Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

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

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

BitKeeper for Kernel Developers

Val Henson
val@nmt.edu

Jeff Garzik
jgarzik@mandrakesoft.com

Abstract

BitKeeper[1] is a revolutionary new distributed source control management suite which is ideal for Linux kernel development. BitKeeper provides tools which automate and simplify many common kernel development tasks. In this paper, we describe basic BitKeeper concepts and operations, BitKeeper solutions for common kernel development problems, and a workflow for interacting with other Linux developers using BitKeeper. We also discuss some of BitKeeper's shortcomings and what is being done to correct them. We conclude that BitKeeper can dramatically improve the efficiency of Linux kernel developers.

1 Introduction

A new source control system is available - why should Linux kernel developers care? Because this particular source control system was designed from the ground up to solve exactly the problems inherent in Linux kernel development. Kernel developers need to manage thousands of files, live and work all over the world, often have limited bandwidth and connectivity, and frequently merge large numbers of changes. Older source control systems were designed for a development model where most developers worked in the same physical building and had 24-hour access to a central repository over high bandwidth local networks. The developers, rarely numbering more than 100 per project, normally checked in changes directly to the central repository, and could easily communicate with other developers working on the same part of the code. Unsurprisingly, the source control software written under these assumptions was not very useful for thousands of loosely connected developers distributed world-wide.

The BitKeeper distributed source control system was designed for, written for, and tested by Linux kernel developers. Linux kernel development provided the perfect test case for a truly distributed source control system, and BitKeeper has been and continues to be shaped by input from kernel developers. As a result, it is preeminently useful for kernel development. The purpose of this paper is to familiarize kernel developers with the most useful and time saving features of BitKeeper, so that developers can spend less time on mechanical make-work and more time on development. After reading this paper, developers new to BitKeeper may consider trying BitKeeper for the first time, and developers already using BitKeeper may learn a few new tricks.

First, we'll briefly review basic BitKeeper concepts and operations (experienced BitKeeper users should skip this section). We'll then examine a variety of problems frequently en-

[1] BitKeeper is a trademark of BitMover, Inc.

countered during kernel development and show how BitKeeper solves these problems. Next, we'll review the workflow involved in using BitKeeper for Linux kernel development. Finally, we'll discuss some of the shortcomings of BitKeeper and what is being done to correct them.

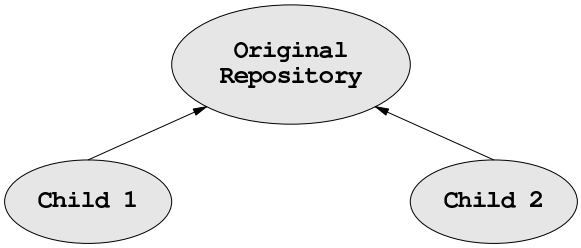

Figure 1: Parent pointers after cloning.

2 Basic BitKeeper Concepts

This section presumes knowledge of basic source control concepts such as "check in" and "check out." We will instead concentrate on the ways in which BitKeeper is different from traditional source control systems. Some of the major differences between the architecture of traditional source control systems and the architecture of BitKeeper exist in order to satisfy one of its key design requirements: Developers should be able to commit work locally, without accessing a remote repository, until the developer is ready to merge with the remote repository. Some other key design goals were reproducibility, data integrity, and performance.

2.1 Running BitKeeper

First, let's go over the nuts and bolts of using BitKeeper: How do you get it, and how do you run it? Download BitKeeper by going to:

```
http://www.bitkeeper.com
```

And clicking on "Downloads." All BitKeeper commands are of the form "`bk <command>`" to avoid namespace clashes. BitKeeper has built-in help, just run "`bk helptool`" (for the GUI tool) or "`bk help`" (for the command line tool). While BitKeeper has many useful graphical tools, a developer can work with BitKeeper using only the command line tools - BitKeeper does not require a windowing environment. We also recommend that first-time users run the demo, which is at:

```
http://www.bitkeeper.com/Test.html
```

2.2 Clones, parents, and children

A BitKeeper repository is a collection of source controlled files. To create a working copy or the equivalent of a CVS sandbox, a developer "clones" a repository. The word "clone" was chosen because cloning a repository creates an exact copy of the original repository. All of the information and administrative files in the original repository are included in the new repository, making it possible to work in any repository completely independent of any other repository. After the clone is completed, the new repository regards the original repository as its "parent." The parent of a tree can be changed at any time, to any other related tree, or to no tree at all (see Figures 1, 2). Because each repository is identical, any repository can be cloned, and the child of one repository can also be the parent of another repository (see Figure 3). Note, however, that despite all the parent-child terminology, BitKeeper repositories interact on a peer-to-peer basis, since the relationship between any two trees can be changed at any time.

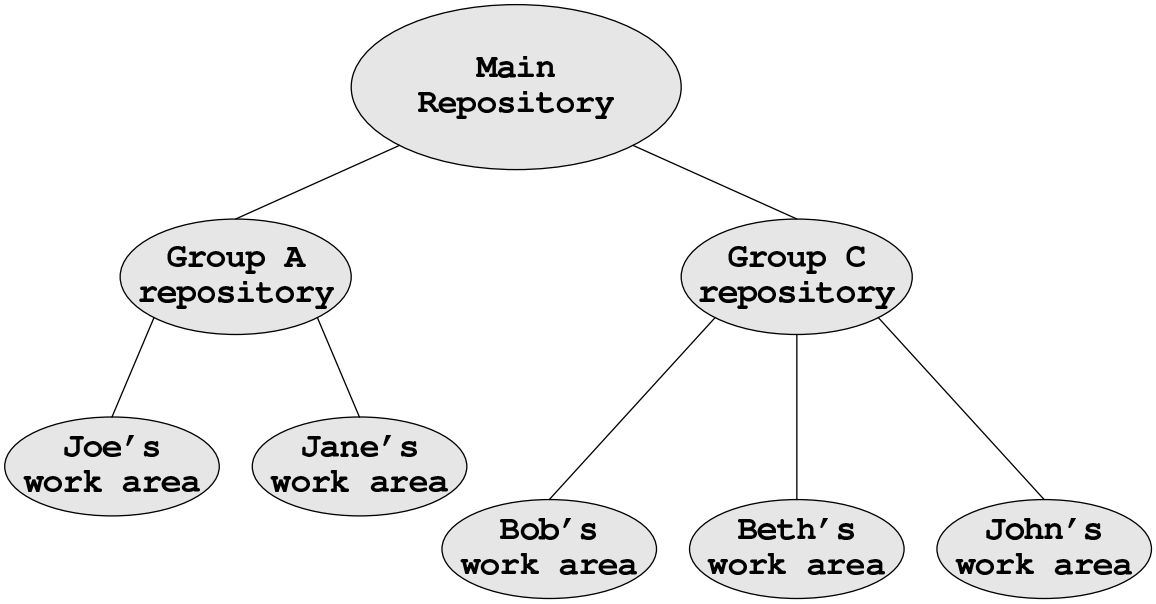

Figure 3: Example BitKeeper repository structure.

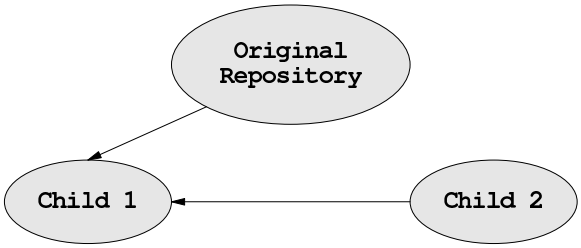

Figure 2: Parent pointers after changing with "`bk parent`".

2.3 Changesets

In BitKeeper, changes to individual files are grouped together into changesets. A changeset is a grouping of one or more deltas to one or more files representing a single logical change. Each changeset can contain multiple deltas to the same file. Each revision to each file in the changeset is commented, as is the changeset as a whole. Logically related changes to separate files can now be explicitly grouped together. For example, if one bug fix requires changes to three different files, all three files' changes can be grouped into one changeset. Being able to explicitly group changes together rather than guessing at their relationships (from last modified date, or location in the same directory) is very useful. Even more useful is that each changeset is an automatic synchronization point, similar to a CVS tag. Users can reproduce the exact state of the repository as of the point that any changeset was committed.

2.4 Push and pull

Changesets are exchanged between repositories using "`bk push`" and "`bk pull`". (Note that commits modify only the local repository, and do not affect the parent repository.) Push will send changesets from the child to the parent, and pull will retrieve changesets from the parent to the child. Each push or pull only sends the changesets which are present in one tree but not in the remote tree (see Figure 4). A push will only send changes which are already merged with the changes in the remote tree, so merging with another tree is

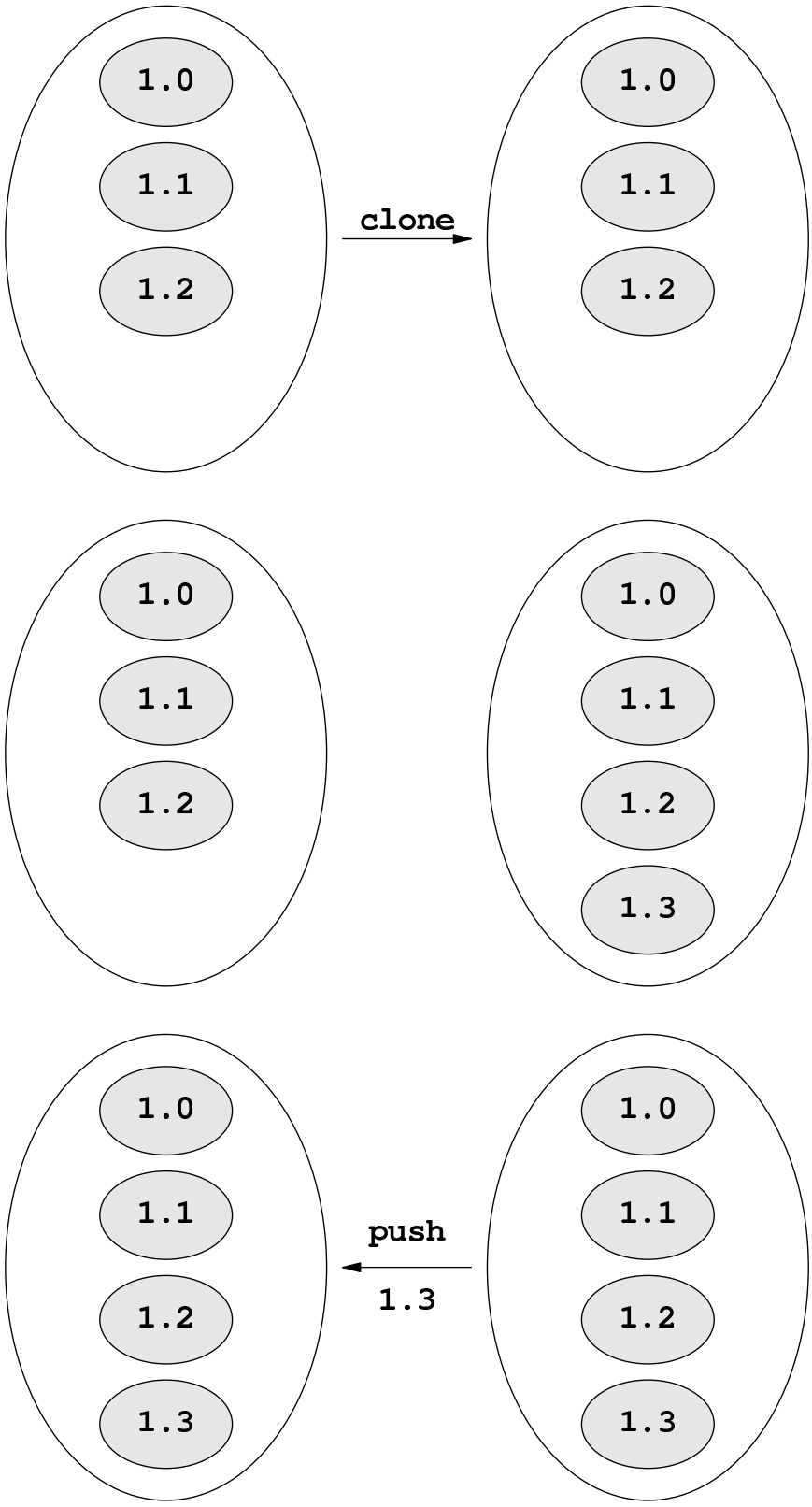

Figure 4: Example of a push: Initial clone, commit a change, push it back.

done by first pulling the remote tree's changes, merging them in the local tree, and then pushing the merged changes back. Push and pull will by default push to or pull from the parent of the local tree, but these commands can also take an argument specifying a different tree to push to or pull from.

Clones can be thought of as creating a personal, private, unnamed "branch," and pulls as a convenient way of merging with the "trunk" without pushing the "branch's" changes to the "trunk." The push-pull model gives developers control over whether or not local changes are pushed to other repositories without sacrificing ease of synchronization with other repositories. CVS users will enjoy the freedom of being able to commit half-finished changes without breaking the main tree.

2.5 Conflict resolution

Usually, a push or a pull that changes a locally modified file will be auto-merged by BitKeeper. In typical use, BitKeeper auto-merges approximately 95% of conflicts that would not have been merged by CVS or `diff` and `patch`. The percentage of successful merges relative to CVS actually increases with the number of developers working on the same repository. BitKeeper improves the auto-merge rate in two ways. First, each merge is only done once - CVS remerges from the point where the trunk and the branch first separated every time a branch's changes are pushed back to the trunk. BitKeeper only needs to merge the changes since the last changeset shared by the two repositories. Second, BitKeeper uses a unique merging algorithm that no other source control system implements. The improvement in the success rate of the merge algorithm is made possible by storing certain kinds of metadata for each file that neither CVS nor `diff` and `patch` can store or generate.

Each pull to a locally modified repository results in the creation of a changeset, which is empty if no files needed to be merged (see Figures 5, 6). Occasionally, a pull will result in a conflict that can't be auto-merged. The BitKeeper command "`bk resolve`" offers a menu of options for each file with conflicts, ranging from "Use local file" to "Merge using graphical three-way file merge tool." Once all the conflicts are resolved, the changes required to resolve the conflicts are saved in the changeset created by the merge, along with your comments. For developers who don't trust auto-merging, "`bk pull`" has an option to disable the auto-merge feature. Each conflict can then be individually hand-merged or auto-merged and the results approved before being committed. We recommend that developers try BitKeeper's auto-merge algorithm even if they have had bad experiences with auto-merging in the past; the new algorithm is an immense improvement over all previous algorithms and, in the authors' experience, always merges changes correctly.

What we just described is only the most common kind of conflict, a conflict in the data of the file itself, or a content conflict. BitKeeper also resolves conflicts in many other file attributes: permissions, ownership, type, pathname, and more. Viewing the pathname of a file as just one more file attribute makes it easy to move files around within a BitKeeper repository.

3 Kernel Development Problems and Solutions

Now that we've explained the basic terminology, let's get to the interesting part: real-life

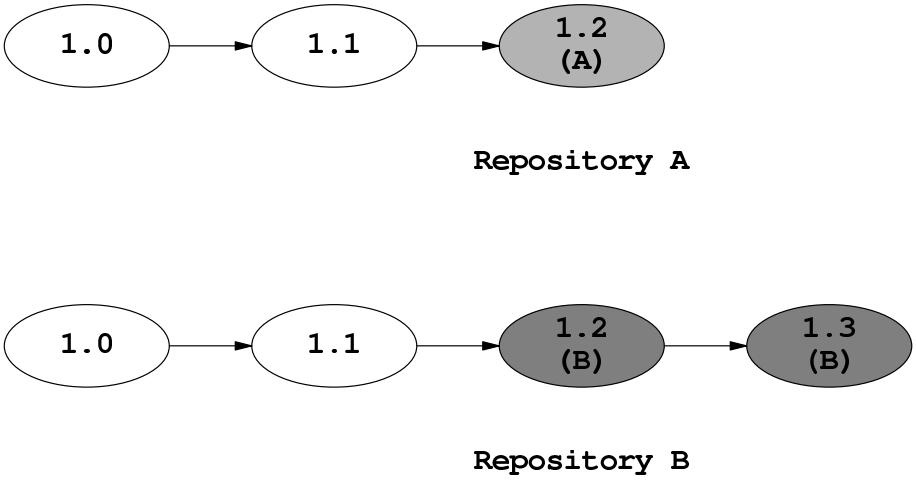

Figure 5: Repositories before merge, shaded changesets were added since clone.

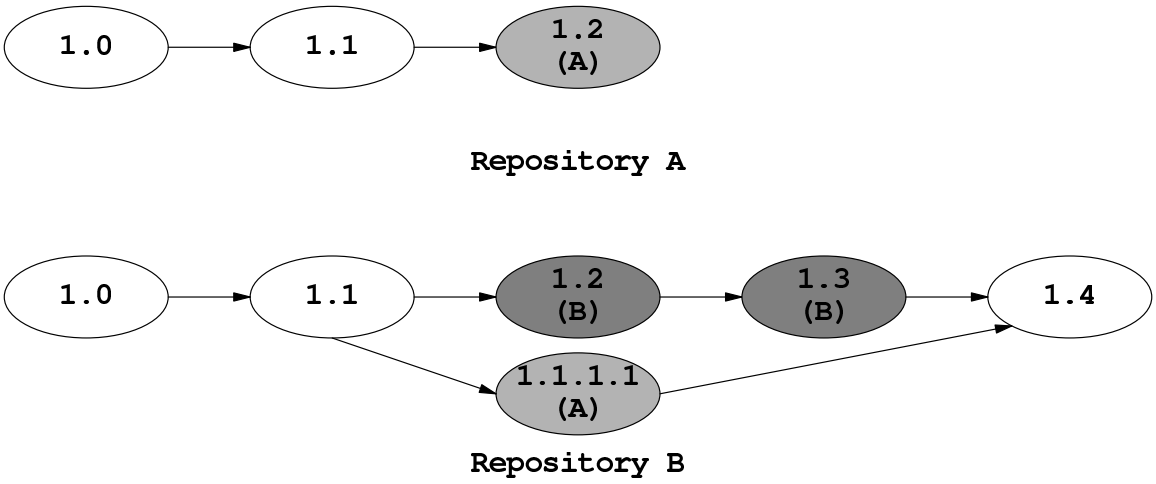

Figure 6: After a pull of A's changes to B, with A's changes merged.

scenarios where BitKeeper makes kernel development less painful. All the scenarios described were experienced first-hand by the authors while actually engaged in useful kernel development. They were not artificially constructed to show off interesting but useless features of BitKeeper but instead are commonly encountered problems solved by using BitKeeper. Some of the described solutions are implemented by other source control systems, but are not easy to do with `diff` and `patch`, the most commonly used tools for working with the Linux kernel source. We'll start with simple scenarios that are relatively easily handled by any source control more complex than `diff` and `patch`, and gradually build up to more difficult scenarios where more advanced source control management systems fail.

3.1 Maintaining different trees

Any serious kernel developer will be familiar with this scenario: You maintain both a stable kernel and a development kernel. The stable kernel contains a few bug fixes and some minor but safe improvements. The development kernel contains some riskier changes, new features that haven't been tested well yet, half-written drivers, and lots of debugging statements. Most likely, it also contains all of the changes in your stable kernel - or it would, if you always remembered to patch it with your latest changes to the stable kernel. But your development kernel is just different enough that `patch` fails to apply your diffs from the stable kernel cleanly, especially if you have moved a few files around. When you want to transfer your development changes into your stable kernel, parts of the patch usually have to be hand-applied. It's generally a pain to keep your development and stable kernels in sync.

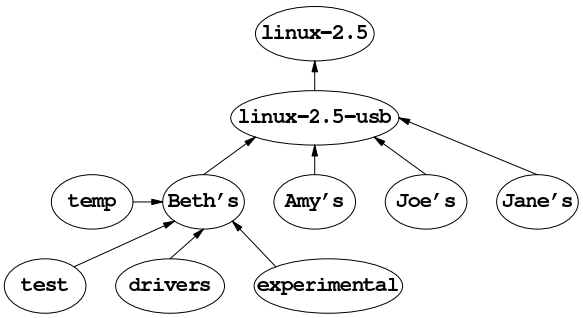

Figure 7: Example BitKeeper repository structure.

3.1.1 Solution

Clone your development tree from your stable tree. Whenever you make a change in your stable tree, run "`bk pull`" from your development tree. When your development changes are ready, "`bk push`" them to your stable tree. BitKeeper's auto-merge algorithm merges the majority of your changes for you, even when you've changed the location of some of the files.

This scenario can be generalized to any number of child repositories, each with their own child repositories. A developer could have "really stable," "stable," "semi-stable," "unstable," and "broken" trees, or one child for each set of experimental changes (see Figures 3, 7). Most developers using BitKeeper have anywhere from 5 to 50 different clones of the same repository, each for for a different set of changes. You might be worried about disk space at this point, but "`bk clone`" has an option to hard-link the files in the new repository to the files in the old repository if they are both on the same filesystem, so only the files that have actually changed take up any significant amount of disk space.[2] A clone can be

[2]This is only a partial solution, see the discussion of "lines of development" in the section "BitKeeper Drawbacks."

thought of as a branch, except that it is far easier to create and merge back to the "trunk" than in most source control systems. Cloning a new repository is so easy that you'll find yourself doing it for the most trivial of purposes.

3.2 Updating to the latest version

You went on vacation for two weeks, and now you are back and 10 patches are pending for various kernel trees. Your automatic patch application script chokes because the naming convention has changed - again. Plus, one of the patches on your local `ftp.kernel.org` mirror was corrupted and won't be updated until midnight, local time. You settle down for a long night of painful hand-application of patches.

3.2.1 Solution

"`bk pull`" downloads and applies the changes for you, regardless of what the latest kernel version was named, Linux 2.3.42, or Linux 2.4.0-test-pre7-sr71-blackbird-unstable. Data integrity checking at every step prevents any part of your tree from getting corrupted. Your update is even faster because BitKeeper compresses the information it sends over the network (it even reports the compression factor when it uncompresses the data locally).

3.3 Merging after long separation

You've decided to concentrate on getting USB working - really working, some major improvements, and you're not going to have time to merge with the vanilla kernel every time a prepatch is released. Two months later, you look up and realize that you now have 2MB of diffs between your tree and the mainline. You apply the patches, run a "`find . -name '*.rej'`" and write off getting any useful work done for the next few hours.

3.3.1 Solution

"`bk pull`" applies and auto-merges most of the changes for you. Occasionally, BitKeeper's auto-merge algorithm finds conflicts it can't resolve. At this point, "`bk resolve`" and the graphical three-way file merge tool turn what is usually 3 hours of work with `patch`, `find`, and your favorite editor into 15 minutes of point and click. The three-way file merge tool shows you the local and remote versions of the file side-by-side, with the differences color highlighted (see Figure 8). The changeset comments for each version of the file are shown above each file. The bottom half of the window shows the partially merged file, and navigation keys are described in the lower right-hand corner. When you've finished merging one conflict, by clicking on the lines from each file that you want and/or hand-editing in the merge window, hit the key to jump to the next conflict. When you're happy with the merged file, save the file and go on to the next file with conflicts. Simpler commands exist for simpler problems, for example, "Use remote file" simply replaces the local file with the remote file.

One of the authors recently merged a heavily modified 2.4.12 kernel tree with a 2.4.16-based kernel tree using `diff` and `patch` (no BitKeeper tree was available for the 2.4.16 version). It took her approximately three hours. She routinely merges from a heavily modified 2.4.12 kernel to 2.4.18 in 15 minutes,[3] using BitKeeper. Using `diff` and `patch` instead of BitKeeper wasted several hours that could have

[3] After a bit of practice. The first few merges took about 30 minutes each.

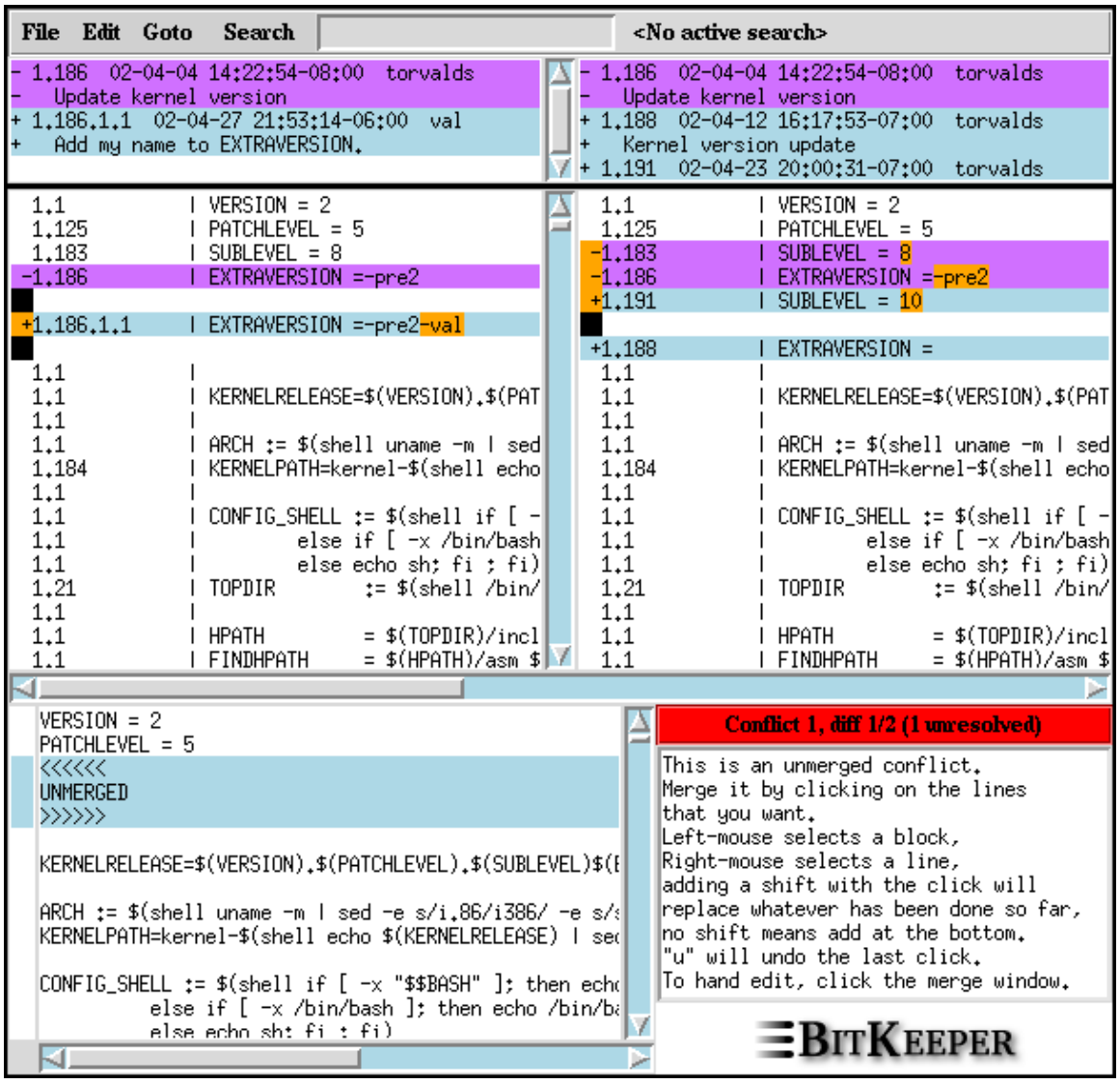

Figure 8: Merging a conflict with the three-way file merge tool.

been spent fixing a particulary vexing timer interrupt bug.

3.4 Creating a patch

Another developer asks you for your bootloader changes. They're in a tree with several other unrelated projects and a number of other changes that you don't want to send to anyone. You create a patch, hand edit out the "`misc.c~`" file that was accidentally included, and send it off. A few minutes later, the other developer emails you back saying that the kernel no longer boots, but it does print out a whole lot of debugging information. You remember that you forgot to include the changes to `head.S`, and you also forgot to remove that debugging statement triggered by the bug you fixed in `head.S`. Several more iterations and hand-edited patches later, you finally create a working patch.

3.4.1 Solution

Run "`bk revtool`" to find the changeset with the comment, "Fixed the bootloader again" and then run "`bk export -tpatch -r1.203 > ../bootloaderpatch`", which exports that changeset in unified diff format. With Bitkeeper, you naturally group related changes into one changeset with a descriptive comment. Once you've found the changeset(s) you want, BitKeeper automatically converts them into the patch format you prefer. The authors frequently have minor unreleased bugfixes requested by other developers or customers; with BitKeeper, creating and sending the proper patch takes seconds.

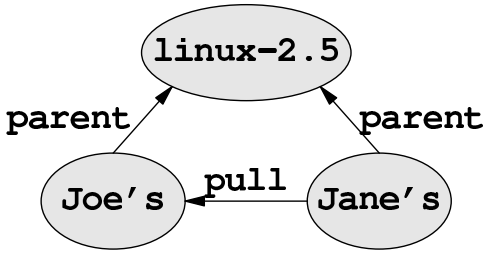

Figure 9: Example of sideways synchronization.

3.5 Sharing changes

You'd like to see another developer's changes to the memory management code, but they're not ready to be merged with the main tree. You send email asking for the patch, but the other developer has just gone to sleep. You're in a different timezone and it'll be 16 hours before you get to see the changes. 16 hours later, the two of you go through the usual "Patch doesn't apply" conversation.

3.5.1 Solution

"`bk clone`" the other developer's public BitKeeper repository. Or, if you're both working in a clone of the same repository, just "`bk pull <location of tree>`" to get the other developer's changes. With BitKeeper, developers don't even have to be at the computer to share their latest patches, as long as they have a publicly accessible tree.

This scenario is an example of sideways synchronization, one of the benefits of the peer-to-peer model (see Figure 9). Changes no longer have to committed to the central repository before any other developer can pull them. Now, any clone of the same repository can be merged with any other clone, regardless of when or how it was cloned. Sets of changes can be eas-

ily pushed or pulled around a group of related repositories in ways that are extremely useful in the day-to-day life of a kernel developer.

3.6 Moving files

You've just reorganized the `linux/drivers/` hierarchy, again. The patch is huge, so you post to the `linux-kernel` mailing list with a brief description of the change, and the eternal question, "Should I submit the changes to Linus as a patch or as a script?" The inevitable debate ensues, you write at least one buggy script, and Linus eventually gets the changes. The next prepatch is even bigger than usual, and armchair kernel hackers complain bitterly about it for weeks.

3.6.1 Solution

"`bk mv`" the files to their new locations. Since BitKeeper really implements renaming of source controlled files, rather than "abandon the old file and create a new file," the resulting changeset is tiny and almost no one even notices it happened. BitKeeper generates a unique id for each file in the repository at its creation, and will never confuse one file with another just because they happen to have the same pathname. Other developers who pull this changeset will find that their changes to the moved files are "magically" merged into the correct files at their new locations.

3.7 Debugging a patch

You apply the patch for 2.4.18-pre2 and discover that it's broken the NFS server. The patch includes changes to nearly every file in `fs/nfsd/`, and most of the changes appear to be cosmetic or related to that API change last week. You wearily page through the diff, searching for something that actually changes the behavior of nfsd.

3.7.1 Solution

The changesets you pulled are all nicely commented. You start up "`bk revtool`" and type "nfs" into the "Search" field to search the check in comments (see Figure 10), or else you select a file in `fs/nfsd/` and examine the most recent changesets affecting that file. Locating an interesting changeset, you click on "View ChangeSet" and quickly skim the beautiful, easy-to-read graphical diffs (see Figure 11). (While many graphical diff viewing tools exist, this graphical tool is integrated with the changeset viewing tools, which is a significant advantage when trying to understand related changes.) In a couple of minutes, you find a changeset with the comment, "Back out Trond's NFS changes, I don't know what they're for." Since you know Trond is the NFS maintainer, you're a little suspicious of this changeset. To check, you use revtool to find the previous changeset for that file, and you discover that it's a changeset from Trond with the comment, "Fix bug in NFS serving." You quickly run "`bk cset -x<rev>`" to exclude the changeset that reverted Trond's fix, recompile, and have NFS serving working again in 5 minutes flat.

3.8 General debugging

You notice a bug in the yellowfin ethernet driver. It's a minor bug, and it's gone unfixed for quite some time. You try a few different versions but can't quickly find the point where

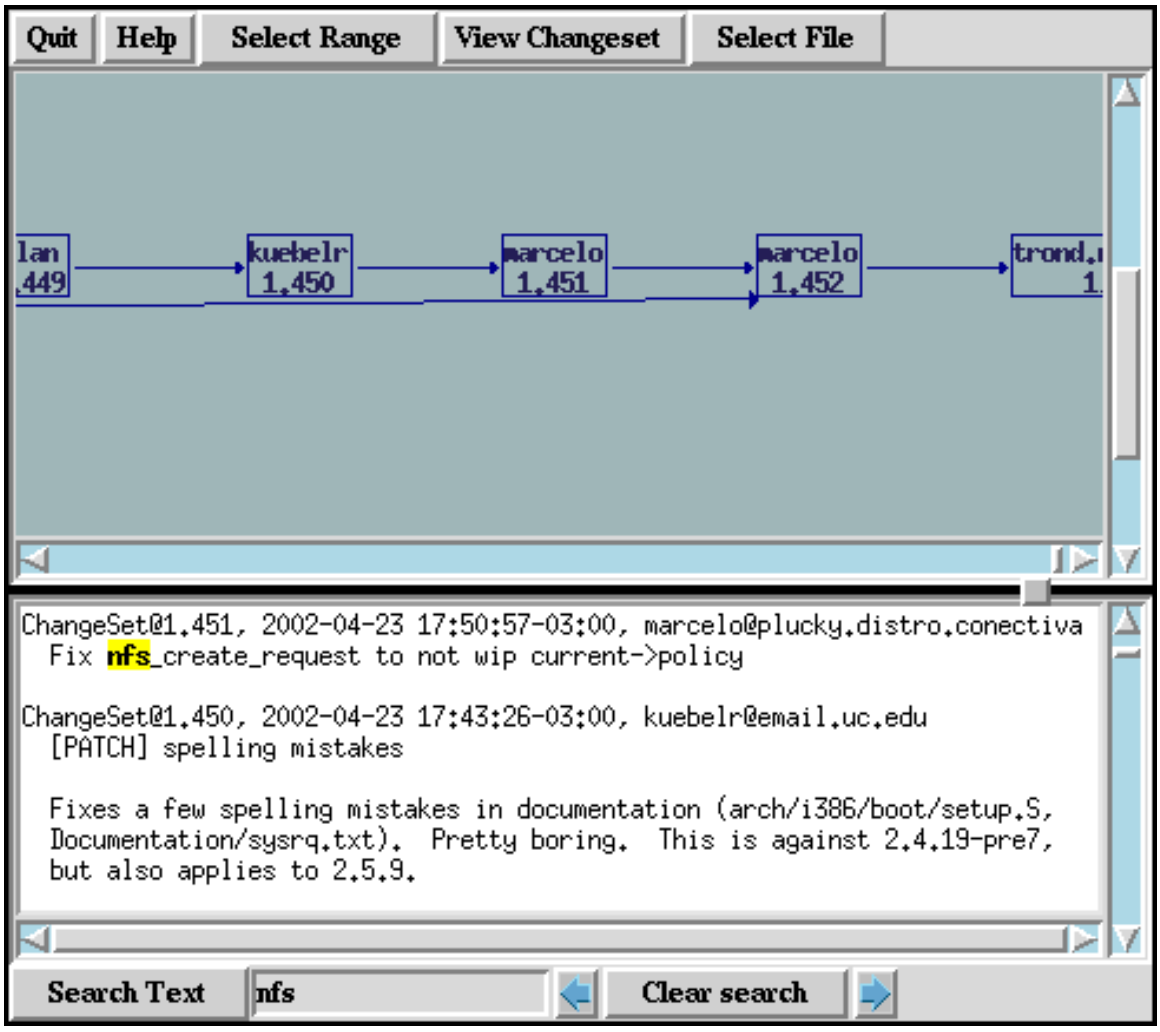

Figure 10: Searching for “nfs” in revtool.

it broke. You type the first of a long series of printks.

3.8.1 Solution

No one has a solution for all bugs, but "`bk revtool`" makes it easy to investigate the changes to a file and the reasons for those changes. Using "`bk revtool drivers/net/yellowfin.c`", you check the history of `yellowfin.c` and find a few suspicious changesets, most notably one from Dave with the phrase, "Totally untested" in the comments. Spending a few minutes with "`bk revtool`" narrows your likely suspects down to a few lines of code and gives you some preliminary ideas of what might have gone wrong. You find that a few endian-ness bugs were introduced during an API change several months ago, and repair the bugs.

3.9 Updating a port

No one has used the Gemini port of the PowerPC branch in 6 months. It doesn't even compile any more. You're new to the PowerPC port and don't know what's changed in the last 6 months. Some major reorganizations have occurred, and it looks like someone attempted to make the required changes for Gemini but never bothered compiling them. You look at other ports but each port varies so wildly that you can't find any easy examples to follow. Resigned, you start learning the PowerPC port from first principles, downloading the occasional 2MB diff and sifting through it for clues.

3.9.1 Solution

Use "`bk revtool`" to look at the context of each change to the Gemini port. After clicking on "View ChangeSet" and looking at the graphical diffs (see Figure 11), the source of many of the compilation errors quickly becomes obvious - a major reorganization of SMP support 5 months ago, where the offending code was cut and pasted without changing the variable names. The changeset that accomplished this reorganization gives you many clues about what you need to do to make the Gemini port compatible with the new system. Other bugs become obvious as soon as you look at the history of the relevant files and their associated changesets. Changes that weren't made for the Gemini were made for other ports, providing a model for your bug fixes. Some bugs are more difficult to fix, but in a day or two, you have repaired 6 months of neglect and Gemini is booting again.

We've shown only a few of the more common ways in which BitKeeper can easily save several hours a day for the active kernel developer. BitKeeper goes above and beyond merely archiving old versions of your code, it also provides a powerful set of tools for understanding code and working with other developers.

4 BitKeeper Workflow for the Linux Kernel

Now that you're using BitKeeper for kernel development, how do you merge your changes with other maintainers and contributors? The Linux development model does not work if all developers are allowed to push to one main repository, which is the only workflow allowed by most other source control systems. In-

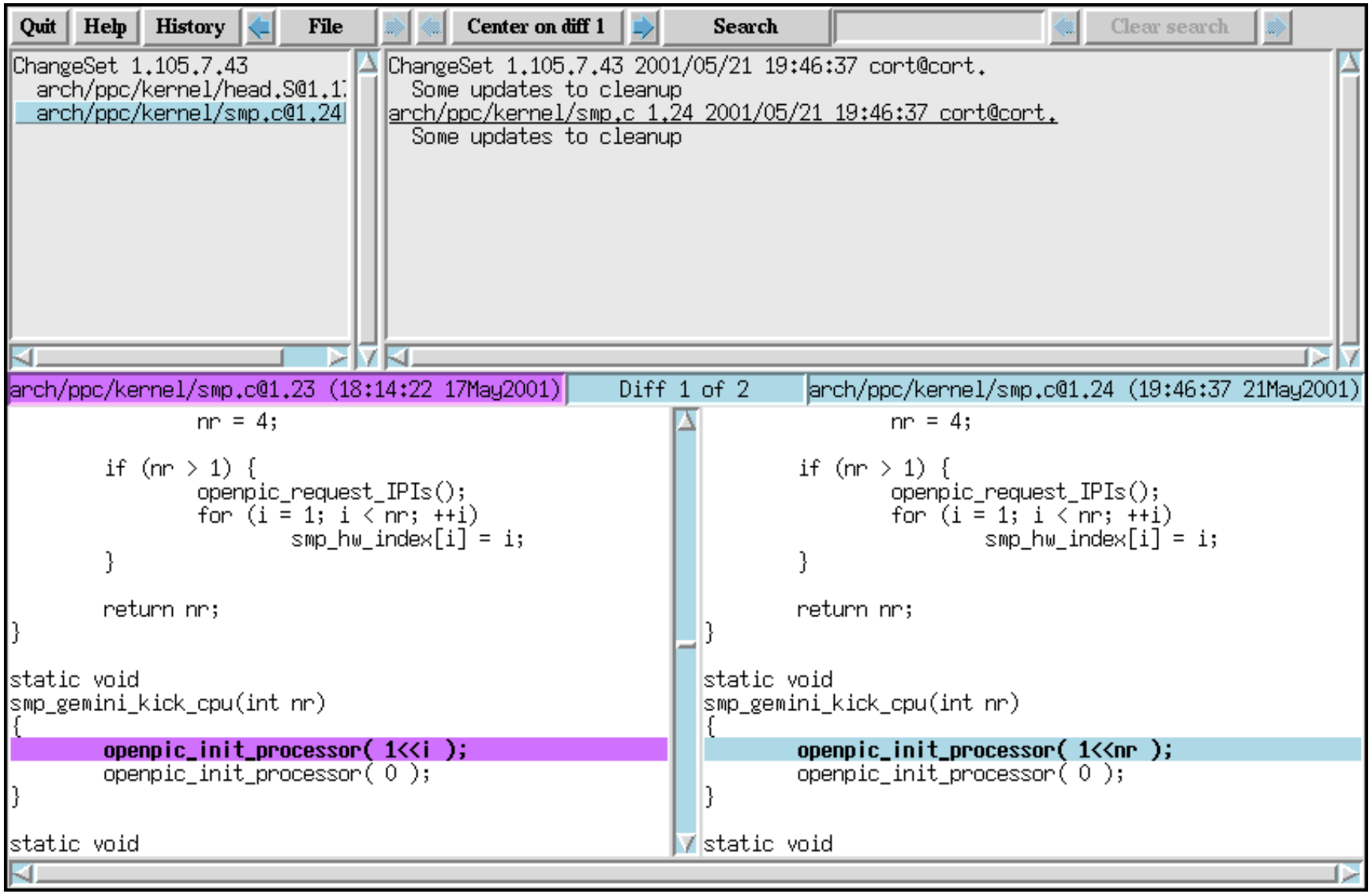

Figure 11: Change to fix a compilation error, viewed with BitKeeper's graphical diff viewer, running inside `bk csettool`.

stead, maintainers farther up in the hierarchy pull changesets from those farther down in the heirarchy, creating a series of staging repositories from the lowest levels of development up to the main repository (see Figure 3 for an example with one level of staging). Developers can also push and pull changesets horizontally between any two trees, regardless of where the trees are located in the staging hierarchy. We'll describe the kernel development workflow between a maintainer at the top of the heirarchy and a maintainer one level down.

4.1 Themes

Often the upstream maintainer will happily accept one group of changes but reject another group. Since "`bk pull`" will pull all of the changes that are in the remote tree and not in the local tree, it's important to separate out your changes into logical "themes."[4] For example, you might have the "`network drivers`" theme, the "`vm hacks`" theme, the "`utterly innocuous bug fixes`" theme, and the "`personal hacks`" theme. Each of these themes has its own tree, and for convenience, you may merge all of your theme trees into one local tree. Each of the theme trees will have as its parent the main Linux tree (see Figure 12).

Generally, the theme trees will not be merged directly with each other, but will only be pulled up into the main tree or down into your working tree. Each of these trees must have been originally created by cloning from the main tree (or a clone of the main tree, or a clone's clone, ad infinitum). The upstream maintainer can then pull your changes up with "`bk pull <location of your tree>`".

4.2 Comments

It is essential to write clear, descriptive check in comments. Not only will your comments be publicly archived for all eternity, but the upstream maintainer will want to read your comments before pulling the associated changes and, once pulled, use the comments to help decide whether or not to accept your changes. Good comments are also valuable as debugging tools, or as landmarks for navigating around the tree's history. If your first attempt at commenting your changes is inadequate, you can and should use "`bk comment -C<rev>`" to update and improve your comments later.[5] As an added bonus, very complete and detailed ChangeLogs are easily generated from your comments.

4.3 Internet accessible repository

The upstream maintainer will need access to your repository one way or another. Either give the maintainer ssh access to a machine with a clone of your repositories, or set up world readable repositories by running bkd, the BitKeeper daemon. The bkd can use HTTP ports and proxies, which allows access to your BitKeeper tree through most firewalls. BitMover provides free hosting for many BitKeeper repositories at `http://www.bkbits.net` and already hosts over a hundred personal Linux kernel repositories, including the main 2.4 and 2.5 repositories. For more information on hosting a repository, see `http://www.bitkeeper.com`

[4] Many people consider the requirement of "theme" trees to be one of BitKeeper's main drawbacks; see the section "BitKeeper Drawbacks" for information on upcoming BitKeeper features to correct this.

[5] Comments changed this way don't propagate; if that changeset is pulled into another tree, and you change the comments afterwards, the comments in the other tree will not be updated, even after a push or pull.

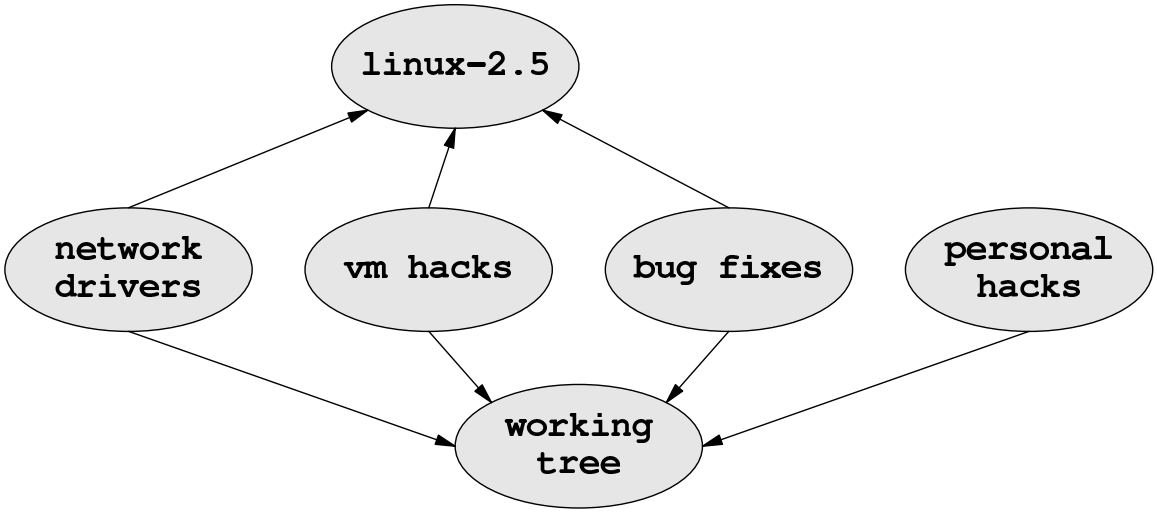

Figure 12: Graph of a typical developer's theme trees. Changes are pulled in the direction of the arrows.

`/Hosted.html`. As a last resort, you may also send your changes through email, using "`bk send`" and "`bk receive`".

4.4 Send a summary of your changes

No maintainer wants to blindly pull a set of changes. At the very least, you should send a summary of the changesets in your repository before asking the upstream maintainer to pull them. "`bk changes -L 2>&1 > ../pending`" will auto-generate a summary of all the pending changesets and their comments.

4.5 Keep your repository up to date

While your upstream maintainer can still pull your changes even if your tree isn't up to date with the upstream tree, you are offloading the work of resolving potential conflicts onto the upstream maintainer. Just like with `diff` and `patch`, your changesets are more likely to be accepted if they merge without conflicts into the main tree. It's good practice to run "`bk pull`" and merge any conflicts yourself before asking the upstream maintainer to pull your changes. Occasionally, the upstream maintainer prefers to do the merging, in which case you should allow the maintainer to pull and merge your changes. You can also perform the equivalent of a "`bk push`" without first doing a "`bk pull`" by using the command "`bk -u<maintainer's tree> send <maintainer's email address>`".

Following these recommendations will result in a smooth flow of patches to the main tree. As long as you comment well, logically separate your changes, and keep your repositories up to date, getting your submissions accepted will be easier than ever.

5 BitKeeper Drawbacks

Like all software, BitKeeper is not perfect. Some commonly requested features are the ability to subdivide repositories, to tell push

and pull to send only a subset of new changes instead of all new changes, and to support true lines of development. The main complaint is that changes currently have to be pushed or pulled in an all-or-nothing manner, requiring the creation of theme repositories in order to be able to "cherry pick" subsets of changesets. The "`bk clone`" command has an option to create the new repository by hard linking the files in the new repository to the files in the original repository, which saves a lot of space if the two clones are on the same filesystem. This is only a partial solution to the problem of needing different theme trees for the Linux style of devlopment.

Two new features addressing these problems are currently being developed for BitKeeper. The first feature is nested repositories, which allows any repository or subrepository to contain multiple subrepositories which can be cloned and checked into separately. The other new feature is lines of development, or LODs. This feature allow a single repository to have more than one "tip" to the tree, allowing two or more independent lines of development to coexist in one repository. A developer will be able to cherry pick changes from one LOD and pull them into another LOD without also pulling all the changesets that came before that changeset.

While BitKeeper is both usable and useful in its current state, development on it is not standing still. Frequently requested features are written and added as quickly as possible. In the meantime, it is fairly easy to implement workarounds for unavailable features.

6 Conclusions

BitKeeper can dramatically improve the efficiency of Linux kernel developers working both alone and with other kernel developers. BitKeeper's tools aid in understanding code, debugging problems, and merging with other developers. Common kernel development tasks, such as updating your tree and sending patches, are trivial when using BitKeeper. Most importantly, kernel developers no longer spend hours on boring tasks which can and should be automated. One of the authors estimates that she saves between 2 and 5 hours a week (about 4-10% of total working hours) by using BitKeeper instead of `diff` and `patch`. Developers who integrate a lot of code from other developers would almost certainly save even more time than that. Using BitKeeper will benefit anyone who works with the Linux kernel source, and will benefit active kernel developers most of all.

Developers interested in using BitKeeper for the Linux kernel may find the BitKeeper Linux kernel development FAQ useful:

http://www.bitkeeper.com
/Documentation.FAQS.Linux.html