Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Evaluating Linux Kernel Crash Dumping Mechanisms

Fernando Luis Vázquez Cao
NTT Data Intellilink
fernando@intellilink.co.jp

Abstract

There have been several kernel crash dump capturing solutions available for Linux for some time now and one of them, kdump, has even made it into the mainline kernel.

But the mere fact of having such a feature does not necessary imply that we can obtain a dump reliably under any conditions. The LKDTT (Linux Kernel Dump Test Tool) project was created to evaluate crash dumping mechanisms in terms of success rate, accuracy and completeness.

A major goal of LKDTT is maximizing the coverage of the tests. For this purpose, LKDTT forces the system to crash by artificially recreating crash scenarios (panic, hang, exception, stack overflow, hang, etc.), taking into account the hardware conditions (such as ongoing DMA or interrupt state) and the load of the system. The latter being key for the significance and reproducibility of the tests.

Using LKDTT the author could constate the superior reliability of the kexec-based approach to crash dumping, although several deficiencies in kdump were revealed too. Since the final goal is having the best crash dumping mechanism possible, this paper also addresses how the aforementioned problems were identified and solved. Finally, possible applications of kdump beyond crash dumping will be introduced.

1 Introduction

Mainstream Linux lacked a kernel crash dumping mechanism for a long time despite the fact that there were several solutions (such as *Diskdump* [1], *Netdump* [2], and *LKCD* [3]) available out of tree . Concerns about their intrusiveness and reliability prevented them from making it into the vanilla kernel.

Eventually, a handful of crash dumping solutions based on *kexec* [4, 5] appeared: *Kdump* [6, 7], *Mini Kernel Dump* [8], and *Tough Dump* [9]. On paper, the kexec-based approach seemed very reliable and the impact in the kernel code was certainly small. Thus, kdump was eventually proposed as Linux kernel's crash dumping mechanism and subsequently accepted.

However, having a crash dumping mechanism does not necessarily imply that we can get a dump under any crash scenario. It is necessary to do proper testing, so that the success rate and accuracy of the dumps can be estimated and the different solutions compared fairly. Besides, having a standardised test suite would also help establishing a quality standard and, collaterally, detecting regressions would be much easier.

Unless otherwise indicated, henceforth all the explanations will refer to i386 and x86_64 architectures, and Linux 2.6.16 kernel.

1.1 Shortcomings of current testing methods

Typically to test crash dumping mechanisms a kernel module is created that artificially causes the system to die. Common methods to bring the system down from this module consist of directly invoking `panic`, making a null pointer dereference and other similar techniques.

Sometimes, to ease testing a user space tool is provided that sends commands to the kernel-space part of the testing tool (via the `/proc` file system or a new device file), so that things like the crash type to be generated can be configured at run-time.

Beyond the crash type, there are no provisions to further define the crash scenario to be recreated. In other words, parameters like the load of the machine and the state of the hardware are undefined at the time of testing.

Judging from the results obtained with this approach to testing all crash dumping solutions seem to be very close in terms of reliability, regardless of whether they are kexec-based or not, which seems to contradict theory. The reason is that the coverage of the tests is too limited as a consequence of leaving important factors out of the picture. Just to give some examples, the hardware conditions (such as ongoing DMA or interrupt state), the system load, and the execution context are not taken into consideration. This greatly diminishes the relevance of the results.

1.2 LKDTT motivation

The critical role crash dumping solutions play in enterprise systems calls for proper testing, so that we can have an estimate of their success rate under realistic crash scenarios. This is something the current testing methods cannot achieve and, as an attempt to fill this gap, the LKDTT project [10] was created.

Using LKDTT many deficiencies in kdump, LKCD, mkdump and other similar projects were found. Over the time, some regressions were observed too. This type of information is of great importance to both Linux distributions and end-users, and making sure it does not pass unnoticed is one of the commitments of this project.

To create meaningful tests it is necessary to understand the basics of the different crash dumping mechanisms. A brief introduction follows in the next section.

2 Crash dump

A variety of crash dumping solutions have been developed for Linux and other UNIX®-like operating systems over the time. Even though implementations and design principles may differ greatly, all crash dumping mechanisms share a multistage nature:

1. Crash detection.
2. Minimal machine shutdown.
3. Crash dump capture.

2.1 Crash detection

For the crash dump capturing process to start a trigger is needed. And this trigger is, most interestingly, a system crash.

The problem is that this peculiar trigger sometimes passes unnoticed or, in the words, *the kernel is unable to detect that itself has crashed*.

The culprits of system crashes are *software errors* and *hardware errors*. Often a hardware error leads to a software errors, and vice versa, so it is not always easy to identify the original problem. For example, behind a panic in the VFS code a damaged memory module might be lurking.

There is one principle that applies to both software and hardware errors: if the intention is to capture a dump, as soon as an error is detected control of the system should be handed to the crash dumping functionality. Deferring the crash dumping process by delegating invocation of the dump mechanism to functions such as `panic` is potentially fatal, because the crashing kernel might well lose control of the system completely before getting there (due to a stack overflow for example).

As one might expect, the detection stage of the crash dumping process does not show marked implementation specific differences. As a consequence, a single implementation could be easily shared by the different crash dumping solutions.

2.1.1 Software errors

A list of the most common crash scenarios the kernel has to deal with is provided below:

- *Oops*: Occurs when a programming mistake or an unexpected event causes a situation that the kernel deems grave. Since the kernel is the supervisor of the entire system it cannot simply kill itself as it would do with a user-space application that goes nuts. Instead, the kernel issues and oops (which results in a stack trace and error message to the console) and strives to get out of the situation. But often, after the oops, the system is left in an inconsistent state the kernel cannot recover from and, to avoid further damage, the system panics (see panic below). For example, a driver might have been in the middle of talking to hardware or holding a lock at the time of the crash and it would not be safe to resume execution. Hence, a panic is issued instead.
- *Panic*: Panics are issued by the kernel upon detecting a critical error from which it cannot recover. After printing and error message the system is halted.
- *Faults*: Faults are triggered by instructions that cannot or should not be executed by the CPU. Even though some of them are perfectly valid, and in fact play an essential role in important parts of the kernel (for example, pages faults in virtual memory management); there are certain faults caused by programming errors, such as divide-error, invalid TSS, or double fault (see below), which the kernel cannot recover from.
- *Double and triple faults*: A double fault indicates that the processor detected a second exception while calling the handler for a previous exception. This might seem a rare event but it is possible. For example, if the invocation of an exception handler causes a stack overflow a page fault is likely to happen, which, in turn, would cause a double fault. In i386 architectures, if the CPU faults again during the inception of the double fault, then it triple faults, entering a shutdown cycle that is followed by a system `RESET`.
- *Hangs*: Bugs that cause the kernel to loop in kernel mode, without giving other tasks the chance to run. Hangs can be classified in two big groups:
  - *Soft lockups*: These are transitory lockups that delay execution and

scheduling of other tasks. Soft lockups can be detected using a software watchdog.

  - *Hard lockups*: These are lockups that leave the system completely unresponsive. They occur, for example, when a CPU disables interrupts and gets stuck trying to get spinlock that is not freed due to a locking error. In such a state timer interrupts are not served, so scheduler-based software watchdogs cannot be used for detection. The same happens to keyboard interrupts, and that is why the `Sys Rq` key cannot be used to trigger the crash dump. The solution here is using the NMI handler.

- *Stack overflows*: In Linux the size of the stacks is limited (at the time of writing i386's default size is 8KB) and, for this reason, the kernel has to make a sensitive use of the stack to avoid bloating. It is a common mistake by novice kernel programmers to declare large automatic variables or to use deeply nested recursive algorithms; both of these practises tend to cause stack overflows. Stacks are also jeopardised by other factors that are not so evident. For example, in i386 interrupts and exceptions use the stack of the current task, which puts extra pressure on it. Consequently, interruption nesting should also be taken into account when programming interrupt handlers.

2.1.2 Hardware errors

Not only software has errors, sometimes machines fail too. Some hardware errors are recoverable, but when a fatal error occurs the system should come to a halt as fast as possible to avoid further damage. It is not even clear whether trying to capture a crash dump in the event of a serious hardware error is a sensitive thing to do. When the underlying hardware cannot be trusted one would rather bring the system down to avoid greater havoc.

The Linux kernel can make use of some error detection facilities of computer hardware. Currently the kernel is furnished with several infrastructures which deal with hardware errors, although the tendency seems to be to converge around EDAC (Error Detection and Correction) [11]. Common hardware errors the Linux kernel knows about include:

- *Machine checks*: Machine checks occur in response to CPU-internal malfunctions or as a consequence of hardware resets. Their occurrence is unpredictable and can leave memory and/or registers in a partially updated state. In particular, the state of the registers at the time of the event is completely undefined.

- *System RAM errors*: In systems equipped with ECC memory the memory chip has extra circuitry that can detect errors in the ingoing and outgoing data flows.

- *PCI bus transfer errors*: The data travelling to/from a PCI device may experience corruption whilst on the PCI bus. Even though a majority of PCI bridges and peripherals support such error detection most system do not check for them. It is worth noting that despite the fact that some of this errors might trigger an NMI it is not possible figure out what caused it, because there is no more information.

2.2 Minimal machine shutdown

When the kernel finds itself in a critical situation it cannot recover from, it should hand con-

trol of the machine to the crash dumping functionality. In contrast to the previous stage (detection), the things that need to be done at this point are quite implementation dependent. That said, all the crash dumping solutions, regardless of their design principles, follow the basic execution flow indicated below:

1. Right after entering the dump route the crashing CPU disables interrupts and saves its context in a memory area specially reserved for that purpose.
2. In SMP environments, the crashing CPU sends NMI IPIs to other CPUs to halt them.
3. In SMP environments, each IPI receiving processor disables interrupts and saves its context in a special-purpose area. After this, the processor busy loops until the dump process ends.
   Note: Some kexec-based crash dump capturing mechanisms relocate to boot CPU after a crash, so this step becomes different in those cases (see Section 7.4 for details).
4. The crashing CPU waits a certain amount of time for IPIs to be processed by the other CPUs, if any, and resumes execution.
5. Device shutdown/reinitialization, if done at all, is kept to a minimum, for it is not safe after a crash.
6. Jump into the crash dump capturing code.

2.3 Crash dump capture

Once the minimal machine shutdown is completed the system jumps into the crash dump capturing code, which takes control of the system to do the dirty work of capturing the dump and saving the dump image in a safe place.

Before continuing, it is probably worth defining what is understood by *kernel crash dump*. A kernel crash dump is an image of the resources in use by the kernel at the time of the crash, and whose analysis is an essential element to clarify what went wrong. This usually comprises an image of the memory available to the crashed kernel and the register states of all the processors in the system. Essentially, any information deemed useful to figure out the source of the problem is susceptible of being included in the dump image.

This final and decisive stage is probably the one that varies more between implementations. Attending to the design principles two big groups can be identified though:

- *In-kernel solutions*: LKCD, Diskdump, Netdump.
- *kexec-based solutions*: Kdump, MK-Dump, Tough Dump.

2.3.1 In-kernel solutions

The main characteristic of the in-kernel approach is, as the name suggests, that the crash dumping code uses the resources of the crashing kernel. Hence, these mechanisms, among other things, make use of the drivers and the memory of the crashing kernel to capture the crash dump. As might be expected this approach has many reliability issues (see Section 5.1 for an explanation and Table 2 for test results).

2.3.2 kexec-based solutions

The core design principle behind the kexec-based approach is that the dump is captured from an independent kernel (the crash dump

kernel or second kernel) that is soft-booted after the crash. Here onwards, for discussion purposes, crashing kernel is referred to as first kernel and the kernel which captures the dump as either capture kernel or second kernel.

As a general rule, the crash dumping mechanism should avoid fiddling with the resources (such as memory and CPU registers) of the crashing kernel. And, when this is inevitable, the state of these resources should be saved before they are used. This means that if the capture kernel wants to use a memory region the first kernel was using, it should first save the original contents so that this information is not missing in the dump image. These considerations along with the fact that we are soft booting into a new kernel determines the possible implementations of the capture kernel:

- *Booting the capture kernel from the standard memory location*
  The capture kernel is loaded in a reserved memory region and in the event of a crash it is copied to the memory area from where it will boot. Since this kernel is linked against the architecture's default start address, it needs to reside in the same place in memory as the crashing kernel. Therefore, the memory area necessary to accommodate the capture kernel is preserved by copying it to a backup region just before doing the copy. This was the approach taken by Tough Dump.

- *Booting the second kernel from a reserved memory region*
  After a crash the system is unstable and the data structures and functions of the crashing kernel are not reliable. For this reason there is no attempt to perform any kind of device shutdown and, as a consequence, any ongoing DMAs at the time of the crash are not stopped. If the approach discussed before is used the capture kernel is prone to be stomped by DMA transactions initiated in the first kernel. As long as IOMMU entries are not reassigned, this problem can be solved by booting the second kernel directly from the reserved memory region it was loaded into. To be able to boot from the reserved memory region the kernel has to be relocated there. The relocation can be accomplished at compile time (with the linker) or at run-time (see discussion in Section 7.2.1). Kdump and mkdump take the first and second approach, respectively.

For the reliability reasons mentioned above the second approach is considered the right solution.

3 LKDTT

3.1 Outline

LKDTT is a test suite that forces the kernel to crash by artificially recreating realistic crash scenarios. LKDTT accomplishes this by taking into account both the state of the hardware (for example, execution context and DMA state) and the load conditions of the system for the tests.

LKDTT has kernel-space and user-space components. It consists of a kernel patch that implements the core functionality, a small utility to control the testing process (`ttutils`), and a set of auxiliary tools that help recreating the necessary conditions for the tests.

Usually tests proceed as follows:

- If it was not built into the kernel, load the DTT (Dump Test Tool) module (see Section 3.2.2).

- Indicate the point in the kernel where the crash is to be generated using `ttutils` (see Section 3.3). This point is called Crash Point (CP).

- Reproduce the necessary conditions for the test using the auxiliary tools (see Section 3.4).

- Configure the CP using `ttutils`. The most important configuration item is the crash type.

- If the CP is located in a piece of code rarely executed by the kernel, it may become necessary to use some of the auxiliary tools again to direct the kernel towards the CP.

A typical LKDTT session is depicted in Table 1.

3.2 Implementation

LKDTT is pretty simple and specialized on testing kernel crash dumping solutions. LKDTT's implementation is sketched in Figure 1, which will be used throughout this section for explanation purposes.

The sequence of events that lead to an artificial crash is summarized below:

- Kernel execution flow reaches a crash point or CP (see 3.2.1). In the picture the CP is called `HD_CP` and is located in the hard disk device driver.

- If the CP is enabled the kernel jumps into the DTT module. Otherwise execution resumes from the instruction immediately after the CP.

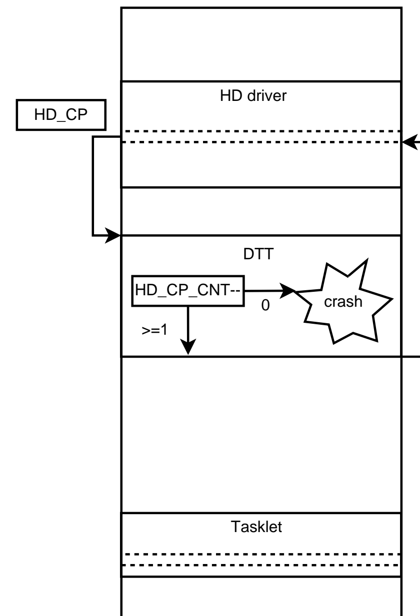

Figure 1: Crash points implementation

- The DTT module checks the counter associated with the CP. This counter (`HD_CP_CNT` in the example) keeps track of the number of times the CP in question has been crossed.

- This counter is a reverse counter in reality, and when it reaches 0 the DTT (Dump Test Tool) module induces the system crash associated with the CP. If the counter is still greater than zero execution returns from the module and continues from the instruction right after the CP.

Both the initial value of the counter and the crash type to be generated are run-time configurable from user space using `ttutils` (see 3.3). Crash points, however, are inserted in the kernel source code as explained in the following section.

```
# modprobe dtt
# ./ttutils ls
id      crash type      crash point name        count   location
1       none            INT_HARDWARE_ENTRY      0       kern
3       none            FS_DEVRW                0       kern
4       panic           MEM_SWAPOUT             7       kern
5       none            TASKLET                 0       kern
# ./ttutils add -p IDE_CORE_CP -n 50
# ./ttutils ls
id      crash type      crash point name        count   location
1       none            INT_HARDWARE_ENTRY      0       kern
3       none            FS_DEVRW                0       kern
4       panic           MEM_SWAPOUT             7       kern
5       none            TASKLET                 0       kern
50      none            IDE_CORE_CP             0       dyn
# ./helper/memdrain
# ./ttutils set -p IDE_CORE_CP -t panic -c 10
```

Table 1: LKDTT usage example

3.2.1 Crash Points

Each of the crash scenarios covered by the test suite is generated at a Crash Point (CP), which is a mere hook in the kernel. There are two different approaches to inserting hooks at arbitrary points in the kernel: patching the kernel source and dynamic probing. At first glance, the latter may seem the clear choice, because it is more flexible and it would not be necessary to recompile the kernel to insert a new CP. Besides, there is already an implementation of dynamic probing in the kernel (Kprobes [12]), with which the need of a kernel recompilation disappears completely.

Despite all these advantages dynamic probing was discarded because it changes the execution mode of the processor (a breakpoint interrupt is used) in a way that can modify the result of a test. Using `/dev/mem` to crash the kernel is another option, but in this case there is no obvious way of carrying out the tests in a controlled manner. These are the main motives behind LKDTT's election of the kernel patch approach. Specifically, the current CP implementation is based on IBM's *Kernel Hooks*.

In any case, a recent extension to Kprobes called Djprobe [13] that uses the breakpoint trap just once to insert a jump instruction at the desired probe point and uses this thereafter looks promising (the jump instruction does not alter the CPU's execution mode and, consequently, should not alter the test results).

As pointed out before, each CP has two attributes: number of times the CP is crossed before causing the system crash and the crash type to be generated.

At the time of writing, 5 crash types are supported:

- Oops: generates a kernel oops.
- Panic: generates a kernel panic.
- Exception: dereferences a null pointer.
- Hang: simulates a locking error by busy looping.

- Overflow: bloats the stack.

As mentioned before, crash points are inserted in the kernel source code. This is done using the macro CPOINT provided by LKDTT's kernel patch (see 3.5). An example can be seen in code listing 1.

3.2.2 DTT module

The core of LKDTT is the DTT (Dump Test Tool) kernel module. Its main duties are: managing the state of the CPs, interfacing with the user-space half of LKDTT (i.e. ttutils) through the /proc file system, and generating the crashes configured by the user using the aforementioned interface.

Figure 2 shows the pseudo state diagram of a CP (the square boxes identify states). From LKDTT's point of view, CPs come to existence when they are registered. This is done automatically in the case of CPs compiled into the kernel image. CPs residing in kernel modules, on the other hand, have to be registered either by calling a CP registration function from the module's init method, or from user space using a special ttutils' option (see add in *ttutils*, Section 3.3, for a brief explanation and Table 1 for a usage example). The later mechanism is aimed at reducing the intrusiveness of LKDTT.

Once a CP has been successfully registered the user can proceed to configure it using ttutils (see set in *ttutils*, Section 3.3, and Table 1 for an example). When the CP is enabled, every time the CP is crossed the DTT module decreases the counter associated with it by one and, when it reaches 0, LKDTT simulates the corresponding failure.

If it is an unrecoverable failure, the crash dumping mechanism should assume control of the

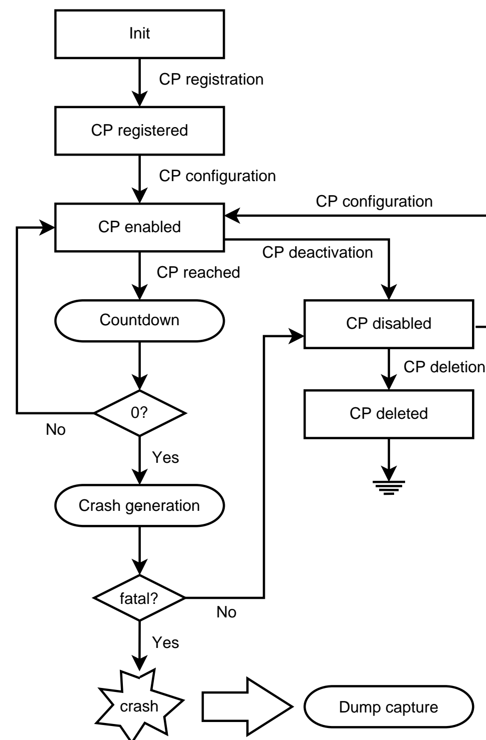

Figure 2: State diagram of crash points

system and capture a crash dump. However, if the kernel manages to recover from this eventuality the CP is marked as disabled, and remains in this state until it is configured again or deleted. There is a caveat here though: in-kernel CPs cannot be deleted.

LKDTT can be enabled either as a kernel module or compiled into the kernel. In the modular case it has to be modprobed:

```
# modprobe dtt [rec_num={>0}]
```

rec_num sets the recursion level for the stack overflow test (default is 10). The stack growth is approximately rec_num*1KB.

3.3 ttutils

ttutils is the user-space bit of LKDTT. It is a simple C program that interfaces with the

DTT module through `/proc/dtt/ctrl` and `/proc/dtt/cpoints`. The first file is used to send commands to the DTT module, commonly to modify the state of a CP. The second file, on the other hand, can be used to retrieve the state of crash points. It should be fairly easy to integrate the command in scripts to automate the testing process.

The ttutils command has the following format:

```
ttutils command [options]
```

The possible commands being:

- `help`: Display usage information.
- `ver(sion)`: Display version number of LKDTT.
- `list|ls`: Show registered crash points.
- `set`: Configure a Crash Point (CP). `set` can take two different options:
  `-p cpoint_name`: CP's name.
  `-t cpoint_type`: CP's crash type. Currently the available crash types are: *none* (do nothing), *panic*, *bug* (oops), *exception* (generates an invalid exception), *loop* (simulates a hang), and *overflow*.
  `-c pass_num`: Number of times the crash point has to be crossed before the failure associated with its type is induced. The default value for `pass_num` is 10.
- `reset`: Disable a CP. Besides, the associated counter that keeps track of the number of times the crash point has been traversed is also reset. The options available are:
  `-p cpoint_name`: CP's name.
  `-f`: Reset not only the CP's counter but also revert its type to none.
- `add`: Register a CP from a kernel module so that it can be actually used. This is aimed at modules that do not register the CPs inserted in their own code. Please note that registering does not imply activation. Activation is accomplished using `set`. `add` has two options:
  `-p cpoint_name`: CP's name.
  `-n id`: ID to be associated with the CP.
- `rmv`: Remove a CP registered using `add`. `rmv` has one single option:
  `-p cpoint_name`: CP's name.

3.4 Auxiliary tools

One of the main problems that arises when testing crash dumping solutions is that artfully inserting crash points in the kernel does not always suffice to recreate certain crash scenarios. Some execution paths are rarely trodden and the kernel has to be lured to take the right wrong way.

Besides, the tester may want the system to be in a particular state (ongoing DMA or certain memory and CPU usage levels, for example).

This is when the set of auxiliary tools included in LKDTT comes into play to reproduce the desired additional conditions.

A trivial example is `memdrain` (see Table 1 for a usage example), a small tool included in the LKDTT bundle. `memdrain` is a simple C program that reserves huge amounts of memory so that swapping is initiated. By doing so the kernel is forced to traverse the CPs inserted in the paging code, so that we can see how the crash dumping functionality behaves when a crash occurs in the middle of paging anonymous memory.

3.5 Installation

First, the kernel patch has to be applied:

```
# cd <PATH_TO_KERNEL_X.Y.Z>
# zcat <PATH_TO_PATCH>/dtt-full-X.
Y.Z.patch.gz | patch -p1
```

Once the patch has been applied we can proceed to configure the kernel as usual, but making sure that we select the options indicated below:

```
# make menuconfig
  Kernel hacking --->
   Kernel debugging [*]
    Kernel Hook support [*] or [M]
     Crash points [*] or [M]
```

The final steps consist of compiling the kernel and rebooting the system:

```
# make
# make modules_install
# make install
# shutdown -r now
```

The user space tools (`ttutils` and the auxiliary tools) can be installed as follows:

```
# tar xzf dtt_tools.tar.gz
# cd dtt_tools
# make
```

4 Test results

The results of some tests carried out with LKDTT against LKCD and two different versions of the vanilla kernel with kdump enabled can be seen in Table 2.

For each crash point all the crash types supported by LKDTT were tried: *oops*, *panic*, *exception*, *hang*, and *overflow*. The meaning of the crash points used during the tests is explained in Section 3.2.1.

The specifications of the test machine are as follows:

- CPU type: Intel Pentium 4 Xeon Hyperthreading.
- Number of CPUs: 2.
- Memory: 1GB.
- Disk controller: ICH5 Serial ATA.

The kernel was compiled with the options below turned on (when available): `CONFIG_PREEMPT`, `CONFIG_PREEMPT_BKL`, `CONFIG_DETECT_SOFTLOCKUP`, `CONFIG_4KSTACKS`, `CONFIG_SMP`. And the kernel command line for the kdump tests was:

```
root=/dev/sda1 ro crashkernel=32M@
16M nmi_watchdog=1 console=ttyS0,
38400 console=tty0
```

The test results in Table 2 are indicated using the convention below:

- `O`: Success.
- `O(nrbt)`: The system recovered from the induced failure. However, a subsequent reboot attempt failed, leaving the machine hanged.
- `O(nrbt, nmiw)`: The dump image was captured successfully but the system hanged when trying to reboot. The NMI watchdog detected this hang and, after determining that the crash dump had already been taken, tried to reboot the system again.

Crash point	Crash type	LKCD 6.1.0	kdump 2.6.13-rc7	kdump 2.6.16
INT_HARDWARE_ENTRY	panic	X	0	0
	oops	X (nmiw, nrbt)	X (nmiw)	0
	exception	X (nmiw, nrbt)	X (nmiw)	0
	hang	X	0	0
	overflow	X	X	X
INT_HW_IRQ_EN	panic	X	0	0
	oops	X	0	X(2c)
	exception	X	0	0
	hang	X	X	X
	overflow	X	X	X
INT_TASKLET_ENTRY	panic	0(nrbt, nmiw)	0	0
	oops	0(nrbt, nmiw)	0	0
	exception	0(nrbt, nmiw)	0	0
	hang	X	X (SysRq)	X(det,SysRq)
	overflow	X	X	X
TASKLET	panic	0(nrbt, nmiw)	0	0
	oops	0(nrbt, nmiw)	0	0
	exception	0(nrbt, nmiw)	0	0
	hang	0(nrbt, nmiw)	0	0
	overflow	X	X	X
FS_DEVRW	panic	0(nrbt, nmiw)	0	X(2c)
	oops	0(nrbt, nmiw)	0	X (log,SysRq)
	exception	0(nrbt, nmiw)	0	X (log,SysRq)
	hang	X	X (SysRq)	X (SysRq)
	overflow	X	X	X
MEM_SWAPOUT	panic	0(nrbt, nmiw)	0	0
	oops	0(nrbt, nmiw)	0	0 (nrbt)
	exception	0(nrbt, nmiw)	0	0 (nrbt)
	hang	X	X	X (unk,SysRq,2c)
	overflow	X	X	X
TIMERADD	panic	0(nrbt, nmiw)	0	0
	oops	0(nrbt, nmiw)	0	0
	exception	0(nrbt, nmiw)	0	0
	hang	X	0	0
	overflow	X	X	X
SCSI_DISPATCH_CMD	panic	X	0	0
	oops	X	0	0
	exception	X	0	0
	hang	X	X (SysRq)	X (det,SysRq)
	overflow	X	X	X

Table 2: LKDTT results

- `X`: Failed to capture dump.

- `X(2c)`: After the crash control of the system was handed to the capture kernel, but it crashed due to a device initialization problem.

- `X(SysRq)`: The crash not detected by the kernel, but the dump process was successfully started using the `Sys Rq` key.
Note: Often, when plugged after the crash the keyboard does not work and the `Sys Rq` is not effective as a trigger for the dump.

- `X(SysRq, 2c)`: Like the previous case, but the capture kernel crashed trying to initialize a device.

- `X(det, SysRq)`: The hang was detected by the soft lockup watchdog (`CONFIG_DETECT_SOFTLOCKUP`). Since this watchdog only notifies about the lockup without taking any active measures the dump process had to be started using the `Sys Rq` key.
Note: Even though the dump was successfully captured the result was marked with an `X` because it required user intervention. The action to take upon lockup should be configurable.

- `X(log, SysRq)`: The system became increasingly unstable, eventually becoming impossible to login into the system anymore (the prompt did not return after introducing login name and password). After the system locked up like this, the dump process had to be initiated using the `Sys Rq` key, because neither the NMI watchdog nor the soft lockup watchdog could detect any anomaly.

- `X(nmiw)`: The error was detected but the crashing kernel failed to hand control of the system to the crash dumping mechanism and hanged. This hang was subsequently detected by the NMI watchdog, who succeed in invoking the crash dumping functionality. Finally, the dump was successfully captured.
Note: The result was marked with an X because the NMI watchdog sometimes fails to start the crash dumping process.

- `X(nmiw, nrbt)`: Like the previous case, but after capturing the dump the system hanged trying to reboot.

- `X(unk,SysRq,2c)`: The auxiliary tool used for the test (see Section 3.4) became unkillable. After triggering the dump process using the `Sys Rq` key, the capture kernel crashed attempting to reinitialize a device.

4.1 Crash points

Even though testers are free to add new CPs, LKDTT is furnished with a set of essential CPs, that is, crash scenarios considered basic and that should always be tested. The list follows:

IRQ handling with IRQs disabled (`INT_HARDWARE_ENTRY`) This CP is crossed whenever an interrupt that is to be handled with IRQs disabled occurs (see code listing 1).

IRQ handling with IRQs enabled (`INT_HW_IRQ_EN`) This is the equivalent to the previous CP with interrupts enabled (see code listing 2).

Tasklet with IRQs disabled (`TASKLET`) If this CP is active crashes during the service of Linux tasklets with interrupts disabled can be recreated.

```
fastcall unsigned int __do_IRQ(
   unsigned int irq, struct
   pt_regs *regs)
{
  .....

  CPOINT(INT_HARDWARE_ENTRY);
  for (;;) {
    irqreturn_t action_ret;

    spin_unlock(&desc->lock);

    action_ret = handle_IRQ_event(
       irq, regs, action);
  .....
}
```

Listing 1: INT_HARDWARE_ENTRY crash point (kernel/irq/handle.c)

```
fastcall int handle_IRQ_event(
  .....

  if (!(action->flags &
     SA_INTERRUPT)) {
    local_irq_enable();
    CPOINT(INT_HW_IRQ_EN);
  }

  do {
    ret = action->handler(irq,
       action->dev_id, regs);
    if (ret == IRQ_HANDLED)
      status |= action->flags;
  .....
}
```

Listing 2: INT_HW_IRQ_EN crash point (kernel/irq/handle.c)

Tasklet with IRQs enabled (INT_TASKLET_ENTRY) Using this CP it is possible to cause a crash when the kernel is in the middle of processing a tasklet with interrupts enabled.

Block I/O (FS_DEVRW) This CP is used to bring down the system while the file system is performing low-level access to block devices (see code listing 3).

Swap-out (MEM_SWAPOUT) This CP is located in the code that tries to allocate space for anonymous process memory.

Timer processing (TIMERADD) This is a CP situated in the code that starts and re-starts high resolution timers.

SCSI command (SCSI_DISPATCH_CMD) This CP is situated in the SCSI command dispatching code.

IDE command (IDE_CORE_CP) This CP is situated in the code that handles I/O operations on IDE block devices.

5 Interpretation of the results and possible improvements

5.1 In-kernel crash dumping mechanisms (LKCD)

The primary cause of the bad results obtained by LKCD, and in-kernel crash dumping mechanism in general, is the flawed assumption that the kernel can be trusted and will in fact be operating in a normal fashion. This creates two major problems.

First, there is a problem with resources, notably with resources locking up, because it is not possible to know the locking status at the time of the crash. LKCD uses drivers and services of the crashing kernel to capture the dump. As a

```
void ll_rw_block(int rw, int nr,
   struct buffer_head *bhs[])
{
  .....
        get_bh(bh);
        submit_bh(rw, bh);
        continue;
      }
    }
    unlock_buffer(bh);
    CPOINT(FS_DEVRW);
  }
}
```

Listing 3: `FS_DEVRW` crash point (`fs/buffer.c`)

consequence, if the operation that has caused the crash was locking resources necessary to capture the dump, the dump operation will end up deadlocking. For example, the driver for the dump device may try to obtain a lock that was held before the crash occurred and, as it will never be released, the dump operation will hang up. Similarly, on SMP systems as operations being run on other CPUs are forced to stop in the event of a crash, there is the possibility that resources needed during the dumping process may be locked, because they were in use by any of the other CPUs and were not released before they halted. This may put the dump operation into a lockup too. Even if this doesn't result in a lock-up, insufficient system resources may also cause the dump operation to fail.

The source of the second problem is the reliability of the control tables, kernel text, and drivers. A kernel crash means that some kind of inconsistency has occurred within the kernel and that there is a strong possibility a control structure has been damaged. As in-kernel crash dump mechanisms employ functions of the crashed system for outputting the dump, there is the very real possibility that the damaged control structures will be referenced. Besides, page tables and CPU registers such as the stack pointer may be corrupted too, which can potentially lead to faults during the crash dumping process. In these circumstances, even if a crash dump is finally obtained, the resulting dump image is likely to be invalid, so that it cannot be properly analyzed.

For in-kernel crash dumping mechanisms there is no obvious solution to the memory corruption problems. However, the locking issues can be alleviated by using polling mode (as opposed to interrupt mode) to communicate with the dump devices.

Setting up a controllable dump route within the kernel is very difficult, and this is increasingly true as the size and complexity of the kernel augments. This is what sparked the apparition of methods capable of capturing a dump independent from the existing kernel.

5.2 Kdump

Even though kdump proved to be much more reliable than in-kernel crash dumping mechanisms there are still issues in the three stages that constitute the dump process (see Section 2):

- Crash detection: hang detection, stack overflows, faults in the dump route.
- Minimal machine shutdown: stack overflows, faults in the dump route.
- Crash dump capture: device reinitialization, APIC reinitialization.

5.2.1 Stack overflows

In the event of a stack overflow critical data that usually resides at the bottom of the stack

is likely to be stomped and, consequently, its use should be avoided.

In particular, in the i386 and IA64 architectures the macro `smp_processor_id()` ultimately makes use of the `cpu` member of struct `thread_info`, which resides at the bottom of the stack. x86_64, on the other hand, is not affected by this problem because it benefits from the use of the PDA infrastructure.

Kdump makes heavy use of `smp_processor_id()` in the reboot path to the second kernel, which can lead to unpredictable behaviour. This issue is particularly serious in SMP systems because not only the crashing CPU but also the rest of CPUs are highly dependent on likely-to-be-corrupted stacks. The reason it that during the minimal machine shutdown stage (see Section 2.2 for details) NMIs are used to stop the CPUs, but the NMI handler was designed on the premise that stacks can be trusted. This obviously does not hold good in the event of a crash overflow.

The NMI handler (see code listing 4) uses the stack indirectly through `nmi_enter()`, `smp_processor_id()`, `default_do_nmi`, `nmi_exit()`, and also through the crash-time NMI callback function (`crash_nmi_callback()`).

Even though the NMI callback function can be easily made stack overflow-safe the same does not apply to the rest of the code.

To circumvent some of these problems at the very least the following measures should be adopted:

- Create a stack overflow-safe replacement for `smp_processor_id`, which could be called `safe_smp_processor_id` (there is already an implementation for x86_64).

```
fastcall void do_nmi(struct
   pt_regs * regs, long error_code
   )
{
  int cpu;

  nmi_enter();
  cpu = smp_processor_id();
  ++nmi_count(cpu);

  if (!rcu_dereference(
     nmi_callback)(regs, cpu))
    default_do_nmi(regs);

  nmi_exit();
}
```

Listing 4: `do_nmi` (i386)

- Substitute `smp_processor_id` with `safe_smp_processor_id`, which is stack overflow-safe, in the reboot path to the second kernel.
- Add a new NMI low-level handling routine (`crash_nmi`) in `arch/*/kernel/entry.S` that invokes a stack overflow safe NMI handler (`do_crash_nmi`) instead of `do_nmi`.
- In the event of a system crash replace the default NMI trap vector so that the new `crash_nmi` is used.

If we want to be paranoid (and being paranoid is what crash dumping is all about after all), all the CPUs in the system should switch to new stacks as soon as a crash is detected. This introduces the following requirements:

- Per-CPU crash stacks: Reserve one stack per CPU for use in the event of a system crash. A CPU that has entered the dump route should switch to its respective per-CPU stack as soon as possible because the

cause of the crash might have a stack overflow, and continuing to use the stack in such circumstances can lead to the generation of invalid faults (such as double fault or invalid TSS). If this happens the system is bound to either hang or reboot spontaneously. In SMP systems, the rest of the CPUs should follow suit, switching stacks at the NMI gate (`crash_nmi`).

- Private stacks for NMIs: The NMI watchdog can be used to detect hard lockups and invoke kdump. However, this dump route consumes a considerable amount of stack space, which could cause a stack overflow, or contribute to further bloating the stack if it has already overflowed. As a consequence of this, the processor could end up faulting inside the NMI handler which is something that should be avoided at any cost. Using private NMI stacks would minimize these risks.

To limit the havoc caused by bloated stacks, the fact that a stack is about to overflow should be detected before it spills out into whatever is adjacent to it. This can be achieved in two different ways:

- *Stack inspection*: Check the amount of free space in the stack every time a given event, such as an interrupt, occurs. This could be easily implemented using the kernel's stack overflow debugging infrastructure (`CONFIG_DEBUG_STACKOVERFLOW`).

- *Stack guarding*: The second approach is adding an unmapped page at the bottom of the stack so that stack overflows are detected at the very moment they occur. If a small impact in performance is considered acceptable this is the best solution.

5.2.2 Faults in the dump route

Critical parts of the kernel such as fault handlers should not make assumptions about the state of the stack. An example where proper checking is neglected can be observed in the code listing 5. The `mm` member of the struct `tsk` is dereferenced without making any checks on the validity of `current`. If `current` happens to be invalid, the seemingly inoffensive dereference can lead to recursive page faults, or, if things go really bad, to a triple fault and subsequent system reboot.

```
fastcall void __kprobes
   do_page_fault(struct pt_regs *
   regs, unsigned long error_code)
{
        struct task_struct *tsk;
        struct mm_struct *mm;

        tsk = current;
        .....
        [ no checks are made on
           tsk ]
        mm = tsk->mm;
        .....
}
```

Listing 5: `do_page_fault` (i386)

Finally, to avoid risks, control should be handed to kdump as soon as a crash is detected. The invocation of the dump mechanism should not be deferred to `panic` or `BUG`, because many things can go bad before we get there. For example, it is not guaranteed that the possible code paths never use any of the things that make assumptions about the current stack.

5.2.3 Hang detection

The current kernel has the necessary infrastructure to detect hard lockups and soft lockups, but they both have some issues:

- *Hard lockups*: This type of lockups are detected using the NMI watchdog, which periodically checks whether tasks are being scheduled (i.e. the scheduler is alive). The Achilles' heel of this detection method is that it is strongly vulnerable to stack overflows (see Section 5.2.1 for details). Besides, there is one inevitable flaw: hangs in the NMI handler cannot be detected.

- *Soft lockups*: There is a soft lockup detection mechanism implemented in the kernel that, when enabled (`CONFIG_DETECT_SOFTLOCKUP=y`), starts per-CPU watchdog threads which try to run once per second. A callback function in the timer interrupt handler checks the elapsed time since the watchdog thread was last scheduled, and if it exceeds 10 seconds it is considered a soft lockup.
  Currently, the soft lockup detection mechanism limits itself to just printing the current stack trace and a simple error message. But, the possibility of triggering the crash dumping process instead should be available.

Using LKDTT a case in which the existing mechanisms are not effective was discovered: hang with interrupts enabled (see *IRQ handling with IRQs enabled* in Section 4.1). In such scenario timer interrupts continue to be delivered and processed normally so both the NMI watchdog and the soft lockup detector end up judging that the system is running normally.

5.2.4 Device reinitialization

There are cases in which after the crash the capture kernel itself crashes attempting to initialize a hardware device.

In the event of a crash kdump does not do any kind of device shutdown and, what is more, the firmware stages of the standard boot process are also skipped. This may leave the devices in a state the second kernel cannot get them out of. The underlying problem is that the soft boot case is not handled by most drivers, which assume that only traditional boot methods are used (after all, many of the drivers were written before kexec even existed) and that all devices are in a reset state.

Sometimes even after a regular hardware reboot the devices are not reset properly. The culprit in such cases is a BIOS not doing its job properly.

To solve this issues the device driver model should be improved so that it contemplates the soft boot case, and kdump in particular. In some occasions it might be impossible to reinitialize a certain device without knowing its previous state. So it seems clear that, at least in some cases, some type information about the state of devices should be passed to the second kernel. This brings the power management subsystem to mind, and in fact studying how it works could be a good starting point to solve the device reinitialization problem.

In the meantime, to minimize risks each machine could have a dump device (a HD or NIC) set aside for crash dumping, so that the crash kernel would use that device and have no other devices configured.

5.2.5 APICs reinitialization

Kdump defers much of the job of actually saving the dump image to user-space. This means that kdump relies on the scheduler and, consequently, the timer interrupt to be able to capture a dump.

This dependency on the timer represents a problem, specially in i386 and x86_64 SMP systems. Currently, on these architectures, during the initialization of the kernel the legacy

i8259 must exist and be setup correctly, even if it will not be used past this stage. This implies that, in APIC systems, before booting into the second kernel the interrupt mode must return to legacy. However, doing this is not as easy as it might seem because the location of the i8259 varies between chipsets and the ACPI MADT (Multiple APIC Description Table) does not provide this information. The return to legacy mode can accomplished in two different ways:

- *Save/restore BIOS APIC states*: All the APIC states are saved early in the boot process of the first kernel before the kernel attempts to initialize them, so that the APIC configuration as performed by the BIOS can be obtained. In the event of a crash, before booting into the capture kernel the BIOS APIC settings are restored back. Treating the APICs as black boxes like this has the benefit that the original states of the APICs can be restored even in systems with a broken BIOS. Besides, this method is theoretically immune to changes in the default configuration of APICs in new systems.
  There is one issue with this method though. It makes sure that the BIOS-designated boot CPU will always see timer interrupts in legacy mode, but this does not hold good if the second kernel boots on some other CPU as is possible with kdump. Therefore, for this method to work CPU relocation is necessary. It should also be noted that under certain rather unlikely circumstances relocation might fail (see Section 7.4 for details).

- *Partial save/restore*: Only the information that cannot be obtained any other way (i.e. i8259's location) is saved off at boot time. Upon a crash, taking into account this piece of information the APICs are reconfigured in such a way that all interrupts get redirected to the CPU in which the second kernel is going to be booted, which in kdump's case is the CPU where the crash occurred. This is the approach adopted by kdump.

6 LKDTT status and TODOS

Even though using LKDTT it is possible to test rather thoroughly the first two stages of the crash dumping process, that is *crash detection* and *minimal machine crash shutdown* (see Section 2), the capture kernel is not being sufficiently tested yet. The author is currently working on the following test cases:

- Pending IRQs: Leave the system with pending interrupts before booting into the capture kernel, so that the robustness of device drivers against interrupts coming at an unexpected time can be tested.

- Device reinitialization: For each device test whether it is correctly initialized after a soft boot.

Another area that is under development at the moment is test automation. However, due to the special nature of the functionality being tested there is a big roadblock for automation: the system does not always recover gracefully from crashes so that tests can resume. That is, in some occasions the crash dumping mechanism that is being tested will fail, or the system will hang while trying to reboot after capturing the dump. In such cases human intervention will always be needed.

7 Other kdump issues

The kernel community has been expecting that the various groups which are interested in crash

dumping would converge around kdump once it was merged. And the same was expected from end-users and distributors. However, so far, this has not been the case and work has continued on other strategies.

The causes of this situation are diverse and, to a great extent, unrelated to reliability aspects. Instead, the main issues have to do with availability, applicability and usability. In some cases it is just a matter of time before they get naturally solved, but, in others, improvements need to be done to kdump.

7.1 Availability and applicability

Most of the people use distribution-provided kernels that are not shipped with kdump yet. Certainly, distributions will eventually catch up with the mainstream kernel and this problem will disappear.

But, in the mean time, there are users who would like to have a reliable crash dumping mechanism for their systems. This is especially the case of enterprise users, but they usually have the problem that updating or patching the kernel is not an option, because that would imply the loss of official support for their enterprise software (this includes DBMSs such as Oracle or DB2 and the kernel itself). It is an extreme case but some enterprise systems cannot even afford the luxury of a system reboot.

This situation along with the discontent with the crash dumping solutions provided by distributors sparked the apparition of other kexec-based projects (such as mkdump and Tough Dump), which were targeting not only mainstream adoption but also existing Linux distributions. This is why these solutions sometimes come in two flavors: a kernel patch for vanilla kernels and a fully modularized version for distribution kernels.

7.2 Usability

There are some limitations in kdump that have an strong impact in its usability, which affects both end-users and distributors as discussed below.

7.2.1 Hard-coding of reserved area's start address

To use kdump it is necessary to reserve a memory region big enough to accommodate the dump kernel. The start address and size of this region is indicated at boot time with the command line parameter `crashkernel=Y@X`, `Y` denoting how much memory to reserve, and `X` indicating at what physical address the reserved memory region starts. The value of `X` has to be used when configuring the capture kernel, so that it is linked to run from that start address. This means a displacement of the reserved area may render the dump kernel unusable. Besides it is not guaranteed that the memory region indicated at the command line is available to the kernel. For example, it could happen that the memory region does not exist, or that it overlaps system tables, such as ACPI's. All these issues make distribution of pre-compiled capture kernels cumbersome.

This undesirable dependency between the second and first kernel can be broken using a run-time relocatable kernel. The reason is that, by definition, a run-time relocatable kernel can run from any dedicated memory area the first kernel might reserve for it. To achieve run-time relocation a relocation table has to be added to the kernel binary, so that the actual relocation can be performed by either a loader (such as kexec) or even by the kernel itself. The first calls for making the kernel an ELF shared object. The second can be accomplished by resolving all the symbols in `arch/*/kernel/head.S` (this is what mkdump does).

7.2.2 Memory requirements

Leaving the task of writing out the crash dump to user space introduces great flexibility at the cost of increasing the size of the memory area that has to be reserved for the capture kernel. But for systems with memory restrictions (such as embedded devices) a really small kernel with just the necessary drivers and no user space may be more appropriate. This connects with the following point.

7.3 Kernel-space based crash dumping

After a crash the dump capture kernel might not be able to restore interrupts to a usable state, be it because the system has a broken BIOS, or be it because the interrupt controller is buggy. In such circumstances, processors may end up not receiving timer interrupts. Besides, the possibility of a timer failure should not be discarded either.

In any case, being deprived of timer interrupts is an insurmountable problem for user-space based crash dumping mechanisms such as kdump, because they depend on a working scheduler and hence the timer.

To tackle this problem a kernel-space driven crash dumping mechanism could be used, and even cohabit with the current user-space centered implementation. Which one to employ could be made configurable, or else, the kernel-space solution could be used as a fallback mechanism in case of failure to bring up user-space.

7.4 SMP dump capture kernel

In some architectures, such as i386 and x86_64, it is not possible to boot a SMP kernel from a CPU that is not the BIOS-designated boot CPU. Consequently, to do SMP in the capture kernel it is necessary to relocate to the boot CPU beforehand. Kexec achieves CPU relocation using scheduler facilities, but kdump cannot use the same approach because after a crash the scheduler cannot be trusted.

As a consequence, to make kdump SMP-capable a different solution is needed. In fact, there is a very simple method to relocate to the boot CPU that takes advantage of inter-processor NMIs. As discussed in Section 2.2 (*Minimal machine shutdown*), this type of NMIs are issued by the crashing CPU in SMP systems to stop the other CPUs before booting into the capture kernel. But this behavior can be modified so that relocation to the boot CPU is performed too. Obviously, if the crashing CPU is the boot CPU nothing needs to be done. Otherwise, upon receiving NMI the boot CPU should assume the task of capturing the kernel, so that the NMI-issuing CPU (i.e. the crashing the CPU) is relieved from that burden a can halt instead. This is the CPU relocation mechanism used by mkdump.

Even though being able to do SMP would boost the performance of the capture kernel, it was suggested that in some extreme cases of crash the boot CPU might not even respond to NMIs and, therefore, relocation to the boot CPU will not be possible. However, after digging through the manuals the author could only find (and reproduce using LKDTT) one such scenario, which occurs when the two conditions below are met:

- The boot CPU is already attending a different NMI (from the NMI watchdog for example) at the time the inter-processor NMI arrives.
- The boot CPU hangs inside the handler of this previous NMI, so it does not return.

The explanation is that during the time a CPU is servicing an NMI other NMIs are blocked, so a lockup in the NMI handler guarantees a system hang if relocation is attempted as described before. The possibility of such a hang seems remote and easy to evaluate. But it could also be seen as a trade-off between performance and reliability.

8 Conclusion

Existing testing methods for kernel crash dump capturing mechanisms are not adequate because they do not take into account the state of the hardware and the load conditions of the system. This makes it impossible to recreate many common crash scenarios, depriving test results of much of their validity. Solving these issues and providing controllable testing environment were the major motivations behind the creation of the LKDTT (Linux Kernel Dump Test Tool) testing project.

Even though LKDTT showed that kdump is more reliable than traditional in-kernel crash dumping solutions, the test results revealed some deficiencies in kdump too. Among these, minor hang detection deficiencies, great vulnerability to stack overflows, and problems reinitializing devices in the capture kernel stand out. Solutions to some of these problems have been sketched in this paper and are currently under development.

Since the foundation of the testing project the author could observe that new kernel releases (including release candidates) are sometimes accompanied by regressions. Regressions constitute a serious problem for both end-users and distributors, that requires regular testing and standardised test cases to be tackled properly. LKDTT aims at filling this gap.

Finally, several hurdles that are hampering the adoption of kdump were identified, the need for a run-time relocatable kernel probably being the most important of them.

All in all, it can be said that as far as kernel crash dumping is concerned Linux is heading in the right direction. Kdump is already very robust and most of the remaining issues are already being dealt with. In fact, it is just a matter of time before kdump becomes mature enough to focus on new fields of application.

9 Future lines of work

All the different crash dumping solutions do just that after a system crash: capture a crash dump. But there is more to a crashed system kexec than crash dumping. For example, in high availability environments it may be desirable to notify the backup system after a crash, so that the failover process can be initiated earlier.

In the future, kdump could also benefit from the current PID virtualization efforts, which will provide the foundation for process migration in Linux. The process migration concept could be extended to the crash case, in such a way that after doing some sanity-checking, tasks that have not been damaged can be migrated and resume execution in a different system.

Acknowledgements

I would like to express my gratitude to Itsuro Oda for his original contribution to LKDTT and valuable comments, as well as to all those who have laid the foundation for a reliable kernel crash dumping mechanism in Linux.

References

[1] Diskdump patches. http://www.redhat.com/support/wpapers/redhat/netdump/.

[2] Michael K. Johnson. Red Hat, Inc.'s network console and crash dump facility, 2002. http://www.redhat.com/support/wpapers/redhat/netdump/.

[3] Linux kernel crash dump (LKCD) home page, 2005. http://lkcd.sourceforge.net/.

[4] Hariprasad Nellitheertha. Reboot linux faster using kexec, 2004. http://www-128.ibm.com/developerworks/linux/library/l-kexec.html.

[5] Kexec-tools code. http://www.xmission.com/~ebiederm/files/kexec/.

[6] Vivek Goyal, Eric W. Biederman, and Hariprasad Nellitheertha. A kexec based dumping mechanism. In *Ottawa Linux Symposium (OLS 2005)*, July 2005.

[7] Kdump home page. http://lse.sourceforge.net/kdump/.

[8] Itsuro Oda. Mini Kernel Dump (MKDump) home page, 2006. http://mkdump.sourceforge.net/.

[9] Linux tough dump (TD) home page (japanese site), 2006. http://www.hitachi.co.jp/Prod/comp/linux/products/solution.html.

[10] Fernando Luis Vázquez Cao. Linux kernel dump test tool (LKDTT) home page, 2006. http://lkdtt.sourceforge.net/.

[11] EDAC wiki. http://buttersideup.com/edacwiki/FrontPage.

[12] Prasanna Panchamukhi. Kernel debugging with kprobes, 2004. http://www-128.ibm.com/developerworks/linux/library/l-kprobes.html.

[13] Djprobe documentation and patches. http://lkst.sourceforge.net/djprobe.html.