteed minimum amount of CPU resources. However, more CPU resource can be utilized if no other task is competing for those resources.

It is important to note that unlike some emulated virtual machines, Containers do not have a performance tax. Adding a Container consumes approximately 60 MB of disk for per-Container files, and a negligible extra amount of overhead to check which Container is in use during a system call. This is comparable in performance to domains, BSD jails, or the IBM mainframe logical partitions.

Finally, the programs used to report the performance on a standalone machine work just as one would hope in a Container. They report the performance and resource usage of the programs in the Container, just as if they were on a standalone machine.

Balancing Changing Loads — An Example Script

The second part to our solution is to find a way to break the work down into small chunks and apply a dynamic balancing algorithm. Consider a program calling `billing` that accepts one parameter, a range of letters. If the program interprets a range of `A-A` as a command to process transactions beginning with just the letter A, then we can easily break down the work into 26 chunks, one for each letter of the alphabet, and balance it with a shell script like the following:

```
#!/bin/sh
#
# balance-billing -- run 7 instances to do a queue of work
#
ProgName=‘basename $0‘

main() {

            loadQueue
            for i in 1 2 3 4 5 6 7; do
                    processQueue $i &
                    sleep 5
            done
            wait
}
#
# processQueue -- do work as long as getQueueEntry returns
#                 it to us.
#
processQueue() {
            myNumber=$1

            letter=‘getQueueEntry $myNumber‘
            while [ "$letter" != "" ]; do
                    billing $letter-$letter
                    say "Server $myNumber processed $letter"
                    sleep 5
                    letter=‘getQueueEntry $myNumber‘
            done
            say "Server $myNumber done"
}
```

In this example script, seven copies of `processQueue` each run a copy of `billing`. Each copy processes a letter of the alphabet until the work for that letter id done, then goes back to get another from getQueueEntry. They each stop `when getQueueEntry` returns them a null string.

The `getQueueEntry` function can be as simple as the following.

```
#
# getQueueEntry -- get a single letter from the queue, and blank
#            if there are none.
#
getQueueEntry() {
             myNumber=$1

             # First, grab the queue (equivalent of locking it)
             for tries in 1 2 3 4 5 6 7 8 9 10; do
                    mv ./TheQueue ./$myNumber 2>/dev/null
                    if [ $? -eq 0 ]; then
                        # Then get and return a queue element
                        popQueue $myNumber
                        return
                    else
                        sleep 6
                        say "Server $myNumber retrying getQueueEntry"
                    fi
             done
}
```

The above example works well for straightfoward cases. Note that more complex situations may require additional C or Java code.

Getting from the Problem to the Solution

Converting to the desired solution can be made easier by taking advantage of the application isolation provided by Solaris Containers. In the example that follows, the same Solaris Container is used for preparation as well as deployment. Installing a program, like our billing example, in a Solaris Container is equivalent to setting it up on another machine — the program in the billing Container cannot interfere with other Containers on the system. As a result, the Solaris Container provides a safe (non-interfering) environment in which to install, test and configure a new copy of the billing program on the target machine, without disrupting the existing billing program.

Once the new “machine” is set up and the billing program installed, the unmodified billing program can be run to get a baseline measurement of the resources needed for successful operation. The `SAR` and `prstat` utilities can be used to obtain the resources used. From this information, system administrators can decide what initial share of the CPU resources to grant to the Container. For example, if a seven-instance billing program drives an eight processor system to 80% CPU utilization, that equates to 6.4 processors running at 100% utilization. This means that the

Container should be given a 10 percent share of the resources on a 64 processor system, and a 27 percent share on a server with 24 processors. Note that the `prctl` command can be used to adjust the CPUs assigned to a given Container.

```
# prctl -n zone.cpu-shares -r -v $SHARES -i zone $CONTAINER_NAME
```

Implementing the Billing Container

In our scenario, a 24-processor machine runs the Oracle and billing software. We need to create a separate Container for billing, and put a copy of the billing program in it, with the load balancing script detailed in the section "Balancing Changing Loads — An Example Script" on page 8.

Creating SRM Projects

The first step involves creating the projects used by the Solaris 10 Resource Manager (SRM) component of the Container. We want to create a billing project with 27% of total CPU resources, on a server with 24 CPUs is in use. Oracle is allocated 40 shares, and the default projects are assigned another five shares between them. Because we want a billing project with 27% of total CPU resources, we need (27 * 45) / 73 or 17 shares.

Right now the `/etc/project` file looks like the following:

```
# cat etc/project
user.root:1::::
noproject:2::::
default:3::::
group.staff:10::::
oracle:100:Oracle:oracle:oracle:project.cpu-shares=(privileged,40,none)
```

Because the Solaris 10 Resource Manager is already in use, the Fair Share Scheduler is probably being used for CPU management. Just in case, set the default scheduler with the `dispadmin` command and the current scheduler with the `priocntl` command.

```
# dispadmin -d FSS
# priocntl -s -c FSS -i class TS
# priocntl -s -c FSS -i pid 1
```

Next, add a billing group and project, and set the CPU shares for the project:

```
# groupadd billing
# projadd -c 'Billing processes' -U billing -G billing billing
# projmod -sK "project.cpu-shares=([priveleged,17,none)" billing
```

To automatically assign the billing user to this project, edit the `/etc/user_attr` file and add a line for billing:

```
# cat /etc/user_attr
# Copyright (c) 2003 by Sun Microsystems, Inc. All rights reserved.
#
# /etc/user_attr
#
# user attributes. see user_attr(4)
#
#pragma ident              "@(#)user_attr 1.103/07/09 SMI"
#
adm::::profiles=Log Management
lp::::profiles=Printer Management
root::::auths=solaris.*,solaris.grant;profiles=Web Console Management,All;lock!
oracle::::project=oracle
billing::::project=billing
```

This means that all processes started by the `billing` user are assigned to the `billing` project. This can be tested by becoming the `billing` user and running a load.

```
# su - billing
project.cpu-shares resource control assignment failed for project "billing"
su: unable to set credentials
Whoops, there's a spelling error in our /etc/projects line, "priveleged" instead of
"privileged". We fix that and su secedes.
```

If a load generating program is now run, it is possible to see if the CPU usage is assigned to the billing project. In our example, the load generator is `yes`, which writes a steady stream of `'y'` and newline characters to `stdout`. The `yes` program happily uses as much CPU resources as it can get. With `yes` running, we can see what project it is in and how much CPU resources it uses with the `prstat -J` command:

```
# su - billing
Sun Microsystems Inc.   SunOS 5.10 Generic January 2005
$ yes >/dev/null &
1238
PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
  1238 billing  1120K  816K run     22    0   0:01:51  29% sh/1
  1228 oracle   1120K  816K run     22    0   0:02:20  28% oracle/1
  1247 oracle   1120K  816K run     31    0   0:00:09  25% oracle/1
  1248 oracle   1120K  816K run     31    0   0:00:03  13% oracle/1
   646 davecb     14M   10M sleep   49    0   0:00:25 1.3% gnome-terminal/1
   371 root       43M   33M sleep   49    0   0:02:10 1.2% Xsun/1
   638 davecb   9856K 7248K sleep   59    0   0:00:28 0.4% metacity/1
  1249 oracle   5848K 4896K cpu0    59    0   0:00:00 0.3% prstat/1
   642 davecb     15M   11M sleep   59    0   0:00:29 0.2% gnome-panel/1
   748 davecb     19M   17M sleep   59    0   0:00:57 0.2% maker5X.exe/1
   675 davecb   6040K 5096K sleep   49    0   0:00:14 0.1% wish8.3/2
   670 davecb   7936K 5248K sleep   59    0   0:00:02 0.1% galf-server/1
   644 davecb     13M 9096K sleep   49    0   0:00:06 0.0% gnome-perfmeter/1
   640 davecb     29M   26M sleep   59    0   0:00:05 0.0% nautilus/6
   636 davecb   3808K 2344K sleep   59    0   0:00:01 0.0% gnome-smproxy/1
PROJID    NPROC  SIZE   RSS MEMORY      TIME  CPU PROJECT
   101        3 3360K 2448K   0.5%   0:02:32  66% oracle
   100        1 1120K  816K   0.2%   0:01:51  29% billing
    10       33  379M  259M    53%   0:05:17 3.5% group.staff
     3        7  388M   70M    14%   0:00:17 0.3% default
     0       34   82M   47M   9.7%   0:00:01 0.0% system
Total: 78 processes, 176 lwps, load averages: 2.58, 1.67, 1.08
```

From this output we can identify three `oracle` processes currently using 66% of available CPU resources. The fake billing process trying to use everything it can get, but is limited to 29% of CPU resources. The remaining 5% is used by other processes, such as `gnome-terminal`.

As mentioned earlier, `billing` can use more than 27% of CPU resources if the other processes on the machine do not need them. However, when those other processes do need access to those CPU resources, `billing` will be restricted to 27%.

Creating the Container

Next the Container is created, starting with its directory under `/export/zone/billing`. This must reside in a filesystem with enough space for local files, plus the minimum zone size of 60 MB. Once the directory is created, the Container is created with the `zonecfg` command.

```
# mkdir /export/zone/billing
# zonecfg -z billing
billing: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:billing> create
zonecfg:billing> set zonepath=/export/zone/billing
zonecfg:billing> add net
zonecfg:billing:net> set address=10.9.129.221
zonecfg:billing:net> set physical=dfme0
zonecfg:billing:net> end
zonecfg:billing> add rctl
zonecfg:billing:rctl> set name=zone.cpu-shares
zonecfg:billing:rctl> add value (priv=privileged,17,none)
zonecfg:billing:rctl> end
zonecfg:billing> verify
zonecfg:billing> commit
zonecfg:billing> exit
```

Next, try to make the Container runable.

```
# zoneadm -z billing install
/export/zone/billing must not be group readable.
/export/zone/billing must not be group executable.
/export/zone/billing must not be world readable.
/export/zone/billing must not be world executable.
could not verify zonepath /export/zone/billing because of the above errors.
zoneadm: zone billing failed to verify
```

While we set the permissions of the `/export/zone/billing` directory to `755`, it could be a possible security risk. For safety, permissions for the zone path must be set to `0700`. Make that change and try again to make the Container runable.

```
# zoneadm -z billing install
Preparing to install zone <billing>.
Creating list of files to copy from the global zone.
Copying <2523> files to the zone.
Initializing zone product registry.
Determining zone package initialization order.
Preparing to initialize <897> packages on the zone.
Percent complete: 6%         Initializing
# zoneadm -z billing boot
```

Before booting the billing container with its 17 shares, first set the global container to the same 45 shares tested using /etc/projects.

```
# prctl -n zone.cpu-shares -r -v 45 1
```

Once booted, the billing container is assigned17 shares, and the global Container 45, shares. As a result, the billing process will not be able to starve the Oracle software of CPU resources.

The normal way one use containers is to put each of the applications in a container, keeping the global container for root logins. In this example, however, the global container is used to run Oracle. Therefore, the `rcntl` command must be added to the `/etc/rc2.d` file to ensure it runs every time the system is booted.

Taking the Next Step

Now that one problem of unmanaged change has been solved, it is possible to take advantage of the resource management lessons learned and apply them to other change management problems that arise in the business.

Nearly every organization experiences two particular types of change — seasonal rushes and an increase in load due to business growth. Both are very good things for the business, but place great demands on the IT department. To address these challenges organizations can:

- Put each major application in its own Container to gain visibility into the resources each application utilizes. Use the `prstat -Z` command to obtain resource utilization statistics.

- Identify the most important programs during busy periods and give them a larger share of CPU resources with the `prctl` command.
- If it is possible to determine how much extra load is likely to be experienced during the annual rush, compute the appropriate balance of resource beforehand so that they can be allocated quickly when needed.
- If insufficient resources are available, plan ahead and obtain pricing for the additional resources needed to handle any expected increased load conditions. Give management an estimate of the cost of potentially lost business, as well as the cost of hardware to keep that business.
- If resources are available but not distributed correctly for an impending rush, take a Solaris Container with a less important program and move it from the machine with critical programs to another machine. Because the Container has its own name and IP address, it can be moved as a unit without breaking other programs with which it communicates.

Remember that management keeps its eye focused on costs more than any other factor. They are constantly struggling to provide more services and better availability while keeping their costs constant.

This means that you can contribute by finding ways to use their resources more effectively, and use an architecture that is flexible enough to handle changes. With Containers, Sun provides you another technology designed with your management's investment protection in mind.

Each of the solutions presented addresses problems which cost management money, including the cost of:

- Not meeting defined service levels during a rush
- Additional hardware to meet demand
- Having programs on the wrong hardware
- Moving the programs to the right place

Helping solve each of these problems provides a monetary advantage to management, one that will be remembered come annual review time.

About the Author

Dave is an currently an engineer in ASE, on the Managed Storage project. At the time he wrote this he was performance engineer at Data Center Works in Toronto, "The Organization Formerly Known as ACE", finishing up a large performance project from which the scenarios were adapted.

In his copious spare time he writes about Samba and Unix, and in the two weeks of Canadian summer, repairs the cottage.

Acknowledgements

The author would like to recognize the following individuals for their contributions to this article:

- Menno Lageman, a Containers expert, for his help with the solution and its presentation.
- Ron Lipsius and George Fytikas, for encouraging me to write about the lessons learned and reviewed the resulting paper.

References

More information on Solaris Containers can be found at
http://www.sun.com/bigadmin/content/zones/

More information on Solaris 10 Resource Manager can be found at
http://docs.sun.com/app/docs/doc/817-1592

A Blastwave tutorial is available at
http://www.blastwave.org/docs/Solaris-10-b51/DMC-0002/dmc-0002.html

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals through this program.

Accessing Sun Documentation Online

The `docs.sun.com` web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com/`

To reference Sun BluePrints OnLine articles, visit the Sun BluePrints OnLine Web site at: `http://www.sun.com/blueprints/online.html`