APPLYING WORKING SET HEURISTICS TO THE LINUX KERNEL

Adrian McMenamin
MSc Computer Science project report,
Department of Computer Science and Information Systems,
Birkbeck College,
University of London,
2011

This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

ABSTRACT. The Linux kernel uses a page replacement policy based on global least recently used (LRU) lists. Having demonstrated the continued validity of the model of program behaviour found in the working set model, formulated by Denning, we attempted to increase the efficiency of running Linux systems, particularly those with low memory, by applying local page replacement heuristics. However, our experiments resulted in degraded performance, as they imposed additional locking burdens on the kernel, or cause the kernel to push out of memory pages that are still needed. We conclude that Denning's argument that the working set method delivers a lower space time product than simpler algorithms, such as global LRU, is flawed, but that there remains scope for local page replacement approaches that take into account the changing phase locality of running programs.

CONTENTS

LIST OF FIGURES

1. VIRTUAL MEMORY, THE WORKING SET AND LOCAL AND GLOBAL REPLACEMENT POLICIES

Multiprogramming computer systems face a fundamental problem of being able to run programs that, in sum, require more memory[1] than is physically available (Tanenbaum, 2009, pp. 173 -174). Historically, various approaches have been used to address this issue, but modern systems typically use *virtual memory* (Denning, 1970) in combination with *paging* (Randell & Kuehner, 1968). Memory is divided into a series of equal sized *page frames* and a mapping, through *page tables*, is applied, allowing programs to be in a physically disjoint set of page frames while it appears to the program itself that the memory is linearly addressable. Not all of the program need to be present at any given time and pages of the program can be *swapped* to and from secondary storage as required[2].

Intuitively, it can be seen that this model allows more programs to run than a simple summation of program memory requirements would suggest. Active programs may have needed pages swapped into memory, replacing the pages of programs that are not in the running state or even pages of the active program which are not currently needed. But each of these swaps takes time, and, as secondary storage may be of orders of magnitude slower to access and read than main memory, it is important that an efficient algorithm is used to minimise swapping by accurately determining which pages are kept in memory and which are selected for swapping out. Too much paging can cause *thrashing* (Denning, 1968a) where the CPU is idle while the I/O subsystem attempts to load the necessary pages.

In general, the optimal algorithm, Belady's MIN (Belady, 1966), will not be available: it relies on foreknowledge of what page will next be required by the system and has been described as the "clairvoyant" algorithm as a result (Love, 2010, p. 325). Instead, algorithm design has to rely on modeling and measuring program behaviour (Denning, 1980).

In 1968 Denning proposed the *working set model* (Denning, 1968b) as the basis of a practical paging and scheduling policy. As we reported in the project proposal, Denning defines the working set of information of a process at time t as $W(t,\tau)$: the collection of information referenced in the time $(t-\tau,t)$ and τ is defined as the *working set parameter*.

Let a program's reference string, the pages it accesses in time τ, be $S(\tau)$. In that time there are n references $S(\tau) : \{s_0, s_1, s_2, \ldots, s_{n-1}\}$. If, for a reasonable value of n, the program repeatedly accesses the same pages, it is said to show *locality of reference*. Our results, discussed below, confirm that programs in execution do show strong *phases* of locality, therefore supporting the contention that the working set at time t, $W(t,\tau)$ might be very similar to that for $t-\tau$, $W(t-\tau,\tau)$. Denning's proposal was that an efficient paging algorithm might be found if the pages accessed in the previous τ seconds, were in some way cached in memory while those that were not accessed in the previous τ seconds were available for swapping[3]. Additionally,

[1] We use the term memory in the every-day sense of random access memory (RAM); disk-based memory will be referred to as such or as secondary storage.

[2] We discussed this model at more length in the project proposal and it is only briefly described here. A fuller description can also be found in (Tanenbaum, 2009, Chapter 3, pp. 173 - 252).

[3] In fact Denning's proposal was that those pages not accessed in τ be automatically discarded, though practical implementations, discussed below, only swap out when it is necessary to free space.

he stipulated that a program only be allowed to run if all the pages in its working set were available in memory.

Denning has also noted (Denning, 1980), and again our results confirm this, that the transitions between the phases of locality are frequent and disruptive. His argument is that a working set policy, where pages are held in memory because they have been accessed in the previous τ seconds, is an effective way of managing the transitions between phases, where more than one region of locality is likely to be accessed: in the period of phase transition the working set can expand as the program accesses pages from different localities.

Denning points towards a *local* replacement policy. His proposal is that the pages cached for use by a given program be determined by that program's access patterns, and not global considerations. Thus, in a time of phase transition the number of pages cached will likely rise, while in a period of strong locality it will shrink.

Pure working set models would require the time of every access to a page to be recorded, increasing the complexity and storage space or hardware required (Wilkinson, 1996, p. 137). Modern practical local replacement implementations accordingly use simpler approaches which only approximate to a pure working set model. One such model is found in the Open VMS operating system, which uses a local page replacement policy based on a modified form of a first in, first out (FIFO) queue to select as its first candidate for replacement the page that has been in a process's working set for the longest amount of time, though it is not automatic that this page is released (Goldenberg, 2002, pp. 320 - 321). Windows NT and its successors share a design heritage with the VMS family (David N. Cutler's foreword, Custer, 1993, pp, xv - xvii), and that operating system and its descendants also implement a local page replacement policy with some global aspects (Friedman, 1999). Both operating systems avoid setting a working set parameter based on time, and Windows NT's page replacement algorithm is similar to the *page-fault-frequency replacement algorithm* (Chu & Opderbeck, 1976) with limits on per-process caches strongly influenced by the page fault rate.

The Linux kernel, however, uses a *global* replacement policy, essentially a variant of a global LRU (least recently used) policy, where the page frame, from the global pool of allocated page frames, that was least recently accessed is made available for replacement. A global policy is favoured because it is said to have a lower overhead (Jiang & Zhang, 2005). But, while this approach is also designed to hold a program's working set in memory (Gorman, 2004, p. 164) and Denning (Denning, 1980) regarded global LRU as a "relative" of his working set model, plainly there are differences, as Denning himself has argued elsewhere (Denning, 1968a).

Linux's global replacement policy is a variation on the 2Q approach for database management outlined by Johnson and Shasha (Johnson & Shasha, 1994). The operating system maintains two *LRU queues* for both file-backed and anonymous (such as data allocated on the heap) memory[4]. Pages move from one queue to another as shown[5] in Figure 1.1 (cf. Tanenbaum, 2009, p. 767).

Most page frames are first added to the 'inactive' queue and are moved to the 'active' queue after a second subsequent reference. When page frame reclaim is

[4]An additional list of unevictable pages is also defined: see `/include/linux/memzone.h` in the Linux kernel sources.

[5]This figure was first prepared for the project proposal

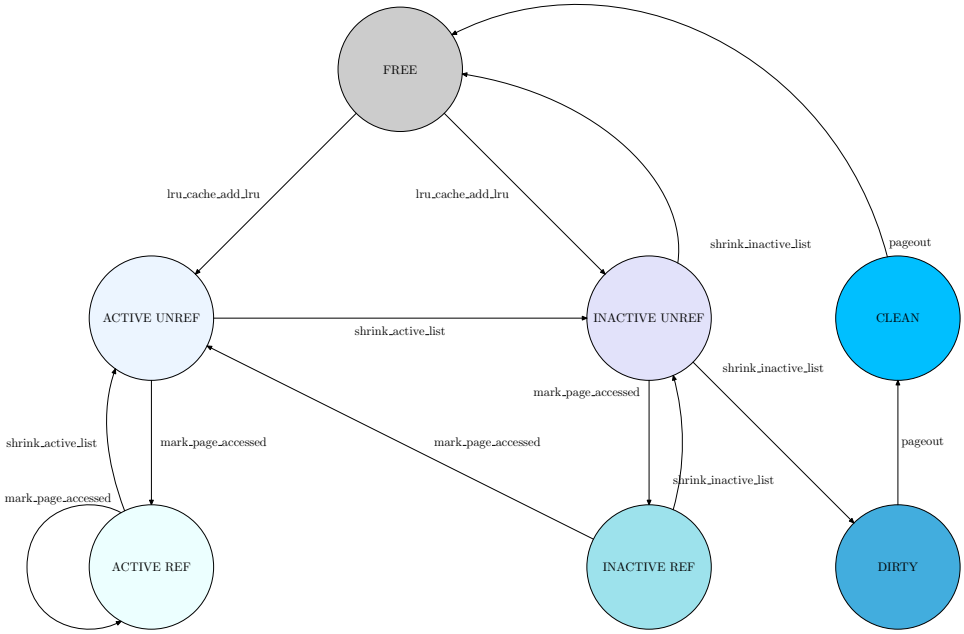

FIGURE 1.1. Life cycle of a Linux page frame

required, pages are taken from the inactive queue (with updates flushed to disk if required). This two-queue (2Q) system is designed to guard against pages with a long or even infinite *reuse distance* (time until a subsequent access) but small *recency* (time since last access) remaining cached at the expense of pages that are often accessed but have not been recently.

Candidates for reclamation (and for eviction from the active list and on to the inactive list) are selected through a modified (CLOCK-PRO) form of the CLOCK algorithm (Jiang *et al.* , 2005)[6]: the kernel aims to keep the active and inactive lists in balance and pages can be moved between the lists or evicted depending on whether they have been accessed since the last balance (this corresponds to `shrink_active_list` and `shrink_inactive_list` in Figure 1.1). The mechanism is highly portable between the many architectures Linux supports(Mauerer, 2008, p. 1029).

In the sections that follow we describe how we first tested whether the experimental results from which the working set model was developed in the 1960s and 1970s still hold true for the GNU/Linux operating system, before describing how we tested ways in which the insights that the model brings can be applied to the Linux kernel.

[6]The Linux implementation of the CLOCK-PRO algorithm outlined in this paper has only been partial, however - see `http://linux-mm.org/PageReplacementDesign` - accessed 15 August 2011.

2. TESTING FOR LOCALITY IN MODERN PROGRAMS

Our tests show that modern programs show both both phases of locality and disruptive transitions. Using the Valgrind framework (Nethercote & Seward, 2007) we were able to record the pattern of memory accesses made by programs in execution and we developed a small suite of tools (described in Appendix A) to interpret and graph the results.

We tested the memory access patterns of the GNU Compiler Collection (GCC) C compiler[7], the Mozilla Firefox browser[8], the MySQL database server daemon[9] and Xterm, the standard X Windows terminal emulator[10]: all running on Debian GNU/Linux Wheezy[11] on a machine with 12 Intel i7 processors and 25GB of memory.

Valgrind's behaviour as an emulator means that the absolute memory addresses accessed are altered, but the pattern of memory accesses is not. Valgrind also allowed us to plot memory accesses against process virtual time as we could mark accesses against the number of instructions executed.

Figure 2.1 shows the memory access patterns for Xterm[12] across the whole of its virtual memory map. The stack accesses can be seen at the highest memory addresses, but at this scale little detail can be seen of the pattern of memory accesses at lower addresses (cf. Mauerer, 2008, pp. 290 - 294).

[7] See `http://gcc.gnu.org/` - accessed 16 August 2011

[8] See `http://www.mozilla.com/en-US/about/` - accessed 16 August 2011

[9] See `http://www.mysql.com/` - accessed 16 August 2001

[10] See `http://www.x.org/wiki/Home` - accessed 16 August 2011

[11] See textttthttp://www.debian.org/releases/ for an explanation of the different releases of Debian GNU/Linux. At the date of access (16 August 2011), Wheezy was the distribution in 'testing' while Squeeze was the 'stable' distribution.

[12] Xterm was opened from the command line and closed manually.

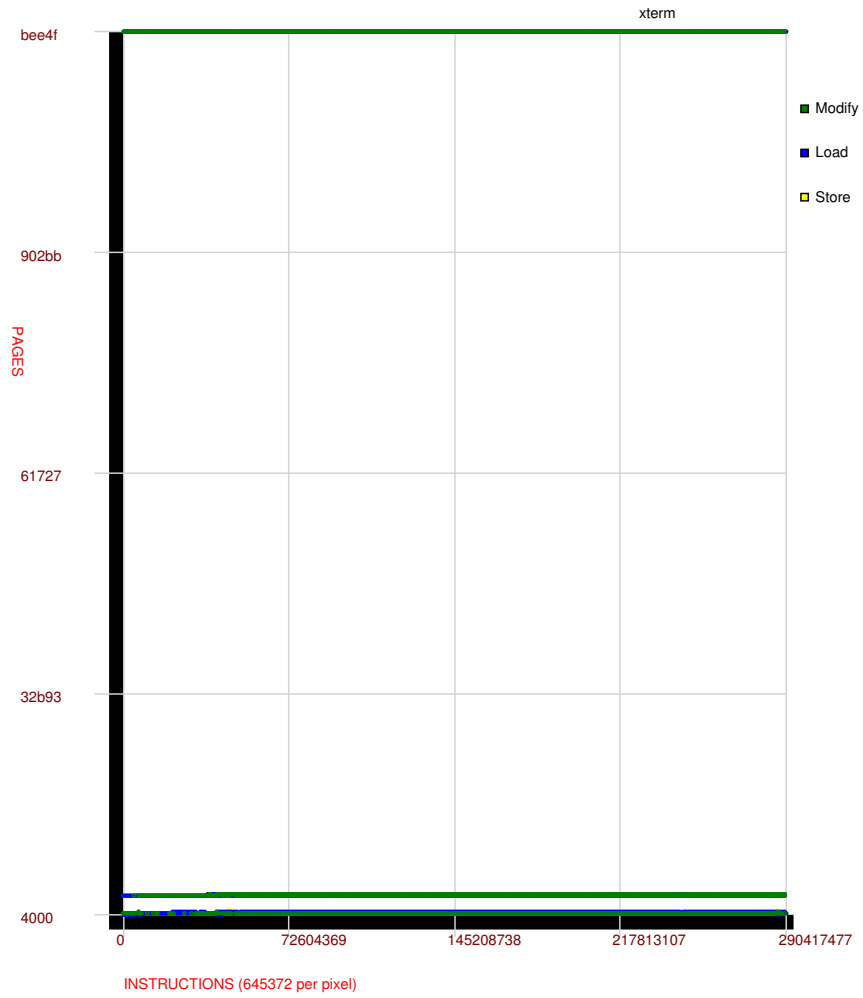

FIGURE 2.1. Xterm memory accesses over all full virtual memory range

Concentrating on memory references in the lowest 1% of the process's address space we can immediately see both the phases of locality and the transitions between them (Figure 2.2).

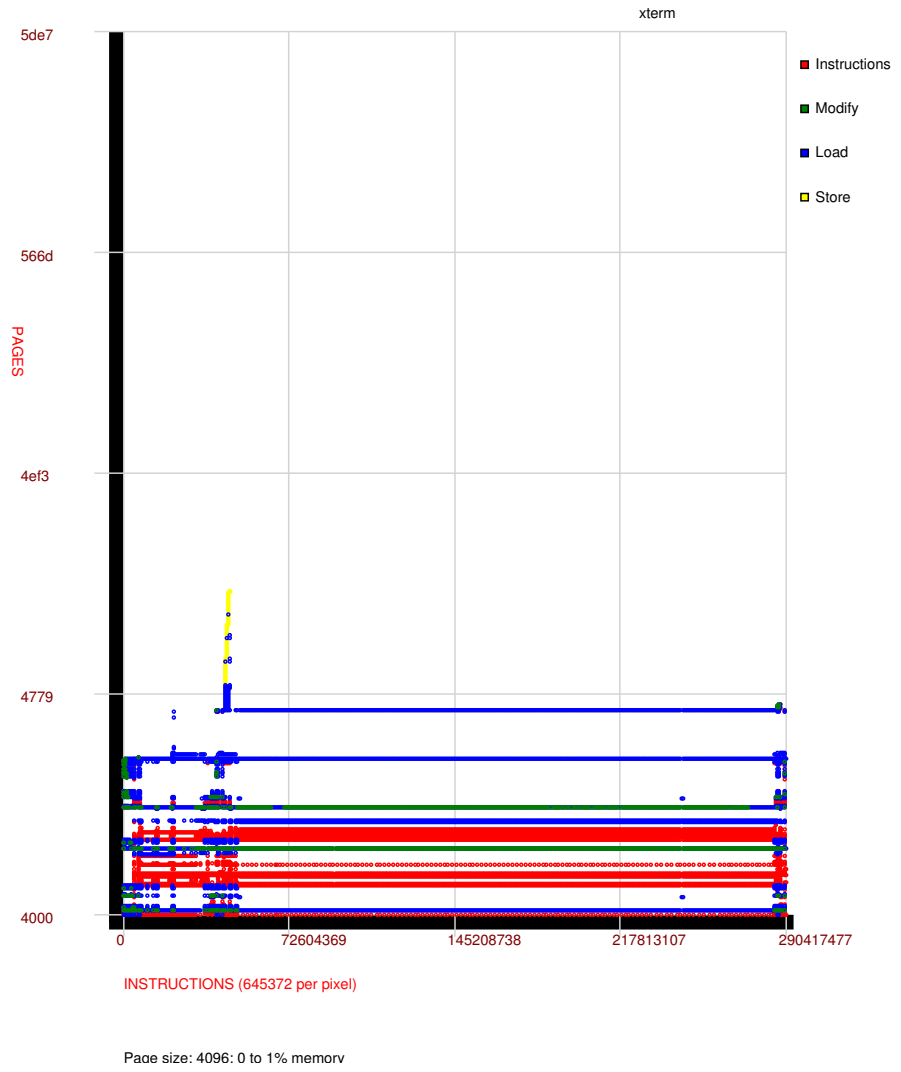

FIGURE 2.2. Xterm memory accesses at low addresses

With Mozilla Firefox, looking at the lowest 2% of the address space, the transitions between phases of locality are even more apparent, as shown in Figure 2.3[13].

[13]We configured Firefox so it would automatically close when it loaded a page with a Javascript instruction to close the window.

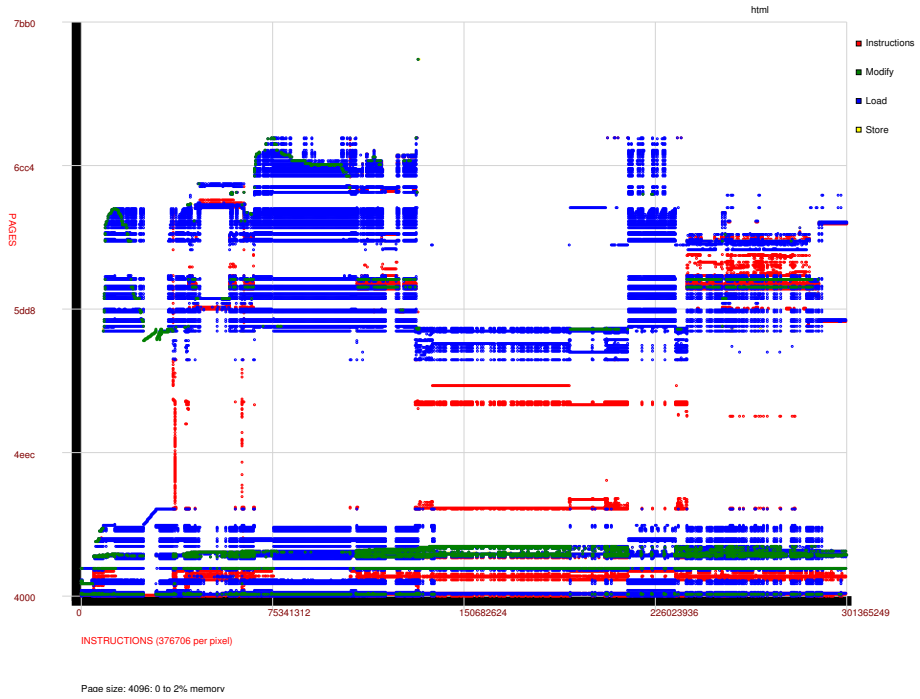

FIGURE 2.3. Mozilla Firefox memory access patterns

The GCC C Compiler[14] (Figure 2.4) and the MySQL daemon[15] (Figure 2.5) complete this brief survey. In all cases whilst phases, often repeated over time, of locality are clearly visible, the relatively sudden nature of the switches between these different phases are also visible. We examine this more closely below.

[14]Compiling the valext.c file (see Appendix B)

[15]The daemon was run under Valgrind, and a MySQL client session briefly opened and closed on the same machine, before the daemon was shut down.

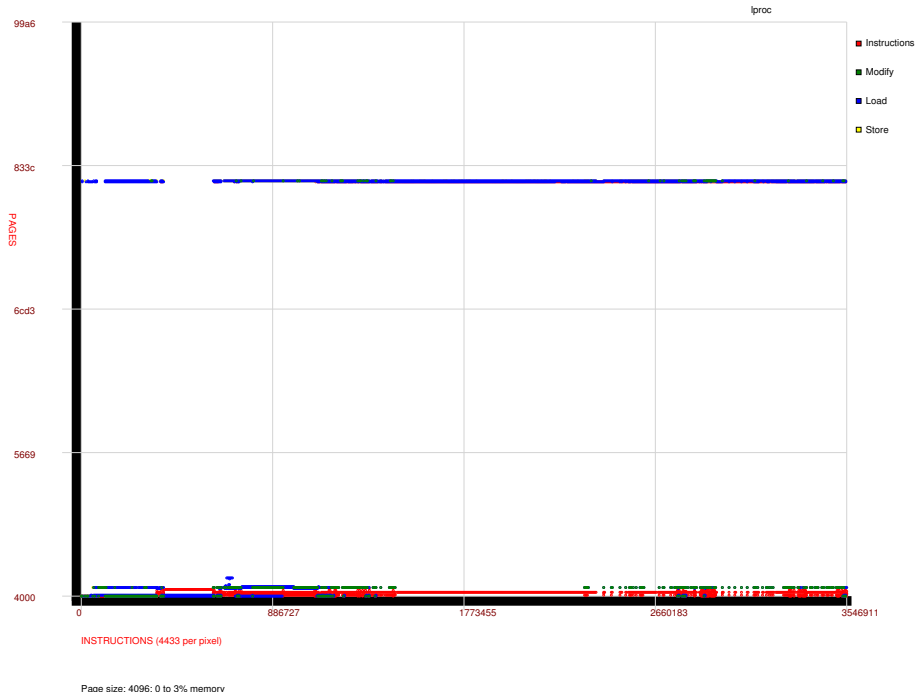

FIGURE 2.4. GCC Compiler memory access pattern

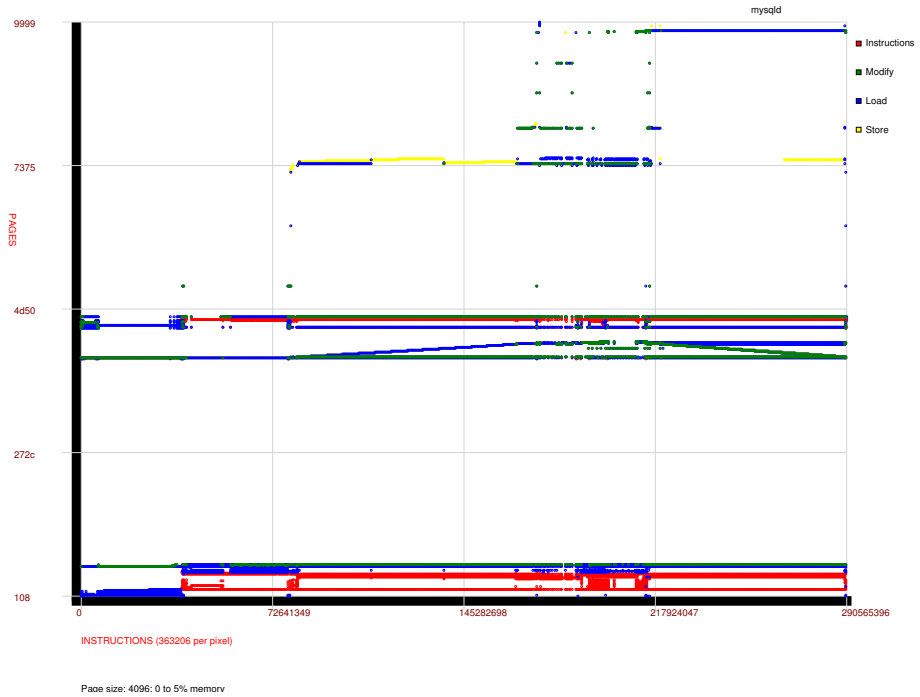

FIGURE 2.5. MySQL daemon memory access pattern

3. TRANSITIONS BETWEEN PHASES OF LOCALITY

It is plain from all the results above, most clearly in the case of Mozilla Firefox, if less prominently with the GCC C Compiler, that switches between phases of locality are frequent, and our results confirm that the interaction between the phases can mean the size of the working set for a given τ can change rapidly. Figure 3.1 illustrates this for Xterm using the same data set used to generate Figures 2.1 and 2.2 for an arbitrary value of τ. Picking a larger value for τ will smooth out the peaks and troughs, but at the expense of caching more memory on average. In Figure 3.2 different working set sizes for Mozilla Firefox (using the same lackeyml data from which Figure 2.3 was generated) with varying τ are shown. The first graph shows the working set for τ of 376,076 instructions (the figure is an artifact of the width of the graph, being the smallest value for τ that could sensibly be shown for a graph 800 pixels wide using this dataset), the second shows the working set for a τ of 1,000,000 instructions and the third for a τ of 10,000,000 instructions. The third graph is somewhat smoother than the first, though it will be noted that sharp increases and decreases in the working set size are not avoided, but also has a peak working set size of 982 pages, as opposed to 628 pages for the τ of 376,076 and 749 pages for the τ of 1,000,000.

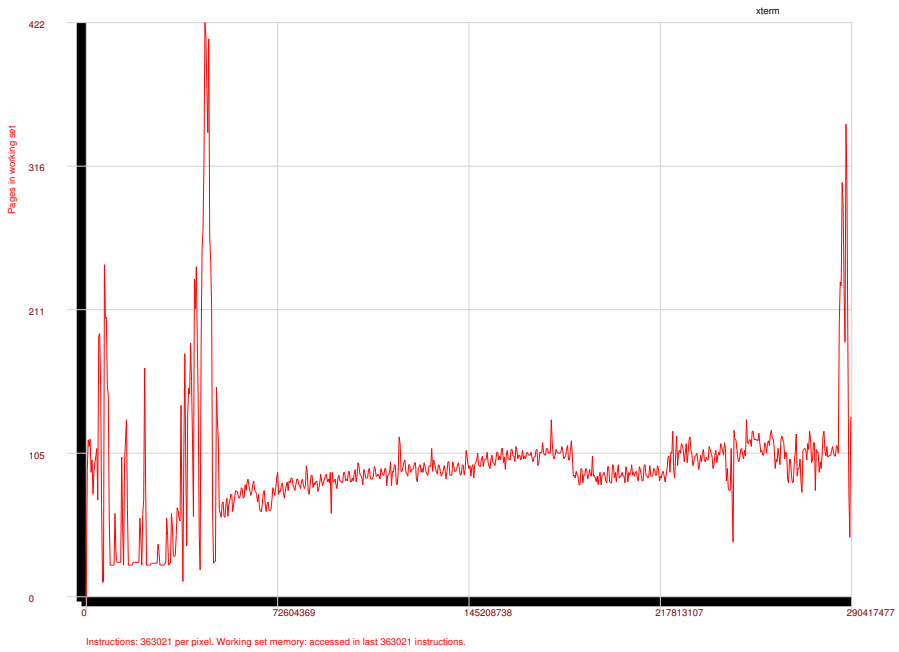

FIGURE 3.1. Working set size for Xterm for an arbitrary τ (as approximated by instruction count): the working set size changes rapidly.

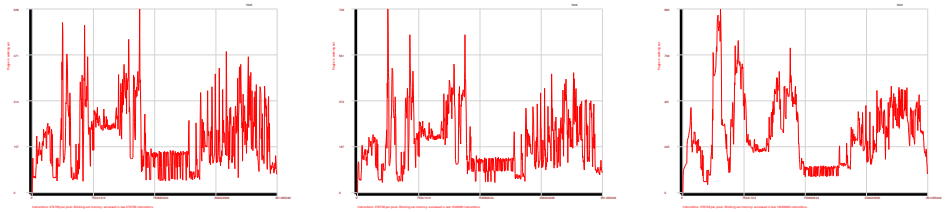

FIGURE 3.2. Working set size for Mozilla Firefox with different values of τ: (left - right) 376,076 instructions, 1 million instructions, 10 million instructions. Larger values of τ smooth the curve but rapid changes remain.

To further investigate this we developed an additional software tool, described in Appendix B, that used both the Linux kernel's ptrace facility (Padala, 2002) and the `/proc/pid/pagemap`[16] interface to step through program execution and examine the actual use and availability of memory. Figure 3.3 shows the actual use of memory by a running instance of Mozilla Firefox. As this reflects the real page replacement policy of the Linux kernel, it is of limited comparison to the examples presented above[17], but it does clearly show that memory demand can spike sharply, as the results above suggest.

[16] See `Documentation/vm/pagemap.txt` in the Linux kernel distribution

[17] In addition the x-axis measures steps in the program's execution as opposed to the number of instructions read

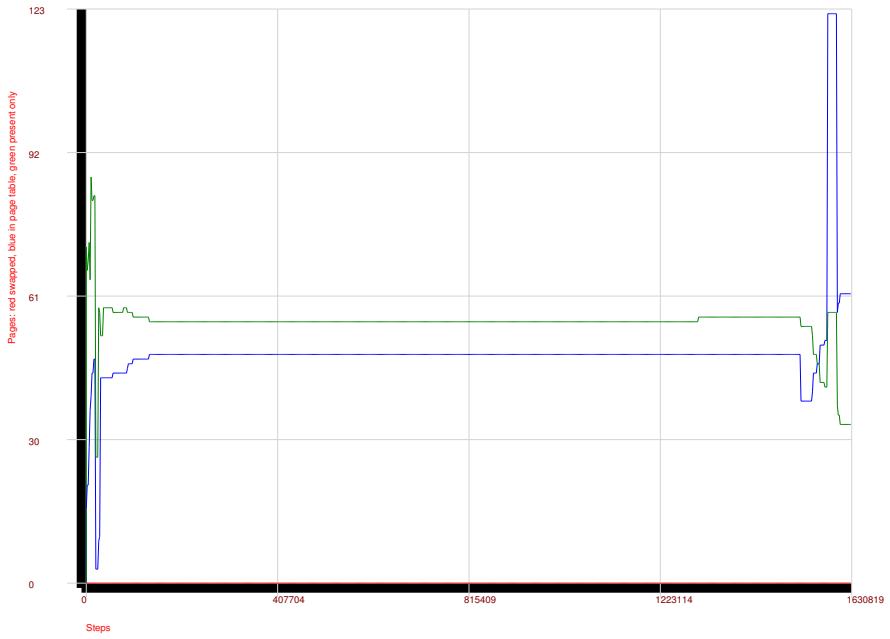

FIGURE 3.3. Actual memory use in Mozilla Firefox, blue line represents pages mapped and in page table, the green line space reserved in virtual address space only. Pages on y-axis, program steps on x-axis.

4. PROGRAM LIFETIME FUNCTIONS

From Belady and Kuehner (Belady & Kuehner, 1969) we take the concept of a program's *lifetime function*:

> "In any given system the lifetime function relates the average length of the execution intervals (e)[18] to the different storage sizes (s) in which the program can be compelled to run."

In the same paper they define a program's *space-time product*, C (for the cost of storage), as the integral of memory occupied over the (wall clock) time of execution (here between t_0 and t_1):

$$C = \int_{t_0}^{t_1} s(t)dt \tag{4.1}$$

Where $s(t)$ is the memory used at time t. They state:

> "It is important to note that the real time occupancy of information in store can be much larger than the amount of processing time given to the associated task during that time interval."

[18] ie., between faults which require access to secondary storage - report author's added note.

Denning (Denning, 1980) restates the space-time product in an importantly different way. He describes the space-time product as:

> "A program's space-time product is the integral of its resident set size over the time it is running or waiting for a missing segment to be swapped into main memory."

It will be noted that this definition, unlike the former one, excludes the time in which the program is idle because another process is running or because operating system code (eg., the scheduler) is being executed. We think this is an important difference and is a weakness in Denning's subsequent argument for the working set method as best practicable scheduling and replacement algorithm, despite the experimental evidence we have also collected that supports many of his other arguments. We return to this in Section 8.2.

Using his definition Denning restates C:

$$C = \sum_{t=1}^{T} s(t) + D \cdot \sum_{i=1}^{K} s(t_i) \tag{4.2}$$

Where $s(t)$ is the resident set at *virtual (program) time* t, D is the mean delay while waiting for a page (segment) to be swapped in (which happens K times during program execution) and $s(t_i)$ the resident set while waiting for the i^{th} swap to complete. The first term in (4.2) simplifies to $x \cdot T$, where x is the average resident set size while the program is executing. The second, argues Denning, may be approximated by $D \cdot x \cdot K$, and as $x \cdot K = T \cdot (x \cdot K/T)$ we can restate (4.2) as:

$$C = x \cdot T \cdot (1 + D \cdot K/T) \tag{4.3}$$

K/T is the average fault rate, which, after Denning, we call $f(x)$ (as it is obviously dependent on the size of x) of the program, the inverse of the lifetime function as defined above, which we call, again after Denning, $g(x)$ giving:

$$C = x \cdot T + x \cdot T \cdot D/g(x) \tag{4.4}$$

In (4.2) T represents virtual time of execution, so is a constant independent of the number of faults or the size of the resident set,[19], hence:

$$C \propto x + x \cdot D/g(x) \tag{4.5}$$

Accepting Denning's formulation, we can therefore see that C is likely to be minimised, given a sufficiently large D, when $g(x)$ reaches a point of inflection (Belady and Kuehner locate the point at where $g''(x)$ is zero or changes sign). Denning's argument is that the working set method will deliver a 'primary knee', the global maximum of $g(x)/x$, at a smaller value of x than the alternatives, making it a more efficient approach as measured by the space-time product.

Using the data produced by Valgrind and processed by Lackey_ml, we plotted (as shown in clockwise order in Figure 4.1) the lifetime functions for Xterm, Mozilla Firefox, the MySQL daemon and the GCC C Compiler.

[19] Of course, as Denning acknowledges, it is possible or even likely that D may vary with the fault rate, as swap requests are queued, but we ignore this here for sake of clarity of exposition.

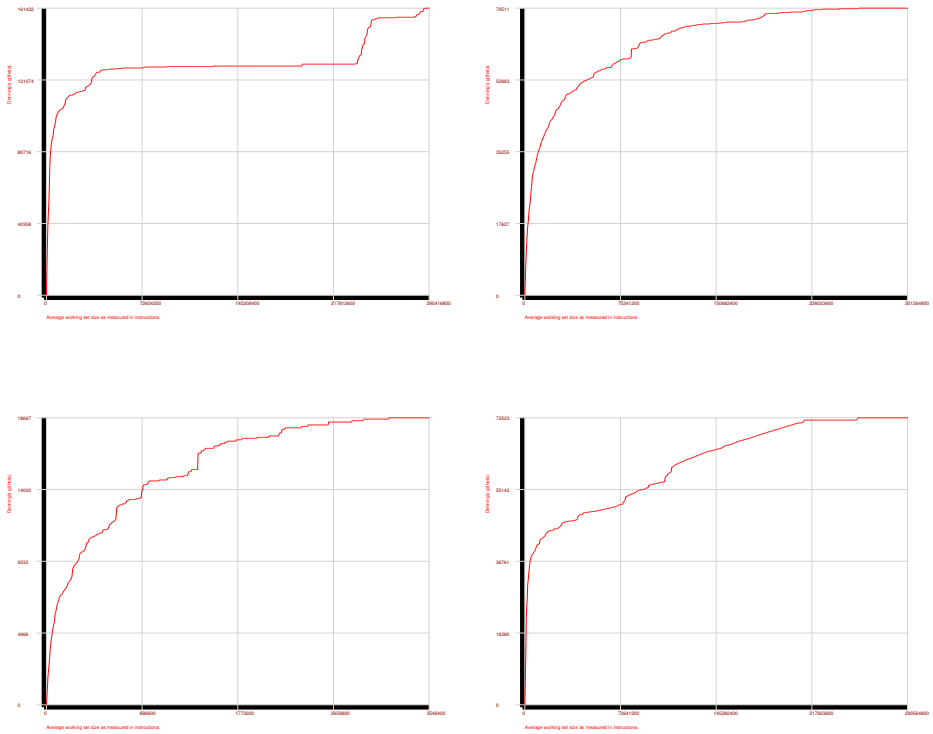

FIGURE 4.1. Lifetime functions for various programs using the working set model. Clockwise from top left: Xterm, Mozilla Firefox, MySQL daemon, GCC C Compiler.

As suggested by Denning, all these show a sharp 'knee' at a relatively small value for what is referred to here (after Denning) as the working set parameter θ (the time, as measured by instructions executed, over which the working set is measured). Beyond this 'primary' knee pages may be held for longer times with diminishing impact on $g(\theta)$ - in the case of Xterm (top left) the graph is close to flat for a long range of increasing θ: clearly page frames allocated to hold the pages of one program's working set cannot be used to hold those of another and this long flat line illustrates that an over-large θ may not just limit multi-programming in this way, it may also deliver little benefit even to the 'over-allocated' program.

We also compared these results to the lifetime functions of the same programs under an LRU replacement policy (again arranged clockwise from the top left as Xterm, Mozilla Firefox, the MySQL daemon and the GCC C Compiler[20] in Figure 4.2). In this case θ is taken to be the average working set (or cached pages) count under either method (red for the working set policy, blue for the LRU policy). In all cases the working set approach shows a 'primary knee' at a lower value for the average resident page count than for LRU, and at larger values for the average resident set, LRU generally delivers longer periods of execution between faults. In his Figure 4 in (Denning, 1980), Denning appears to suggest that the working

[20]The seeming zig zag in the top right of the LRU lifetime function appears to be an artifact of the data

set approach will deliver better results than LRU even at some higher values of θ: our results do not support that, though the performance difference between the two approaches may not be great.

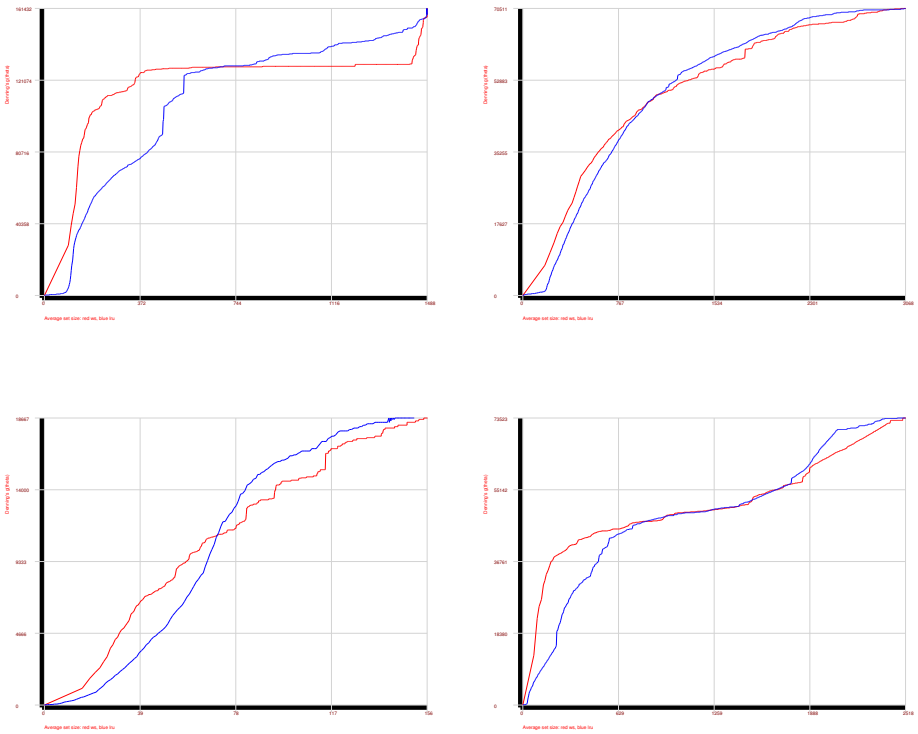

FIGURE 4.2. Working set and LRU lifetime functions compared. Working set in red, LRU in blue. Clockwise from top left: Xterm, Mozilla Firefox, MySQL daemon, GCC C Compiler.

It should, of course, be noted that the Linux kernel operates a global modified LRU policy and not the local policy that we are testing here. If our program P runs under a local LRU policy then its fault rate $f(x)$ will be $1/g(x)$, under a global LRU, we can show that the average fault rate for all programs $\bar{f}_G$ will be the same as the average fault rate $\bar{f}_L$ under a local policy.

Let F_G be the global fault rate in a system with N programs running:

$$F_G = \sum_{i=1}^{N} f_i(x_i) \tag{4.6}$$

Then:

$$\bar{f}_G = \frac{1}{N} F_G \tag{4.7}$$

and:

$$\frac{1}{N}F_G = \frac{1}{N}\sum_{i=1}^{N} f_i(x_i) = \bar{f}_L \tag{4.8}$$

Hence, while we cannot use the above graphs as a guide to how the *individual* program might fare under a global LRU, we can use the *pattern* they have in common as a guide to how programs behave under LRU.[21]

4.1. **Distribution of working set sizes.** With the data available it was thought useful to also examine the distribution of working set sizes. In the past different results have been reported by different researchers. Denning and Schwartz (Denning & Schwartz, 1972) predicted that working set sizes would be normally distributed, and this assumption was still being used in recent, eg., (Schlesinger & Garrido, 2007, p. 286) textbooks. However, as Denning later acknowledged (Denning, 1980) experimental results do not always support this, eg., (Bryant, 1975).

Using XSLT to extract data from the graphs of working set size and the R programming language (Hornik, 2011) we examined the working set size for the MySQL daemon with three different values of θ.

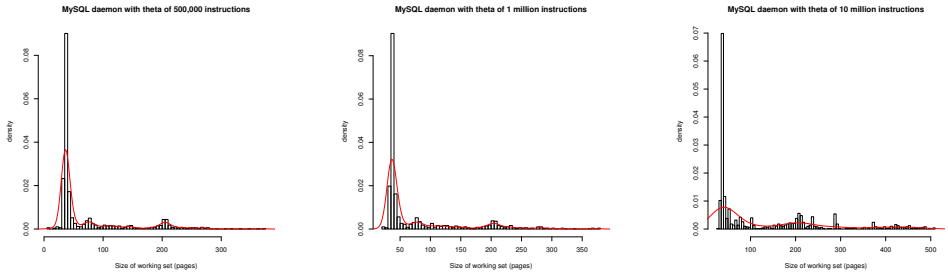

FIGURE 4.3. Working set size distributions for various values of θ (500,000, 1 million and 10 million) for the MySQL daemon, all show multiple maxima.

As can be seen in Figure 4.3 none of the examples (for a θ of 500,000 instructions, of 1,000,000 instructions and of 10,000,000 instructions) show a normal distribution, though the plots for the two smaller values do show a region at the low end which may be locally normal: suggesting that Spirn's view that working set sizes may be normally distributed within locality phases may be correct (Spirn, 1977, p. 62) and that "within each phase there is a preferred working set size". For Mozilla Firefox, a visual comparison of Figures 3.2 and 2.3 points to larger working sets at times of phase disruption and smaller, more stable sized sets during periods of strong phase locality. However, we did not investigate this further.

[21] In fact the consensus appears to be that global LRU delivers better performance than local LRU - see `http://www.sinenomine.net/sites/www.sinenomine.net/files/Hillgang_Wheeler-1.pdf` - accessed 28 August 2011

5. APPLYING LOCAL REPLACEMENT POLICIES IN THE LINUX KERNEL

Our aim[22] was to test the thesis that *applying techniques used in local page replacement and suggested in the working set model can lead to improved performance in the Linux kernel*. The preceding sections should make it clear why this was considered worthwhile:

- Phase disruption is an important part of a program's life cycle and the working set method, and a local replacement policy with variable sized resident sets is adapted to that (Sections 2 and 3 above);
- Our experiments appear to confirm that, if we accept Denning's formulation for the space time product in equation (4.2), working set policies can operate with a smaller space-time product than the LRU approach used in the Linux kernel (Section 4 above).

The working set method was proposed as both a means of understanding the origin of thrashing, and of countering it, and in particular we proposed to test ways in which local page replacement and scheduling policies could improve Linux's handling of low memory situations and minimise the problem of thrashing.

Thrashing is caused by a inability of processes to establish their working sets in memory: instead processes wait as the input/output system attempts to place the needed pages in memory. With too many pages required by too many processes, none can make progress and CPU efficiency collapses (Jiang & Zhang, 2005). Despite the efforts of many, thrashing is still a problem for systems with small memory.

There was, of course, the issue of how we measured system performance and judge how successful or otherwise any measure we proposed was in improving performance and limiting thrashing. Eeckhout (Eeckhout, 2010, p. 11), discussing how to measure hardware performance, states that, in assessing multiprogramming systems both "system throughput" and "average normalised turnaround time" should be measured, as there may be trade-offs that deliver faster turnaround time at the expense of lower throughput and *vice versa*. In practice, though, we concentrated exclusively on turnaround time as the most easily accessible measure, using the "real" value returned by the Linux shell command `time`[23]. Of course, had any of the patches we tested delivered significant improved turnaround times, then throughput measurements would also have been useful.

The kernel building process was chosen as the test basis because it is a common and well-understood process that can be constrained to the same task on each run and it requires minimal operator intervention. That does not mean it was perfect. Figure 2.4 suggests that the C compiler shows a higher degree of locality than the other tested programs, for instance.

Our first test of turnaround time showed that the relative memory shortage that causes thrashing can have a serious impact on performance. Figure 5.1 (generated with R), which shows the time taken to compile a Linux kernel on a virtualised system[24], illustrates this. The red line shows a tentative fit (derived using the least

[22] As stated in the project proposal.

[23] See `man 7 time` or `http://linux.die.net/man/1/time` - accessed 29 August 2011

[24] We compiled the default configuration, using `time make -j3`, for Intel x86 machines for Linux 3.0.0 on a virtualised system, using the KVM

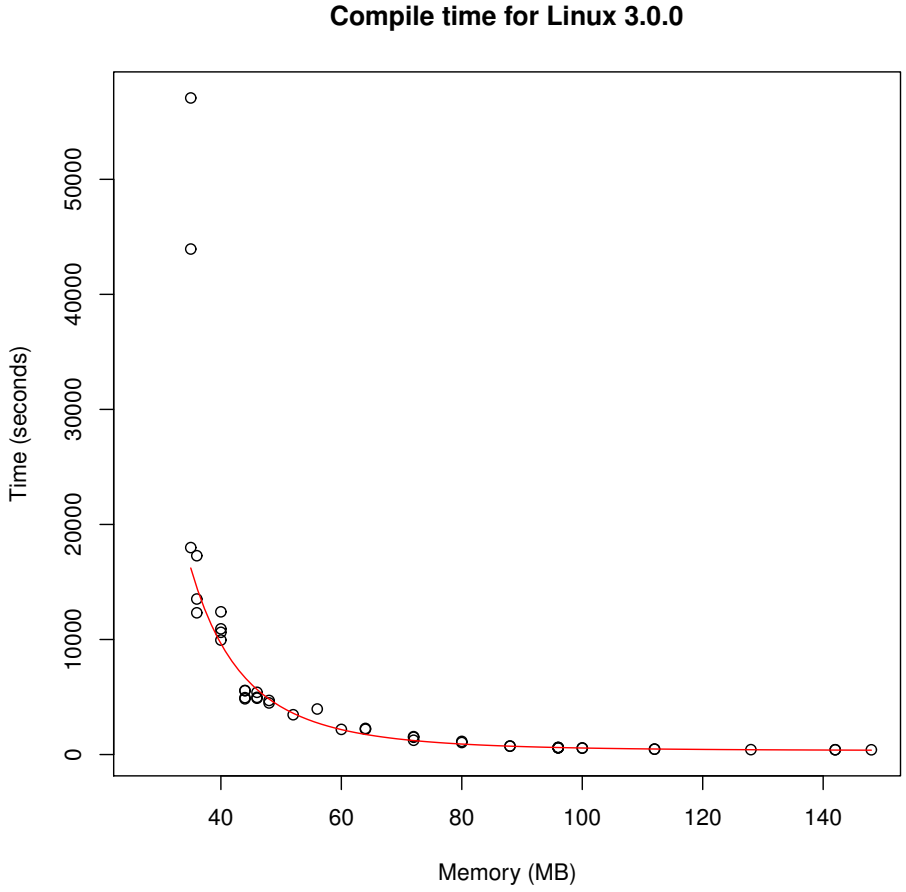

FIGURE 5.1. Time taken to compile Linux 3.0.0 kernel, x86 default configuration

squares method, but excluding the most extreme measurements at either end) of the compile time T seconds, to available memory, M megabytes, of the form:

$$T = \frac{k}{M^4} + c \tag{5.1}$$

Where k and c are constants. It is certainly plain that computing efficiency deteriorates rapidly as available memory decreases.

It will be noted that the `time` command can give variable results - as Figure 5.2, with the observations used in the production of Figure 5.1 redrawn as box plots[25], shows, running the same command with the same amount of memory on the same

(`http://www.linux-kvm.org/page/Main_Page` - accessed 10 August 2011) modules supplied with a standard Debian (`http://www.debian.org/` - accessed 10 August 2011) "Wheezy" GNU/Linux distribution, to virtualise a two CPU Debian "Lenny" system.

[25]All box plots shown in this report were produced by R. The line in a box shows the median, while the box shows the interquartile range between the 25th and 75th percentile. The "whiskers" show the adjacent values, a guide to the extremes of the range (Adler, 2009, p. 240).

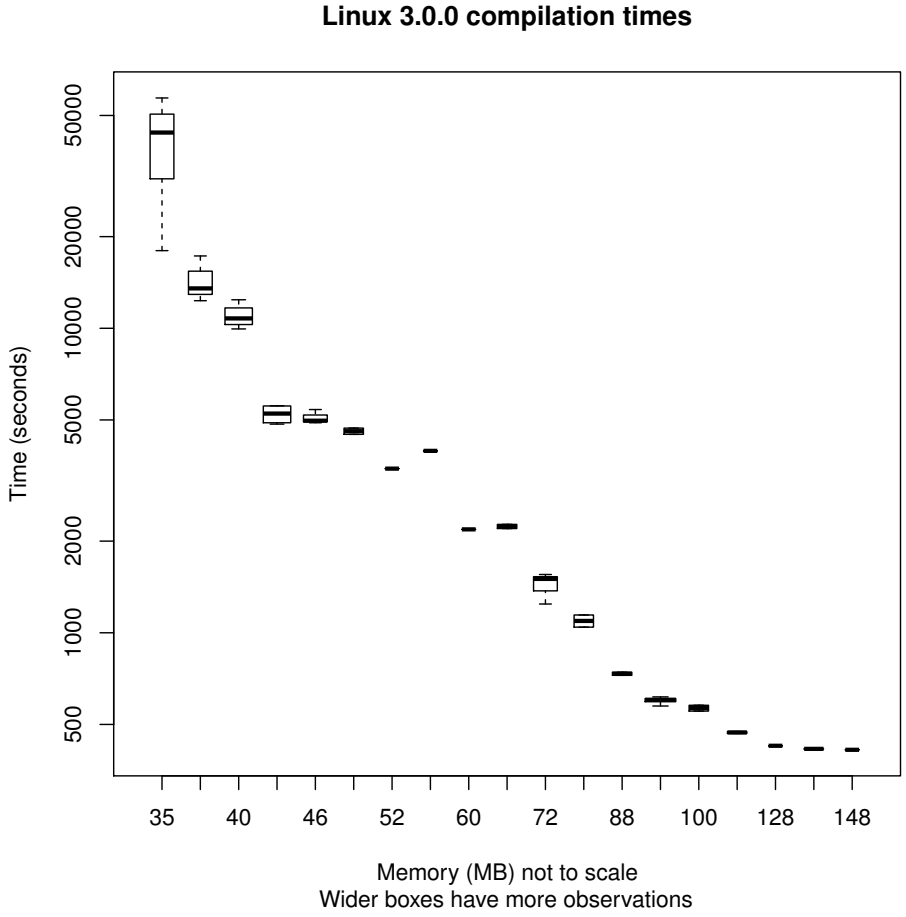

FIGURE 5.2. Boxplot showing range of values returned by `time` shell command

machine can give noticeably different results - a possible reflection of the behaviour of the other processes running (such as over-night *crond* runs.)

In seeking to attack the thrashing problem we did not believe there was a general failure in Linux's virtual memory (VM) subsystem, nor any reason to demur from the view of one of its designers that "the VM performs well in practice." (Gorman, 2004, p. xiii) However, a software tool, Memball, and its related programs, (described in Appendix C), suggested that the largest consumer of memory in a running system was often consuming many times more memory than the next biggest process, suggesting that managing the memory demand from the biggest process running on a system may help diminish the thrashing problem.

Figure 5.3 offers a typical example. Here a fragment of the red-black (semi-balanced) binary tree, ordered by memory allocation, and produced by Memball and the related Treedraw program are shown. The nodes shown are the eight largest running programs, but the largest, the Mozilla Firefox browser component, firefox-bin, is approximately four times larger than the next biggest, an instance of the Evince document viewer.

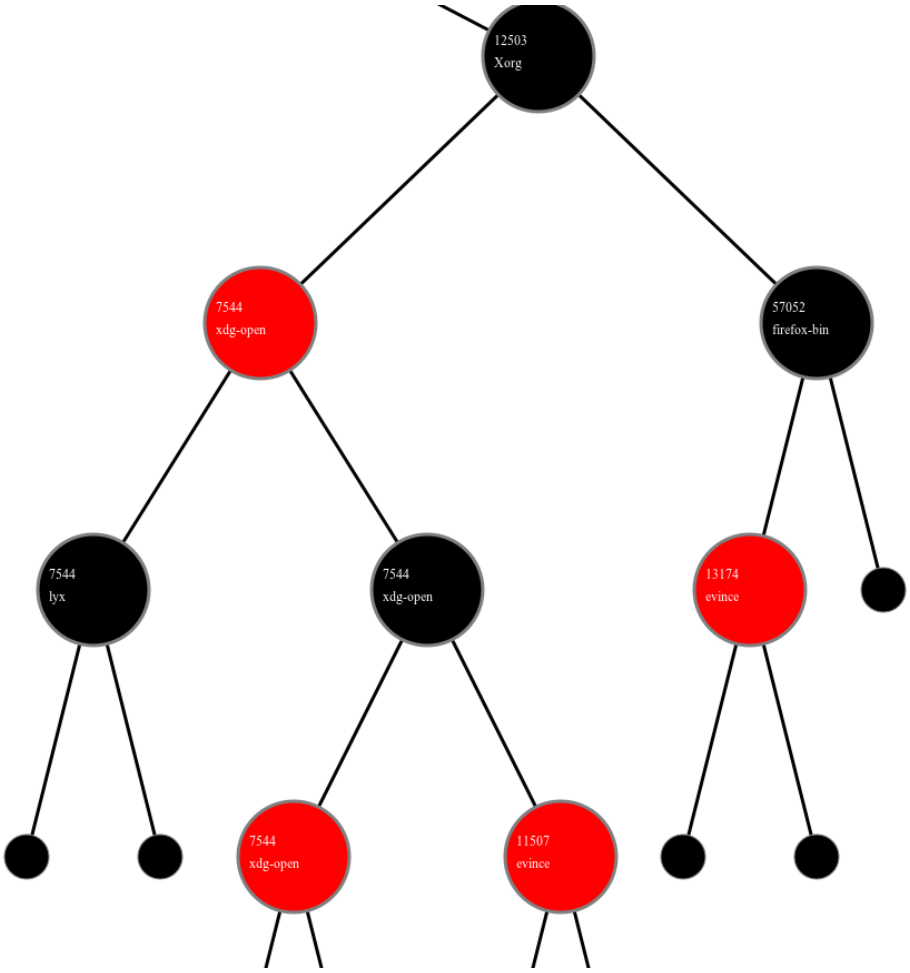

FIGURE 5.3. Fragment of red-black tree output showing largest processes running on Linux machine. Figures are allocated memory in kilobytes.

This behaviour is an indication that treating the largest memory consumer in a running system as a special case could be worthwhile, suggesting, at least to some degree, a local allocation policy. As the designers of the Linux virtual memory system state that their approach is “rather empirical in nature” (Gorman, 2004, p. 163) and the Linux kernel already contains local memory allocation methods, such as the *swap token* (Mauerer, 2008, pp.1079 - 1082), an implementation of the token-ordered LRU proposed in (Jiang & Zhang, 2005), the application of a local replacement heuristic is not alien to the Linux kernel. However, it was not proposed to re-write large parts of the kernel code, but rather to test ideas for small patches that might improve turnaround times.

6. PATCHING SCANNING CODE

6.1. **The basic operation of page scanning and reclaim.** The first series of patches we tested focused on the operations in `mm/vmscan.c`. A brief description of the basic operations carried out in this code may help when we subsequently explain our approach to patching. Linux divides available memory into *zones*, in the x86 architecture these are typically `ZONE_DMA`, the bottom 16MB of physical memory which were the only addresses accessible to DMA[26] controllers on older ISA[27] devices, `ZONE_NORMAL` for normal memory access and `ZONE_HIGHMEM` which contains those pages with physical addresses higher than 896MB which cannot be permanently mapped into the kernel's address space (Love, 2010, pp. 233 - 235). The kernel maintains a series of *watermarks* for each of these zones and will seek to 'balance' the zone (ie., free page frames) if the zones are not in balance. These watermarks are `WMARK_LOW`, `WMARK_MIN` and `WMARK_HIGH`. The code extract in listing 1 shows how the watermarks are setup.

LISTING 1. mm/page_alloc.c code to establish watermarks

```
/**
 * setup_per_zone_wmarks - called when min_free_kbytes changes
 * or when memory is hot-{added|removed}
 *
 * Ensures that the watermark[min,low,high] values for each zone are set
 * correctly with respect to min_free_kbytes.
 */
void setup_per_zone_wmarks(void)
{
        unsigned long pages_min = min_free_kbytes >> (PAGE_SHIFT - 10);
        unsigned long lowmem_pages = 0;
        struct zone *zone;
        unsigned long flags;

        /* Calculate total number of !ZONE_HIGHMEM pages */
        for_each_zone(zone) {
                if (!is_highmem(zone))
                        lowmem_pages += zone->present_pages;
        }

        for_each_zone(zone) {
                u64 tmp;

                spin_lock_irqsave(&zone->lock, flags);
                tmp = (u64)pages_min * zone->present_pages;
                do_div(tmp, lowmem_pages);
                if (is_highmem(zone)) {
                        /*
                         * __GFP_HIGH and PF_MEMALLOC allocations usually don't
                         * need highmem pages, so cap pages_min to a small
                         * value here.
                         *
                         * The WMARK_HIGH-WMARK_LOW and (WMARK_LOW-WMARK_MIN)
                         * deltas controls asynch page reclaim, and so should
                         * not be capped for highmem.
                         */
                        int min_pages;

                        min_pages = zone->present_pages / 1024;
                        if (min_pages < SWAP_CLUSTER_MAX)
                                min_pages = SWAP_CLUSTER_MAX;
                        if (min_pages > 128)
                                min_pages = 128;
                        zone->watermark[WMARK_MIN] = min_pages;
                } else {
                        /*
```

[26]Direct Memory Access - see `http://en.wikipedia.org/wiki/Direct_memory_access` - accessed 29 August 2011

[27]Industry Standard Architecture - see `http://en.wikipedia.org/wiki/Industry_Standard_Architecture` - accessed 29 August 2011

```
5180                            * If it's a lowmem zone, reserve a number of pages
                                * proportionate to the zone's size.
                                */
                               zone->watermark[WMARK_MIN] = tmp;
                        }
5185
                        zone->watermark[WMARK_LOW]  = min_wmark_pages(zone) + (tmp >> 2);
                        zone->watermark[WMARK_HIGH] = min_wmark_pages(zone) + (tmp >> 1);
                        setup_zone_migrate_reserve(zone);
                        spin_unlock_irqrestore(&zone->lock, flags);
5190            }

                /* update totalreserve_pages */
                calculate_totalreserve_pages();
}
```

Should the free pages in any zone fall to `WMARK_LOW` for that zone, then code is called to shrink (potentially all) the zones, until all the zones are above the `WMARK_HIGH` watermark for free pages. A 'priority' is recorded for each zone which indicates how many iterations were required to reach `WMARK_HIGH` (the priority is a count-down from 12, so a lower figure indicates a higher level of memory pressure). Should available memory fall to the `WMARK_MIN` watermark then pages may be freed synchronously with attempted allocations[28] (cf. Gorman, 2004, pp. 18 - 22).

6.2. **Flushing dirty pages.** The first patch of `mm/vmscan.c` was focused on ensuring that 'dirty' pages were more quickly written to disk, so making those page frames available for replacement earlier and possibly easing pressure on memory. It was very quickly obvious that this was unlikely to result in any noticeable improvement in performance, but it also serves to illustrate some of the approaches we took throughout and so we will describe it here.

The patch[29], as applied to a kernel in the Linux "mainline" is shown below in unified[30] format:

```
01: --- mm_vmscan.c       2011-08-29 21:54:41.000000000 +0100
02: +++ mm_vmscan.c_pt     2011-08-30 19:59:44.000000000 +0100
03: @@ -2402,6 +2402,70 @@
04:         else
05:                 return !all_zones_ok;
06:  }
07: +/*
08: + * local_alloc_reduce: accelerate releasing pages
09: + */
10: +void force_writeback(struct address_space *biggest,
11: +       struct address_space *second)
12: +{
13: +       int x, y;
14: +       struct writeback_control wbc = {
15: +               .nr_to_write = 0x100,
16: +               .sync_mode = WB_SYNC_ALL,
```

[28] See `/Documentation/vm/balance` in the kernel distribution

[29] This is the point reached by commit `4c345f7bd74cfdd5bdabd9633de7bc214b648492` in the author's git repository, browsable at `http://newgolddream.dyndns.info/cgi-bin/gitweb.cgi` - accessed 3 September 2011

[30] See `man 1 diff` or `http://linux.die.net/man/1/diff` - accessed 1 September 2011

```
17: +               .range_start = 0,
18: +               .range_end = LLONG_MAX,
19: +       };
20: +       if (biggest && biggest->a_ops && biggest->a_ops->writepage)
21: +               x = write_cache_pages(biggest, &wbc, biggest->a_ops->writepage,
22: +                       biggest);
23: +       if (second && second->a_ops && second->a_ops->writepage)
24: +               y = write_cache_pages(second, &wbc, second->a_ops->writepage,
25: +                       second);
26: +}
27: +
28: +/*
29: + * report_scanning_zone: report on scanned zones
30: + */
31: +void report_scanning_zone(struct zone *zone, int end_zone)
32: +{
33: +       struct task_struct *curprocess;
34: +       struct mm_struct *proc_mm;
35: +       struct vm_area_struct *proc_vma;
36: +       struct address_space *addrsp, *bigaddr = NULL, *secondaddr = NULL;
37: +       int largest, secondlargest;
38: +
39: +       largest = secondlargest = 0;
40: +
41: +       /* find biggest user of memory */
42: +       /* begin with init */
43: +       curprocess = &init_task;
44: +       curprocess = next_task(curprocess);
45: +       while (curprocess != &init_task)
46: +       {
47: +               proc_mm = curprocess->mm;
48: +               if (!proc_mm)
49: +                       goto advance;
50: +               proc_vma = proc_mm->mmap;
51: +               if (!proc_vma)
52: +                       goto advance;
53: +               if (proc_vma->vm_file) {
54: +                       addrsp = proc_vma->vm_file->f_mapping;
55: +                       if (addrsp) {
56: +                               if (addrsp->nrpages > largest) {
57: +                                       secondlargest = largest;
58: +                                       largest = addrsp->nrpages;
59: +                                       secondaddr = bigaddr;
60: +                                       bigaddr = addrsp;
61: +                               }
62: +                       }
63: +               }
64: +advance:
```

```
65: +            curprocess = next_task(curprocess);
66: +        }
67: +    if (bigaddr && secondaddr)
68: +            force_writeback(bigaddr, secondaddr);
69: +    return;
70: +}
71:
72: /*
73:  * For kswapd, balance_pgdat() will work across all this node's zones until
74: @@ -2518,7 +2582,10 @@
75:                  struct zone *zone = pgdat->node_zones + i;
76:                  int nr_slab;
77:                  unsigned long balance_gap;
78: -
79: +
80: +                if (!zone_watermark_ok_safe(zone, order,
81: +                        low_wmark_pages(zone), 0, 0))
82: +                        report_scanning_zone(zone, end_zone);
83:                  if (!populated_zone(zone))
84:                          continue;
85:
86:
```

We begin (the patch's logical flow is shown in a simplified form in Figure 6.1) at line 79 in the patch, in `balance_pgdat` which is regularly called by the *`kswapd`* kernel thread to check that the zones are in balance (Bovet & Cesati, 2005, p. 708). If a zone was at or below the `LOW` watermark - then a function added by our patch, `report_scanning_zone` was called. This relied on the kernel's maintenance of a circular linked list of processes to iterate through all the processes[31], checking to see if the process is backed by a file and if it is, looking for the process with the largest number of pages recorded by its `struct address_space`, a structure which records information about files and their backing devices (a second big process, which may or may not be the second biggest, is also sought). If, having scanned through all the processes, two were found that are backed by a file, then another new function, `force_writeback` was called which attempted to flush to the backing store up to 256 pages which are mapped through the `struct address_space` and which are marked as dirty (ie., have been altered).

The `struct address_space` is shown in listing 2 below:

LISTING 2. struct address_space from /include/linux/fs.h

```
struct address_space {
        struct inode            *host;          /* owner: inode, block_device */
        struct radix_tree_root  page_tree;      /* radix tree of all pages */
        spinlock_t              tree_lock;      /* and lock protecting it */
        unsigned int            i_mmap_writable;/* count VM_SHARED mappings */
        struct prio_tree_root   i_mmap;         /* tree of private and shared mappings */
        struct list_head        i_mmap_nonlinear;/*list VM_NONLINEAR mappings */
        struct mutex            i_mmap_mutex;   /* protect tree, count, list */
        /* Protected by tree_lock together with the radix tree */
        unsigned long           nrpages;        /* number of total pages */
        pgoff_t                 writeback_index;/* writeback starts here */
```

[31]The process ID, PID, of the `init` process, which is always running on a standard GNU/Linux system, is always 0, so providing a convenient point at which to begin the iteration.

```
        const struct address_space_operations *a_ops;   /* methods */
        unsigned long           flags;          /* error bits/gfp mask */
        struct backing_dev_info *backing_dev_info; /* device readahead, etc */
        spinlock_t              private_lock;   /* for use by the address_space */
        struct list_head        private_list;   /* ditto */
        struct address_space    *assoc_mapping; /* ditto */
} __attribute__((aligned(sizeof(long))));
        /*
         * On most architectures that alignment is already the case; but
         * must be enforced here for CRIS, to let the least significant bit
         * of struct page's "mapping" pointer be used for PAGE_MAPPING_ANON.
         */
```

As can be seen, a `struct address_space` holds a pointer to the underlying `inode` of the backing file, as well as a `struct radix_tree_root`, which the root of a search tree used to look up all the pages associated with this `struct address_space` (we refer further to this structure below when describing how it is used to search through the pages).

There is a flaw in this approach, in that pages that are backed in this way are unlikely to be dirty - for instance a Linux program will not alter its code (the so-called *text segment*) while running (Mauerer, 2008, Chapter 4, pp. 289 - 346). We quickly realised this and moved on, not recording any timings after using the `top`[32] shell command showed there were few dirty pages to write back.

6.3. **Increasing CLOCK pressure.** We then tried a second patch with the aim of increasing the rate at which pages were reclaimed from large processes: the results outlined in Section 2 and 3, above, show that there are disruptive phase transitions during program execution and in these periods working set sizes increase as pages may be accessed from two different phases of locality. However once the program has fully moved on to the new region of locality, the pages from the old phase are no longer needed: but while their *reuse distance* (time until their next access) may be very high or infinite, their *recency* (time since their last access) may be low. This is a well-known problem of LRU type replacement policies (Jiang *et al.* , 2005) and it is one that the 2Q policy may only make worse - for in the now finished phase of locality the unneeded page may have been frequently accessed, and so protected from fast removal by being on the active LRU list in a referenced state.

Therefore our aim with this patch was to, in effect, ensure that the 'clock hands' in the CLOCK replacement policy (Stallings, 2008, pp. 370 - 374) moved through the pages of the largest process with extra rapidity.

The patch[33] is shown below in unified format:

```
01: --- mm_vmscan.c        2011-08-29 21:54:41.000000000 +0100
02: +++ mm_vmscan.c_pt        2011-08-31 22:12:27.000000000 +0100
03: @@ -2402,6 +2402,83 @@
04:          else
05:                  return !all_zones_ok;
06:  }
07: +/*
08: + * speedhands: increase clock pressure for biggest process
09: + */
10: +#define PAGEGRAB 0x100
```

[32] See `man 1 top` or `http://linux.die.net/man/1/top` - accessed 31 August 2011

[33] As reached in commit `bf670c6984953fa230f4c89f0a09b161c9c04ae9` - accessed 3 September 2011

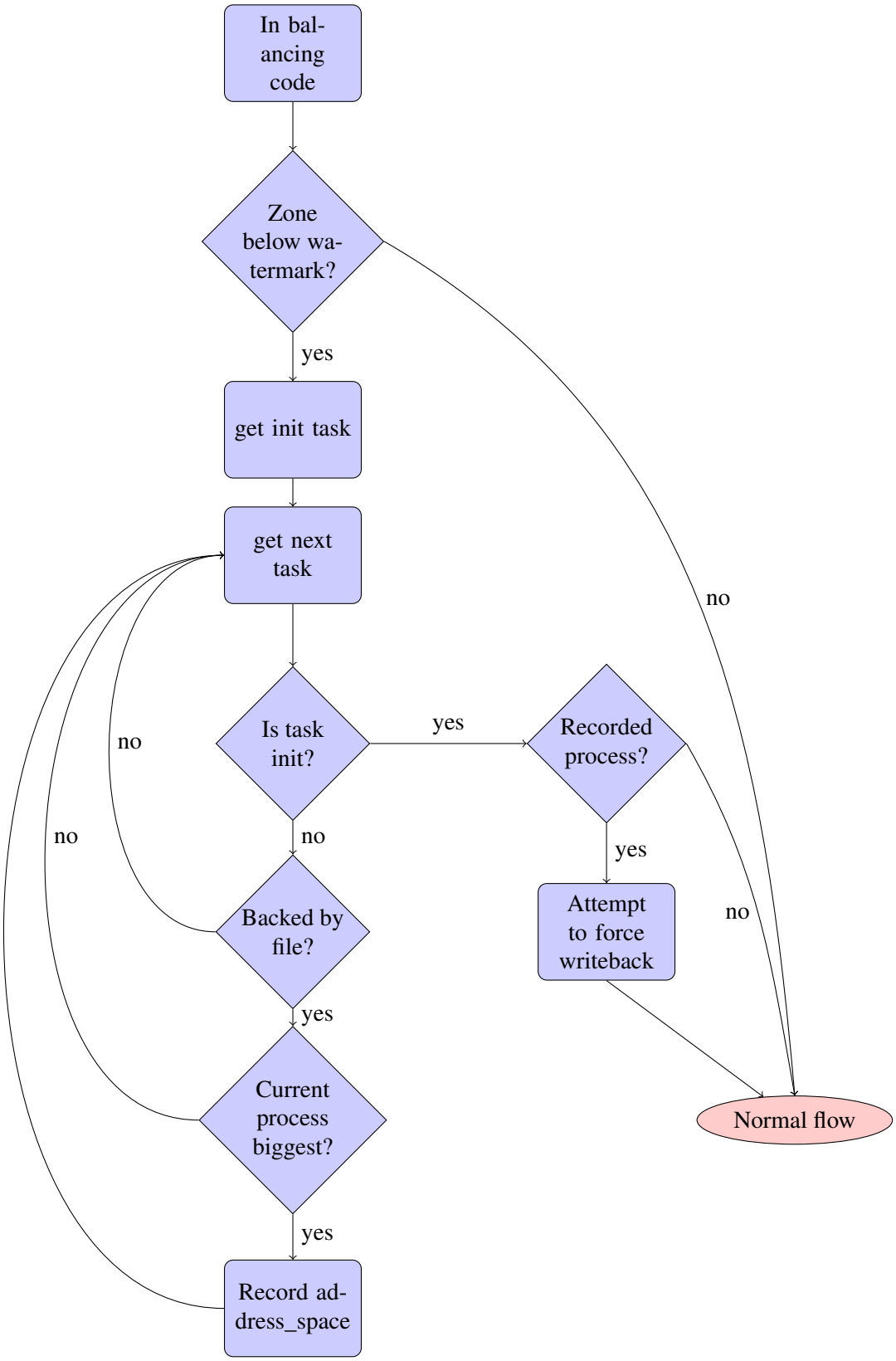

FIGURE 6.1. Flow in page reclaim patch

```
+void speedhands(struct address_space *biggest)
+{
+        int x, y, nr_pages, pos = 0;
+        struct page *biggestpages[PAGEGRAB];
+        struct writeback_control wbc = {
+                .nr_to_write = 0x100,
+                .sync_mode = WB_SYNC_ALL,
+                .range_start = 0,
+                .range_end = LLONG_MAX,
+        };
+        //writeback any dirty pages
+        if (biggest && biggest->a_ops && biggest->a_ops->writepage)
+                x = generic_writepages(biggest, &wbc);
+        if (x)
+                printk("write_cache_pages fails with %d\n", x);
+
+        //fetch pages from the radix-tree
+        do {
+                spin_lock_irq(&biggest->tree_lock);
+                nr_pages = radix_tree_gang_lookup(&biggest->page_tree,
+                        (void **)biggestpages, pos, PAGEGRAB);
+                spin_unlock_irq(&biggest->tree_lock);
+                x  = 0;
+                for (y = 0; y < nr_pages; y++)
+                {
+                        if (PageReferenced(biggestpages[y])) {
+                                ClearPageReferenced(biggestpages[y]);
+                                x++;
+                        }
+                }
+                pos += nr_pages;
+        } while(nr_pages == PAGEGRAB);
+}
+
+/*
+ * report_scanning_zone: report on scanned zones
+ */
+void report_scanning_zone(struct zone *zone, int end_zone)
+{
+        struct task_struct *curprocess;
+        struct mm_struct *proc_mm;
+        struct vm_area_struct *proc_vma;
+        struct address_space *addrsp, *bigaddr = NULL;
+        int largest = 0;
+
+        /* find biggest user of memory */
+        /* begin with init */
+        curprocess = &init_task;
```

```
+        curprocess = next_task(curprocess);
+        while (curprocess != &init_task)
+        {
+                proc_mm = curprocess->mm;
+                if (!proc_mm)
+                        goto advance;
+                proc_vma = proc_mm->mmap;
+                if (!proc_vma)
+                        goto advance;
+                if (proc_vma->vm_file) {
+                        addrsp = proc_vma->vm_file->f_mapping;
+                        if (addrsp) {
+                                if (addrsp->nrpages > largest) {
+                                        largest = addrsp->nrpages;
+                                        bigaddr = addrsp;
+                                }
+                        }
+                }
+advance:
+                curprocess = next_task(curprocess);
+         }
+        if (bigaddr)
+                speedhands(bigaddr);
+        return;
+}

 /*
  * For kswapd, balance_pgdat() will work across all this node's zones until
@@ -2518,7 +2595,10 @@
                         struct zone *zone = pgdat->node_zones + i;
                         int nr_slab;
                         unsigned long balance_gap;
-
+
+
+                        if (!zone_watermark_ok_safe(zone, order,
+                                low_wmark_pages(zone), 0, 0))
+                                report_scanning_zone(zone, end_zone);
                         if (!populated_zone(zone))
                                 continue;
```

Here the test inside `balance_pgdat`, and the code in `report_scanning_zone` are essentially as before, though only the largest process is sought and a different function, `speedhands` is called. This function, as with the previous patch, does look for any dirty pages to write out (in theory writing up to 256 dirty pages), before

trying clear the `PG_referenced` flag[34] on any page in the `struct address_space` of the process.

Some of the key points of this patch are:

- It uses the `page_tree` to search through pages that are held in the `struct address_space`. The `page_tree` is a root of a radix-tree, a binary search tree which can associate leaf nodes with 'tags' and is thus used by the kernel to mark dirty pages as well as to search to see if a page exists in a given address space (Love, 2010, p. 330). We use the `radix_tree_gang_lookup`[35] function in an attempt to pull out pages in the address space in groups of 256 at a time.
- The `page_tree` is protected by a lock, which our patch insisted on holding before moving on (in a spin lock (Love, 2010, pp. 183 - 186)). In fact we later realised that using such an expensive form of locking was not necessary - see below. Holding a lock on the radix-tree of the biggest process in a pre-emptive kernel almost certainly imposes a performance cost - as other parts of the kernel would be locked out.
- The patch focuses exclusively on file-backed pages. If these are clean then they can be dropped from the LRU lists with the minimum of delay, as no write-back is required. If we had, for instance, also looked to increase clock pressure on anonymous pages then we may have forced unnecessary writes to swap (swap being the effective file backup for anonymous pages). If we pushed a page into swap and it was shortly thereafter paged back in, then the kernel would have suffered the delay of two transport times (ie the time it takes for the page to be read from storage), one of which would be a write. If we pushed a clean file-backed page to disk and it was read back in then there is no time lost for the write (obviously it would have been better not to push this page out of the LRU lists at all, but that is the risk we run here.) But focusing on file-backed pages also means ignoring many pages used by the biggest process.
- The patch is adaptive in that `speedhands` will only be called when memory pressure is such that free memory in at least one zone has fallen to or below the `LOW` watermark.
- The patch only minimally interferes with the mechanics of the Linux page replacement algorithm, doing little more than 'hint' to the kernel that pages from the biggest program should be pushed from the active to inactive list.
- When testing the patch we confirmed that pages were being 'marked down' - we estimated the modal number pages being marked in this way was just 1, though frequently it was as high as 56.

The results for turnaround time or the standard `time make -j3` for the Linux kernel, on a 96MB virtual machine, are shown in Figure 6.2: the median time was greater than for the unpatched kernel, but results were not so bad as to suggest that the patch did not have potential. Accordingly, attempts were made to improve it.

6.3.1. *Changing the test point to `MIN`.* The first thing we tried was to make the watermark test harsher, only calling `report_scanning_zone` when a zone fell to

[34] See `/include/linux/page-flags.h` in the Linux kernel distribution

[35] See `http://lwn.net/Articles/9524/` - accessed 1 September 2011

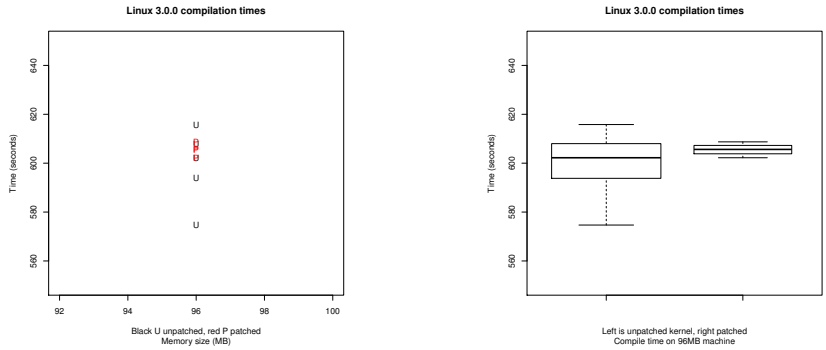

FIGURE 6.2. Unpatched and patched kernel performance compared: on left, patched (P) times in red, unpatched in black, on right box plot, unpatched kernel to left, patched to right

or below `MIN`, (in addition this patch ensured that the testing line would not be reached if the zone was empty) as seen in the unified format patch[36] below:

```
01 : --- a/mm/vmscan.c
02 : +++ b/mm/vmscan.c
03 : @@ -2596,12 +2596,13 @@ loop_again:
04 :                 int nr_slab;
05 :                 unsigned long balance_gap;
06 :
07 : -               if (!zone_watermark_ok_safe(zone, order,
08 : -                       low_wmark_pages(zone), 0, 0))
09 : -                       report_scanning_zone(zone, end_zone);
10 :                 if (!populated_zone(zone))
11 :                         continue;
12 :
13 : +               if (!zone_watermark_ok_safe(zone, order,
14 : +                       min_wmark_pages(zone), 0, 0))
15 : +                       report_scanning_zone(zone, end_zone);
16 : +
17 :                 if (zone->all_unreclaimable && priority != DEF_PRIORITY)
18 :                         continue;
19 :
```

This failed to show any improvement, as demonstrated in Figure 6.3.

6.3.2. *Removing writeback.* Next we removed the attempts to speed page write-back , increased the attempted grab from the radix-tree to 512 pages and reverted to calling `report_scanning_zone` when the free pages level reached the `LOW` level:

```
01 : --- a/mm/vmscan.c
02 : +++ b/mm/vmscan.c
03 : @@ -2405,22 +2405,11 @@ static bool sleeping_prematurely(pg_data_t *pgdat, int order, long remaining,
```

[36] As reached in commit `bc7d3ae359e191912e0b36d3d6d979ba4151c928` - accessed 3 September 2011

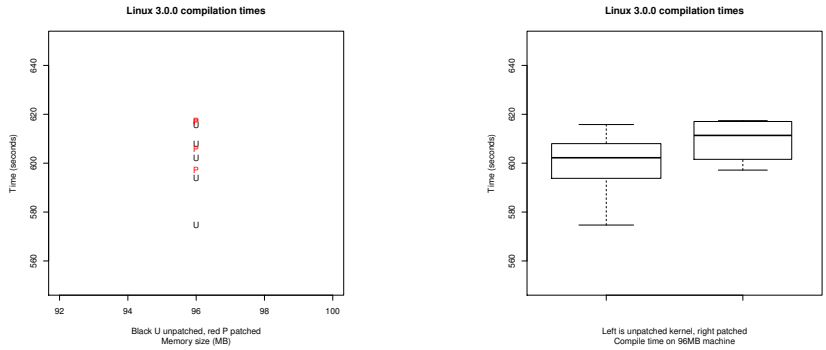

FIGURE 6.3. Patch performance when only calling `report_scanning_zone` when the watermark falls to `MIN`: on left patched (P) times in red, unpatched in black, on right box plot, unpatched kernel on left

```
04:  /*
05:   * speedhands: increase clock pressure for biggest process
06:   */
07: -#define PAGEGRAB 0x100
08: +#define PAGEGRAB 0x200
09:  void speedhands(struct address_space *biggest)
10:  {
11:          int x, y, nr_pages, pos = 0;
12:          struct page *biggestpages[PAGEGRAB];
13: -        struct writeback_control wbc = {
14: -                .nr_to_write = 0x100,
15: -                .sync_mode = WB_SYNC_ALL,
16: -                .range_start = 0,
17: -                .range_end = LLONG_MAX,
18: -        };
19: -        //writeback any dirty pages
20: -        if (biggest && biggest->a_ops && biggest->a_ops->writepage)
21: -                x = generic_writepages(biggest, &wbc);
22: -        if (x)
23: -                printk("write_cache_pages fails with %d\n", x);
24:
25:          //fetch pages from the radix-tree
26:          do {
27:  @@ -2600,7 +2589,7 @@ loop_again:
28:                                  continue;
29:
30:                          if (!zone_watermark_ok_safe(zone, order,
31: -                                min_wmark_pages(zone), 0, 0))
32: +                                low_wmark_pages(zone), 0, 0))
33:                                  report_scanning_zone(zone, end_zone);
34:
```

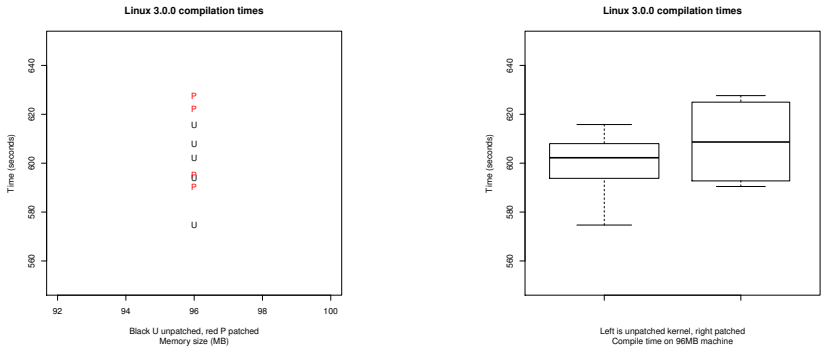

FIGURE 6.4. Patch performance with writeback removed: on left patched (P) times in red, unpatched in black, on right box plot, unpatched kernel is left plot.

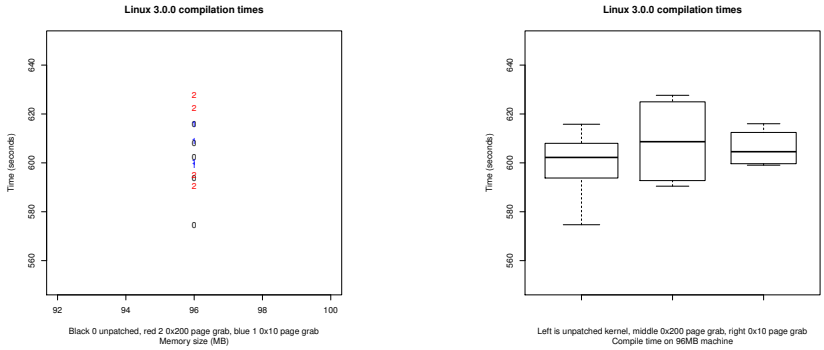

FIGURE 6.5. Patch performance for varied sizes of `PAGEGRAB`: On left, unpatched kernel (black 0), 0x200 `PAGEGRAB` (red 2) and 0x10 `PAGEGRAB` (blue 1). On right, unpatched kernel is left box plot, 0x200 `PAGEGRAB` in middle, 0x10 `PAGEGRAB` on right.

```
35 :            if (zone->all_unreclaimable && priority != DEF_PRIORITY)
36 :
```

As shown in Figure 6.4, this patch[37] too failed to deliver improved performance.

6.3.3. *Reducing the pull from the radix tree.* Holding a spin lock on the root of the radix-tree for as long as it might take to recover 512 pages was thought likely to impose a high penalty, especially as a thread of execution in the kernel contending for a spin lock will not sleep. To reduce the time that a process might 'spin' on a lock, `PAGEGRAB` was reduced to 16.[38]

Test results, as shown in Figure 6.5 (which also shows the performance of the kernel with the previous 0x200 value for `PAGEGRAB`), reveal that the patch did not appear to deliver better performance than the unpatched kernel, but again showed comparable performance.

[37]As reached in commit `7d00ad13ea992f28bf63e79c27bbe7eada5f2174` - accessed 3 September 2011

[38]As reached with commit `bb824acd192e37163bc1e74fcce55930a1bb9f75` - accessed 3 September 2011

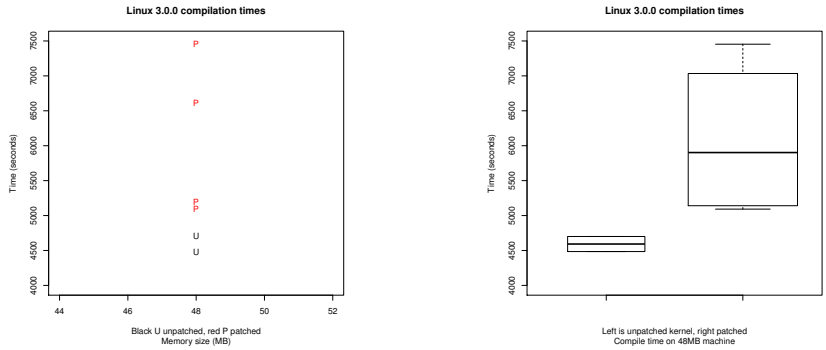

FIGURE 6.6. Testing patched kernel with 48 MB of memory and `PAGEGRAB` of 0x10: On left patched kernel marked with red P, unpatched kernel black U. On right, unpatched kernel shown in left box plot.

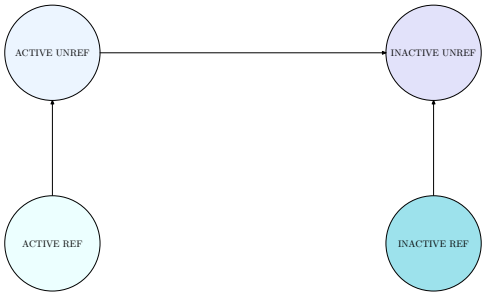

FIGURE 6.7. Possible effects of patch described in Section 6.4

A version[39] of this patch (though with a stray and unnecessary additional integer counting variable added to the loop where `ClearPageReferenced` is called) was then tested on a virtual machine with 48MB of memory. If a patch was working as hoped, then reducing memory ought to deliver better performance relative to the unpatched kernel: as `report_scanning_zone` should be called more often. In fact, the test results, as shown in Figure 6.6 were disappointing (though there are only two test values for the unpatched kernel). They clearly suggest that the patch was contributing to a deterioration in performance. A different approach was needed.

6.4. **Pushing pages out of the active list.** After the failure of patches describe in 6.3 above, we decided to retain the idea of increased CLOCK pressure on the biggest program, but to also interact more directly with the 2Q system of LRU lists by, as well as zeroing the `PG_referenced` flag where it was set, to remove to the inactive list any page found on the active list without `PG_referenced` set. The possible effects are represented in Figure 6.7 (cf. Figure 1.1).

The patch[40] is shown below:

[39]As reached by commit `0d76bc56f67accc46ca38d4c3965472fda180d59` - accessed 3 September

[40]As reached by commit `4d0c1e248ba8be5b17652750b691a7923dbbec21` in the author's git repository - accessed 3 September 2011

```
001 : --- mm_vmscan.c       2011-08-29 21:54:41.000000000 +0100
002 : +++ mm_vmscan.c_pt      2011-09-03 21:55:03.000000000 +0100
003 : @@ -2402,6 +2402,90 @@
004 :          else
005 :                  return !all_zones_ok;
006 :  }
007 : +/*
008 : + * speedhands: increase clock pressure for biggest process
009 : + */
010 : +#define PAGEGRAB 0x10
011 : +void speedhands(struct address_space *biggest)
012 : +{
013 : +       int y, nr_pages, pos = 0;
014 : +       struct page *biggestpages[PAGEGRAB];
015 : +
016 : +       //fetch pages from the radix-tree
017 : +       do {
018 : +               spin_lock_irq(&biggest->tree_lock);
019 : +               nr_pages = radix_tree_gang_lookup(&biggest->page_tree,
020 : +                       (void **)biggestpages, pos, PAGEGRAB);
021 : +               spin_unlock_irq(&biggest->tree_lock);
022 : +               for (y = 0; y < nr_pages; y++)
023 : +               {
024 : +                       if (!PageLRU(biggestpages[y]))
025 : +                               continue;
026 : +                       else if (PageUnevictable(biggestpages[y]))
027 : +                               continue;
028 : +                       else if (PageWriteback(biggestpages[y]))
029 : +                               continue;
030 : +                       else if (PageDirty(biggestpages[y]))
031 : +                               continue;
032 : +                       else if (PageReferenced(biggestpages[y])) {
033 : +                               ClearPageReferenced(biggestpages[y]);
034 : +                               continue;
035 : +                       }
036 : +                       else if (PageActive(biggestpages[y])){
037 : +                               struct zone *zone;
038 : +                               int lru;
039 : +                               zone = page_zone(biggestpages[y]);
040 : +                               lru = page_lru_base_type(biggestpages[y]);
041 : +                               del_page_from_lru_list(zone,
042 : +                                       biggestpages[y], lru + LRU_ACTIVE);
043 : +                               ClearPageActive(biggestpages[y]);
044 : +                               add_page_to_lru_list(zone,
045 : +                                       biggestpages[y], lru);
046 : +                       }
047 : +               }
048 : +               pos += nr_pages;
```

```
049: +       } while(nr_pages == PAGEGRAB);
050: +}
051: +
052: +/*
053: + * report_scanning_zone: report on scanned zones
054: + */
055: +void report_scanning_zone(struct zone *zone, int end_zone)
056: +{
057: +       struct task_struct *curprocess;
058: +       struct mm_struct *proc_mm;
059: +       struct vm_area_struct *proc_vma;
060: +       struct address_space *addrsp, *bigaddr = NULL;
061: +       int largest = 0;
062: +
063: +       /* find biggest user of memory */
064: +       /* begin with init */
065: +       curprocess = &init_task;
066: +       curprocess = next_task(curprocess);
067: +       while (curprocess != &init_task)
068: +       {
069: +               proc_mm = curprocess->mm;
070: +               if (!proc_mm)
071: +                       goto advance;
072: +               proc_vma = proc_mm->mmap;
073: +               if (!proc_vma)
074: +                       goto advance;
075: +               if (proc_vma->vm_file) {
076: +                       addrsp = proc_vma->vm_file->f_mapping;
077: +                       if (addrsp) {
078: +                               if (addrsp->nrpages > largest) {
079: +                                       largest = addrsp->nrpages;
080: +                                       bigaddr = addrsp;
081: +                               }
082: +                       }
083: +               }
084: +advance:
085: +               curprocess = next_task(curprocess);
086: +       }
087: +       if (bigaddr)
088: +               speedhands(bigaddr);
089: +       return;
090: +}
091:
092: /*
093:  * For kswapd, balance_pgdat() will work across all this node's zones until
094: @@ -2518,10 +2602,14 @@
095:                         struct zone *zone = pgdat->node_zones + i;
096:                         int nr_slab;
```

```
097:                unsigned long balance_gap;
098: -
099: +
100:                if (!populated_zone(zone))
101:                      continue;
102:
103: +               if (!zone_watermark_ok_safe(zone, order,
104: +                     low_wmark_pages(zone), 0, 0))
105: +                     report_scanning_zone(zone, end_zone);
106: +
107:                if (zone->all_unreclaimable && priority != DEF_PRIORITY)
108:                      continue;
109:
110:
```

In this patch, lines 51 and onwards are familiar as the essentially unchanged `report_scanning_zone` and then the watermark check, but `speedhands` from line 11 behaves in a different way from before, though lines 16 - 21 are the familiar call to `radix_tree_gang_lookup`. From line 24 onwards the code first checks that the page returned is from an LRU list, then if it is on the list of unevictable pages, then if it is being written back to the backing store or if it is dirty, before checking if it is referenced (ie., has `PG_referenced` set). If the page is referenced then `PG_referenced` is cleared and the loop continued: this is the action represented by the arrow from `ACTIVE REF` to `ACTIVE UNREF` and from `INACTIVE REF` to `INACTIVE UNREF` in Figure 6.7.

If, however, the page does not have `PG_referenced` set, then, if it is a member of the `ACTIVE` LRU list, the code from lines 36 to 46 first removes it from the LRU list, clears the `PG_active` bit for the page, and then inserts it in the `INACTIVE` list for that memory zone.

We tested this patch on both a virtual machine with 96MB of memory and 48MB of memory. The results are presented in Figure 6.8: it can be seen that the performance of the patched kernel when running under 96MB of memory was comparable to the unpatched kernel, but at 48MB the results were very disappointing, suggesting that the mechanisms in the patch were seriously degrading performance when memory pressure was high.

Moving page frames between the LRU lists requires whole memory zones to be locked, which is likely to impose a high cost when memory levels are low and the zone locks consequently more contended, a point we discuss further below in Section 7.4.

In response we made a number of changes to the patch that were designed to speed up the testing of the page status. This patch[41] (presented here as the changes to the previous patch) is below:

```
01: --- mm_vmscan.c_pt       2011-09-03 21:55:03.000000000 +0100
02: +++ mm_vmscan.c_pt_a     2011-09-04 11:51:23.000000000 +0100
03: @@ -2405,7 +2405,7 @@
04: /*
```

[41] As reached by commit `9fe689f36d589cad33f1d9f5d9ad60056f8996a2` - accessed 4 September 2011

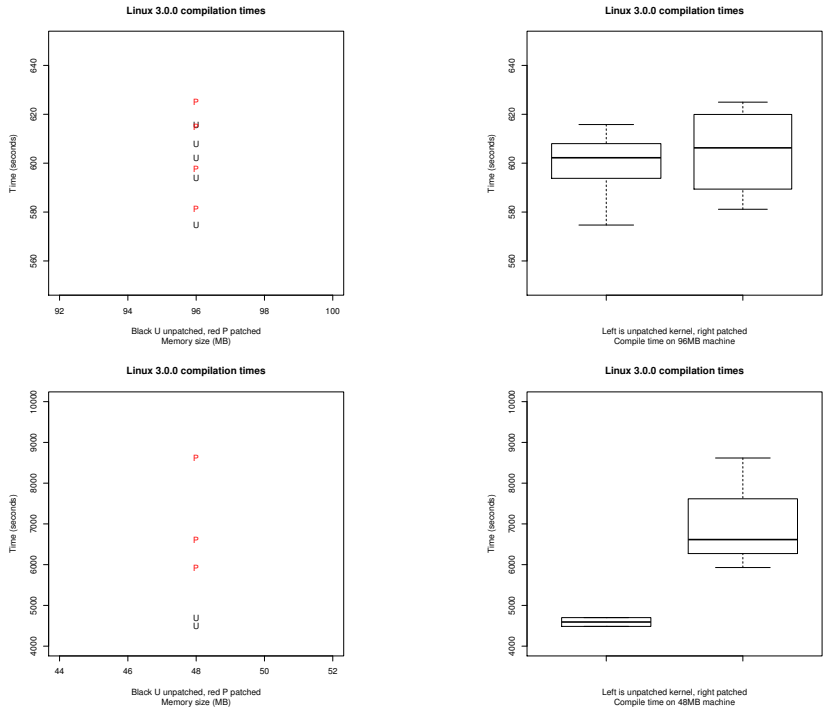

FIGURE 6.8. Test results from 96MB (top) and 48MB virtual machine with patch that pushes pages from `ACTIVE` LRU list, patched times marked on left with red P, unpatched with black U. In box plots, unpatched kernels on left.

```
05 :  * speedhands: increase clock pressure for biggest process
06 :  */
07 : -#define PAGEGRAB 0x10
08 : +#define PAGEGRAB 0x100
09 :  void speedhands(struct address_space *biggest)
10 :  {
11 :          int y, nr_pages, pos = 0;
12 :  @@ -2423,24 +2423,16 @@
13 :                          continue;
14 :                  else if (PageUnevictable(biggestpages[y]))
15 :                          continue;
16 : -                else if (PageWriteback(biggestpages[y]))
17 : -                        continue;
18 : -                else if (PageDirty(biggestpages[y]))
19 : -                        continue;
20 :                  else if (PageReferenced(biggestpages[y])) {
21 :                          ClearPageReferenced(biggestpages[y]);
22 :                          continue;
23 :                  }
24 :                  else if (PageActive(biggestpages[y])){
25 : -                        struct zone *zone;
26 :                          int lru;
```

```
27 : -                      zone = page_zone(biggestpages[y]);
28 : -                      lru = page_lru_base_type(biggestpages[y]);
29 : -                      del_page_from_lru_list(zone,
30 : -                              biggestpages[y], lru + LRU_ACTIVE);
31 : -                      ClearPageActive(biggestpages[y]);
32 : -                      add_page_to_lru_list(zone,
33 : -                              biggestpages[y], lru);
34 : +                      if (isolate_lru_page(biggestpages[y])) {
35 : +                              lru = page_lru_base_type(biggestpages[y]);
36 : +                              lru_cache_add_lru(biggestpages[y], lru);
37 : +                      }
38 :                   }
39 :               }
40 :               pos += nr_pages;
41 : @@ -2467,17 +2459,17 @@
42 :               proc_mm = curprocess->mm;
43 :               if (!proc_mm)
44 :                       goto advance;
45 : +             if (proc_mm->total_vm > largest)
46 : +                     largest = proc_mm->total_vm;
47 : +             else
48 : +                     goto advance;
49 :               proc_vma = proc_mm->mmap;
50 :               if (!proc_vma)
51 :                       goto advance;
52 :               if (proc_vma->vm_file) {
53 :                       addrsp = proc_vma->vm_file->f_mapping;
54 : -                     if (addrsp) {
55 : -                             if (addrsp->nrpages > largest) {
56 : -                                     largest = addrsp->nrpages;
57 : -                                     bigaddr = addrsp;
58 : -                             }
59 : -                     }
60 : +                     if (addrsp)
61 : +                             bigaddr = addrsp;
62 :               }
63 :  advance:
64 :               curprocess = next_task(curprocess);
65 :
```

The principal changes are:

- Lines 42 - 62 use a different test for the largest process than the previous approach of looking for the `struct address_space` with the largest value of `nr_pages`. Now the overall size of the process's virtual memory space is tested, more quickly eliminating smaller processes from the testing but also making it more likely the patch will operate on the biggest consumer of memory.

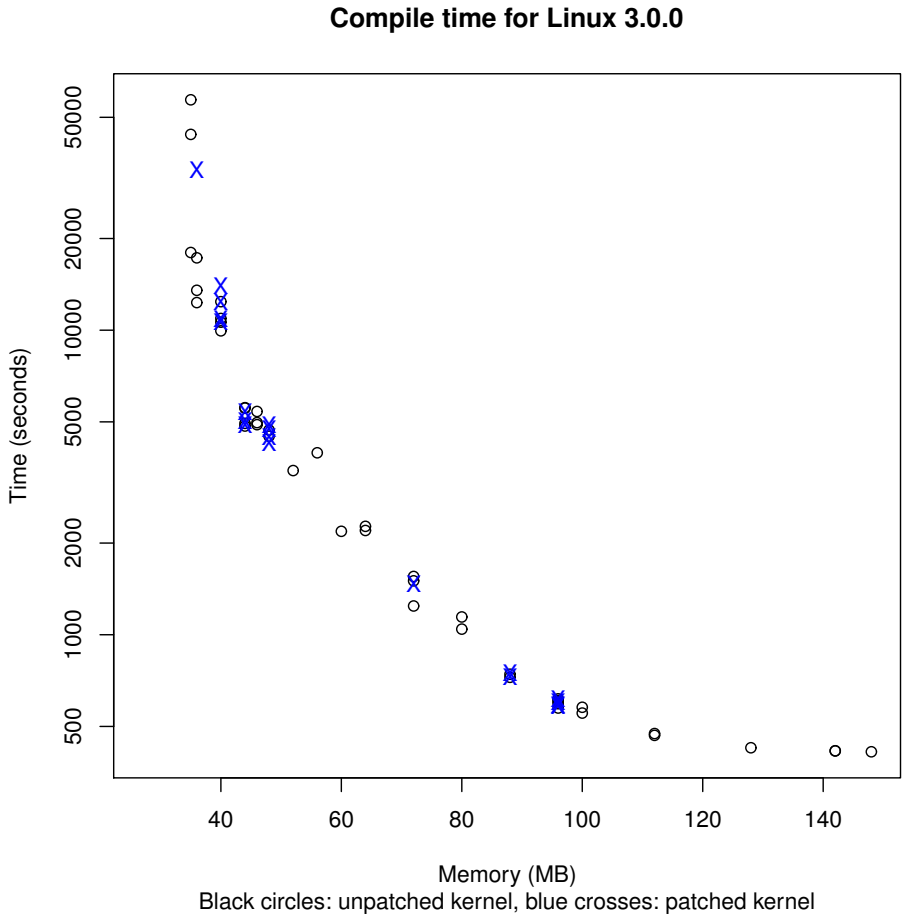

FIGURE 6.9. Patch performance, with writeback eliminated, different amounts of available memory, patched times marked with blue cross, unpatched times with black circles

- Lines 16 -19 eliminate the test for whether a page is dirty or being written back as this status was judged to be independent of what LRU list a page was on.
- Lines 24 - 37 use kernel helper functions when removing a page from the `ACTIVE` LRU list.

In addition the patch returns `PAGEGRAB`, used when searching through the radix-tree, to 256.

We tested this patch using several different amounts of available memory. As shown in Figure 6.9, the patch broadly performed as well as the unpatched kernel at higher memory levels, but there was no sign of any performance improvement. At lower levels of memory the performance was noticeably worse than the unpatched kernel.

6.5. **Using the read-copy-update methods.** As remarked above, using a spin lock to serialise use of the radix tree is potentially a costly operation, and there is an alternative - *read-copy update (RCU)* - which we began to use at this point.

RCU is a form of mutual exclusion which allows readers to have lock-free access to data marked as protected by a *critical section* established by the RCU API. Writers are required to maintain copies of the objects readers are accessing, so writes may be expensive - but as we are not writing to the radix-tree that problem is not a direct concern (McKenney *et al.* , 2001)(McKenney & Walpole, 2008)[42]. It should be noted, though, that, especially in low memory situations, this requirement on writers could slow the global replacement mechanisms.

We adopted RCU in a new patch[43] - shown below as against the unpatched kernel:

```
001: --- mm_vmscan.c       2011-08-29 21:54:41.000000000 +0100
002: +++ mm_vmscan.c_pt       2011-09-04 15:38:23.000000000 +0100
003: @@ -2402,6 +2402,77 @@
004:         else
005:                 return !all_zones_ok;
006:  }
007: +/*
008: + * speedhands: increase clock pressure for biggest process
009: + */
010: +#define PAGEGRAB 0x100
011: +void speedhands(struct address_space *biggest)
012: +{
013: +       int y, nr_pages, pos = 0;
014: +       struct page *biggestpages[PAGEGRAB];
015: +
016: +       //fetch pages from the radix-tree
017: +       do {
018: +               rcu_read_lock();
019: +               nr_pages = radix_tree_gang_lookup(&biggest->page_tree,
020: +                       (void **)biggestpages, pos, PAGEGRAB);
021: +               rcu_read_unlock();
022: +               for (y = 0; y < nr_pages; y++)
023: +               {
024: +                       if (PageReferenced(biggestpages[y])) {
025: +                               if (PageLRU(biggestpages[y]) &&
026: +                                       !PageUnevictable(biggestpages[y])) {
027: +                                       ClearPageReferenced(biggestpages[y]);
028: +                                       continue;
029: +                               }
030: +                       }
031: +                       else if (PageActive(biggestpages[y])){
032: +                               int lru;
033: +                               if (PageLRU(biggestpages[y]) &&
034: +                                       !PageUnevictable(biggestpages[y]) &&
```

[42] See also `http://en.wikipedia.org/wiki/Read-copy-update` - accessed 4 September 2011

[43] As reached by commit `59159338a6d95cdb60d9456c5876dc37fbc2d489` in the author's git repository - accessed 4 September 2011

```
035: +                                  isolate_lru_page(biggestpages[y])) {
036: +                                  lru = page_lru_base_type(biggestpages[y]);
037: +                                  lru_cache_add_lru(biggestpages[y], lru);
038: +                          }
039: +                  }
040: +          }
041: +          pos += nr_pages;
042: +      } while(nr_pages == PAGEGRAB);
043: +}
044: +
045: +/*
046: + * report_scanning_zone: report on scanned zones
047: + */
048: +void report_scanning_zone(struct zone *zone, int end_zone)
049: +{
050: +      struct task_struct *curprocess;
051: +      struct mm_struct *proc_mm, *largest_mm;
052: +      struct vm_area_struct *proc_vma;
053: +      int largest = 0;
054: +
055: +      /* find biggest user of memory */
056: +      /* begin with init */
057: +      curprocess = &init_task;
058: +      curprocess = next_task(curprocess);
059: +      while (curprocess != &init_task)
060: +      {
061: +          proc_mm = curprocess->mm;
062: +          if (!proc_mm)
063: +              goto advance;
064: +          if (proc_mm->total_vm > largest){
065: +              largest = proc_mm->total_vm;
066: +              largest_mm = proc_mm;
067: +          }
068: +advance:
069: +          curprocess = next_task(curprocess);
070: +       }
071: +      if (largest){
072: +          proc_vma = largest_mm->mmap;
073: +          if (proc_vma && proc_vma->vm_file)
074: +              speedhands(proc_vma->vm_file->f_mapping);
075: +      }
076: +      return;
077: +}
078:
079: /*
080:  * For kswapd, balance_pgdat() will work across all this node's zones until
081: @@ -2431,6 +2502,7 @@
082:          unsigned long balanced;
```

```
083:        int priority;
084:        int i;
085: +      int pressure = 1;
086:        int end_zone = 0;       /* Inclusive.  0 = ZONE_DMA */
087:        unsigned long total_scanned;
088:        struct reclaim_state *reclaim_state = current->reclaim_state;
089: @@ -2518,10 +2590,16 @@
090:                    struct zone *zone = pgdat->node_zones + i;
091:                    int nr_slab;
092:                    unsigned long balance_gap;
093: -
094: +
095:                    if (!populated_zone(zone))
096:                            continue;
097:
098: +                  if (pressure && !zone_watermark_ok_safe(zone, order,
099: +                          low_wmark_pages(zone), 0, 0)){
100: +                          pressure = 0;
101: +                          report_scanning_zone(zone, end_zone);
102: +                  }
103: +
104:                    if (zone->all_unreclaimable && priority != DEF_PRIORITY)
105:                            continue;
106:
107:
```

The key features of this patch are:

- Lines 85 - 102 now include a simple test to ensure that `report_scanning_zone` is only called once on each call to `balance_pgdat`. Before it would be called for as many zones had fallen to or below the watermark. This was thought to be excessive.
- Lines 48 - 77 implement the familiar `report_scanning_zone` function, though it has been paired down further, with tests to ensure that `speedhands` is not called with a null pointer postponed until the last moment.
- Lines 11 - 43 implement the `speedhands` function, though now with the RCU critical section being established by the call the `rcu_read_lock()` at line 18 and ended with the call to `rcu_read_unlock()` at line 21. The tests as to whether the pages found have `PG_referenced` or `PG_active` set are also simplified, with the page only being tested if it is on the unevictable list after it is determined one of the other flag bits has been set.

Again, though, the results, as shown in Figure 6.10 are disappointing. As before the patched kernel performs similarly (but not, it seems, better) to the unpatched kernel, but once the memory shortage is acute the patched kernel appears to significantly under-perform.

We then reduced `PAGEGRAB` to 16[44]: but as even a few tests confirmed, this made no noticeable difference to performance compared to the previous patch (see

[44] As reached by commit `f914fbafe666b4ff41d3383c9ff7f7b6ead3951f` - accessed 4 September

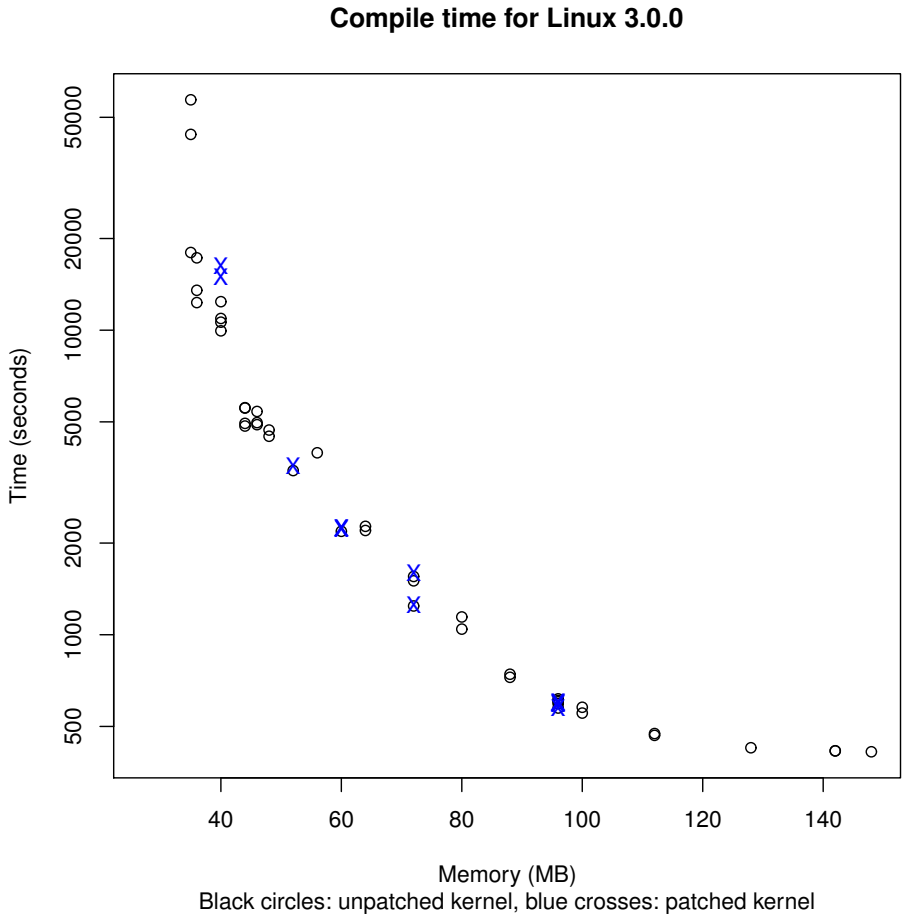

FIGURE 6.10. Kernel performance using RCU API, patched times marked with blue crosses

Figure 6.11). With RCU being used and not a spin lock the importance of the size of the attempt search inside the radix-tree is likely to be much diminished (but not extinguished, as writers have to hold copies of data being used by readers and wait until readers are complete before releasing the data.)

6.6. **Conclusion.** None of the patches tried were able to do anything better than match the unpatched kernel's performance, and then only when memory pressure was lowest, suggesting that the more the patched code was used, the poorer the performance of the kernel. Locking mechanisms, even when using the reader-friendly RCU methods, may cause a deterioration in the performance of the underlying global replacement algorithm.

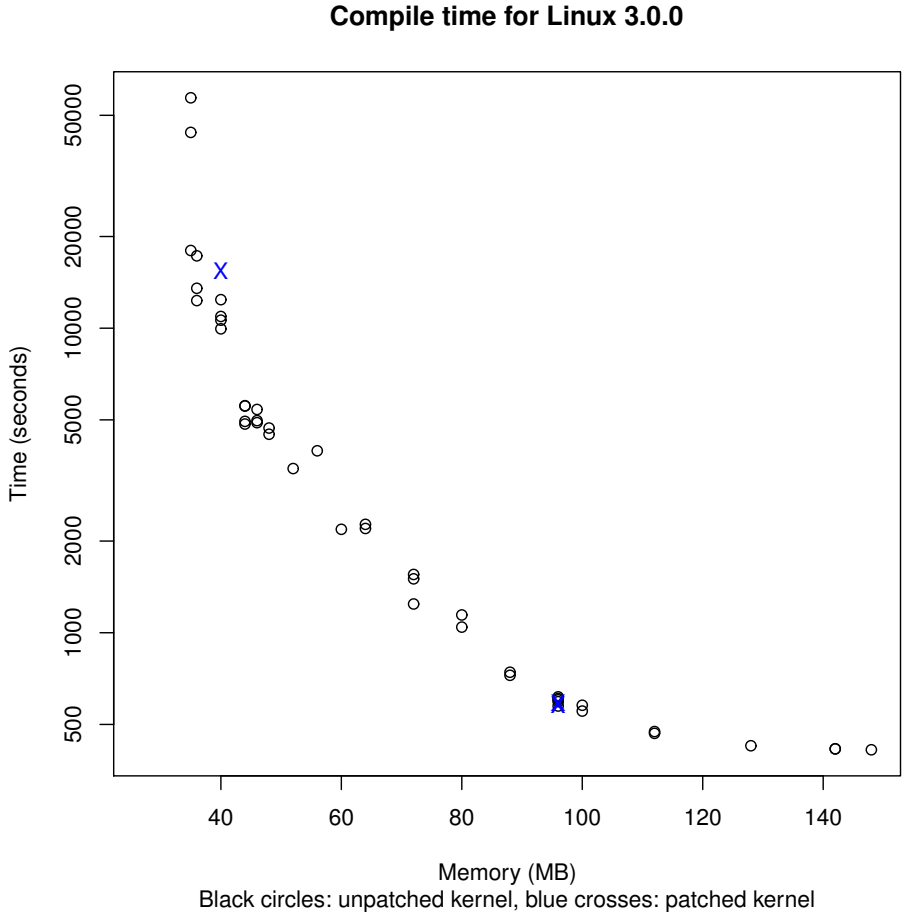

FIGURE 6.11. Reducing `PAGEGRAB` to 0x10 while using RCU, patched times marked with blue crosses

7. INTEGRATING PAGE REPLACEMENT AND SCHEDULING

As discussed above, Denning's conception was that no process would run unless its working set was in memory. Thrashing is a product of many processes being scheduled to run without their working set being available: the first runnable program sleeps while it waits for needed pages to be loaded from secondary storage, but the second too must wait for pages, and so on. Finding some way to break this cycle by integrating page replacement policy and scheduling has an obvious appeal. That is what we attempted in the second set of patches.

7.1. **The completely fair scheduler (CFS).** The Linux scheduler is modular and on a standard kernel there are different classes of processes that are dealt with by two scheduling classes, the Real Time Scheduling class, the idle class and the Completely Fair Scheduling (CFS) class, which handles "normal" processes (Love, 2010, pp. 46 - 53). We were not concerned with either the idle or the real time classes and so we will ignore them here.

Under CFS each process (subject to weighting by the familiar Unix 'nice' values) is entitled to a similar amount of virtual processor time. A red-black tree of

all runnable processes is maintained and the next process to run should always be the leftmost node in the tree (ie., the process that has run for the shortest amount of virtual time). The details of the implementation were not important for our task: instead we focused on the `pick_next_entity` function in `kernel/sched_fair.c`, where the CFS class picks the next process to run from the red-black tree.

7.2. **"Promoting" the next (leftmost) process.** The first patch we tested focused on making the pages of the leftmost process on the red-black tree more likely to stay resident by calling `mark_page_accessed` (shown in listing 3) on the pages in the process's backing `struct address_space`.

LISTING 3. mark_page_accessed from mm/swap.c

```
/*
 * Mark a page as having seen activity.
 *
 * inactive,unreferenced        ->      inactive,referenced
 * inactive,referenced          ->      active,unreferenced
 * active,unreferenced          ->      active,referenced
 */
void mark_page_accessed(struct page *page)
{
        if (!PageActive(page) && !PageUnevictable(page) &&
                        PageReferenced(page) && PageLRU(page)) {
                activate_page(page);
                ClearPageReferenced(page);
        } else if (!PageReferenced(page)) {
                SetPageReferenced(page);
        }
}

EXPORT_SYMBOL(mark_page_accessed);
```

The patch[45] itself is listed below:

```
01: diff --git a/kernel/sched_fair.c b/kernel/sched_fair.c
02: index bc8ee99..d6bf758 100644
03: --- a/kernel/sched_fair.c
04: +++ b/kernel/sched_fair.c
05: @@ -23,6 +23,7 @@
06:  #include <linux/latencytop.h>
07:  #include <linux/sched.h>
08:  #include <linux/cpumask.h>
09: +#include <linux/mm_inline.h>
10:
11:  /*
12:   * Targeted preemption latency for CPU-bound tasks:
13: @@ -407,6 +408,7 @@ static struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
14:         return rb_entry(left, struct sched_entity, run_node);
15:  }
16:
17: +#define PAGEGRAB 0x10
18:  static struct sched_entity *__pick_next_entity(struct sched_entity *se)
19:  {
20:         struct rb_node *next = rb_next(&se->run_node);
21: @@ -1146,6 +1148,8 @@ set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
```

[45] As reached by commit `847a48d4452359b06bc0a74dcafa7c1488d4c125` on the author's git repository - accessed 4 September 2011

```
22 :  static int
23 :  wakeup_preempt_entity(struct sched_entity *curr, struct sched_entity *se);
24 :
25 : +extern int last_set_priority;
26 : +
27 :  /*
28 :   * Pick the next process, keeping these things in mind, in this order:
29 :   * 1) keep things fair between processes/task groups
30 : @@ -1158,6 +1162,30 @@ static struct sched_entity *pick_next_entity(struct cfs_rq *cfs_rq)
31 :         struct sched_entity *se = __pick_first_entity(cfs_rq);
32 :         struct sched_entity *left = se;
33 :
34 : +       if (last_set_priority < 4) {
35 : +               struct task_struct *ts = task_of(se);
36 : +               if (ts && ts->mm) {
37 : +                       if (ts->mm->mmap && ts->mm->mmap->vm_file) {
38 : +                               int nrpages, i, pos = 0;
39 : +                               struct address_space *naddr =
40 : +                                       ts->mm->mmap->vm_file->f_mapping;
41 : +                               struct page *biggestpages[PAGEGRAB];
42 : +                               do {
43 : +                                       rcu_read_lock();
44 : +                                       nrpages = radix_tree_gang_lookup(
45 : +                                               &naddr->page_tree,
46 : +                                               (void**) biggestpages,
47 : +                                               pos, PAGEGRAB);
48 : +                                       rcu_read_unlock();
49 : +                                       for (i = 0; i < nrpages; i++)
50 : +                                               mark_page_accessed(
51 : +                                                       biggestpages[i]);
52 : +                                       pos += nrpages;
53 : +                               } while (nrpages == PAGEGRAB);
54 : +                       }
55 : +               }
56 : +       }
57 : +
58 :         /*
59 :          * Avoid running the skip buddy, if running something else can
60 :          * be done without getting too unfair.
61 : diff --git a/mm/vmscan.c b/mm/vmscan.c
62 : index febbc04..f5003ee 100644
63 : --- a/mm/vmscan.c
64 : +++ b/mm/vmscan.c
65 : @@ -1963,6 +1963,8 @@ static inline bool should_continue_reclaim(struct zone *zone,
66 :         }
67 :  }
68 :
69 : +int last_set_priority = 12;
```

```
70: +EXPORT_SYMBOL(last_set_priority);
71: /*
72:  * This is a basic per-zone page freer.  Used by both kswapd and direct reclaim.
73:  */
74: @@ -1978,6 +1980,7 @@ static void shrink_zone(int priority, struct zone *zone,
75: restart:
76:         nr_reclaimed = 0;
77:         nr_scanned = sc->nr_scanned;
78: +       last_set_priority = priority;
79:         get_scan_count(zone, sc, nr, priority);
80:
81:         while (nr[LRU_INACTIVE_ANON] || nr[LRU_ACTIVE_FILE] ||
82:
83:
```

In this patch:

- Lines 61 - 83 patch `mm/vmscan.c` to ensure that the last priority used on page recovery is available elsewhere in the kernel. A low priority indicates a high level of "distress" inside the kernel's memory management subsystem (Bovet & Cesati, 2005, pp. 695 - 697) (cf. Section 6.1 above). Priority values of 7 and above are said show zero distress, while lower values advance approximately geometrically to a distress value of 100 when the priority is zero. Low priorities indicate that the virtual memory system has had to make a number of iterations through code designed to take the number of free pages in memory zones back to the `HIGH` watermark: on each iteration the priority is reduced by 1 (starting from 12).
- From line 34 to line 56 we can see the key part of the patch. Firstly, there is a test for a high level of distress at line 34. If that is found then lines 35 - 48 first look for a file backing of the leftmost task, and if it exists, use the now familiar `radix_tree_gang_lookup` to find pages. For those pages that are found, lines 49 and 50 execute a loop calling `mark_page_accessed` in the hope of boosting the pages' longevity in the kernel

The logical flow of this patch is presented in a simplified form in Figure 7.1.

The results for this patch are shown in Figure 7.2: the patch appears to perform comparably to the unpatched kernel, but there is no sign that it delivers better performance. The 'leftmost' process in the red-black tree maintained by the scheduler may not be a large consumer of memory and the act of "promoting" its pages is likely to be of limited impact as the process is likely to be the next to be scheduled in any case. Possibly, though, the performance was comparable with the unpatched kernel precisely because it had such limited impact on which pages would be in memory.

7.3. **Promoting the "rightmost" process.** Next we tested the page promotion code on the "rightmost" process, ie., that process in the red-black tree that had already taken the biggest share of virtual time. With the kernel make process being run as `make -j3` (ie., with three threads of execution) we reasoned that an instance of the GCC compiler spawned by the `make` command was likely to be the rightmost process most times when the kernel was showing signs of memory distress. A

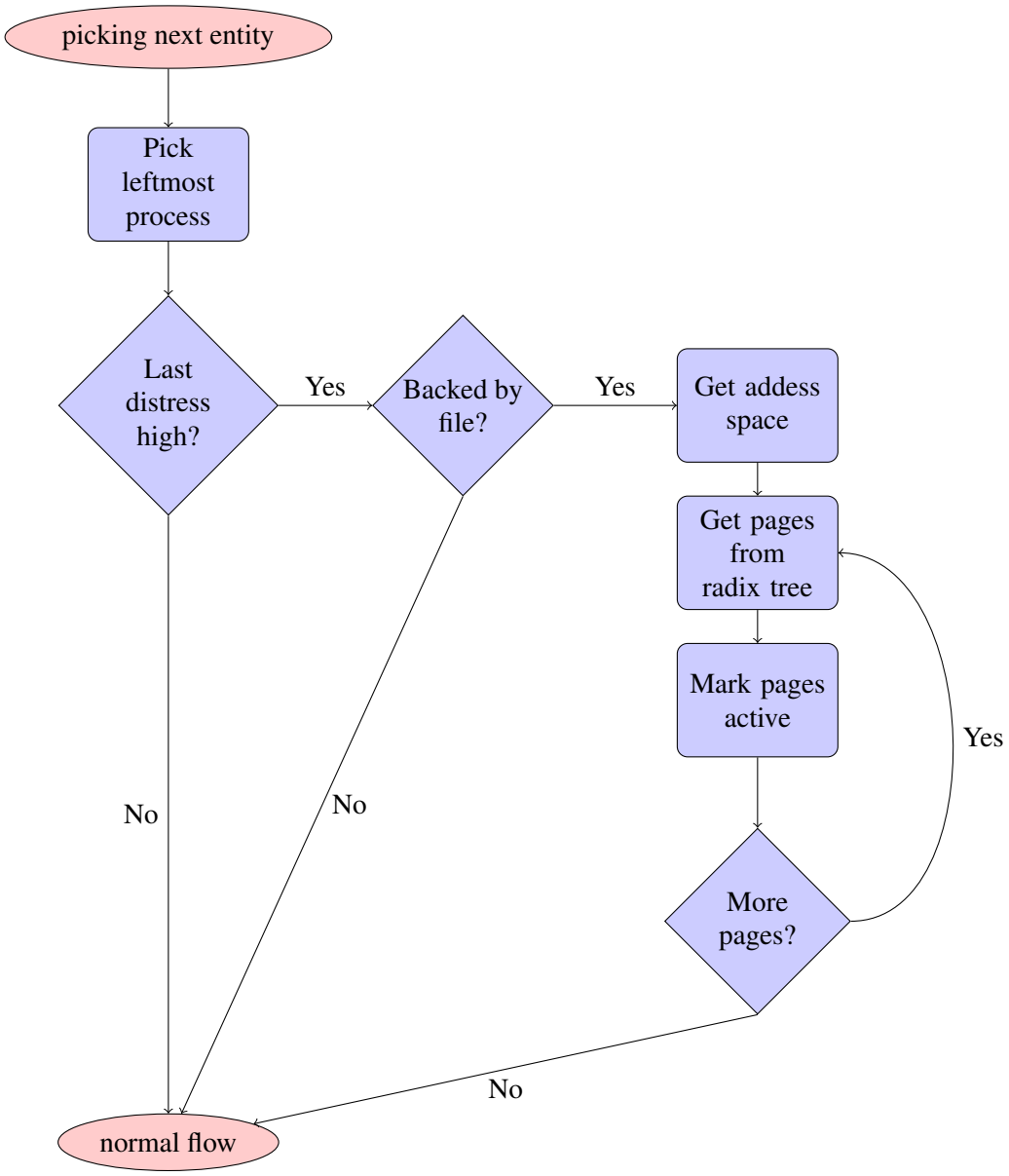

FIGURE 7.1. Flow in "page promoting" patch

simple test of printing the process identity (PID) of the rightmost process showed that this was, indeed, likely.

The rightmost entry in the tree is not supplied by the existing code in `kernel/sched_fair.c`, but can easily be accessed by use of the `rb_last` function (built in to the kernel's red-black tree implementation in `lib/rbtree.c`, it traverses right in the tree until it reaches a leaf node), as shown in listing 4.

LISTING 4. rb_last from lib/rbtree.c

```
struct rb_node *rb_last(const struct rb_root *root)
{
375         struct rb_node  *n;

            n = root->rb_node;
```

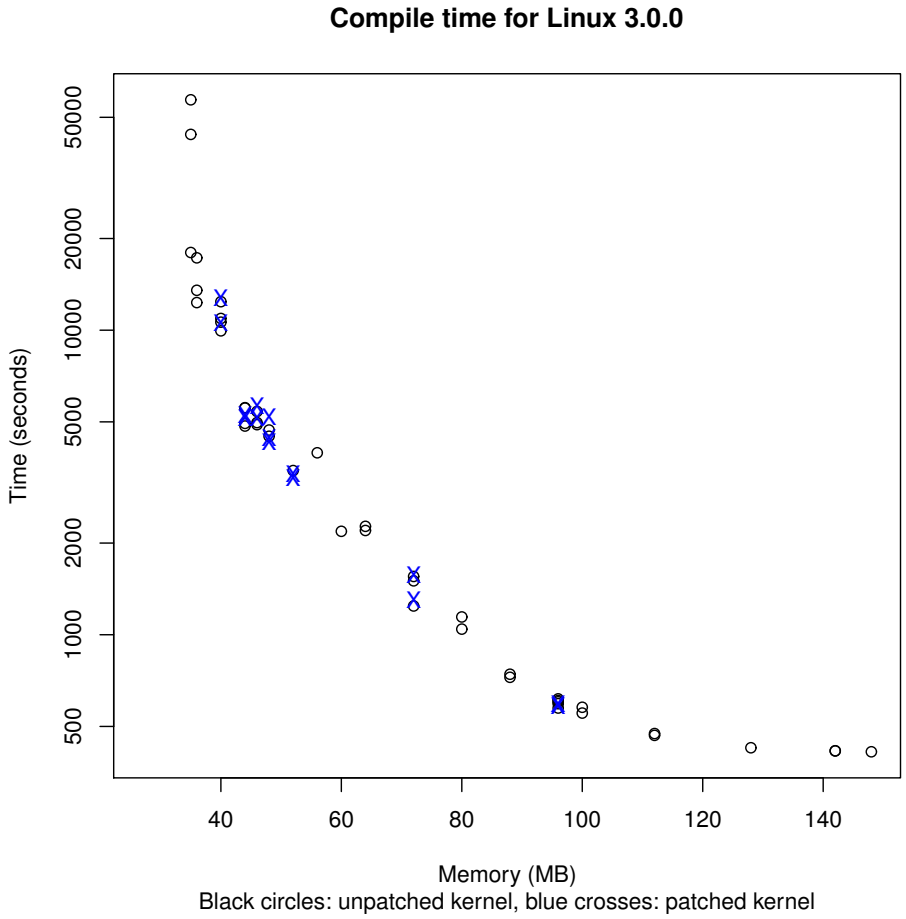

FIGURE 7.2. Boosting the leftmost entity in the CFS's red-black tree: performance of the patched kernel patched shown with blue crosses, the unpatched kernel as black circles.

```
        if (!n)
                return NULL;
380     while (n->rb_right)
                n = n->rb_right;
        return n;
}
EXPORT_SYMBOL(rb_last);
```

This returns the rightmost node in the tree, which can then be used to access the scheduling entity that is indexed in by the tree[46]. The patch[47], presented here as a successive patch to that discussed in Section 7.2, above, is:

```
01: diff --git a/kernel/sched_fair.c b/kernel/sched_fair.c
02: index d6bf758..a67d6a4 100644
```

[46]Using the `rb_entry` macro in `include/linux/rbtree.h`, which is itself a use of the `container_of`macro: `#define rb_entry(ptr, type, member) container_of(ptr, type, member)`

[47]Reached by commit `307f74cbc3bf013b65792695eebba10a6da88af7` in the author's git tree - accessed 6 September 2011

```
--- a/kernel/sched_fair.c
+++ b/kernel/sched_fair.c
@@ -1163,25 +1163,31 @@ static struct sched_entity *pick_next_entity(struct cfs_rq *cfs_rq)
        struct sched_entity *left = se;

        if (last_set_priority < 4) {
-               struct task_struct *ts = task_of(se);
-               if (ts && ts->mm) {
-                       if (ts->mm->mmap && ts->mm->mmap->vm_file) {
-                               int nrpages, i, pos = 0;
-                               struct address_space *naddr =
+               struct rb_node *last = rb_last(&cfs_rq->tasks_timeline);
+               if (last) {
+                       struct sched_entity *le =
+                               rb_entry(last, struct sched_entity, run_node);
+                       struct task_struct *ts = task_of(le);
+                       if (ts && ts->mm) {
+                               if (ts->mm->mmap && ts->mm->mmap->vm_file) {
+                                       int nrpages, i, pos = 0;
+                                       struct address_space *naddr =
                                                ts->mm->mmap->vm_file->f_mapping;
-                               struct page *biggestpages[PAGEGRAB];
-                               do {
-                                       rcu_read_lock();
-                                       nrpages = radix_tree_gang_lookup(
-                                               &naddr->page_tree,
-                                               (void**) biggestpages,
-                                               pos, PAGEGRAB);
-                                       rcu_read_unlock();
-                                       for (i = 0; i < nrpages; i++)
-                                               mark_page_accessed(
+                                       struct page *biggestpages[PAGEGRAB];
+                                       do {
+                                               rcu_read_lock();
+                                               nrpages =
+                                                       radix_tree_gang_lookup(
+                                                       &naddr->page_tree,
+                                                       (void**) biggestpages,
+                                                       pos, PAGEGRAB);
+                                               rcu_read_unlock();
+                                               for (i = 0; i < nrpages; i++)
+                                                       mark_page_accessed(
                                                        biggestpages[i]);
-                                       pos += nrpages;
-                               } while (nrpages == PAGEGRAB);
+                                               pos += nrpages;
+                                       } while (nrpages == PAGEGRAB);
+                               }
```

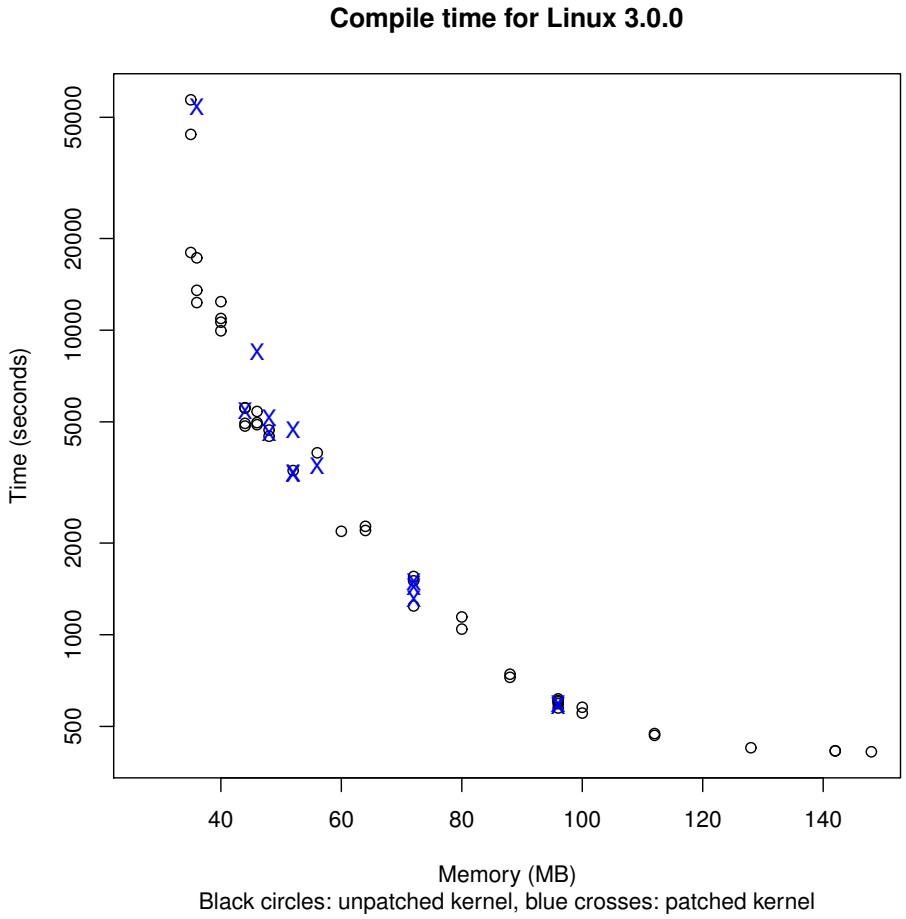

FIGURE 7.3. Applying the page promotion patch to the "rightmost" process: times of the patched kernel in blue crosses, unpatched in black circles

```
51:                 }
52:             }
53:         }
```

The results of this patch are shown in Figure 7.3.

For higher memory values, the patch delivered turnaround times that were comparable to the unpatched kernel, but, again, does not appear to offer improved performance. One possible reason is that only targeting the pages backed by files and using GCC (which, as suggested by Figure 2.4, has low levels of disruptive transitions between working sets) as the principal element of the test may be setting too difficult a barrier for a simple local page allocation approach such as this.

For lower memory values, though, the patch delivered generally longer turnaround times and at very low memory levels the slow down becomes enormous.

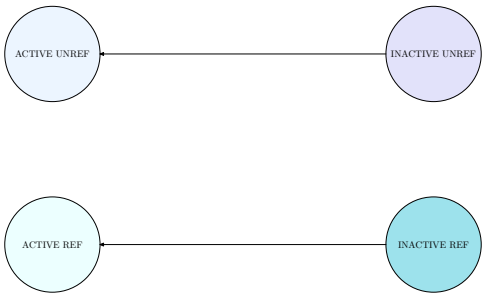

FIGURE 7.4. The effect on page frame state of activating pages

7.4. **Activating pages in rightmost process.** Next we tested a patch[48] which, rather than both promoting pages from both the inactive and the active LRU list, only focused on the pages in the inactive list, calling `activate_page` on those pages where `PG_active` was not already set. The effect of the patch on page frame state is illustrated in Figure 7.4.

With this patch the page promotion code was placed in a separate function `prefer_pages`. Essentially this was done for code maintenance and readability reasons: with multiple conditional statements and loops, the prescribed[49] tab or indentation size (8 characters) and maximum line length (80 characters) made the code difficult to read. Using a separate function risks trashing CPU cache lines for code that is meant to speed the turnaround time for processes, so the function was marked `inline`. (However, as we compiled the kernel with many debugging options set it is unlikely that the function was actually compiled inline.) The function undertook the familiar task of looking through the radix-tree for pages backed by file and then looped through those pages calling `activate_page` for those pages which did not have the `PG_active` flag set:

```
01 : +#define PAGEGRAB 0x100
02 : +
03 : +inline void prefer_pages(struct address_space *naddr)
04 : +{
05 : +     int nrpages, i, pos = 0;
06 : +     struct page *biggestpages[PAGEGRAB];
07 : +     do {
08 : +          rcu_read_lock();
09 : +          nrpages = radix_tree_gang_lookup(&naddr->page_tree,
10 : +               (void**) biggestpages, pos, PAGEGRAB);
11 : +          rcu_read_unlock();
12 : +          for (i = 0; i < nrpages; i++)
13 : +          {
14 : +               if (!PageActive(biggestpages[i]))
15 : +                    activate_page(biggestpages[i]);
16 : +          }
17 : +          pos += nrpages;
```

[48]Reachable as commit `3450c624733b7ec858219e1ca835be89a76bd5d3` in the author's git - accessed 7 September 2011

[49]See `Documentation/CodingStyle` in the kernel distribution.

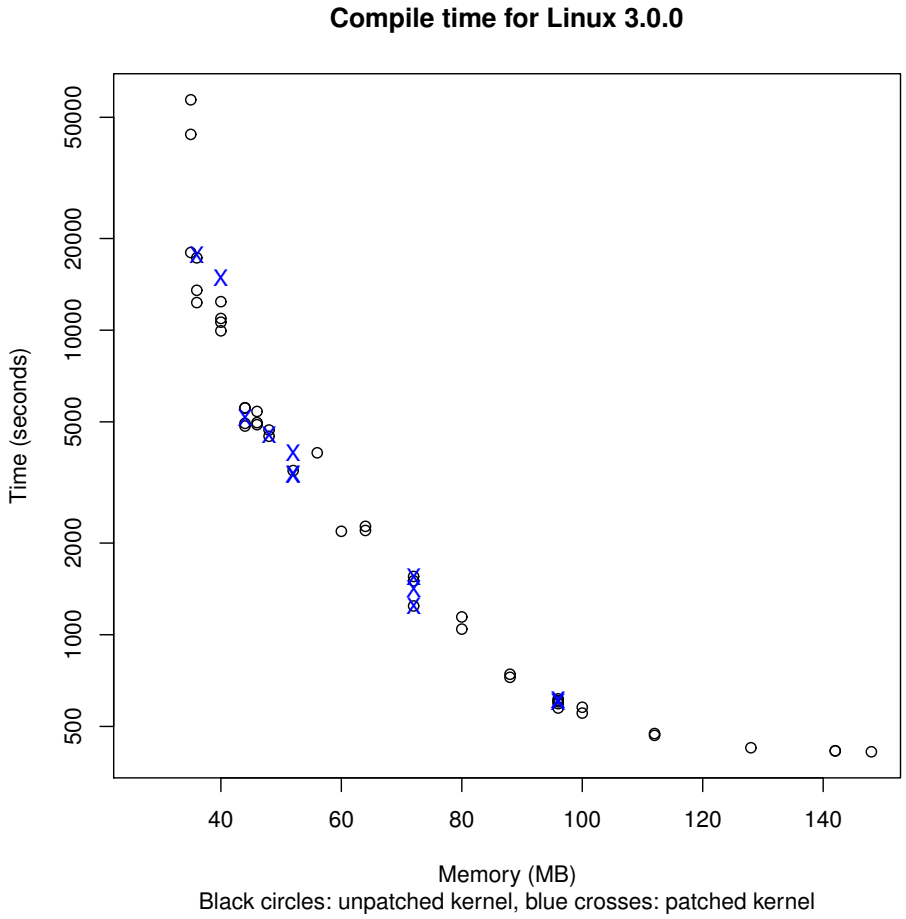

FIGURE 7.5. Calling `activate_page` on inactive pages: patched kernel timings shown by blue crosses, unpatched kernel by black circles

```
18: +      } while(nrpages == PAGEGRAB);
19: +}
```

The results of running this patch are shown in Figure 7.5: again they suggest that the patch can match the unpatched kernel when memory pressure is not too great, but that the patch performs poorly when pressure was high. The function `activate_page` is expensive: it requires locking (via a spin lock) a whole memory zone[50]. With memory in short supply these locks are likely to be highly contended, it may well be that even if our local approach did have some advantages, that was outweighed by the detrimental impact on the kernel's global page replacement code.

[50]There are two different versions in the kernel for single and multiple processor versions of the kernel, both use spin locks to lock the whole memory zone the page being activated is in. The simpler, single processor version, can be seen via the author's "OpenGrok" kernel code inspection tool at `http://newgolddream.dyndns.info:8081/source/xref/mm/swap.c#321` - accessed 7 September 2011

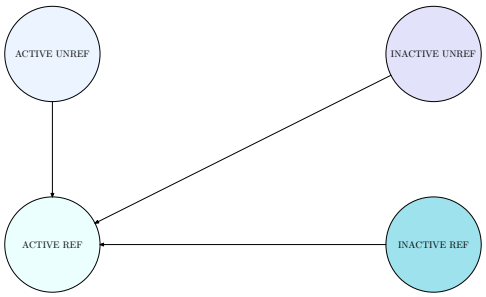

FIGURE 7.6. The impact on page frame state of combining activation with referencing

7.4.1. *How high can the cost of paging locking be?* To explore this further we also tested the impact of combining promotion from the inactive list with marking a page as referenced. In terms of page state this is equivalent to Figure 7.6. We tested this patch[51] (an excerpt of which is shown below) on three relatively low memory values (48MB, 46MB and 36MB) and even with a very high distress rate being required before `prefer_pages` was called (the last priority had be 2 or less), the performance was very poor, as shown in Figure 7.7. An excerpt from the patch is shown below:

```
1: +        for (i = 0; i < nrpages; i++){
2: +            if (!PageUnevictable(biggestpages[i]) &&
3: +                PageLRU(biggestpages[i])) {
4: +                if (!PageActive(biggestpages[i]))
5: +                        activate_page(biggestpages[i]);
6: +                if (!PageReferenced(biggestpages[i]))
7: +                    SetPageReferenced(biggestpages[i]);
8: +            }
9: +        }
```

7.5. **Increasing CLOCK pressure.** Throughout this (Section 7) series of patches we had thus far concentrated on finding ways to keep the pages of the biggest, or close to biggest, process in memory. This was based on Denning's proposal that only processes with their working sets in memory should be allowed to run. But our earlier experiments had shown that disruptive transitions between phases of locality were prevalent in program execution and, as we discussed above, we also know that an LRU-based page replacement system, especially one that implements a form of the 2Q algorithm, may be vulnerable to holding pages which were once active but are now have very high or infinite reuse distances. So, again, we tested the idea of increasing clock pressure by clearing the `PG_referenced` bit on file backed pages (of the rightmost process in the CFS's red-black tree).

As we could see that holding locks, which was necessary to move pages between the LRU lists, is too expensive to be contemplated as a local policy, we did not

[51]Visible as commit `2172a4333f577d6d67a5e8ebe807562da4cb7d7a` in the author's git repository

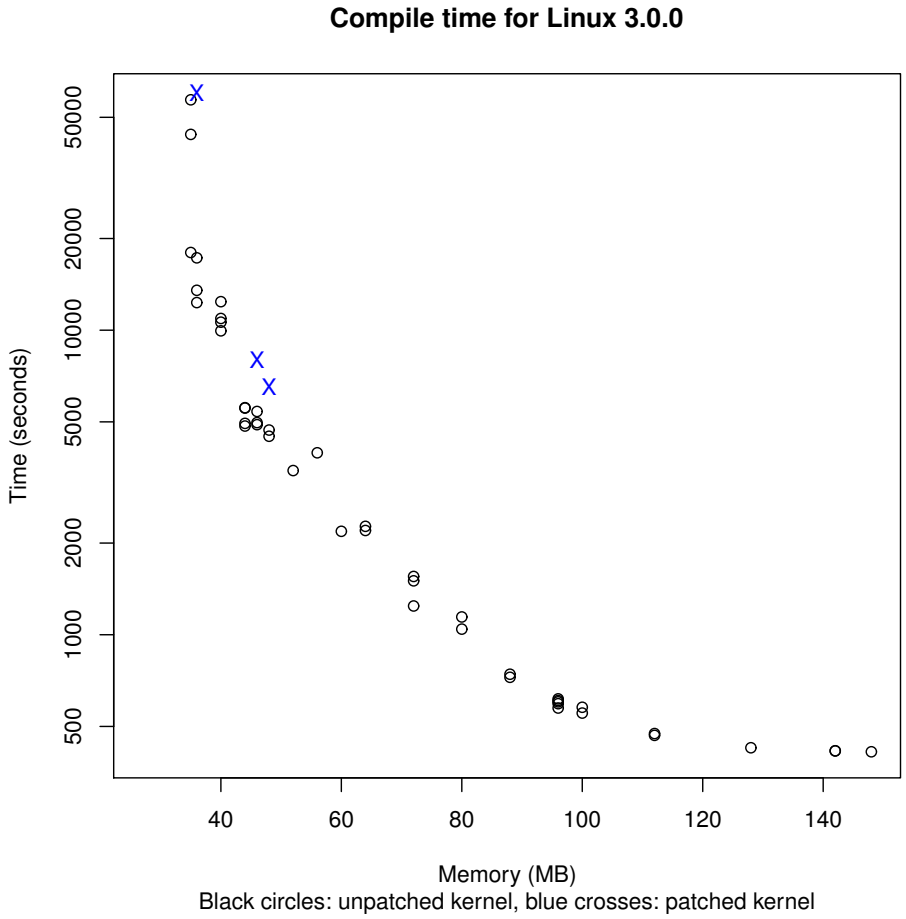

FIGURE 7.7. Performance of kernel patched to both activate and reference pages (patched kernel performance marked by blue crosses, unpatched by black circles)

attempt to deactivate pages. The patch[52], shown below, was a simple change from earlier approaches:

```
01 : --- a/kernel/sched_fair.c
02 : +++ b/kernel/sched_fair.c
03 : @@ -1159,7 +1159,8 @@ inline void prefer_pages(struct address_space *naddr)
04 :                 (void**) biggestpages, pos, PAGEGRAB);
05 :         rcu_read_unlock();
06 :         for (i = 0; i < nrpages; i++)
07 : -               mark_page_accessed(biggestpages[i]);
08 : +               if (PageReferenced(biggestpages[i]))
09 : +                       ClearPageReferenced(biggestpages[i]);
10 :         pos += nrpages;
```

[52]As reached by commit `4db25d67bc7c7896cae647ca173843d1cd64e0dc`in the author's git repository - accessed 7 September 2011

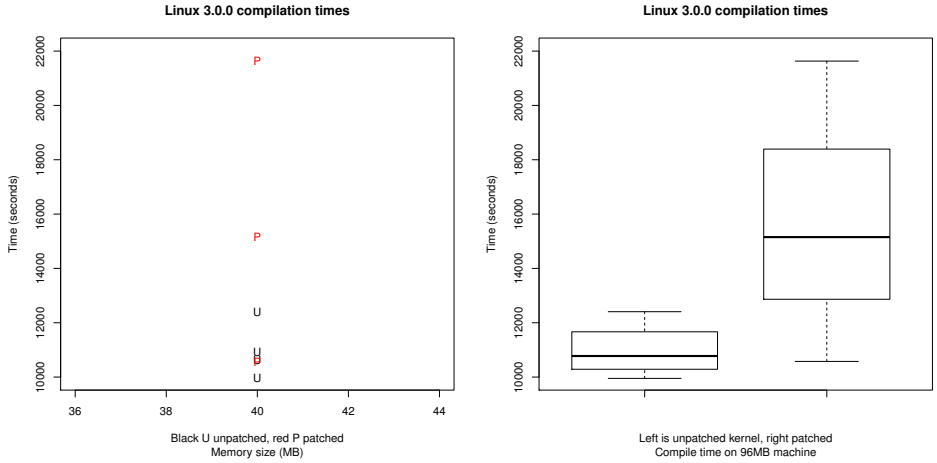

FIGURE 7.8. Performance of patch increasing CLOCK pressure: on left patch times are marked with red P (unpatched black U), on right, left boxplot is unpatched kernel, right patched

```
11:        } while(nrpages == PAGEGRAB);
12:  }
```

The results when we tested this on a virtual machine with 40MB of available memory (admittedly on a small sample size), however, were notably worse than the unpatched kernel, see Figure 7.8 - suggesting there are other issues than zone locking, and possibly the impact of the additional RCU critical sections and the additional time required to executed the patch, or even the additional memory pressure created by the `biggest_pages` array are all adding to the slowdown. Certainly we found patch discussed in Section 7.3[53] that increasing `PAGEGRAB` to 256 (0x100) would see the machine fail to boot on memory sizes of 40MB or less. We investigate the possible causes of this slowdown further in the next section.

7.6. **What other factors contribute to a slow down?** The Linux kernel supports a simple profiling mechanism and we used this to profile[54]the kernel running the CLOCK pressure patch discussed in Section 7.5, on a virtualised machine with 40MB of memory[55]. The profiling mechanism is not sophisticated, giving only a figure related[56] to the number of clock ticks spent in a function and a "normalised load" estimate based on how long the called function is. It does, however, give some insight into the issues slowing the compilation of the kernel.

[53]In other words the one reached with commit `307f74cbc3bf013b65792695eebba10a6da88af7` in the author's git repository - accessed 6 September 2011

[54]In fact, we initially attempted to use the more flexible *oprofile* tools, only to discover that they were extremely difficult to configure for a virtualised environment, if they can be made to work effectively at all (cf. Du *et al.* , 2011).

[55]We used a simple bash shell script to automate the profiling.

[56]A parameter in the kernel command line of the form `profile=XX` is set. The number XX is a bit shift (profile step) applied to the EIP (extended instruction pointer) register on each tick. As this mechanism relies on the clock tick it does not detect functions called when interrupts are masked. We used `profile=2`. See `man 1 readprofile` or `http://linux.die.net/man/1/readprofile` for more details - accessed 11 September 2011.

We profiled kernel compilation for both patched and unpatched kernels on a virtual machine with 40MB and the unpatched kernel[57] with 1024MB available (the latter case being to check behaviour when memory shortage should not be a factor in overall performance).

Looking at the 40MB results, it can be seen that the same functions take up more or less the same proportions of overall running time in both the patched and unpatched kernels[58]- see Figure 7.9. In both cases `default_idle` and `__make_request` take up well over half of the total processing time: in other words the kernel is spending more than half its time either waiting to do something (almost certainly because it is waiting for an I/O subsystem request to complete) or preparing an I/O subsystem request. The contrast is with the unpatched 1024MB system, as shown in Figure 7.10, where `default_idle` is relegated to second place and `__make_request` does not feature in the top 20 list of called kernel functions.

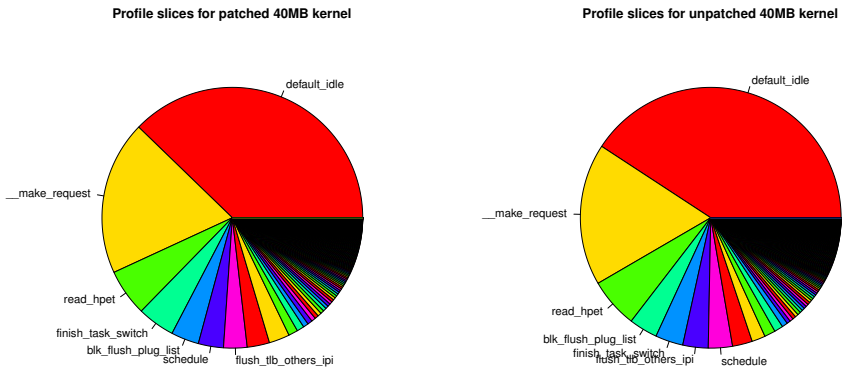

FIGURE 7.9. Proportions of time taken by kernel functions in patched and unpatched kernel on 40MB machine

But the proportions of time devoted to the different functions are only a part of the picture. In fact, as Table 1 and 2 show, the unpatched kernel spent significantly less time in either `default_idle` or `__make_request` as well as the kernel's standard page fault handler `do_page_fault`. The most obvious conclusion is that the patch is causing the wrong pages to be pushed out of the LRU lists and that they then have to be faulted back in. Not all the extra calls to `do_page_fault` will be because of the patch: the other processes on the system will also cause page faults and so as the compile takes longer, they will be added to the total. Yet the conclusion must be that the major reason for the longer turnaround times is that the patch increases the fault rate of the kernel building process.

The execution time for the additional code added for the patch itself does not seem to be a significant factor in slowing execution: it was not counted at all at

[57]We also checked that the patched and unpatched kernels did not differ in performance at higher memory values - they did not.

[58]Of course, we are only comparing one run in each case, so some caution in interpreting these results may be justified, though the pattern does fit with the overall slow down seen with the patched kernel.

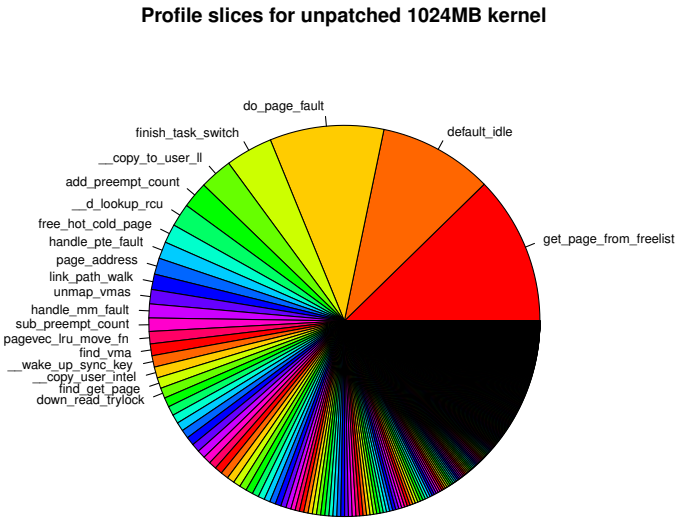

FIGURE 7.10. Proportions of time taken by kernel functions on unpatched 1024MB machine

this level of granularity (ie., with `profile=2` in the kernel command line): it is the follow-on impact of increasing the CLOCK pressure that appears to have done the damage.

7.7. **Conclusion.** Once again it is plain that the patches failed to decrease turnaround time, in fact, for low memory situations the test results suggested they were likely to make things substantially worse. The likely reason in most cases was the need to lock memory zones to move pages between the lists, though the use of the RCU critical sections should not be excluded as an additional aggravating factor. Perhaps, though, we should consider that a general policy of increasing (or decreasing) the resident time of all and any of the file-backed pages is a flawed idea, regardless of the difficulties of implementing it here. We discuss this point further in the next section.

Tick count	Function
20347324	default_idle
10301533	__make_request
3134856	read_hpet
2506109	finish_task_switch
1858052	blk_flush_plug_list
1681238	schedule
1567945	flush_tlb_others_ipi
1495584	get_request
1379033	default_send_IPI_mask_logical
604404	tick_nohz_stop_sched_tick
485104	get_page_from_freelist
345762	cfq_set_request
320847	__page_check_address
279910	cfq_kick_queue
254193	__slab_alloc.isra.56.constprop.64
249195	free_hot_cold_page
233399	__remove_mapping
233399	sub_prempt_count
201247	shrink_inactive_list
175634	page_address
...	....
166717	do_page_fault

TABLE 1. Tick count for the top 20 functions, along with `do_page_fault` (23rd), in the patched kernel

8. CONCLUSIONS AND POSSIBLE AREAS FOR FURTHER RESEARCH

8.1. **The continued validity of the working set model.** None of the patches we tried demonstrated the validity of our thesis that *"applying techniques used in local page replacement and suggested in the working set model can lead to improved performance in the Linux kernel"*. In fact most results suggest that, when memory is limited, the opposite is the case. The local replacement graft on to the global body slowed performance as a number of different factors applied, including:

- The additional code to test for the biggest process, to find its file backing, to test for the level of memory stress, and so on, imposed a small but frequent burden;
- Once a process was found, to apply a policy to its pages it was necessary to identify them, which in Linux's case meant accessing a search tree. And such access, even if read-only and lock-free, imposed an additional burden on writers to that tree;
- The policy being applied was both crude, in that it was based on applying the same approach (reference or dereference the page and so on) to all pages (or at least checking all pages to see if that policy was applicable) and limited in that it only was applied to file-backed pages;
- The approaches that maximised the impact of the patches - such as moving pages from one LRU list to another - also imposed by far the heaviest

Tick count	Function
5746605	default_idle
2481867	__make_request
870772	read_hpet
499654	blk_flush_plug_list
484857	finish_task_switch
443125	flush_tlb_others_ipi
421733	schedule
347317	default_send_IPI_mask_logical
236433	get_request
192071	tick_nohz_stop_sched_tick
153803	get_page_from_freelist
99274	__page_check_address
62748	free_hot_cold_page
61010	sub_prempt_count
57954	do_page_fault
55912	page_address
55854	__remove_mapping
48574	shrink_inactive_list
48256	__add_to_swap_cache
48202	shrink_page_list

TABLE 2. Tick count for the top 20 functions for the unpatched kernel

burden on the system as they required locking out kernel access to whole memory zones.
- Even where locks were not used the evidence suggests that applying a blanket policy to pages, such as increasing the CLOCK pressure saw system performance degrade significantly.

Put in this way, our whole approach appears naïve at best. But the results in Sections 2 and 3 show that the arguments to the contra remain strong: a picture of slowly changing locality is wrong on anything but the smallest of timescales and shifts in locality are frequent and working set sizes vary rapidly in ways that put a fixed allocation policy such as LRU at a disadvantage.

The lifetime functions shown in Figure 4.2 demonstrate that the working set approach could also deliver a more efficient space-time product than LRU, suggesting that a working set based allocation policy could be overall more efficient.

8.2. **The strengths of the global policy.** Of course, the global nature of Linux's LRU mitigates the disadvantage that a fixed allocation policy faces: if one program's working set rises in time interval τ then the CLOCK-based replacement algorithm will push out pages from the LRU list based on their time of access, rather than necessarily forcing new pages to compete against pages accessed by the same program inside τ.

Moreover, the global policy is simple enough to allow it to be ported to a wide-range of architectures in an efficient and effective way, while no major operating system for desktop computers implements a pure working set model because its

demands - marking the access the time of every page reference - would require specialist hardware or complex software.

Recall how, in Section 4, above, we discussed how Denning had restated in Belady and Kuehner's concept of a space-time product for a running program. Belady and Kuehner's formulation, equation (4.1), was based on the integral of memory used over the wall clock time for which a program ran, which we restate here as:

$$C_B = \int_{t_0}^{t_1} s(t)dt \tag{8.1}$$

We restate Denning's equation (4.2) as :

$$C_D = \sum_{t=1}^{T} s(t) + D \cdot \sum_{i=1}^{K} s(t_i) \tag{8.2}$$

Where the first sum is equivalent to the average size of the resident set while the program is being run and the second clause can be approximated as the product of the total number of faults, (K), the average time required (D) to fetch a page into memory and the average resident set size. It can seen that:

$$C_B = C_D + \Delta \tag{8.3}$$

And:

$$\Delta = \sum_{j=1}^{P} R_j s(t_j) \tag{8.4}$$

Where R_j is the time (other than that needed for paging) the process pauses on the j^{th} process halt, which might be caused by a page fault, or by some other reason, eg., pre-emption to allow other processes to run or to execute operating system code out with of a process context. Here $s(t_j)$ is the resident set size on the j^{th}pre-emption.

To simplify this for illustrative purposes we will imagine a machine where there are only two processes, a and b, running. Both have the same, constant, fault rate and the same, constant, transport time. Process a runs for a time ι until it needs to fetch a page from secondary storage, when it stops, waiting for the input-output process to complete (taking time D) and process b then runs for time slice ι, which is less than D, before itself waiting for D seconds for its missing page to be fetched into memory. Once D for process a is over and the missing page is fetched into memory, process a runs for a time slice ι and so on. In this simple case, where $D > \iota$ and so $R = 0$, it can be seen that $\Delta = 0$ and $C_B \equiv C_D$. But, if, after every fault, there was also a period of operating system house keeping, γ, required, then we would have to restate (8.2) to preserve this equality:

$$C_\gamma = \sum_{t=1}^{T} s(t) + D \cdot \sum_{i=1}^{K} s(t_i) + \gamma \sum_{i=1}^{K} s(t_i) \tag{8.5}$$

Our argument is that this is indeed a better statement of the space-time product than (8.2) and that γ is much higher for a working set implementation than for a

global LRU policy. This practical difference is what makes global LRU policies seem the better choice for operating system implementations.

In this sense our experiments can be seen as an attempt to reduce K, by increasing γ, the operating system's house keeping time, but have resulted in increases in both K and γ.

8.3. **Where a local policy could work.** Linux's 2Q policy offers protection from the problem of pages with infinite or very long reuse distances being held in memory at the expense of those pages which are frequently accessed but have been used recently. But our results also show that 2Q has real risk of holding on to pages which were once frequently accessed but which now have a very long reuse distance as a result of a phase change.

A possible area of future research and experimentation would be to find an algorithm, perhaps one that is statistically based (eg., if each phase of locality does have a different size of working set then phase changes could be estimated by measuring changes in working set size), or is based on a better understanding of reference strings, to spot phase changes and act appropriately.

8.4. **A wider range of tests should be used.** Using the GCC C Compiler to compile the Linux kernel was an effective mechanism for measuring turnaround times: the task was substantial but still measurable (the longest measured time being approximately 16 hours and 40 minutes), as well as relatively easy to set up and requiring the minimum of human interaction. But our results in Sections 2 and 3 also suggest that the compiler may be atypical in its patterns of memory access.

Even when concentrating on just the bottom one per cent of virtual memory space, the compiler confines its accesses to a relatively small range of memory addresses and shows long phases of locality: suggesting that the compiler was likely to be CPU-bound (compare Figure 8.1 to Figure 3.3 for the Mozilla Firefox browser, perhaps the tool that is used most heavily by typical users of desktop machines.) The failure of the patches to deliver improved performance for the GCC C Compiler is enough to show they do not work, but, had they succeeded for the compiler, a wider range of tests against a more typical mix of loads would have been required before it could be stated such an approach was likely to have general benefits.

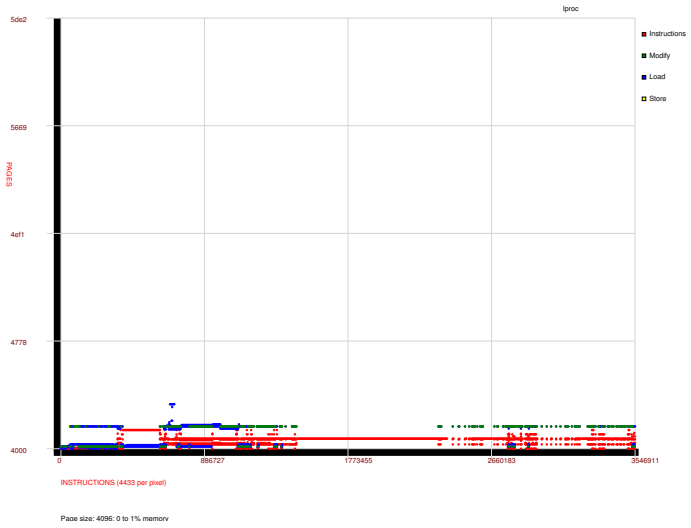

FIGURE 8.1. GCC C Compiler memory accesses in lowest 1% of virtual memory space

Appendix A. Lackey_xml and related tools to analyse Valgrind output

Valgrind's (Nethercote & Seward, 2007) Lackey sub-program is provided as foundation tool for building other Valgrind tools[59] and it has a minimal level of functionality: "Lackey is a simple Valgrind tool that does various kinds of basic program measurement." One of those basic measurements is to record, when the `-trace-mem=yes` option is selected, every access to memory made by a program running under Lackey's control.

The output format is described in Lackey's `lk_main.c` file:

LISTING 5. Valgrind Lackey output explained

```
55 // Specific Details about --trace-mem=yes
   // ----------------------------------------
   // Lackey's --trace-mem code is a good starting point for building Valgrind
   // tools that act on memory loads and stores.  It also could be used as is,
   // with its output used as input to a post-mortem processing step.  However,
60 // because memory traces can be very large, online analysis is generally
   // better.
   //
   // It prints memory data access traces that look like this:
   //
65 //   I  0023C790,2  # instruction read at 0x0023C790 of size 2
   //   I  0023C792,5
   //    S BE80199C,4  # data store at 0xBE80199C of size 4
   //   I  0025242B,3
   //    L BE801950,4  # data load at 0xBE801950 of size 4
70 //   I  0023D476,7
   //    M 0025747C,1  # data modify at 0x0025747C of size 1
   //   I  0023DC20,2
   //    L 00254962,1
   //    L BE801FB3,1
75 //   I  00252305,1
   //    L 00254AEB,1
   //    S 00257998,1
   //
   // Every instruction executed has an "instr" event representing it.
```

[59] Lackey's on line manual may be found at `http://valgrind.org/docs/manual/lk-manual.html` - accessed 18 August 2011

```
// Instructions that do memory accesses are followed by one or more "load",
// "store" or "modify" events.  Some instructions do more than one load or
// store, as in the last two examples in the above trace.
//
// Here are some examples of x86 instructions that do different combinations
// of loads, stores, and modifies.
//
//    Instruction           Memory accesses                  Event sequence
//    -----------           ---------------                  --------------
//    add %eax, %ebx        No loads or stores               instr
//
//    movl (%eax), %ebx     loads (%eax)                     instr, load
//
//    movl %eax, (%ebx)     stores (%ebx)                    instr, store
//
//    incl (%ecx)           modifies (%ecx)                  instr, modify
//
//    cmpsb                 loads (%esi), loads(%edi)        instr, load, load
//
//    call*l (%edx)         loads (%edx), stores -4(%esp)    instr, load, store
//    pushl (%edx)          loads (%edx), stores -4(%esp)    instr, load, store
//    movsw                 loads (%esi), stores (%edi)      instr, load, store
//
// Instructions using x86 "rep" prefixes are traced as if they are repeated
// N times.
//
// Lackey with --trace-mem gives good traces, but they are not perfect, for
// the following reasons:
//
// - It does not trace into the OS kernel, so system calls and other kernel
//   operations (eg. some scheduling and signal handling code) are ignored.
//
// - It could model loads and stores done at the system call boundary using
//   the pre_mem_read/post_mem_write events.  For example, if you call
//   fstat() you know that the passed in buffer has been written.  But it
//   currently does not do this.
//
// - Valgrind replaces some code (not much) with its own, notably parts of
//   code for scheduling operations and signal handling.  This code is not
//   traced.
//
// - There is no consideration of virtual-to-physical address mapping.
//   This may not matter for many purposes.
//
// - Valgrind modifies the instruction stream in some very minor ways.  For
//   example, on x86 the bts, btc, btr instructions are incorrectly
//   considered to always touch memory (this is a consequence of these
//   instructions being very difficult to simulate).
//
// - Valgrind tools layout memory differently to normal programs, so the
//   addresses you get will not be typical.  Thus Lackey (and all Valgrind
//   tools) is suitable for getting relative memory traces -- eg. if you
//   want to analyse locality of memory accesses -- but is not good if
//   absolute addresses are important.
//
// Despite all these warnings, Lackey's results should be good enough for a
// wide range of purposes.  For example, Cachegrind shares all the above
// shortcomings and it is still useful.
//
// For further inspiration, you should look at cachegrind/cg_main.c which
// uses the same basic technique for tracing memory accesses, but also groups
// events together for processing into twos and threes so that fewer C calls
// are made and things run faster.
```

The first program in our suite transforms the raw text output from Lackey into an extensible markup language (XML)[60] format, which we called lackeyml. The document type definition for lackeyml is shown below:

[60] See `http://www.w3.org/XML/` for more details about XML - accessed 19 August 2011

```
<!DOCTYPE lackeyml [
<!ELEMENT lackeyml (application,(instruction|store|load|modify)*)>
<!ATTLIST lackeyml version CDATA #FIXED "0.1">
<!ATTLIST lackeyml xmlns CDATA #FIXED "http://cartesianproduct.wordpress.com">
<!ELEMENT application EMPTY>
<!ATTLIST application command CDATA #REQUIRED>
<!ELEMENT instruction EMPTY>
<!ATTLIST instruction address CDATA #REQUIRED>
<!ATTLIST instruction size CDATA #REQUIRED>
<!ELEMENT modify EMPTY>
<!ATTLIST modify address CDATA #REQUIRED>
<!ATTLIST modify size CDATA #REQUIRED>
<!ELEMENT store EMPTY>
<!ATTLIST store address CDATA #REQUIRED>
<!ATTLIST store size CDATA #REQUIRED>
<!ELEMENT load EMPTY>
<!ATTLIST load address CDATA #REQUIRED>
<!ATTLIST load size CDATA #REQUIRED>
]>
```

XML was chosen as the format because it is both human-readable and widely supported with programing tools and libraries, meaning that once a lackeyml file had been created, any future researcher would likely have the tools needed to examine its contents. The URI chosen as the XML name space (xmlns) is the author's blog and has no other significance.

The program to convert the raw Valgrind/Lackey output into lackeyml was written in Groovy (Koenig *et al.* , 2007), chosen for its support for XML and its suitability for rapid development and for scripting. The conversion program (lackey_xml) could be contained in one Groovy file[61]:

LISTING 6. Groovy code to convert Lackey output to lackeyml

```
/**
 *
 * @author Adrian McMenamin, copyright 2011
 */
class lackeyXmlFile {

        /**
         * Will automatically generate an output file name
         *
         * @param inFile  name or path of raw Valgrind Lackey output
         */
        lackeyXmlFile(String inFile) {
                def dateStr = new Date().time.toString()
                processXml(inFile, "proc_${inFile}_${dateStr}.xml")
        }
        /**
         *
         * @param inFile name or path of raw Valgrind Lackey output
         * @param outFile name or path of output lackeyml file
         */
        lackeyXmlFile(String inFile, String outFile) {
                processXml(inFile, outFile)
        }
        /**
         * Writes DTD for lackeyml file and manages processing of raw file -
```

[61]The development history is publicly available through `https://github.com/mcmenaminadrian/lackey_xml` - accessed 19 August 2011

```
         * calling the appropriate write method on each line in turn
         * @param iFile name or path of the input file
         * @param oFile name or path of the output file
         */
30       void processXml(String iFile, String oFile)
         {
                 println "Reading $iFile Writing $oFile"
                 def inFile = new File(iFile)
                 def outFile = new File(oFile)
                 def writer = new FileWriter(outFile)
                 writer.write("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n")
                 writer.write("<!DOCTYPE lackeyml [\n")
                 def elStr = new String("<!ELEMENT lackeyml (application,")
                 elStr += "(instruction|store|load|modify)*)>\n"
40               writer.write(elStr)
                 writer.write("<!ATTLIST lackeyml version CDATA #FIXED \"0.1\">")
                 def attStr = new String("<!ATTLIST lackeyml xmlns CDATA #FIXED")
                 attStr += " \"http://cartesianproduct.wordpress.com\">"
                 writer.write(attStr)
                 writer.write("<!ELEMENT application EMPTY>\n")
                 writer.write("<!ATTLIST application command CDATA #REQUIRED>\n")
                 writer.write("<!ELEMENT instruction EMPTY>\n")
                 writer.write("<!ATTLIST instruction address CDATA #REQUIRED>\n")
                 writer.write("<!ATTLIST instruction size CDATA #REQUIRED>\n")
50               writer.write("<!ELEMENT modify EMPTY>\n")
                 writer.write("<!ATTLIST modify address CDATA #REQUIRED>\n")
                 writer.write("<!ATTLIST modify size CDATA #REQUIRED>\n")
                 writer.write("<!ELEMENT store EMPTY>\n")
                 writer.write("<!ATTLIST store address CDATA #REQUIRED>\n")
                 writer.write("<!ATTLIST store size CDATA #REQUIRED>\n")
                 writer.write("<!ELEMENT load EMPTY>\n")
                 writer.write("<!ATTLIST load address CDATA #REQUIRED>\n")
                 writer.write("<!ATTLIST load size CDATA #REQUIRED>\n")
                 writer.write("]>\n")
60               writer.write(
                          "<lackeyml xmlns=\"http://cartesianproduct.wordpress.com\">\n")
                 inFile.eachLine { line->
                          if (line =~/Command:/) {
                                   writer.write("<application command=\"")
                                   def cmdStr = line =~ /(\w)+$/
                                   writer.write("${cmdStr[0][0]}\"/>\n")
                          }
                          if (!(line =~ /^==/)) {
                                   if (line =~ /^I/)
70                                         writeInstruction(line, writer)
                                   else if (line =~ /^ S/)
                                            writeStore(line, writer)
                                   else if (line =~ /^ L/)
                                            writeLoad(line, writer)
                                   else if (line =~ /^ M/)
                                            writeModify(line, writer)
                                   else
                                            println("could not process $line")
                          }
80               }
                 writer.write("</lackeyml>\n\n")
                 writer.close()
         }
         /**
          * Called by write methods:
          * Writes out address and size attributes and closes xml element
          * @param line
          * @param writer
          */
90       void writeAddressSize(String line, FileWriter writer)
         {
                 def aStr = line =~/(\w*),(\w*)$/
                 if (aStr && aStr[0].size() >= 3)
                          writer.write(
                                   " address=\"0x${aStr[0][1]}\" size=\"0x${aStr[0][2]}\"")
                 writer.write("/>\n")
         }

         /**
100       * Called by processXML: opens an instruction xml element
          * @param line
          * @param writer
          */
```

```
        void writeInstruction(String line, FileWriter writer)
        {
                writer.write("<instruction ")
                writeAddressSize(line, writer)
        }

        /**
         * Called by processXML: opens a store xml element
         * @param line
         * @param writer
         */
        void writeStore(String line, FileWriter writer)
        {
                writer.write("<store ")
                writeAddressSize(line, writer)
        }

        /**
         * Called by processXML: opens a load xml element
         * @param line
         * @param writer
         */
        void writeLoad(String line, FileWriter writer)
        {
                writer.write("<load ")
                writeAddressSize(line, writer)
        }

        /**
         * Called by processXML: opens a modify xml element
         * @param line
         * @param writer
         */
        void writeModify(String line, FileWriter writer)
        {
                writer.write("<modify ")
                writeAddressSize(line, writer)
        }

}

def lackeyIn
if (args.size() == 0)
        println "Have to specify input file"
else if (args.size() == 1)
        lackeyIn = new lackeyXmlFile(args[0])
else
        lackeyIn = new lackeyXmlFile(args[0], args[1])
```

The resulting lackeyml files can then be processed by another Groovy program, lackeySVG.

This program, lackeySVG[62], will plot several different types of graph, based on the lackeyml input. The lackeyml is parsed using a "Simple API for XML" (SAX)[63] parser, a widely-used event-driven parser for XML. Graphs that can be output include both the memory reference maps shown above (eg., in Figure 2.1) and a modeled lifetime curve (Denning, 1980) for the executing program using both a working set and a least recently used model (examples of these graphs are shown above, eg., in Figure 4.2).

The graphs are plotted as scalable vector graphics (SVG)[64], a widely supported XML-based vector graphic format which is lightweight, preserves some of the calculated information in a human readable form and is easily manipulated through software tools, including via extensible stylesheet language transformations (XSLT)

[62]The development history is publicly available through `https://github.com/mcmenaminadrian/lackeySVG` - accessed 19 August 2011

[63]See `http://www.saxproject.org/` - accessed 19 August 2011

[64]See `http://www.w3.org/Graphics/SVG/` - accessed 19 August 2011

stylesheets[65], a standard and widely-supported means of extracting information from and transforming XML documents. To illustrate this, we have used the XSLT stylesheet shown in below to transform the memory maps generated by the MySQL daemon to show (Figure A.1), clockwise from top left, the instructions, the memory modifications, the memory stores and the memory loads[66]: the stylesheet uses XSL version 2, as opposed to the more widely supported version 1, to ease handling of the namespace declared for the SVG file.

```
<?xml version="1.0"?>
<xsl:stylesheet version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xpath-default-namespace="http://www.w3.org/2000/svg">
<xsl:param name="colour">Black</xsl:param>
<xsl:template match="/">
<xsl:apply-templates select="svg"/>
</xsl:template>
<xsl:template match="svg">
<xsl:copy>
<xsl:for-each select="@*">
<xsl:copy/>
</xsl:for-each>
<xsl:text>&#10;</xsl:text>
<xsl:apply-templates select="rect"/>
<xsl:apply-templates select="line"/>
<xsl:apply-templates select="text"/>
<xsl:apply-templates select="circle"/>
</xsl:copy>
</xsl:template>
<xsl:template match="line">
<xsl:copy>
<xsl:for-each select="@*">
<xsl:copy/>
</xsl:for-each>
</xsl:copy>
<xsl:text>&#10;</xsl:text>
</xsl:template>
<xsl:template match="rect">
<xsl:copy>
<xsl:for-each select="@*">
<xsl:copy/>
</xsl:for-each>
</xsl:copy>
```

[65] See `http://www.w3.org/TR/xslt` - accessed 19 August 2011

[66] The Saxonb-xslt command line processor was used to generate these files - see `http://saxon.sourceforge.net/` - accessed 22 August 2011

```
<xsl:text>&#10;</xsl:text>
</xsl:template>
<xsl:template match="text">
<xsl:copy>
<xsl:for-each select="@*|node()">
<xsl:copy/>
</xsl:for-each>
</xsl:copy>
<xsl:text>&#10;</xsl:text>
</xsl:template>
<xsl:template match="circle">
<xsl:if test="@stroke=$colour">
<xsl:copy>
<xsl:for-each select="@*">
<xsl:copy/>
</xsl:for-each>
</xsl:copy>
<xsl:text>&#10;</xsl:text>
</xsl:if>
</xsl:template>
</xsl:stylesheet>
```

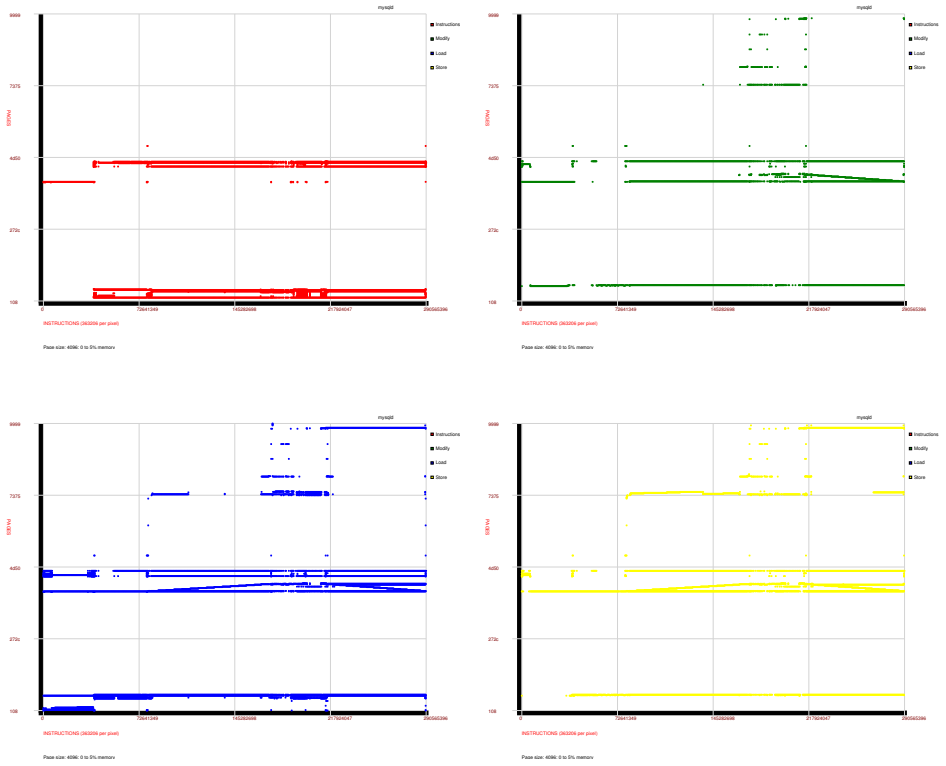

FIGURE A.1. Different types of memory access for the MySQL daemon

The core Groovy code for the lackeySVG program is shown below:

LISTING 7. Groovy code to handle lackeyml files

```
import javax.xml.parsers.SAXParserFactory
import org.xml.sax.helpers.DefaultHandler
import org.xml.sax.*
import java.util.concurrent.*


/**
 *
 * @author Adrian McMenamin, copyright 2011
 *
 */
class LackeySVGraph {

        def MEMPLOT = 0x01
        def WSPLOT = 0x02
        def LIFEPLOT = 0x04
        def LRUPLOT = 0x08

        /**
         * Build various graphs from a lackeyml file
         *
         * @param width width of the graph in pixels
         * @param height height of the graph in pixels
         * @param inst graph instructions
         * @param fPath path to the lackeyml file being processed
         * @param verb verbose output
         * @param oF name of reference map output file
         * @param percentile percentile of memory to form base of reference map
         * @param range range of memory to be examined in reference map
         * @param pageSize bit shift used for pages (eg 12 for 4096KB pages)
         * @param gridMarks number of grid marks to be used on graphs
         * @param workingSetInst number of instructions
         *                       against which to plot working set
         * @param threads size of thread pool
         * @param boost size of margin on graphs
         * @param PLOTS bitmask for graphs to be drawn
         */
        LackeySVGraph(def width, def height, def inst, def fPath, def verb,
        def oF, def percentile, def range, def pageSize, def gridMarks,
        def workingSetInst, def threads, def boost, def PLOTS) {
                def thetaLRUMap
                def thetaMap
                def thetaAveMap
                def thetaLRUAveMap
                println "Opening $fPath"
                def handler = new FirstPassHandler(verb, pageSize)
                def reader = SAXParserFactory.newInstance().newSAXParser().XMLReader
                reader.setContentHandler(handler)
                reader.parse(new InputSource(new FileInputStream(fPath)))

                println "First pass completed"
                println "Instruction range is:"
                println "${handler.minInstructionAddr} - ${handler.maxInstructionAddr}"
                println "Instruction count is ${handler.totalInstructions}"
                println "Memory range is:"
                println "${handler.minHeapAddr} - ${handler.maxHeapAddr}"
                println "Biggest access is ${handler.maxSize}"
                if (PLOTS & MEMPLOT)
                        println "Writing to $oF width: $width height: $height"
                if (inst) println "Recording instruction memory range"
                if (pageSize)
                        println "Using page size granularity of ${2**pageSize} bytes"
                if (percentile)
                        println "Starting from $percentile with range $range%"
                def maxPg = (handler.pageMap).size()
                println "Total pages referrenced $maxPg"
                def pool = Executors.newFixedThreadPool(threads)


                def memClosure = {
                        def handler2 = new SecondPassHandler(verb, handler, width, height,
                                        inst, oF, percentile, range, pageSize, gridMarks, boost)
                        def saxReader = SAXParserFactory.newInstance().
                                        newSAXParser().XMLReader
                        saxReader.setContentHandler(handler2)
```

```
        saxReader.parse(new InputSource(new FileInputStream(fPath)))
        println "Memory use mapping complete"
}

def wsClosure = {
        def handler3 = new ThirdPassHandler(verb, handler, workingSetInst,
                        width, height, gridMarks, boost)
        def saxReader = SAXParserFactory.newInstance().
                        newSAXParser().XMLReader
        saxReader.setContentHandler(handler3)
        saxReader.parse(new InputSource(new FileInputStream(fPath)))
        maxWS = handler3.maxWS
        println "Working set mapping complete"
}

if (PLOTS & MEMPLOT)
        pool.submit(memClosure as Callable)
if (PLOTS & WSPLOT)
        pool.submit(wsClosure as Callable)

if (PLOTS & LIFEPLOT) {
        println "Plotting life with variable WSS"
        thetaMap = Collections.synchronizedSortedMap(new TreeMap())
        thetaAveMap = Collections.synchronizedSortedMap(new TreeMap())
        def stepTheta = (int) handler.totalInstructions/width
        (stepTheta .. handler.totalInstructions).step(stepTheta){
                def steps = it
                Closure passWS = {
                        if (verb)
                                println "Setting theta to $steps"
                        def handler4 = new FourthPassHandler(handler, steps,
                                        12)
                        def saxReader =
                                        SAXParserFactory.newInstance().
                                                newSAXParser().XMLReader
                        saxReader.setContentHandler(handler4)
                        saxReader.parse(
                                        new InputSource(new FileInputStream(fPath)))
                        def g = (int)(handler.totalInstructions /
                                        handler4.faults)
                        thetaMap[steps] = g
                        thetaAveMap[handler4.aveSize] = g
                        println "Ave. working set ${handler4.aveSize}"
                }
                pool.submit(passWS as Callable)
        }
}

pool.shutdown()
pool.awaitTermination 5, TimeUnit.DAYS


def pool2 = Executors.newFixedThreadPool(threads)

if (PLOTS & LRUPLOT) {
        thetaLRUMap = Collections.synchronizedSortedMap(new TreeMap())
        thetaLRUAveMap = Collections.synchronizedSortedMap(new TreeMap())
        def memTheta = (int) maxPg/width
        if (memTheta == 0)
                memTheta = 1
        (memTheta .. maxPg).step(memTheta){
                def mem = it
                Closure passLRU = {
                        if (verb)
                                println "Setting LRU theta to $mem"
                        def handler5 = new FifthPassHandler(handler, mem,
                                        12)
                        def saxLRUReader = SAXParserFactory.newInstance().
                                        newSAXParser().XMLReader
                        saxLRUReader.setContentHandler(handler5)
                        saxLRUReader.parse(
                                        new InputSource(new FileInputStream(fPath)))
                        def g = (int)(handler.totalInstructions /
                                        handler5.faults)
                        thetaLRUMap[mem] = g
                        thetaLRUAveMap[handler5.aveSize] = g
                }
                pool2.submit(passLRU as Callable)
```

```
                        }
                }

                pool2.shutdown()
                pool2.awaitTermination 5, TimeUnit.DAYS

                if (PLOTS & LIFEPLOT)
                        def graphTheta = new GraphTheta(thetaMap, width, height,
                                        gridMarks, boost)
                if (PLOTS & LRUPLOT)
                        def graphLRUTheta = new GraphLRUTheta(thetaLRUMap, width, height,
                                        gridMarks, boost)
                if ((PLOTS & LRUPLOT) && (PLOTS & LIFEPLOT))
                        def graphCompTheta = new GraphCompTheta(thetaAveMap,
                                thetaLRUAveMap, width, height, gridMarks, boost)
        }
}


def svgCli = new CliBuilder
                (usage: 'lackeySVG [options] <lackeyml file>')
svgCli.w(longOpt:'width', args: 1,
                'width of SVG ouput - default 800')
svgCli.h(longOpt:'height', args: 1,
                'height of SVG output - default 600')
svgCli.i(longOpt: 'instructions', 'graph instructions - default false')
svgCli.u(longOpt: 'usage', 'prints this information')
svgCli.v(longOpt: 'verbose', 'prints verbose information - default false')
svgCli.p(longOpt: 'percentile', args:1, 'lowest percentile to graph')
svgCli.r(longOpt: 'range', args:1, '(percentile) default is 10')
svgCli.g(longOpt: 'pageshift', args:1, 'page size in power of 2 - 4KB = 12')
svgCli.m(longOpt: 'gridmarks', args: 1, 'grid marks on graph - default 4')
svgCli.s(longOpt: 'workingset', args: 1, 'instructions per working set')
svgCli.t(longOpt: 'threadpool', args: 1, 'size of thread pool (default 3)')
svgCli.b(longOpt: 'margins', args: 1, 'margin size on graphs (default 100px)')
svgCli.xm(longOpt: 'nomemplot', 'do not plot memory use')
svgCli.xw(longOpt: 'nowsplot', 'do not plot working set')
svgCli.xl(longOpt: 'nolifeplot', 'do not plot ws life time curve')
svgCli.xr(longOpt: 'nolruplot', 'do not plot lru life time curve')

def oAss = svgCli.parse(args)
if (oAss.u || args.size() == 0) {
        svgCli.usage()
}
else {

        def PLOTS = 0xFF
        def width = 800
        def height = 600
        def percentile = 0
        def range = 1
        def pageSize = 0
        def inst = false
        def verb = false
        def gridMarks = 4
        def wSSize = 100000
        def oFile = "${new Date().time.toString()}.svg"
        def threads = 3
        def boost = 100
        if (oAss.w)
                width = Integer.parseInt(oAss.w)
        if (oAss.h)
                height = Integer.parseInt(oAss.h)
        if (oAss.i)
                inst = true
        if (oAss.v)
                verb = true
        if (oAss.t) {
                threads = Integer.parseInt(oAss.t)
                if (threads < 1)
                        threads = 3
        }
        if (oAss.b) {
                boost = Integer.parseInt(oAss.b)
                if (boost < 0)
                        boost = 100
        }
        if (oAss.p) {
```

```
                def tPer = Integer.parseInt(oAss.p)
                if (tPer > 0 && tPer <= 100)
                        percentile = tPer
                if (oAss.r) {
                        def tRange = Integer.parseInt(oAss.r)
                        if (tRange >= 1 && tRange <= (101 - percentile))
                                range = tRange
                }
        }
        if (oAss.m)
                gridMarks = Integer.parseInt(oAss.m)
        if (oAss.g)
                pageSize = Integer.parseInt(oAss.g)
        if (oAss.s)
                wSSize = Integer.parseInt(oAss.s)

        if (oAss.xm)
                PLOTS = PLOTS ^ 0x01
        if (oAss.xw)
                PLOTS = PLOTS ^ 0x02
        if (oAss.xl)
                PLOTS = PLOTS ^ 0x04
        if (oAss.xr)
                PLOTS = PLOTS ^ 0x08

        def lSVG = new LackeySVGraph(width, height, inst, args[args.size() - 1],
                        verb, oFile, percentile, range, pageSize, gridMarks, wSSize,
                        threads, boost, PLOTS)
}
```

The code is multi-threaded: due to the size of the lackeyml files (500MB is a typical size for a program that runs for less than a minute of virtual time on a fast machine, while files for the MySQL daemon ran to over 5GB) and the large number of computations required to compute a lifetime function for the program using both the working set and LRU models, typical calculations run for weeks in virtual time and days in wall clock time, even on fast machines with several threads. Our experience was that, while relative processor utilisation would fall as more threads were used, with 10 or more threads we saw a sudden and complete collapse in computational efficiency. As the hosting computer had 12 CPUs and 25GB of memory, this seems likely to be a function of the Java Virtual Machine (JVM) rather than a limit imposed by the available hardware: perhaps as a result of internal memory fragmentation. This phenomenon was seen with both the open source JVM and the proprietary Sun/Oracle JVM that were available with the Debian distribution. We cannot comment further on this behaviour as we did not conduct any further investigations, simply limiting the maximum thread pool used to 8 in most cases.

Appendix B. Valext and mapWSS

Valext[67] - the title reflects that it was originally conceived as an extension to Valgrind - uses the Linux ptrace mechanism (Padala, 2002)along with its `fork()` mechanism (Love, 2010, p. 32) to launch a process under the control of Valext:

LISTING 8. main function from Valext

```
int main(int argc, char* argv[])
{
        FILE* outXML;
        char filename[MEMBLOCK];
        if (argc < 2)
                return 0; /* must supply a file to execute */
        srand(time(NULL));
        pid_t forker = fork();
        if (forker == 0) {
                //in the child process
                if (argc == 3) {
                        ptrace(PTRACE_TRACEME, 0, 0, 0);
                        execlp(argv[1], argv[2], NULL);
                } else {
                        ptrace(PTRACE_TRACEME, 0, 0, 0);
                        execvp(argv[1], NULL);
                }
                return 0;
        }
        //in the original process
        if (forker < 0) {
                printf("Could not get %s to run\n", argv[1]);
                return 0;
        }
        /* Open XML file */
        sprintf(filename, "XMLtrace%d_%d.xml", forker, rand());
        outXML = fopen(filename, "a");
        if (!outXML) {
                printf("Could not open %s\n", filename);
                return 0;
        }
        fputs("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n", outXML);
        fputs("<!DOCTYPE ptracexml [\n", outXML);
        fputs("<!ELEMENT ptracexml (trace)*>\n", outXML);
        fputs("<!ELEMENT trace EMPTY>\n", outXML);
        fputs("<!ATTLIST trace step CDATA #REQUIRED>\n", outXML);
        fputs("<!ATTLIST trace present CDATA #REQUIRED>\n", outXML);
        fputs("<!ATTLIST trace swapped CDATA #REQUIRED>\n", outXML);
        fputs("<!ATTLIST trace presonly CDATA #REQUIRED>\n", outXML);
        fputs("]>\n", outXML);
        fputs("<ptracexml>\n", outXML);
        getWSS(forker, outXML, CHAINSIZE);
        fputs("</ptracexml>\n", outXML);
        fclose(outXML);
        return 1;
}
```

After the fork the child process marks itself with `PTRACE_TRACEME` before calling `exec` while the parent process goes on to write out the header for an XML file and then calls the `getWSS` function to step through the child process's execution:

LISTING 9. trace the child process

```
/* run the child */
void getWSS(pid_t forked, FILE* xmlout, int size)
{
        int i = 0, status;
        struct blockchain *header = newchain(size);
        /*create a string representation of pid */
        char pid[MEMBLOCK];
        sprintf(pid, "%u", forked);
        /* loop while signalling child */
        while(1)
        {
```

[67] Valext's development history can be publicly traced at `https://github.com/mcmenaminadrian/valext` - accessed 21 August 2011

```
                wait(&status);
                ptrace(PTRACE_SINGLESTEP, forked, 0, 0);
                if (WIFEXITED(status))
                        break;
                getblocks(pid, header, size);
                if (header->head[0])
                        getblockstatus(pid, header, xmlout, i++, size);
        }
        cleanchain(header);
}
```

On each step the program interrogates the /proc/*pid*/maps to find which memory ranges are present:

LISTING 10. interrogating /proc/pid/maps

```
/* query /proc filesystem */
void getblocks(char* pid, struct blockchain* header, int size)
{
        FILE *ret;
        int t = 0;
        char buf[MEMBLOCK];
        /* open /proc/pid/maps */
        char st1[MEMBLOCK] = "/proc/";
        strcat(st1, pid);
        strcat(st1, "/maps");

        ret = fopen(st1, "r");
        if (ret == NULL) {
                printf("Could not open %s\n", st1);
                goto ret;
        }
        while (!feof(ret)){
                fgets(buf, MEMBLOCK, ret);
                if (!getnextblock(header, buf, size, &t)) {
                        goto close;
                }
        }
close:
        fclose(ret);
ret:
        return;
}
```

The addresses of the blocks which are marked as present are found through a regular expression query and then stored in arrays chained through a linked list - the arrays are only allocated once and use a guard (or sentinel) to mark the end of current series of values, a method suggested in (Bentley, 2000, p. 90):

LISTING 11. finding which blocks are present

```
/* set up a list */
int getnextblock(struct blockchain *header, char *buf, int size, int *t)
{
        int match;
        uint64_t startaddr;
        uint64_t endaddr;
        uint64_t i;
        struct blockchain* chain = header;
        const char* pattern;
        int retval = 0;
        regex_t reg;
        regmatch_t addresses[3];

        pattern = "^([0-9a-f]+)-([0-9a-f]+)";
        if (regcomp(&reg, pattern, REG_EXTENDED) != 0)
                goto ret;
        match = regexec(&reg, buf, (size_t)3, addresses, 0);
        if (match == REG_NOMATCH || match == REG_ESPACE)
                goto cleanup;
        startaddr = strtoul(&buf[addresses[1].rm_so], NULL, 16) >> PAGESHIFT;
        endaddr = strtoul(&buf[addresses[2].rm_so], NULL, 16) >> PAGESHIFT;
        for (i = startaddr; i < endaddr; i++)
        {
                chain->head[*t]  = i;
```

```
                (*t)++;
                if (*t == size) {
                        if (chain->tail == 0) {
                                struct blockchain *nxtchain =
                                        newchain(size);
                                if (!nxtchain)
                                        goto cleanup;
                                chain->tail = nxtchain;
                        }
                        chain = chain->tail;
                        *t = 0;
                }
                chain->head[*t] = 0; //guard
        }
        retval = 1;

cleanup:
        regfree(&reg);
ret:
        return retval;
}
```

The DTD for the XML file produced is shown below:

```
<!DOCTYPE ptracexml [
<!ELEMENT ptracexml (trace)*>
<!ELEMENT trace EMPTY>
<!ATTLIST trace step CDATA #REQUIRED>
<!ATTLIST trace present CDATA #REQUIRED>
<!ATTLIST trace swapped CDATA #REQUIRED>
<!ATTLIST trace presonly CDATA #REQUIRED>
]>
```

This ptracexml file can then be parsed by the Groovy script mapWSS, which will output a graph - the core `mapWSS.groovy` code is shown below:

LISTING 12. Groovy code to graph real memory use

```
import org.xml.sax.Attributes;
import javax.xml.parsers.SAXParserFactory
import org.xml.sax.helpers.DefaultHandler
import org.xml.sax.*


class GraphWSS {

        GraphWSS(def iFile, width, height, marks, margins)
        {
                println("Beginning␣first␣pass")
                def handler = new PtraceFirstHandler()
                def reader =
                        SAXParserFactory.newInstance().newSAXParser().XMLReader
                reader.setContentHandler(handler)
                reader.parse(new InputSource(new FileInputStream(iFile)))

                def maxSteps = handler.maxSteps
                def maxPagesP = handler.maxPagesP
                def maxPagesS = handler.maxPagesS
                def maxPagesM = handler.maxPagesM
                def sampleRate = (int) maxSteps/width
                println("Beginning␣second␣pass")
                def handler2 = new PtraceSecondHandler(sampleRate)
                reader.setContentHandler(handler2)
                reader.parse(new InputSource(new FileInputStream(iFile)))

                def listP = handler2.wssListP
                def listS = handler2.wssListS
                def listM = handler2.wssListM
                println("Drawing␣graph")
                new PtraceDrawWSSGraph(listP, listS, listM, width, height,
                        marks, margins, maxSteps, maxPagesP,
```

```
                        maxPagesS, maxPagesM);
        }

}


def ptraceCli = new CliBuilder
        (usage: 'mapWSS [options] <ptracexml file>')
ptraceCli.w(longOpt:'width', args: 1,
'width of SVG ouput - default 800')
ptraceCli.h(longOpt:'height', args: 1,
 'height of SVG output - default 600')
ptraceCli.u(longOpt: 'usage', 'prints this information')
ptraceCli.m(longOpt: 'gridmarks', args: 1, 'grid marks on graph - default 4')
ptraceCli.b(longOpt: 'margins', args: 1,
        'margin size on graphs (default 100px)')

def width = 800
def height = 600
def marks = 4
def margins = 100

def pAss = ptraceCli.parse(args)
if (pAss.u || args.size() == 0) {
        ptraceCli.usage()
} else {
        if (pAss.w)
                width = Integer.parseInt(pAss.w)
        if (pAss.h)
                height = Integer.parseInt(pAss.h)
        if (pAss.m)
                marks = Integer.parseInt(pAss.m)
        if (pAss.b)
                margins = Integer.parseInt(pAss.b)

        def gWSS = new GraphWSS(args[args.size() -1], width, height, marks,
                margins)
}
```

Valext records both pages which are both present and mapped into the program's page tables and space that has been reserved in the virtual address space of the program but for which no page table entry exists. That space may be used to later map in pages from, for example, the backing file. The mapWSS program records the numbers of both types of pages, both those for which a page table entry exists (in blue) and those for which space has been reserved but no page table entry exists (in green): it can be seen in Figure 3.3 that the green and blue lines appear to be in close relation, rising together, or acting in opposition, but certainly not moving independently, suggesting pages for which space has been reserved are then being read in or that as a file is removed from the virtual address space, the empty but reserved pages also go.[68]

[68]The author's discussion of this on the Greater London Linux User Group's mailing list can be read starting from here - `http://lists.gllug.org.uk/pipermail/gllug/2011-August/088110.html` - accessed 1 September 2011.

Appendix C. Memball and related programs

Memball, and the related Treedraw and TreeQT are a small suite of programs that will display a red-black binary tree of processes running on a Linux machine and ordered by a user set parameter related to memory use or process time.

The principles of the red-black tree algorithm were first described by Rudolf Bayer in 1972 (Bayer, 1972). Using colours for nodes, the algorithm ensures that trees are self-, semi-balanced: all direct paths from the root (black) node to a leaf node must pass through the same number of black nodes. The algorithm is described in some detail in (Cormen *et al.* , 2009, chapter 13, pp. 308 - 338).

The self-balancing nature of the inserts and deletes on the tree mean that the deepest leaf can never be more than twice as deep as the shallowest, making the red-black tree algorithm a good choice for use in the Linux kernel (Love, 2010, pp. 105 - 108). For our purposes a balanced-tree algorithm was used to ensure a good visual result: a lopsided tree would be wasteful of display space. To ensure the tree was displayed in an aesthetically pleasing manner the *Reingold-Tilford algorithm* (Reingold & Tilford, 1981) was applied.

The three programs were written to reflect Doug McIlroy's dicta: "This is the Unix philosophy: Write programs that do one thing and do it well. Write programs that work together. Write programs to handle text streams, because that is a universal interface." (McIlroy, as quoted in Raymond, 2003, p. 12).

Memball reads data from the `proc` file system, and outputs data in one of three text formats - the XML format GraphML[69], for direct use in the LaTeXtext processing system (Goossens *et al.* , 2007, pp. 366 - 378) or a simple plain text format. Treedraw can read a GraphML file or from "standard in" and output a scalable vector graphic (SVG) file or as a text stream which can in turn be read by TreeQT, which uses the Qt framework[70] to produce an X Windows application that will display the red-black tree.

All three programs are written in C++ and at their core is a template-based implementation of the red-black tree algorithm as stubbed in the `redblack.hpp` header:

LISTING 13. Red-black tree classes

```
25  template <typename T>
    class redblacknode{

            template <typename Z> friend ostream& operator<<(ostream& os,
                    redblacknode<Z>* rbtp);
30          template <typename Z> friend void
                    streamrbt(ostream& os, redblacknode<Z>* node);
            template <typename Z> friend void
                    drawnextroot(redblacknode<Z>* node, int, ostream&);
            template <typename Z> friend void
35                  drawTEXtree(redblacknode<Z>* node, ostream&);
            template <typename Z> friend void
                    drawnextxml(redblacknode<Z>* node, int, int&, ostream&);
            template <typename Z> friend void
                    drawGraphMLtree(redblacknode<Z>* node, ostream&);
40
            private:
                    T value;
    #ifdef ADDITIONAL_INFO
                    const string additional_info() const;
45  #endif

            public:
```

[69] See `http://graphml.graphdrawing.org/`, accessed 6 August 2011

[70] See `http://qt.nokia.com/products/` accessed 6 August 2011

```
                int colour;
                redblacknode* up;
                redblacknode* left;
                redblacknode* right;
                redblacknode(const T& v);
                redblacknode(redblacknode* node);
                redblacknode(redblacknode& node);
                redblacknode* grandparent() const;
                redblacknode* uncle() const;
                redblacknode* sibling() const;
                bool bothchildrenblack() const;
                bool equals(redblacknode*) const;
                bool lessthan(redblacknode*) const;
                void assign(redblacknode*);
                void showinorder(redblacknode*) const;
                void showpreorder(redblacknode*) const;
                void showpostorder(redblacknode*) const;
};

template <typename NODE>
class redblacktree {
        private:
                void balanceinsert(NODE*);
                void rotate3(NODE*);
                void rotate2(NODE*);
                void rotate2a(NODE*);
                void rotate1(NODE*);
                void transform2(NODE*);
                void free(NODE*);
                NODE* maxleft(NODE*) const;
                NODE* minright(NODE*) const;
                NODE* locatenode(NODE*, NODE*) const;
                void countup(NODE*, int&) const;
        public:
                NODE* root;
                void insertnode(NODE*, NODE*);
                bool removenode(NODE&);
                bool find(NODE&) const;
                NODE* min() const;
                NODE* max() const;
                int count() const;
                redblacktree();
                ~redblacktree();
};
```

Treedraw re-implements Reingold and Tilford's Pascal for node positioning in C++:

LISTING 14. Reingold-Tilford algoithm in C++

```
void Tree::calcpoints(Node* n, int level, Extreme& lmost, Extreme& rmost)
{
        // algorithm from Reingold and Tilford
        // "Tidier Drawing of Trees"
        // IEEE Transactions on Software Engineering
        // Vol SE-7 No 2 March 1981
        Extreme rr, rl, lr, ll;
        int loffsum = 0;
        int roffsum = 0;
        n->yco = level;
        if (n->left != -1)
                calcpoints(items[n->left], level + 1, lr, ll);
        if (n->right != -1)
                calcpoints(items[n->right], level + 1, rr, rl);

        Node* left = NULL;
        Node* right = NULL;
        int ileft = n->left;
        if (ileft != -1)
                left = items[ileft];

        int iright = n->right;
        if (iright != -1)
                right = items[iright];

        if (iright == -1 && ileft == -1)
        {
```

```
                //leaf node most be most extreme
                lmost.n = n;
                lmost.offset = 0;
                lmost.level = level;
                rmost.n = n;
                rmost.offset = 0;
                rmost.level = level;
                n->offset = 0;
                return;
        }

        const int minsep = distance;
        int rootsep = minsep;
        int cursep = rootsep;

        while (left && right) {
                if (cursep < minsep) {
                        rootsep = rootsep + (minsep - cursep);
                        cursep = minsep;
                }

                if (left->right != -1) {
                        loffsum = loffsum + left->offset;
                        cursep = cursep - (left->offset + 1) / 2;
                        ileft = left->right;
                        left = items[ileft];
                }
                else
                {
                        loffsum = loffsum - left->offset;
                        cursep = cursep + (left->offset + 1) /2;
                        if (left->left != -1) {
                                ileft = left->left;
                                left = items[ileft];
                        }
                        else
                                left = NULL;
                }

                if (right->left != -1 ) {
                        roffsum = roffsum - right->offset;
                        cursep = cursep - (right->offset + 1) / 2;
                        iright = right->left;
                        right = items[iright];
                }
                else
                {
                        roffsum = roffsum + right->offset;
                        cursep = cursep + (right->offset + 1) / 2;
                        if (right->right != -1) {
                                iright = right->right;
                                right = items[iright];
                        }
                        else
                                right = NULL;
                }
        }

        n->offset = rootsep;
        loffsum = loffsum - (rootsep + 1)/2;
        roffsum = roffsum + (rootsep + 1)/2;

        //update extreme descendants details
        if (rl.level > ll.level || n->left == -1)
        {
                lmost = rl;
                lmost.offset = lmost.offset + (n->offset + 1) / 2;
        }
        else {
                lmost = ll;
                lmost.offset = lmost.offset - (n->offset + 1) / 2;
        }

        if (lr.level > rr.level || n->right == -1)
        {
                rmost = lr;
                rmost.offset = rmost.offset - (n->offset + 1) / 2;
        }
```

```
250         else {
                    rmost = rr;
                    rmost.offset = rmost.offset + (n->offset + 1) / 2;
            }

255         //threading
            if (left != NULL && left != items[n->left])
            {
                    rr.n->thread = true;
                    rr.n->offset = abs((rr.offset + n->offset) - loffsum);
260                 if (loffsum - n->offset <= rr.offset)
                            rr.n->left = ileft;
                    else
                            rr.n->right = ileft;
            }
265         else if (right != NULL && right != items[n->right])
            {
                    ll.n->thread = true;
                    ll.n->offset = abs((ll.offset - n->offset) - roffsum);
                    if (roffsum + n->offset >= ll.offset)
270                         ll.n->right = iright;
                    else
                            ll.n->left = iright;
            }
}
```

REFERENCES

Adler, Joseph. 2009. *R in a Nutshell: A Desktop Quick Reference*. 1st edn. O'Reilly Media, Inc.

Bayer, Rudolf. 1972. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, **1**, 290–306. 10.1007/BF00289509.

Belady, L. A. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, **5**(June), 78–101.

Belady, L. A., & Kuehner, C. J. 1969. Dynamic space-sharing in computer systems. *Commun. ACM*, **12**(May), 282–288.

Bentley, Jon. 2000. *Programming pearls (2nd ed.)*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Bovet, Daniel, & Cesati, Marco. 2005. *Understanding The Linux Kernel*. Oreilly & Associates Inc.

Bryant, Peter. 1975. Predicting working set sizes. *IBM J. Res. Dev.*, **19**(May), 221–229.

Chu, W.W., & Opderbeck, H. 1976. Program Behavior and the Page-Fault-Frequency Replacement Algorithm. *Computer*, **9**, 29–38.

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., & Stein, Clifford. 2009. *Introduction to Algorithms, Third Edition*. 3rd edn. The MIT Press.

Custer, Helen. 1993. *Inside Windows NT*. Suisun City, CA, USA: Microcomputer Applications.

Denning, Peter J. 1968a. Thrashing: its causes and prevention. *Pages 915–922 of: Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. AFIPS '68 (Fall, part I). New York, NY, USA: ACM.

Denning, Peter J. 1968b. The working set model for program behavior. *Commun. ACM*, **11**(May), 323–333.

Denning, Peter J. 1970. Virtual Memory. *ACM Comput. Surv.*, **2**(September), 153–189.

Denning, Peter J. 1980. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, **6**, 64–84.

Denning, Peter J., & Schwartz, Stuart C. 1972. Properties of the working-set model. *SIGOPS Oper. Syst. Rev.*, **6**(June), 130–140.

Du, Jiaqing, Sehrawat, Nipun, & Zwaenepoel, Willy. 2011. Performance profiling of virtual machines. *Pages 3–14 of: Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. VEE '11. New York, NY, USA: ACM.

Eeckhout, Lieven. 2010. *Computer Architecture Performance Evaluation Methods*. 1st edn. Morgan & Claypool Publishers.

Friedman, Mark B. 1999 (December). Windows NT Page Replacement Policies. *Pages 234–244 of: Proceedings of 25th International Computer Measurement Group Conference*.

Goldenberg, Ruth. 2002. *OpenVMS Alpha Internals and Data Structures: Memory Management*. Burlington, MA: Elsevier.

Goossens, Michel, Mittelbach, Frank, Rahtz, Sebastian, Roegel, Denis, & Voss, Herbert. 2007. *LaTeX Graphics Companion, The (2nd Edition) (Tools and Techniques for Computer Typesetting)*. Addison-Wesley Professional.

Gorman, Mel. 2004. *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Hornik, Kurt. 2011. *The R FAQ*. ISBN 3-900051-08-9.

Jiang, Song, & Zhang, Xiaodong. 2005. Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems. *Perform. Eval.*, **60**(May), 5–29.

Jiang, Song, Chen, Feng, & Zhang, Xiaodong. 2005. CLOCK-Pro: an effective improvement of the CLOCK replacement. *Pages 35–35 of: Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '05. Berkeley, CA, USA: USENIX Association.

Johnson, Theodore, & Shasha, Dennis. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. *Pages 439–450 of: Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Koenig, Dierk, Glover, Andrew, King, Paul, Laforge, Guillaume, & Skeet, Jon. 2007. *Groovy in Action*. Greenwich, CT, USA: Manning Publications Co.

Love, Robert. 2010. *Linux Kernel Development*. Third edn. Developer's Library. Upper Saddle River, NJ, USA: Addison-Wesoley.

Mauerer, W. 2008. *Professional Linux Kernel Architecture*. Wrox Programmer to Programmer. John Wiley & Sons.

McKenney, Paul E., & Walpole, Jonathan. 2008. Introducing technology into the Linux kernel: a case study. *SIGOPS Oper. Syst. Rev.*, **42**(July), 4–17.

McKenney, Paul E., Appavoo, Jonathan, Kleen, Andi, Krieger, Orran, Russell, Rusty, Sarma, Dipankar, & Soni, Maneesh. 2001. Read-Copy Update. *In: Ottawa Linux Symposium 2001*. Ottawa Linux Symposium.

Nethercote, Nicholas, & Seward, Julian. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *SIGPLAN Not.*, **42**(June), 89–100.

Padala, Pradeep. 2002. Playing with ptrace, Part I. *Linux J.*, **2002**(November), 5–.

Randell, B., & Kuehner, C. J. 1968. Dynamic storage allocation systems. *Commun. ACM*, **11**(May), 297–306.

Raymond, Eric S. 2003. *The Art of UNIX Programming*. Pearson Education.

Reingold, E. M., & Tilford, J. S. 1981. Tidier Drawings of Trees. *IEEE Trans. Softw. Eng.*, **7**(March), 223–228.

Schlesinger, Richard, & Garrido, Jose. 2007. *Principles of Modern Operating Systems*. 1st edn. USA: Jones and Bartlett Publishers, Inc.

Spirn, Jeffrey R. 1977. *Program Behavior: Models and Measurements*. New York, NY, USA: Elsevier Science Inc.

Stallings, William. 2008. *Operating Systems: Internals and Design Principles*. 6th edn. Upper Saddle River, NJ, USA: Prentice Hall Press.

Tanenbaum, Andrew S. 2009. *Modern Operating Systems*. 3rd international edn. Upper Saddle River, NJ, USA: Pearson Education.

Wilkinson, Barry. 1996. *Computer architecture (2nd ed.): design and performance*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.