Institut für Informatik, Reinhard Riedl, PhD

Master's Thesis in Computer Science and Business Administration

Software: An Economic Perspective and Coping with High Memory Load in Linux

Roger Luethi, Fribourg, Stud. Nr. 93-505-410

January 19, 2004

Abstract

The first part of this thesis examines the economics of software production. Two development models are evaluated for their ability to deal with the peculiarities of software: The proprietary, closed source model and the Free and Open Source Software (FOSS) model. The former is shown to create enormous, usually hidden costs compared to a hypothetical, ideal solution. The combination of current regulations and proprietary, closed source development leads to a suboptimal resource allocation and – eventually – market failure. It is suggested that FOSS offers a solid approximation of the ideal solution made possible by technology and infrastructure advances, poised to become the dominant new development model in free markets unless regulations keep favoring the incumbent and largely obsolete model.

The second part is concerned with the scalability of the Linux kernel, with a focus on its ability to scale down to machines with a limited and limiting amount of memory. The canonical solutions which date back to the time of the introduction of virtual memory in the 1960s are reassessed in the light of hardware developments and modern usage patterns. A prototypical implementation of a load control mechanism for the Linux kernel is presented and evaluated along with the potential of load control in modern general purpose operating systems. Finally, this paper offers a systematic study of performance in high memory overload situations for 88 kernel releases from Linux 2.5.0 to 2.6.0. The study is complemented by a discussion of selected aspects, in particular the impact of unfairness on throughput.

Der erste Teil dieser Arbeit untersucht die Oekonomie der Softwareproduktion. Zwei Entwicklungsmodelle werden untersucht auf ihr Vermögen, mit den Besonderheiten von Software umzugehen: Das proprietäre Modell mit geheimen Quellen und das freie, quelloffene Modell (FOSS). Es wird gezeigt, dass Ersteres im Vergleich zu einem hypothetischen, idealen Modell enorme, vorwiegend unsichtbare Kosten verursacht. Insbesondere führt die Kombination dieses Modells mit gegenwärtigen Regulierungen unweigerlich zu einer suboptimalen Ressourcenallokation und schliesslich zu Marktversagen. Es wird dargelegt, dass FOSS eine solide Annäherung an die ideale Lösung darstellt, die erst durch Fortschritte in Technologie und Infrastruktur ermöglicht wurde und das dominante Modell in freien Märkten ablösen kann, es sei denn, Regulierungen bevorzugen weiterhin das etablierte und überwiegend obsolete Modell.

Der zweite Teil befasst sich mit der Skalierbarkeit des Linux Kernels, insbesondere mit seiner Fähigkeit, auf Maschinen mit begrenztem und begrenzendem Speicher nach unten zu skalieren. Die kanonischen Lösungen, welche aus der Zeit der Einführung virtuellen Speichers nach 1960 stammen, werden im Licht von Hardwareentwicklungen und modernen Anwendungsmustern neu bewertet. Eine prototypische Implementierung eines Lastkontrollmechanismus wird präsentiert und zusammen mit dem Potential von Lastkontrolle in modernen Allzweck-Betriebssystemen evaluiert. Schliesslich bietet dieses Papier eine systematische Leistungsstudie der 88 Kernelversionen von Linux 2.5.0 bis 2.6.0 in Situationen mit massiver Speicherüberlastung. Die Studie wird ergänzt durch eine Diskussion ausgewählter Aspekte, insbesondere der Bedeutung von Unfairness auf den Durchsatz.

Contents

Acknowledgments **7**

Introduction **8**

I. Software: An Economic Perspective **10**

1.1. Introduction . . . 11
1.2. Software and Economics: Setting the Stage . . . 11
1.2.1. Software as a Public Good . . . 11
1.2.2. Copyrighting Secrets . . . 14
1.2.3. A Model for Software Production . . . 14
1.3. Proprietary Software . . . 15
1.3.1. Known and Hidden Costs . . . 15
1.3.2. The Software Market . . . 17
1.3.3. Hedging a Stranglehold . . . 20
1.4. Free and Open Source Software . . . 23
1.4.1. Cost Comparison . . . 23
1.4.2. FOSS Weaknesses . . . 25
1.5. The Software Market Revisited . . . 31
1.5.1. FOSS as a Strategic Weapon . . . 31
1.5.2. The Road Ahead . . . 34
1.6. On Regulation . . . 37
1.6.1. A Call for Free Markets . . . 37
1.6.2. Public Policy . . . 38
1.7. The Microeconomic Angle . . . 40
1.7.1. Considering FOSS Deployment . . . 40
1.7.2. Beyond TCO . . . 41
1.8. Conclusions . . . 44

II. Coping with High Memory Load in Linux **47**

2. Linux Performance Aspects **48**
2.1. Introduction . . . 48
2.2. Linux Scalability . . . 48
2.2.1. Up and Down . . . 48
2.2.2. Linear Scaling . . . 49
2.2.3. Vertical vs. Horizontal Scaling . . . 49
2.2.4. Userspace Scaling . . . 50
2.2.5. Scaling by Hardware Architecture . . . 50
2.3. Beyond Processing Power . . . 51
2.3.1. Challenges . . . 52
2.3.2. Scalability Limits? . . . 53
2.4. Conclusion . . . 55

3. Thrashing and Load Control **56**
3.1. Introduction . . . 56
3.2. Trends in Resource Allocation . . . 56
3.3. Thrashing . . . 57
3.3.1. Models . . . 57
3.3.2. Modern Strategies . . . 60
3.4. Decision Making in System Software . . . 63
3.5. Linux Resource Allocation . . . 64
3.5.1. Process Scheduler . . . 64
3.5.2. Virtual Memory Management . . . 65
3.5.3. I/O Scheduler . . . 66
3.6. Enter the Benchmarks . . . 67
3.7. Load Control . . . 70
3.7.1. A Prototype Implementation . . . 70
3.7.2. Load Control in Modern Operating Systems . . . 73
3.7.3. Prototype Performance . . . 75
3.8. Paging between Linux 2.4 and 2.6: A Case Study . . . 77
3.8.1. Overview . . . 77
3.8.2. Identifying a Culprit . . . 79
3.8.3. Unfairness . . . 83
3.8.4. Notes on Linux Reporting and Monitoring . . . 84
3.9. Conclusions . . . 85

III. Appendices **88**

A. Source Code vs Object Code **89**

B. Technological Means to Prevent Unauthorized Copying **92**
B.1. Watermarks . . . 92
B.2. Software Activation . . . 92
B.3. License Manager . . . 93
B.4. Hardware Dongle . . . 93
B.5. Trusted Computing . . . 93

C. Proprietary Software in Practice **95**
C.1. In Defense of Microsoft . . . 95
C.2. Vaporware and Sabotage . . . 96
C.3. Taxing Hardware . . . 97
C.4. Fear, Uncertainty, and Doubt . . . 98

D. Software Market Numbers **100**

E. Legislation and Overregulation **101**

F. Total Cost of Ownership **107**

G. A Word on Statistics **109**

H. Source Code **111**
H.1. thrash.c . . . 111
H.2. log.c . . . 112
H.3. plot . . . 115
H.4. linuxvmstat.pm . . . 122
H.5. linux24.pm . . . 124
H.6. linux26.pm . . . 127
H.7. freebsd.pm . . . 132
H.8. loadcontrol.diff . . . 134

I. Glossary **142**

Acknowledgments

First and foremost, we would like to thank Reinhard Riedl, Head of the Distributed Systems Group at Department of Information Technology of the University of Zurich, for giving the leeway to explore the subjects freely, and for providing valuable criticism and feedback throughout the past months.

Several Linux kernel hackers influenced this paper: Andrew Morton suggested load control and thrashing in the Linux kernel as a topic. Rik van Riel and William Lee Irwin III contributed to the discussion which is almost entirely recorded in the mailing list archives for linux-kernel and linux-mm (Linux Memory Management). Special thanks go to Linux Torvalds for revealing the UPK mystery [1].

The group around Margit Osterloh, Professor of Business Administration and Organization Theory at the Institute for Research in Business Administration at the University of Zurich, provided insight into the state of FOSS research in their discipline and inspired us to focus on the macroeconomic perspective and the consequences of regulations and proprietary, closed-source software development.

This thesis would contain many more Germanisms had not Daniel G. Rodriguez done the work of an editor all the while traveling Mexico.

Last but not least, we are indebted to the authors, maintainers, and contributors of the FOSS community who created the software that made this thesis possible. From a vast list of excellent software we recognize in particular the crucial role played by LaTeX, gnuplot, gcc and various other GNU tools, perl, vim and – of course – the Linux kernel.

Opinions and any mistakes are the sole responsibility of the author.

[1] Cf. footnote, page 53

Introduction

Part I

Most research papers on Free and Open Source Software (FOSS) are contributions towards a better understanding of the mechanisms that drive it: The motivations of FOSS developers, the reasons why some companies choose to give the source code for their software away for free, the organization of a disparate and distributed community (e.g. [87, 53, 81]). These papers share one common theme: "How could FOSS possibly work?"

Starting from that foundation, this paper tackles a different question: "But what is it good for?" – As most software has been written using a model other than FOSS, we believe that the key to evaluating FOSS is not a description of what it can do, but a comparative study of its economic benefits and costs.

This approach should be most relevant to people who are interested in or concerned with public policy, IT strategy, or software procurement. This is not an introductory paper into FOSS; however, the reader of part I need only be familiar with the basic principles and terminology. Especially those readers without a technical background may find that the appendices contain important explanatory and supporting material:

- Appendix A explains the difference between source code and binary executables that are sold as closed source software.
- Appendix B presents common technological means used by proprietary software vendors to prevent unauthorized copying.
- Appendix C illustrates common practices in the proprietary software industry with examples.
- Appendix D documents the market share distribution for some of the largest software segments.
- Appendix E discusses a recent law initiative as an indicator of regulation changes sponsored by the proprietary software industry.
- Appendix F shows the controversy surrounding Total Cost of Ownership studies that have become increasingly popular in the IT industry.

Part II

Chapter 2 opens with a high-level discussion of Linux scalability. We present some scalability improvements of recent years and note that the focus of Linux development work seems to be shifting to other areas.

In chapter 3, we look at the history of resource allocation in operating systems. We take a classic model for thrashing behavior and extend it to better match our own ideas and observations of this phenomenon. We discuss methods for decision making in system software, central components for resource allocation in Linux, and a number of benchmarks with high memory load – all this in preparation of our own prototypical implementation of a load control mechanism in the Linux kernel. We present results and evaluate the role load control can play in modern operating systems.

In addition, we offer a case study that demonstrates performance regressions of the new Linux kernel 2.6 under high memory load. Finally, we show that some of these regressions cannot be addressed with load control.

The statistics presented throughout part II are explained in Appendix G.

Part I.

Software: An Economic Perspective

Economics: The overeducated in pursuit of the unknowable.

(Robert Solow)

1.1. Introduction

Software developers and economics scholars tend to be fascinated by the Free and Open Source Software (FOSS) development model because it succeeds despite its apparent rejection of both the economic canon and the practice – or rather the theory – of traditional software engineering (cf. [108, 63]).

FOSS roots in the hacker culture and ethics which are clearly not based on business considerations. IT executives who deploy FOSS in their organizations are hardly impressed with ideologies or hacker ethics, however. They rely on FOSS because it provides benefits today that are reflected in the bottom line of their operations.

In our view, supporters and opponents alike tend to overemphasize the real and perceived contradictions between FOSS and classic economic theory. Witness Raymond's contrasting the hacker "*gift culture*" to the traditional "*exchange culture*" [84] on the one side and frequent complaints about the alleged "*anti-business*" nature of FOSS on the other side.

In this thesis, we try to reconcile FOSS with economic theory. In fact, we argue that for many software markets, FOSS has become the most efficient production model.

The history of FOSS has been presented in detail in a number of publications [65, 114], and many papers have explored motivation and mechanisms of FOSS development [53, 87, 81]. We have looked at these aspects ourselves [61, 62] in earlier papers.

This thesis starts with a discussion of Public Goods. It defines the *order* of Public Goods and introduces a model for software production based on assumptions of classic economics in section 1.2.

Proprietary, closed source software production is compared to this model in section 1.3. We describe behavioral patterns of the various participants in software markets and the stable position they are seeking.

Section 1.4 considers how FOSS handles the problems we found with proprietary software in section 1.3 and looks at some weaknesses of FOSS compared to proprietary software. In section 1.5, we list reactions of some major players in the proprietary software industry to the challenge of FOSS and sketch out what we expect in the future. Section 1.6 calls for a software market with less regulations and privileges.

In the closing of part I, section 1.7 abandons the macroeconomic perspective to consider FOSS from the perspective of organizations. We submit a number of arguments to extend the limited view of Total Cost of Ownership (TCO) studies.

1.2. Software and Economics: Setting the Stage

1.2.1. Software as a Public Good

A Public Good exhibits two basic properties [29]:

- *Nondiminishability, which means that one person's use of the good does not diminish the amount of it available for others.*
- *Nonexcludability, which means that it is either impossible or prohibitively costly to exclude people who do not pay from using the good.*

The usually tacit assumption is that the good is desirable, that it is not some sort of negative good that society would be better off without to begin with. In other words, a Public Good creates positive external effects. For a pure Public Good, *overuse* is not a concern due to the nondiminishable nature of the good, but the problem of *underprovision* is very pronounced: Supply will fall short of demand, because firms cannot hope to recoup their investments, let alone make a profit. A classic justification for government, this argument has seen a lot of debate about how specific Public Goods might be provided by private companies (see [12] for a famous example) or whether certain goods are in fact Public Goods or not.

Software is quite obviously nondiminishable. In addition, software exhibits very strong *network effects*: It tends to gain in worth when it is being used by others (cf. [112]), an effect that Eric S. Raymond folds into what he calls "*Inverse Commons*" [89].

Popular software compels a symbiotic support industry to offer books, training, consulting services, and more.

Popular software attracts a symbiotic industry producing supporting goods like books, training, or consulting.

Also, a large user base – paying or not – builds mind share, which is beneficial for the software author as long as lost revenue[2] today translates into higher revenue in the future.

Software could also be seen as nonexcludable. But is it really? After all, it is not their sense of fair compensation alone that is keeping people from copying software freely. The first obstacle is technological provisions employed by software producers to make copying difficult [3], the second is laws that give a software author exclusive rights to copy her work. It is at this point where we have to revisit the definition of nonexcludability: Where does "*prohibitively costly*" start? And whose costs are taken into account when determining that?

Consistent with macroeconomic literature we demand that the distortion caused by *external effects* be taken into account. Therefore, the costs to any members of society are to be accumulated and we suggest further that:

- If free markets can sustain production of a nondiminishable good using technological or other means, the Public Good is a *second-order Public Good*.
- The costs for excludability of a nondiminishable good are prohibitive if interventions by the state are necessary for a sufficient supply. We call this a *first-order Public Good*.

The order of a Public Good may change as technological progress creates new methods of excludability. Many roads may cease to be first order Public Goods when advances in technology make road pricing economical, and encrypted distribution allows radio and TV stations to exclude people from their broadcasts.

Typical state interventions for first order Public Goods include subsidies, special regulations, and production by the state itself.

One key question in arriving at a solution – whether by private entities, the state, or a combination thereof – is whether the approach is based on excludability: In that case, the good can be sold on a market which provides the producer with important information about the demand in terms of quality and quantity. The nondiminishability aspect remains intact and the marginal

[2] There are no additional *costs* to the producer due to the nondiminishability.

[3] Appendix B.

production costs are still virtually zero, which usually creates new problems based on the circumstances:

- If the sales price asked by the vendor is close to zero to reflect the marginal costs, then it may be hard to recoup the mere costs of ensuring excludability, let alone any initial fixed cost investment. This is an extreme form of the perfect competition that all vendors face in the ideal market of classic economics – "extreme" due to the enormous ratio between fixed and marginal costs; in a recent example, mobile phone carriers found themselves in a similar situation, although their product clearly fails to qualify for any definition of a Public Good.

  This case is rare in practice: If the initial investment was miniscule, there would be no underprovision problem worth debating. The vast economies of scale that are typical for Public Goods mean that a market will rarely support more than a few dominant suppliers. Combining this with the premise of excludability above, we find that suppliers will likely have some substantial pricing discretion.

- To be able to set a price, the vendor should have a monopoly or at least a dominant position in the market or for a product for which no cheap substitute exists. If the vendor sets a price that is significantly different from zero, a further distinction should be made:

  – The vendor offers the product at a fixed price. In that case, the problem of *underuse* arises: The higher the price, the more people choose not to buy the product because the asking price is higher than their respective benefits. For the economy as a whole, the sum of all individual benefits is an uncompensated welfare loss compared to an optimal solution.

    In this case, one form of underprovision replaces another: Excludability creates an incentive to increase production, but the distribution of the resulting goods becomes suboptimal.

  – The vendor charges customers individually based on their presumed benefits. This is called *price discrimination* and depends on an additional prerequisite: The good should not be transferable. Otherwise, some customers will buy at a low price only to resell the good to high-margin customers, a phenomenon which results in *gray markets* [4].

    Goods that are tailored to an individual and many services are typical examples of products that are not transferable.

  According to standard economic theory, perfect price discrimination results in an optimal supply and distribution of the good, but it shifts the surplus consumers enjoy in a competitive market to the supplier.

Recapitulating our discussion of production and business models for nondiminishable goods that are based on excludability, we note that they their key advantage is the provision of market

[4] Gray markets are less important if a good is transferable but not storable, like perishable food. That is, however, rarely a property of nondiminishable goods.

information about actual demand; the quality of that information can vary greatly in the presence of market distortions caused for instance by regulations or dominant suppliers. Negative consequences of the excludability approach include a mixture of underuse and the disappearance of the consumer surplus.

1.2.2. Copyrighting Secrets

Copyright started as a protection for literary works and over time expanded to cover other artistic expressions like paintings and music. Economically, copyright is a deal between society and creators of certain kinds of Public Goods.

Society defined and enforced an artificial excludability as an incentive for creators to produce their goods. Not only ordinary consumers benefited from works that were created due to this incentive: Potential competitors – writers, musicians, and architects – found inspiration in the creations of their contemporaries. And after some time, the copyright expired and the works entered the public domain, to be used freely by anyone interested without fee for any purpose. Many argue that in the past decades and even centuries, this balance has been changed to massively favor copyright holders over the public interest (for example [56]), and leading economists have weighed in to condemn the repeated extension of copyrights [3]. For the purpose of this paper, however, the description of the intent of copyright still holds: It is a deal offered by society to encourage the production of creative works.

It is important to realize that it was not a Law of Nature, but a political decision that extended copyright to cover computer programs as "*literary works*". It was another decision to grant protection regardless of their "*form of expression*", thus including both source code and binary executables which are not human-readable [5] [80].

The justification for this regulation must have been that private measures alone cannot provide excludability and thus without help, a free market would fail to supply a sufficient amount of software goods. The arguments to support this assertion certainly seemed more solid at the time than they do today.

1.2.3. A Model for Software Production

In oder to compare the costs of software production models, we introduce an imaginary solution. This solution is based on the classic economic assumptions of zero transaction costs and full rationality. In support of this choice we note the following:

- Although transaction costs and *bounded rationality* clearly influence the market under consideration, the impact of these phenomena is hard to quantify.
- The share of transaction costs has changed over the past decades – for instance, global computer networks provide new channels for information and offer a potential for vastly lower transaction costs.
- Information technology has been used since its inception to push the information constraints imposed by a bounded rationality.

[5] Appendix A elaborates on the difference between source code and binary executables.

For nondiminishable goods such as software, marginal costs are negligible compared to the fixed development costs. The marginal cost of producing an additional copy of a program has no relevance on the sales price.

Under this model, producers and consumers alike can predict costs and individual benefits of a software product – this thanks to full rationality. Perfect price discrimination becomes possible, a practice usually associated with monopolists appropriating the supposed consumer surplus of competitive markets to maximize their own profit.

In our case, however, price discrimination is used to distribute costs and surplus among consumers. The producers will face a traditional, perfectly competitive market and offer to write software at cost, meaning that they can recoup the fixed costs for development.

Adam Smith's theorem of the invisible hand, also known as the first theorem of welfare economics, asserts that an equilibrium resulting from exchanges in a competitive, free market will be *Pareto optimal*: Since exchanges continue as long as any are left that benefit both parties, an equilibrium is reached only when no one can gain except at the expense of others.

In our model, each transaction consisting of a pooled payment and the corresponding software output is one step towards Pareto optimality, because every participant in the transaction yields a net gain.

A program or feature is implemented if and only if its costs are at least outweighed by benefits to customers, and development may start only after sufficient payment has been pledged. In other words: With transaction costs zero, the software will be produced and paid for if the combined benefits to all people make it worthwhile, which is of course to be expected from an efficient resource allocation scheme.

And thus in a classic economic world, an optimal software supply is sustained and funded without copyrights or any other special regulations.

1.3. Proprietary Software

1.3.1. Known and Hidden Costs

Starting from the baseline our model provides, we can proceed to compare to the situation resulting from the combination of copyright protection, secret source code, and proprietary software licenses that is common today. Costs and drawbacks of proprietary, closed source software compared to an ideal solution include:

- Efforts for technological measures to prevent unauthorized copying: development and production of license managers, software activation, hardware dongles, etc.
- Hassle for software users due to said technological measures [6].
- Law enforcement: cost of police, lawyers, and a judicial system investigating crimes that did "*not diminish the amount of [a good] available for others*", crimes that often result in superior allocations: Distributing a nondiminishable good to a person who gains something from its use is an overall gain, unless the person now using an unauthorized copy

[6] See Appendix B for a discussion of technical measures and some problems experienced by paying customers.

would have bought the program had it not been copied – in that case, the gain by the copier equals an actual loss by the producer. Due to network effects even the software producer may yield a substantial benefit from uncompensated copying.

Unauthorized copying is, of course, harmful in the long run if it significantly hampers production in the future – however, the incentive problem did not exist in our ideal world of fully rational individuals, and neither was there unauthorized copying.

- As discussed in section 1.2.1, a production model based on individual sales of a nondiminishable good requires massive price discrimination. Without price discrimination, an enormous gap between optimal and actual supply will persist, since many people would benefit from using the software, but not enough to justify the price the vendor asks. Many mutually beneficial exchanges between software producer and users would fail to take place and in a massive violation of the first theorem of welfare economics the economical benefits of a nondiminishable product are all but eliminated.

  Software is by nature a perfectly storable, transferable good. It follows that price discrimination for software can only work if additional technical or legal barriers are put in place in order to prevent gray markets[7]. If a monopoly is in place, though, such barriers are fatal for consumers.

  The sum of potential gains lost due to imperfect price discrimination and the costs of measures to allow selective pricing are additional costs compared to the ideal solution.

- The practice of keeping source code secret facilitates all kinds of illegal and anti-competitive behavior. Short of an inspection of the closely guarded source code it is, for instance, virtually impossible to positively determine whether a closed source software vendor has misappropriated somebody else's work [8].

  Additional costs incur due to the resulting market inefficiencies and failures.

- If the typical duration of copyright these days is questionable for other works ([3], again), it is ludicrous for software. Unlike books or music, computer programs will for all practical purposes be utterly worthless once their copyright protection expires.

  Moreover, in all likelihood no computers will exist to run closed source software once it enters the public domain. Even if such computers or – more likely – *emulators* existed, software protected by hardware dongles or the increasingly common software activation would refuse to work regardless.

- Closed source software is rather useless for educational purposes, at least as far as learning how to write programs is concerned. In a world of proprietary, closed source software, programmers and computer scientists are forced to study toy programs or whatever source code their respective employer may have access to.

[7] See also page 104 in Appendix E, for a discussion of a law that makes price discrimination easy to enforce.
[8] Further, more precarious options open to closed source software vendors will be discussed later in this chapter.

- Welfare losses caused by source code dying along with a defunct software company are very common [5]. In some cases, at least the damage for existing customers can be mitigated by *source code escrow services* which come at an extra cost and are hard to implement correctly.

- Since proprietary code is not shared, the same functionality must be reimplemented numerous times. Binary software libraries can alleviate that issue by providing limited means for software reuse [9].

- There are strict limits to the customizations a given proprietary program allows a buyer to make, if any. The program she "bought" is not hers to use any way she sees fit or to improve at her discretion, even if she is content to never keep more copies than she paid for. She has no choice but to accept the software vendors price for any incremental improvement she would like.

- In addition, massive market distortions and inefficiencies are caused by specific properties of the software market. Most of them are exacerbated by the proprietary, closed source development model. We dedicate a separate section (1.3.2) to these effects.

It seems quite obvious that society got shortchanged when copyrights were extended to cover software. The economy pays a steep price for the mere providing of a sufficient incentive for software producers to supply their goods – other benefits that come with copyrighted books and music are obliterated by proprietary, closed source software as explained above.

And yet, since computers and the software they run are seen as beneficial to the economy despite those staggering costs, all these points are moot unless there is a better way to produce software, one that avoids at least some of the extra costs while still managing to maintain production. Before considering alternatives, though, we will explore somewhat further the economic consequences of today's dominant software production model.

1.3.2. The Software Market

Software production ought to be a highly competitive market. After all, it takes little more than a computer to enter the competition. There are some obstacles, however, that are specific in quantity and quality to the software market:

- High development costs and astronomical economies of scale favor companies that manage to move large volumes, making it hard to break into a mature market.

- Regulation changes and initiatives of the past decades have almost invariably favored large, proprietary software vendors. Examples include the introduction of software patents and the UCITA initiative in the USA [10].

[9] Some pitfalls and typical problems of proprietary, closed source software libraries are discussed in [88].
[10] Appendix E.

- Despite some advances in the past decades, data is still tied to specific applications. Database management systems have their extensions of the SQL standard, word processors their secret file formats, file servers their proprietary protocols. This leads to a phenomenon known as *vendor lock-in* [11] which is the next hurdle awaiting a new contender who manages to offer a superior product at a competitive price.

  The effect of having data tied to applications is twofold:

  – Once a system has been picked , a company will likely stick with its supplier, because the costs of switching to a different vendor are substantial and typically eat up all the benefits of switching. The purchaser becomes dependent on the supplier and thus subject to *holdup*.

    Williamson calls this the *fundamental transformation* of contracting [115]: The customer's data and other assets like human capital become *relationship-specific assets*, giving the current supplier a competitive advantage that is hard to beat, no matter how good alternative products are. Customers are stuck with their initial choice.

    It is this transformation that allows every vendor who has his customers tied to his application to enjoy what amounts to a bilateral monopoly[12] which in turn leads to market failure.

    Eric S. Raymond notes [89]:

    > *If the supplier doesn't perform, you will have no effective recourse because you are effectively locked in by your initial investment and training costs. You need your supplier more than your supplier needs you.*

  – In order to avoid an expensive migration to a different product, a rational buyer will likely buy from the supplier with the least risk of discontinuing its products. This argument favoring large companies usually trumps first mover advantage or the higher flexibility of small software firms.

- Buying what everybody else buys offers benefits to the purchaser other than the assurance that the vendor is not going away. It is also easier to find people who know how to work the software, and it improves interoperability with the majority of companies that bought the same system. Additional third party software will be readily available for a widespread platform and much less so for others. This is the core of the network effects of software.

- Sales of proprietary software reflect the number of users who benefit from it beyond the asking price – that is how consumers decide to buy software, after all. However, sales figures provide little information about what exactly convinces users to prefer one program over another, a problem that becomes worse as more and more functionality is sold in a bundle that is only offered as a whole.

  The worth of this information in a mature software market is frequently and highly overrated: A high impact of network effects on attractiveness and market price of a product

[11] An effect that works by association as well: For instance, a monopoly may extend to operating systems and even hardware if a program that has become critical is only available on one platform.

[12] He is, for instance, not anymore in the database management systems market. As far as his locked-in customers are concerned, he now owns his BrandName™DBMS market.

mean that only a small portion of the price reflects the quality of the product itself. Thus, prices in a mature software market will rarely reveal anything beyond the well-known fact that most consumers prefer to buy from the dominant vendor. If anything, this information will tell a rational vendor to focus on market dominance and not on product improvements. We will see in section 1.3.3 that as far as the software market is concerned, these two goals rarely amount to the same.

- Large, proprietary vendors can use their monopoly rents to cross-subsidize other segments of their operations which leads to market distortions. When Microsoft started to provide revenue information for individual business segments, it turned out that the margins for client operating systems and the office productivity software were around 80% while four of the remaining five business segments were losing money [14] in markets where the company had been a late entrant.

  It is worth noting that some denounce as cross-subsidization what others call investment. The aspect we find alarming, however, is that *monopoly rents* are used as a lever to conquer existing markets rather than to innovate and create new ones.

- We have shown why software tends to increase its attractiveness just by being common. This leads us to predict a seemingly paradoxical effect for the software market: In this industry with its extremely high economies of scale the most frequently sold products tend to cost *more* than comparable products by niche vendors, simply because each vendor will ask what the market bears.

  Unless the niche player has a vastly better cost structure, two outcomes are possible: either the dominant player will reap fabulous profit margins or the niche player will eventually go out of the market, in which case fabulous profit margins for the dominant vender have just been postponed to the time when competition has ceased to exist.

  It follows that the software market presents a classic *social dilemma*: While it is in the interest of all buyers to maintain competition on the supplier side, their strategy in the *Nash equilibrium* is to chose the supplier most likely to be the coming monopolist.

As long as some serious competition exists, a software vendor must balance the benefits of actively locking customers in with the risk of alienating existing and prospective customers. Competitors try to improve their products in order to appeal to customers.

The higher a particular vendor's market share, though, the more attractive the products. When competition dwindles and network effects get stronger, we find a self-amplifying mechanism working towards a market structure with one dominant vendor claiming an increasing portion of growing externalities, simply because in a proprietary software market, one vendor ends up with a monopoly on the product that benefits most from externalities. Eventually, customers may come to accept the lock-in as inevitable.

Considering what has been said so far, we expect any software market to develop an oligopoly or a monopoly. This would be most pronounced in areas where lock-in is particularly effective [13]. We present recent numbers for some key products in Appendix D.

[13] An example of a market where vital data are rarely tied to applications is the computer games market, which incidentally is still highly competitive.

When an oligopoly or a monopoly is established, the market changes: The dominating player's main interest becomes to defend the status quo that guarantees a high rent. Customer demands will be ignored if they jeopardize market dominance.

1.3.3. Hedging a Stranglehold

Information Technology is not like other industries, and the peculiarities do not end once a monopoly is in place. Standard economic theory notes that a monopolist gains the discretion to set the product price. Compared to perfect competition, profit-maximizing monopolies are predicted to produce less and sell at a higher price. Famous textbook examples of supplier dominance – Rockefeller's Standard Oil Company or the De Beers monopoly on diamonds – confirm the prediction.

There is no mention of substantially lower quality in diamonds or oil sold by monopolists. Dominant software manufacturers, however, have both opportunity and vested interest to create programs that are inferior from a customer perspective.

In particular, software purchasers and dominant vendors are diametrically opposed on one crucial issue of software quality: Interoperability. To a consumer, software is good if it is flexible and can be used with a large number of other products, which not only maintains competition but allows to use the best tool for any job. Obviously, this clashes with the goals of dominant suppliers and it is they who direct product development.

Having all the factors listed in the previous section going for them, dominant vendors can afford to lag in product quality and raise prices, or to sink huge amounts of money only to prevent potential competitors from arising.

We present a selection of proven methods to fortify such a strong position:

- Use of monopoly power to expand into adjacent markets. A brute-force approach may work simply because the monopoly gains in one market allow operation at a loss in a different market indefinitely; this can be beneficial if it keeps potential competitors from developing a financially sound base that might have posed a threat at some point in the future.

  A more subtle approach has one's monopoly product work exclusively or preferentially with the new product that is about to be pushed into the market. "Integration", a marketing term frequently used to describe this practice is a euphemism with several meanings. What they have in common is best understood in contrast to interoperability:

  - The goal of *interoperability* is to have products of different vendors work together. The path to interoperability tends to be long and tedious: Either because it takes time for multiple competing vendors to agree on a standard, or because a dominant vendor prefers the exclusive control of a de facto standard over an open standard. Interoperability lowers the costs of heterogeneous IT environments and encourages customers to consider the competition.
  - The goal of *integration* is to have products of one vendor work together. It is a fast method of innovation, able to yield impressive results quickly because the company pushing it need not consult anyone else. Integration provides a relative cost

advantage to homogenous IT environments and encourages customers to invest in a software monoculture with products all by the same vendor.

In section 1.5.2, we will see that integration plays a crucial role in the plans of dominant software vendors.

- Lobbying for government regulations to raise barriers to market entry, and to expand the negotiating powers of copyright holders to strengthen their position against their customers [14].

- Spreading *Fear, Uncertainty and Doubt*. FUD is a common industry term and refers "*to any kind of disinformation used as a competitive weapon*" [86].

  – One powerful form of FUD in the IT market insinuates that a competitor's customers will be left stranded with an unsupported product, which appears to be a worse scenario than being dependent on a monopolistic vendor. Such warnings from a dominant player can become self-fulfilling prophecies.

  – New competitors can be stopped by announcing the imminent release of a product better and cheaper than the challenger's; an actual product may or may not be released later on. A young company will hardly survive long enough to see the potential buyers realize that the product promised by the dominant company was little more than an announcement destined to avert a threat. This special form of FUD is called *vaporware* [15].

- With marginal costs that are virtually zero, a monopolist can offer software at any price without losing money. Preferential pricing and price discrimination help win over buyers when they still have a choice, and price wars drive potential competitors out of the market. When the market is ruined and the competition eliminated, prices can be raised again.

  In some cases, a program like the one a potential competitor is selling may even be given away for free. This often takes the form of *bundling*: Customers receive a program "for free" in combination with a monopolistic product they need to buy anyway. The nature of software makes it easy for vendors to claim that bundling is just another form of "integration" and beneficial to customers.

- If a smaller company seems firmly entrenched in its market, it can often be bought. Personal finance management software is one example: After a number of failed attempts to challenge Intuit, the incumbent in that market, Microsoft successfully negotiated to take control of Intuit. The only reason Microsoft does not own the dominating line of products in this area is that the US Department of Justice announced it would file suit to stop a planned merger [100].

  Even a high premium on the purchase is outweighed by the gains of maintaining or expanding a monopoly.

[14] Appendix E.

[15] One classic example of vaporware is included in Appendix C.2.

- Closed sources allow companies to build arbitrarily complex code to prevent competitors from writing interoperable software once a market is captured, thus shutting out competition. This is famously a common practice in the software industry.

- The dominant players can set de facto standards and have a vested interest that there be no open standards which could level the playing field.

  If an open standard threatens to weaken the lock-in, proprietary features can be added to it. The standard will officially be supported, but the "enhanced" implementation will depend on those new features. This strategy is commonly referred to as *embrace and extend*.

- Changing data formats and communication protocols frequently and making them complicated without technical need has a number of benefits for the dominant vendor:

  – Competitors will have to waste resources reverse engineering what the market leader proclaimed the "industry standard".

  – Competitors will always lag behind, trying to catch up with the most recent protocol changes.

  – If a new product release uses a new protocol or file format, a limited number of buyers can compel the rest of the market to buy the new product just to regain the ability to exchange data.

- Hidden code can be added to a program with the sole purpose of sabotaging a competitor's software. Even in the unlikely case that the competitor manages to hold out long enough to win a long war in court, the uncertainty about the outcome will have stopped the competitive threat. Sadly, this is no mere theoretical possibility, either [16].

- A particularly successful software company may coerce hardware vendors to buy a software license for every single machine sold, regardless of the programs the customer actually bought [17].

  Contracts may even contain clauses to flat out prohibit the installation of any competing software by the hardware vendor. This will work if the hardware vendor's profit or even existence depends on the permission to sell the monopoly software.

Many of these methods are known in other industries as well. However, we put forward that the current regulation of software and its associated business models which focus on creating excludability amplify the inherent problems of software markets, most notably their exceedingly strong tendency to favor the dominant player. In consequence, software markets seek to be owned by one supplier.

Moreover, proprietary, closed source code creates a whole slew of golden opportunities to hold relationship-specific investments hostage and a chance for dishonest individuals to take advantage of information asymmetries. Since such behavior is very hard to prove, vendors not resorting to these methods are at a competitive disadvantage.

[16] Appendix C.2 discusses one of the best documented examples.
[17] The taxing of hardware is explained in Appendix C.3.

Last but not least, economic core benefits of software are systematically curtailed or eliminated to provide further incentives for the software producers.

The reader is invited to revisit this section later and consider whether an alternative software production model can limit each of these problems.

1.4. Free and Open Source Software

1.4.1. Cost Comparison

FOSS is the alternative we are looking at in this paper. Definitions of Free Software and Open Source Software can be found in [27] and [83], respectively.

The history of FOSS has been presented in detail in a number of publications [65, 114], and many papers have explored motivation and mechanisms of FOSS development [53, 87, 81]. We have looked at these aspects ourselves [61, 62] as well.

We will not present yet another discussion of whether or how FOSS development can produce competitive products. Excellent papers do this already, and a mountain of evidence in form of highly competitive FOSS software is steadily growing.

Instead, we will focus on the question of whether and how FOSS can overcome the shortcomings we found the in the proprietary production method, and what combination of regulations and business environment would likely yield the highest global utility before we examine the microeconomic perspective in a separate section.

As we recall the problems with proprietary software, we find a very different picture for FOSS:

- FOSS foregoes any attempts at introducing excludability to software and therefore preserves all the gains of a nondiminishable product. Distribution optimality is largely achieved by using the cheap transports the Internet provides.
- There are no costs due to technological or other measures to prevent software copying or transfer, neither for the producer nor for the consumer.
- The strain on the judicial system is very limited. There are few restrictions to enforce, and any wrongful use of code in a FOSS project is visible to anyone who cares to look.
- Common anti-competitive measures that require that the source code be secret and exclusive property of one entity are unworkable with FOSS. This helps maintaining competition which in turn benefits consumers.
- FOSS licenses, too, rely on copyright protection, and it is unlikely that FOSS will be of much value, either, by the time the copyright protection expires.

  This is, however, not a problem since by choosing a FOSS license, software authors relinquish the most important monopoly powers of copyright holders, most notably the *exclusiveness* of their right to copy the work and to create derivative works from it.
- FOSS allows students and scientists to do research on real world source code, some of which is on par in terms of quality with the best proprietary code.

- It is equally possible for programmers to learn or even borrow from any FOSS [18]. Software reuse is actively encouraged, be it in the form of tools, software libraries, code snippets, or simply ideas.

- Some FOSS exponents are lobbying for regulation changes, although often with a focus on reducing the amount of regulation rather than increasing it. We will discuss the merits of various suggestions in section 1.6.1.

- When a FOSS supplier goes out of business or fails to deliver, a user can easily switch suppliers without migrating any software. Any appropriately staffed company can provide support for any FOSS program. More importantly, a consumer willing to pay for a specific feature will find a number of competing companies eager to get the contract.

  Few entities are likely to fund the development of a large FOSS application: Not the producers, because they cannot recoup the investment the way proprietary software vendors can [19]. And not the users, either, because they are not keen on shouldering the financial burden for a big application all by themselves.

  However, a company or a developer may very well pay for a few missing features if that's all that keeps them from finding a program useful. With FOSS, the cost for a program with any given feature set is the cost to extend the best match among existing FOSS programs to meet the requirements.

  That is, by the way, a major reason why FOSS works so much better than similar attempts in literature or music: Software can be (and usually is) written incrementally.

- Consumers might be tempted to wait for somebody else to fund even small features. One reason that FOSS flourishes regardless is that some tend to need an improvement more urgently than others. It is a rational decision for them to fund development and have the result available when they need it.

  Unlike proprietary software, FOSS leverages the incremental nature of software for the benefit of consumers who can calculate the costs for specific functionality relative to the nearest match among FOSS offerings. This mechanism alone means that development work fails to secure sufficient funding only if no user exists for whom the benefits outweigh the development costs.

  Where FOSS is funded this way, two additional effects come into play:

  – A primitive form of price discrimination: Those who benefit the most will often fund development because they are most likely to find the benefits to outweigh the costs of an improvement.

  – Information on actual demand is back without requiring excludability. On the one hand, it is limited to reflect the benefits for those who paid for the development. On the on other hand, the information is available per feature.

[18] The "borrowing" requires the licenses of the affected projects to be compatible, though.

[19] An exception is *dual-licensing* which works well for some projects and companies.

We suppose that the sales figures for proprietary software are a better indicator for demand in young, developing software markets but fail in mature markets where the feedback loop as outlined above is superior.

- With the source code publicly available, it is much harder and less interesting to actively tie a customer's data to an application. The ties to a specific application remain, albeit weaker; customized code, investment in training, etc. form a bond between application and customer. And there is still a case to be made for using what everybody else uses – mindshare has not lost any relevance.

  The crucial difference lies somewhere else: The application has ceased to be controlled by a single supplier. No matter how much a customer depends on a program, there will always be plenty of competition in the market she is in now: The market of support and custom development for an application anyone may work on.

  Depending on her preferences, a corporate user looking for support may chose a multinational or a local company, one that is large enough to be cheap or to hire the best talent, or one that is small enough to be eager to please its customers.

In [89], Eric S. Raymond discusses the economics of proprietary software and FOSS. Pointing out that the vast majority of software written is not for sale but for in-house use [20] and noting that "*the price a consumer will pay is effectively capped by the expected future value of vendor service*" [21] he shares this insight with his readers:

> *Software is largely a service industry operating under the persistent but unfounded delusion that it is a manufacturing industry.*

We have maintained the standard nomenclature and talked about software "vendors" and software "production", but looking at our own findings we second Raymond's observation.

If the events of the past few years are any indication, then FOSS simply means that the software industry is moving towards a model that better fits the service aspect: Remuneration will increasingly be distributed based on value added through improvements and maintenance rather than on monopolistic control of network effects.

This section discussed how FOSS can rectify the imbalance that favored software vendors over their customers. This production model keeps the market competitive and has software follow other industries to become a buyer's market: Large vendors remain strong players, but they have to use their economies of scale and similar advantages to make competitive offerings, or they will lose market share to smaller competitors. Their absolute power is gone, along with the monopoly rent that funded their costly war on the merest hint of competition.

1.4.2. FOSS Weaknesses

Comparing FOSS to our ideal model for software production, we note that the real world lacks two important properties: Perfect information and zero transaction costs.

[20] If most software is written for in-house use, why is this paper focusing on proprietary standard software? – Because the vendors of proprietary software shape both image and regulation of the whole industry, and among them are the most formidable opponents of FOSS.

[21] The sale value of a discontinued product tends to approach zero very rapidly.

This explains why FOSS tends to spread in the wake of Internet access[22]. Ubiquitous, cheap means of communication and data exchange greatly facilitate the massively collaborative effort that most successful FOSS projects are. They allow small projects to reach interested parties anywhere on the planet. The tremendous impact of global computer networks on information and distribution costs mitigated the key weakness of the Public Good software enough to make this production model feasible on a large scale and to let it play out its strengths.

One by one, the FOSS community has tackled problems that were previously deemed unsurmountable. It is therefore too early to draw definite conclusions about inherent limitations of the development model. In this section, we will discuss neither FUD nor issues where at least some cautious optimism seems justified. Rather, we will focus on a select few potential problems and shortcomings that we believe are both significant and unsolved.

Special Purpose Software

As a program becomes tied to a unique purpose, FOSS loses some of its benefits. There may never be global development communities around control software for a particular industrial machine, or FOSS programs to support the peculiarities of a rare business. In some cases, excludability will occur as a natural consequence of the fact that the software is useless without the machine it controls – which may also severely restrict the number of potential contributors.

Typically, those custom solutions are and will be based on standard software components – an operating system, software libraries, programming languages, etc. As far as those components are concerned, the situation looks again like a standard application of FOSS. And indeed, a survey of embedded systems development found in 2002 that the operating systems targeted for over 40% of new projects were FOSS [55].

In other cases, though, proprietary software may be the only way to secure sufficient funds for development. Frequently used programs like operating system kernels, DBMSs, or spreadsheets exist in FOSS versions that are at least good enough for most. Whether the FOSS model ever manages to provide the plethora of polished special purpose software that is available for the dominant proprietary operating system remains to be seen.

Shortcomings of Evolution

The incremental nature of typical FOSS development is often likened to evolution. A goal is reached through numerous small steps, all of which are useful in their own right.

Some applications lend themselves to gradual improvement less than others, though. Many computer games, for instance, require consistent artwork and stories, which moves them closer to traditional arts like literature, painting, and music.

The cost for the minimal useful increment may become prohibitive. In that case, the prospect of exclusive ownership of a program provides the necessary incentive.

[22] Another reason is that FOSS technologies, which have traditionally been the foundation of the Internet, are an impressive show case.

Public Perception: Innovation

There are a number of factors fueling the perception that FOSS does not innovate.

- The FOSS tenet "*Release early, release often*" means that the public is rarely surprised with a host of new features. They are known to exist as soon as they are added to the development tree.
- A popular saying in engineering goes: Good, fast, cheap – pick any two. With FOSS, there is very rarely somebody willing to pay for the consequences of foregoing "cheap" – the prospect of a lucrative monopoly is missing. And since FOSS tends to be driven rather by technology than by sales [23], a tendency of FOSS to focus on cheap and good can be expected.
- A lot of groundbreaking done by FOSS developers is low-profile infrastructure work. This does include the very foundations of the Internet – not only the programs[24] that make it work, but also most of the core protocols like TCP/IP that allow smaller vendors of both proprietary software and FOSS to compete on more equal grounds for once.
- In many areas, FOSS projects just got started. Like any new contender they tend to spend most of their time catching up with what has already been done in that field.
- Another substantial amount of time is invested in reverse engineering proprietary file formats and protocols. In most cases, this is much harder than creating and documenting a new, even superior replacement. In order to ensure interoperability, though, FOSS projects often do both.
- Unlike most proprietary software, eminent FOSS is concurrently developed for a wide range of different open and proprietary hardware and software platforms and is thus very portable. This allows hardware vendors with a small market share to offer a complete software stack with a small investment. A prime example is – somewhat ironically, given its dominance a few decades ago – IBM's line of mainframes which saw renewed interest after IBM ported the Linux kernel and an initial *tool chain* to the S/390 architecture. Because popular FOSS is very portable, there was no or very little additional effort required to make the vast majority of the remaining FOSS applications work on that architecture.

  The development of portability tools is another innovation and significant beyond its promoting competition in additional areas. While writing portable code is good practice, it adds an additional complication FOSS developers typically encounter quite early on.

In summary: FOSS developers frequently innovate in important areas with low visibility; their contributions tend to be underestimated by the public. It is quite possible, though, that monopoly rents allow proprietary software companies to innovate at an even higher rate *if* they chose to do

[23] Using sales instead of marketing here is intentional. FOSS developers tend to be much closer to customers and their wishes than programmers working for large, proprietary software vendors – and knowing what customers want is a key asset of good marketing.

[24] Apache, Bind, and Sendmail, to name a few.

so. Even assuming that proprietary, closed source models for innovation are faster, it does not automatically follow that they are more efficient as well.

In fact, the "winner takes it all" nature of software markets suggests that efficiency is no concern because the company winning the race can recoup the investment, not the one arriving at the same result more efficiently albeit more slowly.

Software Patents

The broadening of patent scope to allow software patents in the USA in 1981 was anything but a planned decision to revitalize innovation in a sluggish industry [90]. It made software the only product that can be protected by both copyright and patents and it gave big, established companies another weapon to wield against smaller competitors until they can afford to build their own defensive patent portfolios – if they manage to survive that long.

Today, the major proprietary software vendors have large patent portfolios that are cross-licensed with each other. An estimated 20'000 software patents *per year* have been issued in the USA by 1999, "*ten times the amount issued six years earlier*" [1], while the average costs of taking out a single software patent are in the range from 20'000 to 50'000 € [37].

The Economist observed in a piece on the patent "*gold rush*" of the past decade [20]:

> *As companies see how valuable patents can be, so the arsenals are building up. IBM is now getting ten new patents every working day. Now that software is patentable, the companies that produce it are rushing to own it [...]. And firms are no longer merely patenting things they have already made: they are using patents to colonise new areas of technology.*
>
> *This is called "strategic patenting". "You start from what you want to do," says Charles Eldering of Telecom Partners, whose business is building patent portfolios for itself and for customers, "and then you look at how you might do it." You do not even have to make the thing you want to patent, so long as you can describe plausibly how you might make it.*

In fall 2003, the United States Federal Trade Commission (FTC) released an eminent report on patent and innovation policy. The report had been in the making for several years and was based on extensive hearings. Before coming to conclusions that in their essence are highly critical of the current software patent regime [25], the FTC paper notes:

> *In some industries, such as computer hardware and software, firms can require access to dozens, hundreds, or even thousands of patents to produce just one commercial product. [...] Many of these patents overlap, with each patent blocking several others. This tends to create a "patent thicket" that is, a "dense web of overlapping intellectual property rights that a company must hack its way through in order to actually commercialize new technology".*
>
> *Much of this thicket of overlapping patent rights results from the nature of the technology; computer hardware and software contain an incredibly large number of*

[25] [13] Chapter 3, V.G, pages 55-56.

> *incremental innovations. Moreover, as more and more patents issue on incremental inventions, firms seek more and more patents to have enough bargaining chips to obtain access to others overlapping patents. One panelist asserted that the time and money his software company spends on creating and filing these so-called defensive patents, which "have no [...] innovative value in and of themselves", could have been better spent on developing new technologies.*

A common misconception claims as universally valid the hypothesis that definition, assignment, and enforcement of property rights lead to optimal outcomes. This hypothesis is quite fundamentally mistaken as far as software patents are concerned:

The fast-paced, incremental nature of innovation in software combined with the aggressive claiming of countless software patents means that any non-trivial program is bound to infringe on a number of them, leading to the situation the FTC paper describes: The lone software author who tries to play by the rules must negotiate patent licenses with a multitude of likely competitors prior to writing a program. Exorbitant transaction costs are associated with these negotiations and make an optimal outcome impossible.

The reasoning on the contrast between fast and efficient innovation applies to software patents as well. Even disregarding the role of FOSS altogether, it remains questionable whether an inflation of patents granted for miniscule, incremental improvements spurs innovation enough to outweigh at least the inevitable rise in transaction costs.

Network effects make patents that define de facto standards virtually impossible to substitute. The prospect of monopoly riches which follows from that would make negotiations difficult even if transaction costs were zero. A monopoly tends to be more lucrative than liberal licensing. The problem here is not the monopoly a patent grants by itself but the fact that there are no substitutes for some of them.

On top of that, vendors without a defensive patent portfolio run a substantial risk of overlooking and violating some patents despite their best efforts, a risk they are not compensated for by the market place because the large vendors can sidestep most of these risks.

We noted at the beginning of section 1.4.2 that the rise of FOSS was enabled by advances in technology and infrastructure that lowered information and distribution costs. An increase in transaction costs as caused by the constant threat of potential patent infringement litigation may be enough to hurt FOSS critically. Even some economists would have us believe that these costs are the result of free markets in action while they have in fact been introduced only as a result of recent regulation.

It is hard to imagine how FOSS projects – despite all their positive effects for the economy and regardless of their own innovations – could build a substantial defensive patent portfolio of their own. By and large, software patents have been little more than a nuisance for FOSS, but where they exist they have the potential to threaten the very existence of every but the largest software vendors.

Public Perception: Backlash

Many people dislike FOSS for a variety of reasons. Those who do use FOSS, though, tend to be happy with it. In 1998, the author of a Microsoft internal memo was impressed with an

international survey of large enterprises[103]:

> *A December 1997 survey of Fortune 1000 IT shops by Datapro asked IT managers to rate their server OS's on the basis of: TCO, Interoperability, Price, Manageability, Flexibility, Availability, Java Support, Functionality, and Performance. [...]*
>
> *When overall satisfaction with the OS's was calculated, Linux came out in first place. Linux was rated #1 in 7 of 9 categories in the DataPro study losing only on: functionality breadth, and performance (where it placed #2 after DEC)*

Little has changed in this regard since then. There is a caveat with those numbers, though: The vast majority of current FOSS users chose the software they use. In other words: There are no unhappy FOSS users because these people simply go back to use proprietary software.

This would change if companies changed their policies and committed to large scale FOSS deployments, especially on the desktop. It remains to be seen if user satisfaction stays up once that happens.

Also, some of those who end up paying for FOSS development – for example because they want professional support – may react angry once they realize that FOSS can very well be a commercial, profitable venture. FOSS companies are walking a thin line between making money and alienating their customers who frequently associate Free Software with "free of cost" instead of "freedom".

Security

In theory, FOSS enables the kind of competition that gives users a strong influence on the quality of the software they buy, and that includes security. One problem with software security in general, though, is that paying for secure software is like an insurance: If all goes well, nothing happens and the investment seems wasted. If the current crop of software is any indication, then software users rate features and price higher than security for most products. Security may therefore be one of those features that are easier to fund with proprietary development: a dominant software vendor might recognize it as a critical property even if few customers were willing to pay the price in a competitive market.

That source code access makes finding security vulnerabilities easier is well established. With regards to FOSS, public access to the source code may yield additional help in finding and fixing problems before they are exploited, but it also benefits crackers looking for security holes. Unfortunately, the FOSS development method is not as well suited for this type of problem as many believe.

Security holes are much better understood if they are thought of as well hidden, undesirable features rather than bugs. Regular bugs are reliably found by a large user base – if something does not work as expected, it will get reported. No qualification is needed to find common bugs, either. Security holes, on the other hand, are rarely noticed except by those determined *and* qualified to find them. On top of that, security audits are expensive: It is hard to appreciate, after all, the incremental improvement of a security audit for a small portion of the source code if the rest may remain vulnerable.

John Viega warned in 2000 of the misconceptions that are prevalent in large parts of the FOSS community [105]:

> *In fact, the wu-ftpd [Washington University ftp daemon] has been used as a case study for vulnerability detection techniques that never identified these problems as definite flaws. One tool was able to identify one of the problems as potentially exploitable, but researchers examined the code thoroughly for a couple of days, and came to the conclusion that there was no way that the problem identified by their tool could actually be exploited. Over a year later, they learned that they were wrong, when an expert audit finally did turn up the problem.*
>
> *[...The] benefits open source provides in terms of security are vastly overrated, because there isn't as much high-quality auditing as people believe, and because many security problems are much more difficult to find than people realize.*

Some FOSS projects have an excellent track record on security. Many other projects, though, might find it hard to compete if a proprietary software producer made security a top priority.

One advantage of FOSS in terms of security can be diversity: The economies of scale that favor popular programs still apply, but the vastly improved opportunities for interoperable, compatible applications mean that if customers have diverging preferences, several solutions to the same problem can coexist, which blunts the risk of monoculture that are common in the proprietary software world, a risk highlighted by renowned security researchers in a recent paper [30]:

> *The only way to stop this is to avoid monoculture in computer operating systems, and for reasons just as reasonable and obvious as avoiding monoculture in farming. Microsoft exacerbates this problem via a wide range of practices that lock users to its platform. The impact on security of this lock-in is real and endangers society.*

The Content Problem

An unsolved problem with FOSS is the lack of games, music, movies, and data collections like dictionaries.

Like proprietary software producers, content owners have been able to gradually extend the scope of their exclusive control over their works far beyond what copyright grants, often using technology to restrict their customers' rights. Content owners of course greatly prefer platforms that control their users – which is hard if not impossible to implement in FOSS.

Unfortunately, creating content is difficult using methods similar to FOSS. Works of art cannot easily be improved incrementally, and they are rarely produced to serve the creator's need. A few success stories like the free encyclopedia Wikipedia are hardly indications that popular movies will be produced FOSS-style anytime soon.

1.5. The Software Market Revisited

1.5.1. FOSS as a Strategic Weapon

The rise of FOSS has caught the proprietary software industry by surprise. Most large software vendors have embraced FOSS as yet another platform for their offerings, several are contributing

to FOSS in one place and fighting it somewhere else. FOSS is frequently used as a competitive weapon.

- IBM commands a fleet of operating systems: OS/2, OS/400, AIX, and MVS, to name a few, all of which have been losing market share to other operating systems, most notably Sun Solaris and Microsoft Windows. In 1999, the year IBM announced big investments into Linux and predicted a bright future for servers running that kernel, IBM's AIX had a 16% share of the revenue-based Unix market compared to Sun Solaris at 38% [101].

- Sun grudgingly sells servers running the Linux kernel, but the company insists that Linux has no place on servers. Quoting Jonathan Schwartz, Sun's executive vice president for software, in a 2003 interview [22]:

  > *Also, let me really clear about our Linux strategy [sic]. We don't have one. We don't at all. We do not believe that Linux plays a role on the server. Period. If you want to buy it, we will sell it to you, but we believe that Solaris is a better alternative, that is safer, more robust, higher quality and dramatically less expensive in purchase price.*

- In 2003, Sun launched a desktop initiative based on GNU/Linux, code-named Mad Hatter. In a press release, Sun explained its offering [44]:

  > *[Dumisani Mtoba, senior systems engineer at Sun Microsystems SA,] notes that the high cost of Windows software, its inherent security vulnerabilities, and proprietary code-base made compelling reasons for users to seek an alternative.*
  >
  > *"Mad Hatter represents a concerted effort to deliver a viable desktop operating system that sacrifices nothing in terms of compatibility with existing software and user-friendliness," says Mtoba.*
  >
  > *[. . .]*
  >
  > *"Industry and government agencies agree that reliance on a single vendor for desktop deployments represents an Achilles heel in the safety and security of the world's network infrastructure," says Jonathan Schwartz, executive vice-president, Software, Sun Microsystems.*

  Sun has its tiny share of the desktop market – mostly in the high performance workstation segment – attacked by systems based on Microsoft Windows and GNU/Linux.

- IBM lost its bid for desktop dominance a decade ago when OS/2 was defeated by Microsoft's Windows. While publicly downplaying the possibility of using Linux anywhere but on servers, IBM migrated 14'000 employees to GNU/Linux desktops, with some 40'000 estimated to follow by 2004 [51].

  In late 2003, IBM Global Services executive Samuel Docknevich has been quoted as saying "*Linux is ready to blossom on the desktop. Support is a big issue in the world of desktops. We're putting together a support plan for the Linux desktop. Big customers want Level 2 and Level 3 support. We're not there today but will be there next year*" [91].

- In 1998, when IBM approached the programmers behind the FOSS Apache web server to suggest a cooperation on web server development, Apache was already used on half of all web sites[26]. IBM's Internet Connection Server was running about 1% of web sites at the time.

- At 36.2%, IBM had the largest share of the market for relational database management systems in 2002 [27] . In 2003, IBM published a paper to tout the advantages of its DB2 over competing FOSS DBMSs [34]. The case study contained therein stresses the costs of a subsequent migration from a FOSS DBMS to DB2:

  > *Moving from an open-source database that cannot grow with the customer's needs could entail costly migration and retraining, not to mention infrastructure upgrades, licensing and consulting fees. These costs can be far greater than the initial savings of acquiring an open-source database system, and can more than outweigh the initial purchase costs of DB2 UDB.*

- Oracle contends that FOSS is no real competitor as far as DBMSs and related enterprise software stacks are concerned. In a 2003 article in Wall Street Journal that highlighted the threat from FOSS DBMSs, Ken Jacobs, Oracle's vice president for product strategy, was quoted as saying that the MySQL FOSS DBMS "*is certainly interesting, but I don't see it as competition for Oracle. Not now and not for some time to come. [. . .] For years I've heard people say the database is being commoditized and I don't believe that.*" [6].

  Since Oracle CEO Larry Ellison stated in June 2002 that "*we'll be running our whole business on Linux*", Oracle has migrated many of its key internal and external systems to GNU/Linux, including all its middle-tier systems – mail servers, application servers, and all the web servers [49]. All its 5000 application developers moved from a proprietary Unix to GNU/Linux [99].

  Oracle owned 33.9% of the RDBMS market in 2002 and made a public bid for PeopleSoft in an attempt to grow its stake in ERP systems in 2003.

- SAP, the leader in ERP systems by a wide margin, collaborates with MySQL, developer of the most popular FOSS DBMS, to "*jointly deliver a next-generation, enterprise-ready open source database for businesses*" [92].

- In a 2003 interview, Sun CEO Scott McNealy suggested customers use a FOSS database [47]:

  > *Then if you want to save more money, make the default database MySQL. It's free, it's bundled, you've got the whole open-source community working on making it better. If Yahoo and Google can run their entire operations on MySQL, then certainly there's a huge chunk of your operations you could run on it as well.*

[26] As far as the largest corporations were concerned, it was still trailing web server offerings from Netscape and Microsoft.

[27] Appendix D.

- Microsoft does not see an opening for FOSS on the desktop, as a server operating system, or as a DBMS. Microsoft's respective market share numbers: 93.8%, 55.1%, and 18.0% – all of them higher than in the previous year [28].

These numbers may serve as anecdotal evidence for our surmise that opinions on the viability of FOSS are strongly related to the position a company has in a particular market. Market participants are quick to realize that their competitors offerings can be replaced with commodity goods.

Proprietary software vendors defending their turf against FOSS contenders will need to adapt some of the strategies outlined earlier in this chapter:

- No proprietary software vendor can win a price war against FOSS – hence the shift to Total Cost of Ownership (TCO) arguments.
- FOSS projects cannot be bought out and shut down, either.
- Switching from one proprietary vendor to another did not affect the nature of a customer's dependency. FOSS changes that, and proprietary software vendors are faced with a new type of argument.
- Traditional FUD does not work well against FOSS. Microsoft's famous internal strategy memorandum noted this already in 1998 [29].

1.5.2. The Road Ahead

Based on standard economic theory and the known history of IT it is possible to sketch out with some certainty likely reactions and strategies of dominant proprietary software vendors in markets under attack by FOSS.

- Lobbying for laws favoring large, proprietary software vendors must look even more appealing after serious competition appeared unexpectedly. Initiatives to extend control far beyond regular copyrights are to be expected [30].
- Better Software. If we accept that monopolists can afford to lag in innovation and general software quality, the reverse should be true as well: Faced with a serious competitive threat, a monopolist will likely make improved software a high priority. The monopoly rent allows huge investments into R&D, and the resulting products may well distract customers from the unique opportunity FOSS presents: The opportunity to establish and maintain choice in software once and for all.
- Extension of existing protocols and creation of new ones. A Microsoft internal memo explains the strategy under the title "*De-commoditize protocols & applications*" [104]:

[28] Appendix D.
[29] Appendix C.4.
[30] Appendix E.

> *OSS projects have been able to gain a foothold in many server applications because of the wide utility of highly commoditized, simple protocols. By extending these protocols and developing new protocols, we can deny OSS projects entry into the market.*

With Windows 2000, Microsoft replaced its own NT LAN Manager authentication with the more efficient Kerberos, an open standard security protocol [31] which was created at the Massachusetts Institute of Technology and is available in a FOSS implementation. Microsoft's version of the protocol was slightly modified to use an optional field for its own extensions, preventing Windows 2000 systems from exchanging authentication information with machines running other implementations [60]. The information needed for interoperability was only made available after a public outcry, and then only under a non-disclosure agreement, making it impossible to add the extensions to FOSS Kerberos servers.

Reports at the time stated that "*Microsoft has yet to decide if it will license the data format so other vendors can support it in their KDCs [Key Distribution Centers, i.e. Kerberos servers] or applications*" [24]. A few months later, the company's lawyers sent threatening e-mails to those who publicly revealed the proprietary protocol amendments, complaining about "*unauthorized reproductions of Microsoft's copyrighted work*" and "*instructions on how to circumvent the End User License Agreement that is presented as part of the download for accessing the Specification*" [69].

- Integration. Powerful software vendors will integrate all their respective products as tightly as possible. The products that most users have no choice but buying serve as a lever: they will work better or exclusively with other offerings by the same vendor. The benefits of integration will be emphasized, the need for interoperability downplayed.

  In a public talk given in 2003, Microsoft Senior Vice President Will Poole described his vision:

  > *And drilling into integrated innovation – and you've, I'm sure, heard that word from us a lot, and you'll continue to, because it is absolutely fundamental to how we deliver value to our customers.*
  >
  > *[. . .]*
  >
  > *Going forward, we see tremendous opportunities for driving innovation. And again, from the Windows client perspective, Longhorn is the main place we do this. We see the opportunity to do rich communications and collaboration solutions that involve new devices, servers, communications – all integrated together and deployed easily, as a great opportunity area for you.*

  The client monopoly will be the lever to discourage the use of servers from other vendors. Customers will increasingly face one trade-off: Integration versus interoperability and choice.

[31] RFC 1510.

- In many countries, the buyer of a typical movie DVD cannot watch the movie legally using only FOSS. Doing so would constitute a circumvention of a technical measure designed to prevent unauthorized copying, which is illegal in many countries even for the buyer of the DVD [32]. Proprietary DVD players exist for a few FOSS platforms and alleviate somewhat the situation for the law-abiding citizen.

  In the future, however, proprietary software vendors may create systems that make unauthorized copying exceedingly hard, by keeping the complete stack from the hardware up to the user space application completely outside the control of the software user [33]. This may result in a situation where content – especially music and movies – is only available to users of completely proprietary, closed source systems, making it hard if not impossible to sell FOSS at least to home users.

- Software markets may experience an increase in price discrimination. David Lancashire predicted in 2001 [53]:

  > *As such, tolerating piracy may become a strategy of self-preservation for certain commercial firms, especially those seeking to establish their products as de facto market standards. Tacitly ignoring piracy - for all of its lost revenue - may yet become one of Microsoft's survival tactics, especially in countries like China which have yet to socially institutionalize open-source development networks. Once open source projects are well-established, however, high levels of piracy very clearly undermine commercial software development. It most prominently lowers the opportunity cost of coding free software over commercial applications, and also ushers in the escalating benefits of free software development modeled by Raymond and Ghosh.*

  In 2003, the government of Thailand embarked on an ambitious project to sell 1'000'000 subsidized computers to help computer literacy in the country. Out of cost considerations, the systems shipped were based on GNU/Linux. After the popularity of the project became apparent, Microsoft changed its pricing policy and offered the "*people's PC project*" a spectacular discount over regular Windows and Office prices – both in one package at the spectacularly low price of 1,490 Baht [34] [52].

In summary of our educated speculation on developments in the software market, we note that the proprietary software industry has woken up to find an unexpected, new contender in several key markets. We predict that dominant, proprietary vendors will deny the previous lack of competition to divert public attention from the fact that their own business model enables them to dominate competition and customers. Forced by alternative offerings provided as FOSS, dominant vendors will improve their own products, albeit with a focus on integration. Exclusive differentiation may also occur with video and audio content. The urge to innovate will last as long as the competition remains a threat.

[32] Appendix E.
[33] Still talking of "owner" would be a bit of a stretch under such circumstances.
[34] About US$ 37 or € 32 in November 2003.

FUD based on litigation against FOSS developers and users will increase; in particular, we expect the likelihood of patent infringement claims against FOSS to rise sharply if the European Union decides to follow the lead of the USA in allowing patents on software, because an official blessing of software patents by the EU would lower the risk of a backlash against proprietary software vendors attempting to "defend their intellectual property".

Finally, proprietary software vendors will try to prevent regulation changes that could jeopardize their current privileges. Instead, we believe they will use their economic power to lobby for the enactment of laws to favor their own business model even further.

1.6. On Regulation

1.6.1. A Call for Free Markets

We have shown that from a macroeconomic perspective FOSS is vastly more desirable than proprietary, closed source development. Evidence of the damaging side-effects of proprietary, closed source software development abounds, while it is widely accepted that FOSS is socially beneficial provided it can sustain production despite the massive change in incentives. Not surprisingly, a link to government policies has been pointed out before.

For instance, David Lancashire states in [53]:

> *Since theory informs us that public goods are always under-provided in market systems (the free-riding problem), the success of packages such as Linux in direct competition with market-driven alternatives offers vindication for policies aimed at improving social welfare through deliberate market distortion. Public policy may prove critical to the continued success of the open source movement [. . .].*
>
> *[. . .] At least on first glance the historical record seems to suggest that - over decades - certain kinds of state intervention may in fact be socially desirable.*
>
> *These questions are not merely academic: our understanding of how open source development actually works has profound implications for what kinds of corporate strategies and public policies firms and governments should pursue over time.*

Following a recurring theme in our paper, we suggest that studying the mechanisms of proprietary software markets is at least as instructive as an understanding of how FOSS development works. The key economic advantages of FOSS lie not so much in its marvelous working, but in its resilience to the systematic market failures that are built into the proprietary model. It is for that reason that our conclusions are less ambivalent than those of many other papers.

We hold that pitting market-driven against FOSS development creates a false dichotomy: The illusion that proprietary software is a free market solution while FOSS is not. What goes unmentioned here deserves particular attention: As we have mentioned earlier in this chapter, *all* solutions to the software production problem involve state interventions of one form or another. The choice for society is merely to pick a lesser evil.

The financial viability of the proprietary software production model hinges on state regulations – most notably the copyright. While *digital convergence* and the rise of the Internet made FOSS and related concepts ever more powerful, they also compromised the effectiveness of previous measures aimed at creating and maintaining excludability. A case in point is that these

days, digital content may be distributed world-wide within hours of one anonymous cracker breaking the code that was written to prevent just that. Every Internet user has access to an infrastructure that has the potential to produce thousands or millions of copies of any reasonably sized digital content.

Complaints about unauthorized copying led to the enactment of additional regulations in the past years, regulations that increased the cost of proprietary software to society even further. Unabashedly, exponents of the old software industry use monopoly rents garnered with government help to attack an economically preferable production model, deriding it for its struggling in some markets to compete with firmly entrenched, proprietary software suppliers.

Of course, the assumption underlying said regulations has been that this was the best way to ensure a sufficient supply with a Public Good. In the case of software, that assumption has been successfully challenged by an alternative production model, making a careful reevaluation imperative: Current regulations subsidize one production model and put FOSS at a distinct disadvantage. The lion share of the costs of the proprietary software production model are indirect and have been externalized. They are paid for by society. Thus, the continuation or even expansion of these privileges as demanded by the most vocal exponents of the proprietary software industry would amount to corporate welfare.

While evidence in support of massive state interventions favoring FOSS seems somewhat lacking, the case for limiting market distortions that favor a proprietary, closed source production model is compelling.

Software may well be a first order Public Good and some regulation hence required. However, it is quite obvious that the current regulation went over the top with incentives, creating lopsided relationships between suppliers and customers, and ultimately fostering software vendors content with innovating monopoly strategies.

FOSS proponents like to point out that their software is “free as in freedom” as opposed to “free beer”. In the light of what has been said it might be appropriate to add that FOSS is also “free as in free markets” – markets that depend only on a bare minimum of regulation, markets with a variety of competing offerings.

1.6.2. Public Policy

Public policy has no liability to make an industry’s chosen business model work for them, once a better alternative is available. We suggest that in order to replace state sanctioned monopolies with working markets some regulations regarding software, in particular those favoring closed source development, should be reconsidered. The main issues are:

- The stream of proposals for additional legislation favoring proprietary software vendors over their customers and their FOSS competition should be countered [35]. Existing laws that were enacted under the assumption that there was no alternative to proprietary, closed source software production should be carefully analyzed and repealed if appropriate.

- Software patents are widely believed to be harmful for FOSS and for small proprietary software producers alike. The evidence for any positive net effects software patents might

[35] Appendix E.

have is decidedly underwhelming. There is no doubt, though, that they pose a critical threat to FOSS. Both abolishing software patents altogether or exempting FOSS from patent infringements seems to be a distinct improvement.

- Software copyright protection for binary executables could be lifted. Proprietary software production would either resort to technological measures to protect their binaries or rely on plain copyright to protect their source code.

- Copyright protection for software should be cut to a period of time fitting the subject, which we suspect to be within a range from five to fifteen years.

- The weakening of the *first sale doctrine* and the *exhaustion of rights doctrine* grants additional powers to monopolists. We discussed the potential problems in appendix E on page 104. Stopping or reversing this trend seems advisable.

- As copyright protection is granted by society to encourage innovation, a case could be made for revoking that privilege from a company that was found to abuse that artificial monopoly to stifle competition and hinder innovation.

  There are a number of drawbacks to this idea. For instance, such a condition would be subject to complex interpretations and inevitably put an additional strain on the justice system.

- A state should not tie the data it holds on behalf of its citizens to a specific software vendor. Proprietary, closed source software can fulfill this requirement as long as it uses open, well documented file formats and communication protocols.

- Besides regulations, public administrations regularly influence markets with their procurement decisions. While it is controversial if the economic benefits of FOSS justify mandating the state to use FOSS whenever possible, it is arguably better than it is to, say, allow an administration to keep handing tax payers money to a company repeatedly convicted of abusing a monopoly position it acquired thanks to state regulations.

  If a state reckons its need of a specific good, there is a lot to be said for an investment that distributes the benefits – in those rare cases where an efficient method exists to do that. Software in the FOSS form is such a case, because large portions of an economy benefit from money that is spent on development and maintenance of FOSS, and the whole economy benefits from the competition FOSS brings to the software market.

  More research is needed in this area, but a policy for the public sector to prefer FOSS products over competing proprietary offerings will likely turn out to be beneficial for the economy. In fall 2003, the Danish Board of Technology joined others to arrive at this conclusion, stressing the leading role IT decision makers in the public sector could and should play in establishing open formats and protocols [76]. The report states:

> *The ordinary market conditions for standard software will tend towards a very small number of suppliers or a monopoly. It will only be possible to*

achieve competition in such a situation by taking political decisions that assist new market participants in entering the market.

[. . .]

It is therefore not sufficient for us in Denmark to follow Britain and Germany, for example, in merely recommending that open source should be 'considered'. A more active decision must be taken in those areas where there is a de facto *monopoly.*

The suggestions above are aimed at reducing the amount of regulation that obstructs competition in the software market, and to encourage FOSS production where the state has no choice but to take a stand either way – in procurement.

The next section will examine how our findings can be applied by IT decision makers in both the private and the public sector.

1.7. The Microeconomic Angle

Most people agree with the general notion that competition is good for business or at least for consumers. And in the past decade the IT industry has faced a rising tide of complaints from its customer base about a distinct lack of competition, leading a number of public institutions to investigate and find that anti-competitive practices abound indeed.

At the time of this writing, though, there is no doubt left that with FOSS a new kind of contender has established a beachhead in development labs and data centers around the globe, reminding IT decision makers of the curse of choice: New opportunities to take wrong decisions.

People who have spent their time in IT are used to change, of course. They have learned to see through the inflation of alleged breakthroughs, revolutions and paradigm shifts. So what is FOSS to an organization? A fad? The dawn of a new age? Or just yet another option?

Based on our findings in previous sections, we e will try to shed some light on important technical and business aspects of FOSS deployment from a corporate perspective.

Our focus will be on public and private sector organizations deploying third-party software or writing applications for in-house use. Companies planning to make money on the sale of software or associated services are a separate category – their stakes in software production are much higher and so are their risks, regardless of whether they choose to fight or embrace FOSS; most of them will need to reevaluate their business strategy to address the rise of FOSS, if they haven't done so already. That is clearly beyond the scope of this paper, though.

1.7.1. Considering FOSS Deployment

Despite the common scathing comments about computers causing more problems than they solve, very few companies could return them and survive. The very term "*mission-critical* application" illustrates how much the business world has come to depend on computers.

No organization would rely on a single supplier for such a crucial resource without a compelling reason. In section 1.3.2 we discussed why deployment of proprietary software constitutes such a reliance and the reasons that make choosing a dominant supplier advantageous, resulting

in market failure: Perfectly rational behavior gradually eradicates the competition that could keep a powerful vendor at bay.

It should not be forgotten, though, that proprietary closed source software once entered the enterprise as a cost-saver. Many big firms have been wary of abandoning their own custom applications and a key reason for doing so eventually were exorbitant cost and complexity of custom development to satisfy the ever increasing demand for new functionality within the company.

On the one hand, there has been a trend towards commercial off-the-shelf software, since it clearly makes no sense for every company to have their own accounting or DBMS software written. On the other hand, proprietary software purchases got customers out of the frying pan into the fire: One-sided relationship-specific investments are a serious threat both in theory and – as experience shows – in practice. Significant information asymmetries between producer and consumer make even comprehensive contracts difficult if not impossible to enforce.

FOSS is a way to solve some of the dilemmas of previous production models: Competition is reinstated in a robust form. The relationship between producer and consumer can be balanced without abandoning the benefits of using a standard software. Each user may directly influence development by underwriting or contributing development work and still enjoy the advantage of cost sharing.

But what are the costs of software to the individual purchaser? Analysts and proprietary software vendors are correct in rejecting product comparisons based on purchasing costs alone – although many of them have been touting for years the virtues of "good enough", cheap standard software. The obvious problem with the alternative *Total Cost of Ownership* analysis is that there is a lot more room for assumption, interpretation, and manipulation. Not surprisingly, the results of TCO studies are highly controversial [36]. More importantly, they fall short of the promise of comprehensiveness the "T" in TCO suggests.

1.7.2. Beyond TCO

The more serious shortcoming of TCO, though, is that the concept is still not comprehensive enough: What is the price tag on vendor lock-in, for instance? If there is some worth to the possibility to direct development and order incremental improvements – what is it? What are the costs of keeping track of proprietary software licenses to ensure no unlicensed copy is installed anywhere? In other words: How does a TCO analysis figure in the negative side-effects of proprietary software? As we pointed out in the previously, these costs are more narrowly defined in the private than in the public sector where additional factors include total economic and social costs.

Nothing short of omniscience can answer these questions conclusively since no universal answers exist that apply to all organizations – measuring TCO remains highly dependent on a specific, well-defined situation.

Traditional TCO considerations are but one of several types of arguments that can influence a decision about FOSS procurement:

- The classic TCO perspective remains important and the results can differ as much as the companies that are studied. Many individual factors need to be taken into account: License

[36] We illustrate this controversy in appendix F.

costs, update frequency, migration and retraining costs, staff attitude towards the issue, impact on IT heterogeneity, etc. Not all costs mentioned in this short list are included in standard TCO models – the examples serve to sketch out our understanding of the scope of the TCO idea.

A standard TCO analysis alone may often find that selective FOSS procurement reduces costs while improving service at the same time.

- Studying the impact of FOSS on supplier dependence and on discretion for decisions in the future is a first step beyond a limited TCO perspective.

  Generally speaking, risks and subsequent costs of vendor lock-in are closely related to the ties of an application to its data and other applications, be it through file formats, protocols, platform-specific code libraries, or investments in human capital.

  Since we expect dominant, proprietary software vendors to try and make their offerings all-or-nothing propositions while they can expect to win that game, customers should carefully evaluate each IT decision lest they find themselves entangled in an IT environment with no choice left.

  Whenever organizations decide to buy proprietary software, they should be wary of becoming dependent on features that are neither properly documented nor a commodity. However, conducting such assessments with due diligence prior to any decisions is not sufficient to address the risk of lock-in.

  A rational, powerful software vendor will anticipate these considerations and avoid inconveniencing customers too much as long as there is an alternative product. When customers grow uncomfortable enough with the behavior of a software vendor to seriously consider migrating to alternatives, they will likely find that the behavior changed only after the alternatives disappeared. Therefore, the alarm signal to watch out for is not misbehavior by a software vendor but the early signs of a market structure moving towards a monopoly or an oligopoly. Unfortunately, experience shows that mature software markets tend to gravitate towards a structure with one dominant product, which puts an end to competition unless the dominant product is FOSS.

- To achieve a better bargaining position against a dominant player, a consumer does not necessarily need actual alternatives. Decisive is what the supplier believes: All it may take is a credible threat. Whether a customer manages to impress a dominating supplier depends on a number of factors: Obviously, large or otherwise influential, high-profile organizations have a distinct advantage since they serve as an indicator of where the market is going, which we have shown to be a key criterion for IT purchasing decisions.

  Publicity is a two-edged sword. It does have the potential to make any case a high-profile case, putting pressure on the supplier. However, a supplier will likely exhaust other means before making any concessions and will likely resent the customer either way, which may not bode well for a small, still highly dependent customer.

  Also, the credibility of a mere threat to switch suppliers is hard to maintain for an extended period of time. A substantial investment in alternative solutions is arguably needed at

some point to keep the pressure up and a powerful vendor at bay, which is another aspect of FOSS procurement considerations.

- The public sector enjoys a certain independence from some social dilemmas that are prevalent in the private sector. The quality of the market conditions, regulations and any other ways public institutions influence the economy are part of the service they provide to their customers and the citizens they are ultimately reporting to.

  To call on a public institution to act in the public interest is asking it to do its job. A number of frequently cited government duties that may be relevant to the case of FOSS deployment in the public sector are listed below:

  – Education. We have mentioned the profound negative impact of closed source software on education in section 1.3.1.

  – Maintaining ability and capacity to produce and maintain critical infrastructure goods. There is no jingoism in recognizing that in a world economy that is commonly cast as a competition of nations, the arguments for the desirability of maintaining competition within a national economy most certainly apply on a larger scale as well.

    The production of proprietary software may be inefficient, but it can still be a net gain for a national economy if the local software industry manages to extract monopoly rents from foreign economies. This makes a few nations winners – for the rest, FOSS provides an opportunity to prevent the corresponding loss of public welfare.

  – Encourage competition. Abolishing regulations that needlessly foster monopolies would be preferable, but if a regulation is seen as mostly beneficial or is largely out of the hands of any individual state [37], decisions that mitigate the adverse effects caused by the regulation are preferable.

  – Overcome social dilemmas. A deal is bad for a company if it hurts the bottom line. For a public institution a deal remains good if additional gains for the citizens outweigh its own loss.

    Problems occur if such a transaction involves large-scale redistribution from one part of the constituency to another. A decision to deploy FOSS, though, will usually not only increase the overall gains but also distribute them more evenly among the population than the procurement of proprietary software.

In software procurement, there are many considerations beyond the limited scope of standard TCO studies. Additional arguments apply for the public sector. All organizations, however, will fare better in the long run if they wake up to the implications of their decisions.

[37] In the case of copyrights due to international pressure and treaties [80].

1.8. Conclusions

Software is Different

Bad regulations and production models of the software world can be traced back to historical failures in recognizing basic differences between software and other nondiminishable goods. Network effects, vendor lock-in, or interoperability issues are unknown or negligible in the areas that inspired the copyright protection for software. We discussed additional eminent peculiarities like the fast pace of innovation, the consequential rapid decay of programs compared to other creative works, the ease of incremental creation, and the impact of closed sources on education.

The special properties of software can roughly be categorized by their economic impact. For some notable examples from this thesis, we suggest:

- Economies of scale, network effects, and the fundamental transformation due to specific investments work in favor of *the dominant software*, be it proprietary or FOSS.

  These aspects can create a strong social dilemma: In many cases, software selection marks only the begin of a long business relationship with a service character. A rational customer will try to minimize the risk of losing relationship-specific investments by anticipating and choosing the coming dominant vendor. As the certainty of this prediction grows, the weight of product and service quality decreases – the utility of software is now derived from externalities like network effects.

  The licensing of FOSS eliminates the mechanism that grants the copyright owner of dominant, proprietary software a market monopoly by association, and it prevents a dominant vendor from rising the sales price to siphon said positive externalities off.

- Complexity hidden in closed source code and software patents can be used to sabotage or prohibit interoperability respectively, an effective weapon for *dominant, proprietary software vendors* to keep the competition at bay.

- Other aspects limit the *macroeconomic efficiency* of proprietary, closed source software production. Examples include the costs of creating excludability and lost opportunities in education and software reuse. In practice we also observe a massively suboptimal distribution of proprietary software due to imperfect price discrimination.

- All *proprietary software vendors* have a monopoly on incremental improvement of their products – for competing offers, the customer would have to add the costs of switching to a different product, and these costs are often not determined by licensing but migration costs. By discontinuing maintenance and support of an earlier release, proprietary vendors may even compel dependent customers without need for further functionality to buy updates.

The list above indicates why open standards provide a false sense of safety: Customers will successfully insist on them as long as the market is competitive. However, factors other than lacking interoperability suffice for the market to reach a mature stage with a dominant vendor. At this point, customers will likely see the vendor's commitment to open standards fade.

We have shown that current regulations encourage production models that allow some vendors to dominate customers and competition. However, the software industry has yet to settle on the most efficient production model. A few decades after coming into existence, information technology continues to dramatically change some fundamental economic conditions, and it stands to reason that few industries are more affected than software production and distribution themselves.

Unfortunately, the dominant, proprietary software vendors insist on a production model that promises an ongoing flow of monopoly rents – and as far as the general public is concerned, these companies *are* the software industry.

We believe that two factors will largely determine the future of FOSS:

- The direction regulations will take: Software patents are the most imminent threat – the decision about their recognition is pending in the European Union and when taken will serve as a signal to other countries.

  Section 1.6.2 discusses other regulations that affect the relative advantages of FOSS and proprietary software.

- It is unclear at this point whether IT decision makers seize the chance to wrest control of software production from their suppliers as sketched out in section 1.7.2. Most FOSS deployments seem to be motivated by TCO considerations rather than by the desire for vendor independence.

  In our view, great responsibility rests with the public sector because additional criteria speak in favor of FOSS use in that area. While we share the skepticism of most economists with regards to states directing production, we argue that states can and should take substantial implications to the economy into account when making procurement decisions.

Future Research

A ruthless meritocracy. A disparate, international community of people publishing the results of their research. Knowledge generated incrementally and submitted to peers for review. The FOSS community is often portrayed in the same terms as the scientific community. FOSS works because software development is more akin to scientific research than to, say, writing a novel.

However, FOSS was not created by science: Practitioners explored licenses, development methods, and business models by trial and error. For several years and in best scientific tradition, both economics and software engineering scholars have been busy reconciling their models with the fact that FOSS has become hard to ignore. The scientific community is moving towards a better understanding of the inner workings of FOSS.

Mainstream economics appears fairly puzzled by the production of nondiminishable goods. This is not too surprising because state interventions are in practice eminently political if not emotional decisions: How much national defense, public fireworks, or subsidized culture a society deems a sufficient supply is hardly determined by scientific means. Which takes us to the next problem: What is a sufficient or appropriate supply of the nondiminishable good "software"? Is the current plethora of proprietary software expression of an actual demand, or is it the result of regulations that divert an abundance of resources into one industry, with venture

capital flowing on the odd chance of cracking the jackpot and earn monopoly riches? We note that if we are to compare the output of different production models, we ought to have criteria to evaluate the resulting quantity and quality.

Our reading of papers in this field confirms our view that at this point, economic theory can provide valuable insights and explanations but hardly conclusive predictions or solid advice – as evidenced by our experience that there is no position on FOSS too outrageous to have some economics scholars equipped with models and statistics to support it. In this thesis, we have not applied economics to gain amazing insights. Instead, we used familiar models and terminology as tools to describe what any avid observer of the IT industry could have noticed. We cited examples taken from the real world and appealed to common sense because we are deeply suspicious about any attempts to use the framework of economics to support arguments for or against FOSS. Consequently, we make no claim in proving the superiority of FOSS with this thesis. However, we presented evidence that indicates that FOSS *might* be superior, and based on such evidence we call for expanded use of FOSS which is a prerequisite for a better evaluation.

What about the voice of academia, though? Unconventional ideas for a key technology of the future, promising first results, and no evidence of major, irreparable damage looming: Scientists should fall over themselves to suggest further experiments, and be it only to gather enough data to determine whether their models need adapting. And that is indeed what we consider the most important point about software and economics: The success of FOSS so far warrants additional research, but the empirical data is lacking because the environment – traditions, economic power, regulations – is biased towards proprietary software. Economists may not have the models or methods to conclusively determine the merits of competing software production models, but they certainly have some influence in public policy decisions.

As for actual research, we recommend business models as a research subject for economists. Companies have tried numerous methods for generating revenue to sustain the creation of FOSS. They need to work in an extremely competitive environment: Customers are not dependent on any single supplier, even the largest vendors cannot command a monopoly rent, competitive advantages tend to be transient. Moreover, FOSS vendors are usually competing with proprietary vendors who are not operating on similarly razor-thin margins. The environment is so harsh, in fact, that the main criticism of FOSS remains that a sufficient supply (however defined) may not be sustainable. The exact circumstances under which some FOSS business models work and others fail are unclear at this point – extensive, current data and analysis should be instructive from an academic and a practical point of view.

In the long run, it will be most interesting to see where proprietary software can survive. Our findings suggest that FOSS will have a clear advantage where the proprietary model resulted in market failure: Standard software sold by a dominant vendor in high volumes and at a steep premium. We expect proprietary software to remain strongest where incremental development is difficult.

Part II.

Coping with High Memory Load in Linux

> If we knew what it was we were doing, it would not be called research, would it?
>
> *(Albert Einstein)*

2. Linux Performance Aspects

2.1. Introduction

This chapter provides an introduction into Linux performance with a focus on scalability. In section 2.2 we discuss aspects, quality, and dimensions of scalability in the context of FOSS and Linux.

Section 2.3 illustrates the scope of scalability work with examples from the development cycle that led to Linux 2.6.0. We estimate future challenges in this area and comment on the recent attitude change of Linux kernel developers towards high-end scalability.

2.2. Linux Scalability

Scalability is one of several areas of interest for enterprise operating systems. The *Carrier Grade Linux Specifications Subgroup* – a multi-vendor working group defining requirements for improved viability of Linux and related components in the telecommunications industry – lists it along with other criteria like availability, serviceability, and security in its *Carrier Grade Linux Requirements Definition*[9] and defines the term as:

> *A requirement that supports vertical and horizontal scaling of carrier server systems such that the addition of hardware resources results in acceptable increases in capacity.*

2.2.1. Up and Down

Scalability is usually understood to refer to scaling *up*, typically expressed in numbers of CPUs or nodes a system supports before the marginal gain of adding more resources becomes too small to be useful. An equally important criterion, though, is the scaling *range* covered – a major reason for the success of FOSS has been its ability to scale not up but down. Previously decommissioned machines often see a second life as headless servers in cash-strapped IT departments. The increasing use of GNU/Linux in small embedded systems attests to its capability for running on hardware resources an order of magnitude below a current entry level personal computer, and that gives it an edge over competing proprietary operating systems (FOSS systems other than GNU/Linux exhibit similar strengths in this area and others).

It is characteristic for Linux that the current 2.5 *development cycle* included not only significant scalability advances for NUMA hardware but also merged uClinux. While external patches for MMU-less Linux date back to 1998, 2.6.0 is the first official, stable Linux kernel to run on microcontrollers without Memory Management Units.

2.2.2. Linear Scaling

As a design goal linear scaling is the Holy Grail of scalability work [1]. And yet, achieving exactly that may in fact indicate a lack of optimization.

Adding SMP capabilities to a kernel, for example, requires synchronization mechanisms like *spin locks* or semaphores. A high scalability target for such a kernel requires fine grained locking to minimize lock contention. If only one version of a kernel exists, the result is more likely to display linear scaling simply because the overhead remains the same regardless of the underlying hardware. Of course that also means the kernel runs slower than need be on low-end hardware. The Linux kernel source can be configured to compile a uniprocessor (UP) kernel where spin locks collapse to empty statements, which allows for better performance on UP systems but makes strictly linear scaling impossible because the extra overhead required for machines with more than one CPU is not imposed on UP systems.

That said, one focus of Linux kernel development has been to find low order algorithms for frequent operations, and machines well below the high-end benefit from some of these scalability improvements.

2.2.3. Vertical vs. Horizontal Scaling

Linux 2.4 works well for 4-way systems and the new version 2.6 is expected to increase that limit to 16 CPUs. Patches exist to allow Linux vertical scaling well beyond, but they are not seen fit for merging into mainstream Linux at this point. In fact, during the writing of this paper Silicon Graphics announced the sale of a 256-processor single-system image Linux machine for technical computing to NASA.

Only few applications need more than four CPUs working on the same data set and even fewer want to go beyond 16 CPUs – huge data bases are the prime example.

Most tasks in high performance computing can be solved with computer clusters. For horizontal scaling on cheap commodity hardware software costs often become a concern and so does support for scripted remote administration. Not surprisingly, one early claim of fame for GNU/Linux was the Beowulf[97], ...

> *a kind of high-performance massively parallel computer built primarily out of commodity hardware components, running a free-software operating system like Linux or FreeBSD, interconnected by a private high-speed network. It consists of a cluster of PCs or workstations dedicated to running high-performance computing tasks.*

Today, computing clusters based on FOSS have found their place among the most powerful super computers. Half of the top ten in the November 2003 "*Top500 Supercomputer Sites*" list, a ranking based on the Linpack benchmark which solves systems of linear equations, are Linux clusters [102].

[1] Obviously, even superlinear scaling is possible *for selected workloads*, for instance if adding CPUs gives independent processes a private CPU cache each. Ideally good scaling is achieved for all types of work loads, though.

2.2.4. Userspace Scaling

Many enterprise-class applications can distribute their work load among several servers. Simple solutions include round-robin DNS load balancing for semi-static web content; more sophisticated server clusters handle even the most demanding web and file serving needs, and in a multi-tier infrastructure, the same holds true for application servers. Computation like simulation are the realm of cluster computing these days.

At the other end of the spectrum database clusters are still somewhat of a novelty item, and huge databases with strict requirements for data consistency and integrity have remained a stronghold of vertically scaled IT infrastructure – without those requirements clusters win again as the most popular Internet search engine of the past few years demonstrates. High-end SMP and NUMA machines, however, come at a premium high enough that competing commodity-based solutions can afford to be fairly inefficient and *still* be the economical choice.

One consequence of the importance of cluster computing is that server operating systems – provided they are not meant to run on big iron exclusively – should run well on those machines with the best price/performance ratio. The ration depends on the work load, but if market forces are any indication the answer for most tasks is an x86 1-way to 4-way machine, despite all the well-known limitations. Further load distribution and synchronization issues are often handled best in user space.

Thus, it is largely irrelevant even for many of the most demanding enterprise-type applications how far up exactly an operating system scales – any modern server operating system will do as far as that is concerned. In fact, for tasks that can be solved through clustering high-end scalability in a kernel can be a *handicap*. This is true whenever design decisions have conversing performance impacts for different classes of machines. With trade-offs being the famously recurring theme in Computer Science literature this problem cannot easily be dismissed.

Consequently an ideal kernel for many types of cluster systems provides little more than easy and robust access to hardware resources while its main virtue is getting out of the way.

2.2.5. Scaling by Hardware Architecture

For most major operating systems, there is little choice in terms of hardware platforms, if any. The way to scale is to add more and faster CPUs, or to switch to a different operating system with different APIs altogether. The Linux kernel and the vast majority of FOSS applications on the other hand have been ported to a wide range of hardware platforms. Applications can be developed on standard workstations and then be deployed on handheld computers, 64-bit 4-way cluster nodes, or mainframes. This means that hardware can be picked to fit the task rather than the operating system.

FOSS hardware agnosticism offers unprecedented opportunities for scalability. It should be noted, however, that the main focus for the popular FOSS *kernels* is x86 – they tend to run quite well on other hardware architectures, but they are rarely a match for a competing operating system that makes no compromise to achieve maximum performance on that single platform. Since the kernel does most of the heavy lifting of abstracting the hardware specifics, the problem is much smaller in userspace. This can be observed in the real world: The web server of choice may well be the FOSS product Apache, but on a SPARC machine, for example, it will likely run

on top of Solaris, not Linux.

2.3. Beyond Processing Power

While CPU scalability has traditionally taken the limelight, server operating systems must scale in many areas which – depending on the task at hand – may well make CPU scalability insignificant in comparison.

To demonstrate the scope of scalability work we point out a number of scalability changes that went into the Linux kernel during the 2.5 development cycle:

- A Linux specific variant of the standard Unix poll mechanism has been introduced: epoll allows a process to simultaneously watch and service tens of thousands of sockets with O(1) overhead [58].

- With Linux 2.4, a user could not be member of more than 32 groups. Some users needed that limit lifted to allow for over 10^4 group memberships per user.

- The old scheme with 8 bit major and minor numbers imposed a limit of 256 devices per device type which is a problem, for instance, when thousands of disks are attached to a single host. Since device numbers are exposed to user space, Linux 2.6 provides 12 bit major and 20 bit minor numbers in a backward compatible manner.

- Under Linux 2.6, a process with appropriate privileges can request large pages: 4 MB instead of 4 KB on x86. This not only reduces the amount of work setting up, maintaining, and tearing down page table entries by orders of magnitude, it also vastly increases the range of memory the translation look-aside buffer (TLB) covers which makes an expensive TLB miss less likely.

- A new O(1) process scheduler reduces the overhead for systems running a large number of processes or threads [73].

- A lot of effort went into improved threading support, which has traditionally been a weak point in Linux: a common sentiment among kernel developers was that Linux processes were light-weight enough to be used as threads, that having more threads than CPUs in a machine was simply a misguided approach to programming, and that the POSIX threads standard was broken anyway. The recent work in this area may be more due to market demand than to a fundamental change of perception in the developer community. Besides better POSIX compliance, good scalability to many CPUs and to large numbers of threads were among the primary design goals for the new Native POSIX Thread Library (NPTL) for Linux[19].

- Read-Copy Update (RCU) mutual exclusion was introduced into Linux 2.5. RCU is a two-phase update method for mutual exclusion. It excels where data is mostly read because readers do not have to acquire a lock [64].

- One weakness of Linux 2.4 network drivers is that they break down under extreme incoming traffic due to an interrupt livelock. The network cards raised an interrupt for each of tens of thousand of incoming packets per seconds – the CPU was busy executing the interrupt service routine and processing the packet only to drop it later on because user space never got a chance to remove any packets from the queue. A New API (NAPI) in Linux 2.6 addresses that problem by replacing the interrupt driven behavior with a polling model under load [36].

A better understanding of current hardware and common programming practices often lead to substantial design changes – most examples listed above fall into this category.

Moreover, advances in hardware technology will continue to challenge operating system designers in the foreseeable future. For instance, the recent introduction of simultaneous multithreading (SMT) in mainstream processors multiplies the number of logical CPUs a scheduler sees. In addition, any SMP system using SMT CPUs exhibits NUMA properties: logical CPUs share caches if they are on the same chip. Thus, if a scheduler decides to move a process to a different CPU run queue, it must be aware that migration costs vastly differ, and it should avoid ending up with all CPU siblings of a chip busy while other CPUs are idle. This is one place where high-end scalability work trickles down to machines that – while not exactly on the low-end – certainly don't qualify as high-end, either.

A mere difference in speed of progress for different hardware components means solutions that used to be perfectly balanced once may need to be reconsidered. From one perspective, for instance, memory is more expensive today than it has been for many years; the ever increasing gap between CPU and RAM speeds raised Level 1 and 2 CPU cache sizes to important design considerations for operating system designers. In terms of the subject at hand: The size of in-kernel working sets and data structures have a huge impact on scalability. Kernel memory usage tends to increase with the resources it manages. If it regularly walks its list of processes or page frames, it evicts more and more pages from the cache causing expensive cache misses somewhere down the road.

Solutions yield new problems as well. Besides the obvious fact that eliminating one bottleneck tends to expose another, some new features move from exotic to required when systems grow. Two areas that are frequently affected are availability and reliability: If a system has to be shut down for maintenance every time a component fails, then it will experience *more* down time as its growth increases the number of possibilities for the system to fail – hence the increasing demands to provide extended hot-plugging support for disks, CPUs, and even memory.

2.3.1. Challenges

Out of 142 requirements in the current version of the Carrier Grade Linux Requirements Definition we quoted before, 21 are clustering requirements and a single one is labeled as a scalability requirement [9]: Efficient low-level asynchronous events, a problem Linux 2.6 addresses with the introduction of epoll as we pointed out above.

The list of features for Linux in 2004 proposed by the *Data Center Linux Technical Working Group* contains only a small number of scalability improvements [15]. Combined with issues

discussed at the 2003 (Linux) Kernel Summit [59], a list of the most pressing scalability problems would likely include these:

- The current implementation of large pages requires applications to explicitly request them. Also, currently memory fragmentation can prevent applications from successfully allocating large pages unless they request them soon after the machine booted. An implementation that is transparent to userspace would make writing and porting applications using large pages easier.

- On big systems with thousands of physical devices attached persistent device naming becomes a requirement. Devices should, for instance, never be renamed just because another device was added or removed.

- Investigation of different allocation schemes for NUMA architectures. For example, a process may get memory preferentially or exclusively on its home node, and memory allocations for one process may rotate through all or several nodes. Currently it seems those decisions will not be left with the application but with the system administrator *if* better performance makes the increased complexity look worthwhile.

Most documents concerned with generic Linux scalability seem to date at least a year back, and they tend to call for solutions that have been implemented in the meantime ([95] is a typical example). We believe this is no coincidence. Scalability work has been an ongoing process with Linux for most of its existence, but in the time frame between 2.2 and 2.6 major IT vendors have undertaken substantial efforts to make this kernel scale – usually in order to make it work across their range of hardware offerings.

2.3.2. Scalability Limits?

Leading Linux kernel developers have always stated their goal is not to conquer the market with the highest margins on hardware, but the commodity market for both server and desktop systems. Patches helping Linux grow in other places are accepted only if they don't hurt the primary objective. And as far as scalability goes, Linux is most certainly ready for whatever may pass as commodity hardware in the next few years.

Only recently did it become apparent that mainstream Linux may eventually scale to high-end hardware. Linus Torvalds explained in a Q&A session in 2003 [7]:

> *I used to think that it made no sense to try to support huge machines in the same source tree as regular machines. I used to think that big iron issues are so different from regular hardware that it's better to have a fork and have some special code for machines with 256 CPUs or something like that. The thing is, the SMP scalability has helped even the UPKs [sic[2]] just by cleaning stuff up and having to be a lot more careful about how you do things. And we've been able to keep all the overheads down.*

[2] A transcript error. Should be "*UP case*" – the uniprocessor case.

> *So that spinlocks, which are there in the source, just go away because you don't need them. We're scaling so well right now that I don't see any reason to separate out the high end hardware. A lot of the reason for using Linux in the first place ends up being that you want to ride the wave of having millions of machines out there that actually incorporate new technology faster than most of the big iron things usually do. So the big iron people want to be in the same tree, because having a separate big tree would mean that it wouldn't get the testing, it wouldn't get the features, it wouldn't get all the stuff that Linux has got, and that traditional Unix usually doesn't have.*

Especially with the massive backing by industry heavy-weights, there seems indeed little reason to believe that there are any major scalability issues that the Linux kernel development community could fail to address. At the same time, a good number of vocal developers and users of Linux on low-end hardware make sure high-end scalability is not achieved at their expense.

Therefore, the real danger of Linux scalability work may lie not in diametrical performance impacts on different systems but elsewhere: Increasing code complexity in parts of the kernel, for instance due to the proliferation of fine-grained locking and better support for high-end architectures, raised the bar for understanding the underlying mechanisms significantly. This may be most obvious in the areas of virtual memory management and process scheduling. Compiling code sections conditionally only for the high-end hardware where they are useful fixes the performance trade-offs, but doesn't improve code readability.

While becoming early on much more than the educational toy Minix is, Linux maintained the appeal of a kernel which looked very familiar to anybody who ever worked through a standard Unix textbook. This has been gradually changing. Of course it can be argued that it is even more exciting to study the code of a high-end kernel, or that a raised bar will keep away those people who should never have taken up programming in the first place, but there is no denying that in recent years it has become increasingly difficult for amateurs to make substantial contributions to the core of the Linux kernel code, and one major contributing factor was complexity introduced for the advancement of Linux on hardware that is hardly going to be a commodity anytime soon.

Understanding the Linux kernel and writing code for it may have become a daunting challenge to many amateurs, but there are reasons for full-time developers to be impressed, too:

- The source distribution for the Linux kernel is rapidly approaching 200 MB, containing over 6000 C source files. Discussions on design and implementation take place on dozens of mailing lists concerned with various aspects of Linux kernel development, with the main list carrying 200 messages a day. A sustained rate of 180 changesets per day in the main source repository tend to overwhelm newcomers and make it hard to keep up with development [67].

- FOSS projects often face additional complexity because in some regards they tend to have more ambitious goals than their proprietary counterparts. For one, most popular FOSS projects are maintained on a wide range of different hardware platforms simultaneously.

  In user space, hardware issues like endianess and word width are rather trivial to solve compared to the problems posed by numerous subtle differences and bugs in system tools

and libraries on the supported platforms (many of which are not FOSS themselves). And members of the FOSS community have developed tools to automate most of this work and writing portable applications is standard practice nowadays.

In kernel space, all the hardware differences are not only clearly visible but need to be dealt with to provide the hardware abstraction applications have come to expect. Consequently, most of the complexity caused by cross-platform support is found in operating system kernels.

2.4. Conclusion

Starting from the common notion that good scalability means linear scaling to high-end hardware, we introduced a different interpretation that is more popular among FOSS developers: Good scalability means optimal performance on any hardware configuration. In case of trade-offs, optimizations favor common rather than high-end hardware. And the significance of the upper scalability limit is trumped by the hardware *range* covered, both in terms of machine size and hardware platforms.

The release of version 2.6.0 is by no means the end of Linux scalability work, but the evidence suggests that little immediate urgency remains. For most tasks, horizontal scaling is more economical; high-end vertical scaling will not emerge from its niche in the foreseeable future. While predictions are always hard to make due to the free-wheeling nature of the Linux kernel development process, we expect the main focus to move to other areas like serviceability.

The perception that high-end hardware should not be supported in mainstream Linux has changed. While the evidence indicates that performance on the low end need not suffer, we suspect that the resultant complexity increase of source code creation and maintenance may not be fully appreciated yet.

3. Thrashing and Load Control

3.1. Introduction

We start off in section 3.2 with a description of how resource allocation has been affected by changes in hardware, software, and usage patterns.

Section 3.3 introduces a model of thrashing used by Peter J. Denning in 1970. We extend it to account for the positive effect of additional processes on system throughput and discuss additional factors that influence the performance of a virtual memory system under load.

Section 3.4 presents three categories of methods for operating systems to take decisions that are not predetermined by standards or protocols.

In section 3.5, we look at aspects of Linux resource allocation that pertain directly to paging and thrashing: Process scheduler, virtual memory management, and I/O scheduling. In anticipation of section 3.8, we focus on significant changes between Linux 2.4 and 2.6.

Section 3.6 describes the benchmarks we used to study system behavior under high memory load.

In section 3.7, we present our own load control implementation for the Linux kernel and the lessons we learned from that project.

In section 3.8, finally, we offer a systematic study of system behavior under high memory load for all Linux kernels from 2.5.0 to 2.6.0. We demonstrate how the data we gathered can be used to track down and fix performance regressions, and we discuss the role of unfairness and our experience with performance figures provided by the Linux kernel.

3.2. Trends in Resource Allocation

Allocation schemes that maximize throughput fell out of favor with operating system designers after timesharing systems were introduced in the 1960s – response time became a critical quality. Computer users needed at least a timely acknowledgment of the order they had given, and they were likely to grow impatient if the result took considerably longer to arrive than expected, making a dynamic approach to fairness and user quotas desirable. The decision of when to start a task was now up to the user which made load control that much harder.

The importance of large multi-user systems dwindled when desktop computers replaced dumb terminals. Users became decoupled from a centrally shared, scarce resource. For desktop systems, though, the need arose to concurrently execute several applications for the same user. On these systems, perceived interactivity was the new dominant performance criterion, while both throughput and fairness became mere side conditions. The term *multitasking* is sometimes used to differentiate this scenario from traditional, throughput-oriented flavors of multiprogramming.

On the server side, hardware below the high-end became a commodity. Large machines continued to sell at a steep premium, though, which was a crucial factor in making servers dedicated to a specific task economical in many situations – a fleet of specialized servers with each essentially running one program was cheaper to obtain than a single machine capable of handling the accumulated load.

Thus, most of the user multiplexing has been moved out of the operating system space. In a multi-tier client/server environment, only one component could not easily be spread among a multitude of computers and still defines the core of a typical enterprise IT center: The database. Modern DBMSs, however, are largely self-contained, that is they frequently bypass the operating system and do, for instance, their own page management and I/O scheduling.

Consequences

At first glance and as far as servers are concerned, the need for sophisticated resource allocation strategies at an operating system level seems to be less distinct than it used to be. On the other hand, several developments kept the field challenging. Noteworthy examples include:

- Threads: Common server problems that used to be coded as *state machines* or *coroutines* are increasingly written using threads, shifting responsibility for resource allocation back to the operating system.
- Convergence: The ratio by which client systems outnumber specialized servers translates to a solid advantage in economies of scale for the former. The successors of smart terminals started to offer services of their own. The clear separation between server and client operating systems faded.

  Today, the majority of the server market belongs to hardware and operating systems that hardly differ from standard desktop systems in terms of underlying principles and architecture [1].
- Single System Image (SSI) Clustering: The goal common to many efforts in this area is to make a number of interconnected machines appear like a single system. Such functionality tends to reside at least partially in the operating system kernel, especially if the SSI illusion is to be presented to unsuspecting application programs.

Emphases differ, but all modern, common operating systems try to strike a reasonable balance between interactivity and throughput.

3.3. Thrashing

3.3.1. Models

With the proliferation of multiprogramming and virtual memory as features in commercial operating systems of the 1960s, thrashing became a well-known phenomenon: With increasing load,

[1] Appendix D.

the process scheduler seemed suddenly unable to find runnable processes, the CPU was mostly idle, and consequently throughput collapsed.

The surprising property of thrashing was not that I/O could become a bottleneck on a computer or that the paging activity of a virtual memory system led to additional disk activity, but that a system running a number of processes could suddenly tip over, resulting in heavy I/O activity but dramatically lower system throughput, while all processes were blocked waiting for pages from the paging disk.

In a seminal paper Peter J. Denning described the conditions that lead to thrashing and a method to prevent it [17]. For a simple model he defined [18]:

Δ The memory reference time is "*measured between the moments at which references to items in memory are initiated by a processor; it is composed of delays resulting from memory cycle time, from instruction execution time, from 'interference' by other processors attempting to reference the same memory module simultaneously, and possibly also from switching processors among programs*". The *average memory reference time* is called Δ. Denning suggested in 1970 that a typical Δ was 1 μs.

T The transport time is "*the time required to complete a transaction that moves information between the two levels of memory; it consists of delays resulting from waiting in queues, from waiting for the requested information transfer to finish, and possibly also from waiting for rotating or movable devices to be positioned*". The *average transport time* is called T and according to Denning was at least 10 ms.

m_i The amount of memory available to process i.

$f_i(m_i)$ The probability that a memory reference by process i will cause a fault and thus make an access to the backing store necessary.

$d_i(m_i)$ The "*expected fraction of time [program i] spends in execution*".

$$d_i(m_i) = \frac{\Delta / f_i(m_i)}{\Delta / f_i(m_i) + T} = \frac{1}{1 + \frac{T * f_i(m_i)}{\Delta}} \tag{3.1}$$

Denning elaborated the model further. To describe his findings, he used the notion of *working sets*: The set of pages an application had referenced within a certain time span [2], a concept based on the observation of reference locality: Since memory references are not random, knowledge about past behavior can be used to predict likely future references. Denning suggested to reserve per-process working sets just big enough that no process was significantly slowed down by page faults. He showed that thrashing occurred if these working sets for the currently running applications did not fit into memory. Thrashing, according to Denning, occurred when overall CPU usage decreased dramatically with the addition of one more process. Solving the problem was simple enough:

> *If we are considering adding the* kth *program to memory, we may do so if and only if [. . .] there is space in memory for its working set" [18].*

[2] The window measures application time or *virtual CPU time*, that is the time the application spent on the CPU.

Unfortunately, additional complexity is needed when programs change the size of their working sets. Most current operating systems sidestep the problem by employing global page replacement algorithms.

Many later papers concurred that the key to solving the thrashing problem was . They tend to fall short of describing a mechanism, though, or propose mechanisms based on admission control or selecting victims and aborting them [39]. These are proven mechanisms for transaction processing systems, but they are not applicable for desktop systems or for servers with similar properties where both starting and stopping processes are done by the user and hence exogenous as far as the operating system is concerned.

An Alternative Model

We introduce a somewhat different perspective on thrashing using the definitions above plus:

q_i The fraction of CPU usage caused by process i. Thus $\sum q_i = 1$, regardless of the total CPU usage.

a_i The probability of disk access operations other than faults by process i during the time period Δ. This value does not depend on m_i but is a property of each process i.

$R(k)$ The CPU-I/O ratio of a system.

$$R(k) = \frac{\Delta}{T \sum_{i=1}^{k} q_i(f_i(m_i) + a_i)} \tag{3.2}$$

If there are processes running, then R can be smaller than 1 if and only if all of them are I/O bound. I/O requests may, however, coincide to make all processes block, arguably a rare event except for systems with an extremely high I/O load.

The denominator in (3.2) calculates the time that is needed to satisfy all I/O requests during Δ. If it is larger than Δ itself the CPU will stall. Obviously, this plays out over a much longer period of time than Δ and with $\Delta/T \gg 10^4$ the $\sum$ must add up to a very small number or the CPU will be mostly idle.

Thrashing was observed as a collapse in performance at some point when a system load that was already high increased slightly. In a system running several programs with virtual memory, starting an additional process shifts a portion of memory for existing processes to the paging disk which increases the fault probability for these programs.

Using some sort of LRU mechanism to pick pages to evict from memory reduces the number of page faults over purely random mechanisms because the best candidates for future references are kept in memory. However, this also means that the growth of the fault probability f_i *accelerates* when m_i shrinks. That is, for $a > b > c$ with $a - b \leq b - c$:

$$f_i(a) \leq f_i(b) \leq fi(c) \wedge f_i(b) - f_i(a) \leq f_i(c) - f_i(b). \tag{3.3}$$

Our model takes into account that throughput does not necessarily suffer if a number of processes are fighting for I/O as long as there is at least one process j that is CPU bound. Its q_j may approach 1 and the other processes may not make any progress, but the overall system

throughput remains high until the last process blocks waiting for I/O, too. For a long time adding processes actually helps preventing a performance loss from the fault rate picking up steam – assuming that all programs are of equal importance. This is one key reason that performance collapses at some point instead of slowly deteriorating when the load increases.

Initially, only those processes with long breaks between some of their memory references are affected. But with a higher load only frequently referenced pages will remain in memory, which affects a growing number of processes. Under a global page replacement policy and intense memory pressure, a process i resuming after waiting for I/O will often find its m_i was shrunk by other processes acquiring memory in the meantime, raising the fault probability $f_i(m_i)$ even further. This effect is exacerbated once I/O is contested and starts building a backlog and T, which was defined to include "*delays from waiting in queues*", takes up growing on its own.

In a multiprogramming system with virtual memory I/O suddenly becomes the determining factor for system throughput when $R(k)$ falls below 1. And our discussion of the behavior of fault probabilities explains why R, and hence performance, may collapse rather than just slightly shrink.

On T and Δ

T has barely changed in the past decades: Disk latency is governed by mechanics, and the improvements in that area pale compared to the exponential growth of CPU and – to a lesser extent – memory speed since Denning described thrashing first in 1968. A typical access time of a hard disk is still well over 1 ms although disk arrays can be used to improve it. While hardware is the determining factor for T by and large, though, it should be noted that the operating system kernel exerts a significant influence as well [3]. The fact remains that the *access time gap* between disks and RAM is ever widening [40], and even RAM has failed to keep up with the speed increases of CPUs. Δ is at least two or three orders of magnitude smaller than thirty years ago, which – looking at (3.2) – makes R even more sensitive to high I/O access rates.

The trend towards dedicated servers we mentioned before affected thrashing in two ways: Running only one service on a machine reduced the thrashing risk but is also made traditional cures virtually obsolete at the operating system level: If such a machine is thrashing then it is the respective server program that is qualified to take action – it knows all about the specifics of the service, unlike the operating system. A multithreaded DBMSs may, for example, decide to abort transactions based on their age or priority.

3.3.2. Modern Strategies

If technology development made thrashing more likely and modern computer usage patterns rendered traditional thrashing prevention methods obsolete, why isn't there more recent research activity in this area? To a large degree, the answer is that the evolution of hardware technology changed the preferred method for thrashing prevention as well. In today's computers, reliance on massive paging is rare and thrashing an unexpected condition – not because the problem has been solved on the operating system level but because the economic solution to the problem has become to throw more memory at it.

[3] See section 3.5.3 for the discussion of I/O scheduling.

The growing access time gap between disks and RAM combined with a tremendous price drop for memory *capacity* had immensely powerful I/O subsystems that are typical for mainframes lose much of their importance: RAM is used as a disk cache.

If adding memory is not an option, (3.2) indicates some options for influencing $R(k)$ besides changing the number of processes k. For instance, an operating system may be able to prevent thrashing for a specified load by having the process scheduler favor certain processes in order to lower the I/O load. A good candidate process i does little I/O (a_i) and has a small fault probability f_i.

Another approach is to stop trying to minimize the number of *major page faults*, that is the number of pages that have to be written to and read from a paging disk. This approach has its own problems, though, if the bottleneck is not the bandwidth of the paging disk but head seeks.

There are many conceivable modifications of a pure LRU based page out algorithm:

- Paging out contiguous data clusters comprising several pages in order reduces the number of disk head seeks.
- Page fault probability can decrease if additional memory is freed and pages next to a faulting memory address are read at the same time. This strategy called *read-ahead* tries to anticipate likely future faults and is another application of reference locality.
- An I/O scheduler can reorder block I/O requests to improve throughput by reducing the number of disk head seeks.
- Pages that have an existing backing store on the disk are cheap to free if they are clean, that is if their image on the disk reflects the current state in memory – they can simply be discarded, saving one write access to the disk. The most common example are pages containing executable code ("*text*" in Unix terminology).
- Shared pages are used in all modern operating systems, both as shared machine code and as means of interprocess communication. A per-application LRU mechanism will routinely underestimate the importance of pages that are shared among a number of applications, since pages may be freed based on the working set of one application although other processes frequently use them. This results in more disk activity and a lower throughput for the affected processes. It is therefore important to keep pages that are "popular" on a global level in memory. With shared pages, picking a page to discard is clearly a global problem: It requires knowledge gathered from all processes.

  Even a global page replacement policy based on LRU tends to evict shared pages too easily. This is because the current position of a shared page in an LRU ordered list reflects merely the position for the most recent user. Other processes that used the same page slightly longer ago do not influence the positioning at all despite their obvious impact on the likelihood of future references to that particular page.

  Also, carelessly freeing shared pages creates a new risk: It is conceivable that all processes end up blocked, waiting for the same shared page.
- The LRU approximation that is common today – Not Recently Used (NRU) – relies on *reference bits* which are set by the processor when a page is used. The kernel regularly

checks and clears those bits. The information on what pages had their reference flags set can be used to determine which pages have not been used in a while, hence the name.

This is important beyond the obvious fact that an approximation may deviate significantly from the perfect LRU order: There are many ways to walk pages and harvest reference bits, and the specific method used has a significant impact on cost and accuracy of the approximation.

- Paging algorithms based on LRU lists take bad decisions in some common scenarios. A server streaming content typically uses a page exactly once before it is discarded. A pure, global LRU algorithm will see a steady stream of recently used pages and needlessly tries to keep them in memory. A simple method for dealing with streaming I/O adds a page to the LRU list only after is has been used for the second time.

  A Least Frequently Used list will also fare better than LRU with streaming because it is not as shortsighted. In [54], the authors show that LFU gains an advantage over LRU with increasing cache size. They suggest a more sophisticated algorithm that combines elements of both LRU and LFU.

  The operations to maintain an LRU priority queue are of a $O(1)$ complexity while the complexity of maintaining LFU priority queues in heaps grows $O(\log n)$ with the size of the heap. The increasing access time gap, however, works to make expensive replacement policies economical if they manage to prevent enough page faults.

- Operating systems often make copious use of memory, for instance to cache file system metadata or recently used disk blocks. The corresponding pages are not mapped into the address space of any process. The algorithm that determines when to free unmapped memory and when to free process-owned pages has a significant impact on thrashing behavior.

  Linux makes this distinction and hesitates to remove a page when a process has a page table entry pointing to it. It will do so when memory pressure is high enough or the amount of process-owned pages exceeds a threshold.

- The size of the time slices allocated by the process scheduler is of limited importance for thrashing. It is, after all, one characteristic of such a situation that processes don't exhaust their time slices.

  Yet, *load control* is a well-known method for attacking the thrashing problem by means of the process scheduler: Temporarily removing a process from the scheduler reduces the combined size of the working sets of all running processes and thus the paging load. Better throughput achieved like this comes at a price, though: Increased worst case latency. A process that has been taken off the scheduler may take quite a while to respond.

- While a process spends time waiting for I/O it cannot reference other pages. We mentioned on page 60 that competition for memory under a global LRU scheme increases the chance that a process waiting for a page wakes up to find even more of its working set missing. A slight imbalance in page fault frequency can amplify itself with all processes mainly evicting pages of the same victim.

This unfairness may be perceived as a flaw. It typically has a positive effect on overall throughput, though, since it amounts to some kind of load control: Primitive, because the victim still contributes to the I/O load by producing a steady stream of page faults, but automatic – the load control aspect is an emergent behavior of the unfairness of a pure global LRU list.

We also note that processes contributing a lot to the disk I/O load tend to be more likely to become victims since they block more frequently – exceptions from this rule include heavy use of asynchronous I/O. Even so, the victim is with some likelihood a process with a fault probability that is very sensitive to a shrinking memory size.

The classic recipes to prevent thrashing are rarely used these days. Where the operating system is still involved, they have been replaced with a number of mechanisms that interact with each other to deal with various aspects of a system that are related to the problem. In the next section, we will discuss how operating systems can address complex decision problems like those presented above.

3.4. Decision Making in System Software

An operating system takes many decisions that affect perceived behavior but are not predetermined by standards and protocols. Trade-offs are often involved and the most "desirable" behavior may even be a matter of user preference.

Several solutions are in use to address this problem:

- A modular design for kernel component makes it possible to switch from one algorithm to another. This solution allows uncompromising solutions for a limited number of scenarios but loses some of its appeal if the problem space is a continuum. Also, applications may unexpectedly change behavior after a kernel component switch, which jeopardizes one key advantage of developer workstations that are largely equivalent to the deployment servers.

- The solution that most closely matches the famous Unix tenet of separating mechanism and policy is arguably one that has a kernel component provide a number of dials that influence its behavior. Unlike the first approach, this solution always exercises and tests the complete core mechanism. However, a lot of responsibility rests with user space to devise a good policy.

  Some decisions are rare enough that the context switch overhead does not matter and too frequent for users to consider individually, although they may have quite strong opinions on specific, sophisticated policies. In such cases, a daemon process may implement a policy given by the user and manage the mechanisms offered by the kernel.

  One such example is cpufreqd, a daemon for recent Linux kernels which can dynamically adjust the frequency for some CPUs depending on a set of rules.

- A kernel component may try to adapt to a variety of scenarios automatically. This solution does not only reduce complexity for users and system administrators, but it can also be

the optimal solution in terms of performance if decisions for a specific problem depend on highly volatile variables.

Automation is weak where no set of rules will reliably result in desirable decisions, which can still be acceptable if bad decisions are rare enough or their consequences negligible in a particular scenario.

These solutions are not mutually exclusive, of course. Combinations are common.

3.5. Linux Resource Allocation

Three Linux components pertain directly to the scope of our discussion: The Process Scheduler, the Virtual Memory Manager (VM), and the I/O scheduler.

3.5.1. Process Scheduler

We already mentioned changes in the job description of process schedulers. These are some of the characteristics of schedulers in modern server operating systems:

- The process scheduler has no control over when processes are started. Number and nature of concurrently executing programs is determined in user space.
- Swapping out complete processes in order to control the load and improve throughput has become a last resort if not impossible: Typical C/S interaction is synchronous, prohibiting the server from delaying answers and having the client wait indefinitely. Also, the vast majority of servers are maintained while they are running, which means that at least some important programs are interactive and require immediate attention when used.
- The convergence of client and server kernels suggests that a server kernel should accommodate the needs of a desktop client system as well.

  Schedulers that work well on the desktop must answer to additional requirements. Media players, for instance, tend to have strict low latency requirements – if the kernel fails to schedule such programs frequently enough, users complain about skipping audio or video.

Linux offers system calls and scheduling policies related to real-time scheduling as defined in POSIX.1 [77]. However, it should be noted that as far as mainstream Linux and other common desktop and server operating systems are concerned *hard real-time* does not exist. Hard real-time operation requires clearly defined and guaranteed worst case latencies. Some recent improvements in Linux which brought the average latency down and reduced the probability of the occasional maverick merely positioned the kernel better for soft real-time tasks where a best effort approach is acceptable.

The system calls mentioned above are not commonly used to improve latency for interactive applications: Programs that chose to run under real-time scheduling policies are privileged and monopolize the CPU among themselves; ordinary processes get to run only if no process on a `SCHED_RR` or `SCHED_FIFO` policy is runnable.

Linux does not support *priority inheritance*, which underscores that any use of those special real-time scheduling policies takes careful design to prevent the consequences of *priority inversion*. Also, a process that depends on low latency enough to select a special scheduler will want to lock all its memory to keep it from being paged out which requires additional privileged systems calls and largely exempts the process from having to compete for memory. Systems that take advantage of real-time extensions are special cases and beyond the scope of this paper. We assume for the rest of our discussion that all processes are scheduled under the standard policy.

Like all Unix flavors, Linux allows users to adjust the *nice value* – that is, the static priority – of a process. It determines the size of the time slice – called *quantum* – a process receives from the standard scheduler. Every runnable process is scheduled to run on the CPU, though, regardless of its priority. This limits the damage that can be done by a priority inversion and prevents the starving of low priority tasks like I/O bound batch processes.

In order to allocate CPU time, Linux 2.4 iterates over all processes and hands out time slices for the next cycle. The processes are then scheduled based on their static priority and the remaining size of their respective time slices until all runnable processes have exhausted their slices – at this point the cycle starts again. The scheduler tries to favor interactive applications based on unused CPU time: A large unused time credit increases the likelihood for a process to be selected by the scheduler. In addition, processes can bring half their unused time over to the next cycle. Obviously, this gives I/O bound batch processes the same priority boost, while an interactive process may lose its bonus too quickly in a short burst of activity.

The early development series that led to Linux 2.6 merged a new process scheduler which scaled much better with the number of CPUs and processes. It became known under the descriptive name *O(1) scheduler*. The runnable processes are collected in per-CPU runqueues. A process that exhausted its time slice has the next quantum calculated and is moved from a runqueue's *active* to its *expired* array – unless a process is deemed interactive, then it is immediately put back to the active array – a preferential treatment that can be vetoed by a mechanism designed to prevent CPU starvation for low priority processes. Once no process is left in the active list, the two arrays switch roles.

While scalability work continued, for instance to make the scheduler NUMA aware, the focus of public attention shifted to interactivity improvements for desktop and other systems requiring low latency. On the dominant hardware platform, Linux 2.6 changed the default frequency of the timer interrupt the scheduler uses for preemption and timekeeping from 100 Hz to 1000 Hz which causes some additional overhead but allows for more fine-grained scheduling.

The mechanism for assessing interactivity is much more sophisticated then in Linux 2.4 [50]. An even more involved change made the kernel preemptible: A process can now be preempted while executing space.

3.5.2. Virtual Memory Management

The Virtual Memory Manager (VM) is a cornerstone of every modern server operating system. Moreover, Virtual Memory has become a common feature for server operating systems over three decades ago. One might expect that VM knowledge is exhaustive and that a reasonably solid implementation is only tuned by its developers and never replaced.

This seems not to be the case, at least as far as Linux is concerned. A new VM was introduced

during the 2.3 development series and in a move surprising many replaced again in 2001 for 2.4.10, a long way into the stable series – one of the more controversial decisions in the history of that kernel. For this paper, the "*Linux 2.4 VM*" signifies the second VM as present in Linux 2.4.21. The 2.4 VM has been thoroughly documented in [32].

The 2.5 development series brought more scalability work which also affected memory management. NUMA machines, for instance, may not only have differing memory access latencies, they can also have holes between the address ranges of various nodes – hence the need for discontiguous memory support. Several data structures became node or CPU specific. The paging work is now done in per-node kernel threads. And, for the purpose of scaling down, support was merged for MMU-less processors.

Arguably the most significant change, however, was the new VM with its most prominent feature, *reverse mapping*:

Every VM frequently derives a physical from a virtual memory address, a calculation that is easy to reverse. Either operation requires a process context, though. Since a virtual address needs the context to be unambiguous anyway, the context is always readily available if a virtual address is to be translated to a physical one. Starting from a physical address, the VM in Linux 2.4 faces two issues: The only way to find a corresponding virtual memory address is to iterate over the memory data structures of all processes. And the result may contain more than one virtual address.

This limitation becomes clearly visible when the Linux 2.4 VM frees memory by paging out shared pages: Due to the first issue, the VM must scan the virtual memory areas of processes in turn to find pages that can be freed. The second issue poses an additional problem because a page frame cannot be freed until the page tables for all processes using it have been updated.

The VM keeps use counters in a data structure associated with each page frame and will not evict a page until its page counter indicates that all page tables are up to date. This makes shared pages hard to page out, which might look like a welcome feature given our previous discussion of page replacement aspects. It becomes a problem, however, on large systems: Instead of growing with the number of processes referencing a shared page, the difficulty in freeing it is now dominated by the sum of memory mapped in any process page tables. Also, there is no simple way to free memory in a specific physical address range – this matters, for instance, on hardware which is capable of doing *DMA* only with physical addresses below 16 MB.

The new reverse mapping functionality in Linux 2.6 provides access to a list containing all relevant page table entries when given a physical address. Therefore, the new paging mechanism does not need to switch to virtual scanning when it decides to free mapped pages – it simply ceases to skip mapped pages it encounters. Consequently, the same change that brought reverse mapping removed the code that looked for pages to evict by walking the process memory data structures.

3.5.3. I/O Scheduler

As mentioned previously, an I/O scheduler can reorder block I/O requests to improve throughput by reducing the number of disk head seeks.

Linux 2.6 offers several I/O schedulers to choose from:

- The noop scheduler does no reordering and is a good choice if a block device is not a hard disk but, for instance, RAM based.
- The deadline scheduler tries to keep the number of disk head seeks low while maintaining an upper limit for how long a request may be delayed. It was introduced in Linux 2.5.39.
- The default scheduler in 2.6 is the anticipatory scheduler which was merged in Linux 2.5.75. It collects read and write requests in separate queues and serves each queue alternately. It leaves the disk idle and its head in the current position for a while after serving a request, based on the observation that a request is often followed by a another one in the vicinity.

The cfq (complete fairness queueing) scheduler was in testing during the Linux 2.5 time frame but not quite ready for inclusion in 2.6.0. The cfq scheduler is an attempt at giving each process a fair share of the I/O bandwidth.

3.6. Enter the Benchmarks

thrash

In order to assess the behavior of a system a simple, portable, and reproducible method for triggering thrashing was needed. For this purpose, a program was created early on in this project [4]. It forks a number of child processes: Each child allocates a part of the main memory and iterates over the pages it owns in an attempt to keep them all in memory. In addition, those pages are also written to, making it virtually impossible for the VM to find and discard clean pages. Since every page is only referenced twice – once for reading and once for writing – before a process moves to the next page, the execution time is extremely sensitive to memory shortage, leading to slowdown factors which are best measured in orders of magnitude.

This program clearly satisfied Denning's condition for thrashing: Performance collapsed with one additional process. As we found out later, it is also an excellent method for triggering the unfairness that is particularly strong in Linux 2.4: Thrashing does not occur because some processes grow at the expense of others.

This program was used to test and tune the load control mechanism described in section 3.7. It was subsequently replaced with qsbench which has similar properties but is used by other kernel developers as well. While a good method to find thrashing and unfairness, thrash did not excel as a benchmark: It is measuring both thrashing and unfairness, and optimizing a VM for this special kind of workload seems a dubious proposition.

kbuild

Building the kernel is a popular benchmark. It measures a real work load that all kernel developers care about. Our kbuild benchmark instructs the make utility to run a maximum of 24 commands simultaneously to build Linux 2.5.70 with 64 MB of RAM available. To keep execution time within limits, only a small portion of the kernel is rebuilt each time.

[4] Appendix H.1.

RAM The amount of RAM made available to the system was limited using a standard Linux boot parameter. A test was conducted to confirm that for our purposes, this method results in the same behavior as actual RAM removal.

Run Med, Avg Median and average number of processes running. In Linux, running and blocked processes are not accurately identified, which is noticeably especially if processes block frequently. It is not uncommon to find a process counted both as running and blocked.

Swap Avg, Max Average and maximum virtual memory use.

CPU Idle The average of idle times reported. For our benchmarks it is equivalent (but not equal) to the time all processes were blocked waiting for I/O.

Idle In, Out The percentage of 1 second intervals for which no disk activity was reported. In and Out refer to reading and writing, respectively.

Slowdown The ratio between median benchmark run times with and without memory limits.

$\hat{s}_r$ The relative standard deviation for run times [a]. For the precision we used, $\hat{s}_r$ is zero if plenty of RAM is available and the first run which leaves all data in the disk cache is discarded.

[a] Statistic are discussed in appendix G.

Figure 3.1.: Labels for figures 3.2, 3.4, and 3.6 explained.

RAM	64 MB	Run Med	1
Swap Avg	57 MB	Run Avg	4.6
Swap Max	108 MB	CPU Idle	50%
Idle In	1%	Slowdown	2.4
Idle Out	19%	$\hat{s}_r$	9.9%

Figure 3.2.: Key figures for kbuild.

```
cd linux-2.5.70
rm arch/*/*/*.o
rm arch/i386/boot/bzImage
make -j24 >/dev/null
```

Figure 3.3.: kbuild core.

RAM	32 MB	Run Med	1
Swap Avg	56 MB	Run Avg	0.6
Swap Max	71 MB	CPU Idle	84%
Idle In	0%	Slowdown	7.4
Idle Out	5%	$\hat{s}_r$	0.6%

Figure 3.4.: Key figures for efax.

```
cd efax-gtk-2.2.2/src
make clean
make main.o >/dev/null
```

Figure 3.5.: efax core.

RAM	256 MB	Run Med	3
Swap Avg	205 MB	Run Avg	2.5
Swap Max	261 MB	CPU Idle	21%
Idle In	49%	Slowdown	1.3
Idle Out	40%	$\hat{s}_r$	1.1%

Figure 3.6.: Key figures for qsbench.

```
qsbench -p 4 -m 96
```

Figure 3.7.: qsbench core.

Dependencies exist between some of the two dozen processes competing for resources: The assembler cannot do its job before the compiler has translated a source to the intermediate format, for instance.

A major drawback of kbuild is the large variance of test results.

efax

The scenario used for the efax benchmark was described in fall 2003 by Chris Vine as a regression of Linux 2.6.0-test9 compared to 2.4.22 [106, 107]. A compile test like kbuild, this benchmark starts from C++ source code and has the make utility issue only one command at a time. Looking at the distinct stages, the source file `main.cpp` weighs a mere 19 KB initially, 2249 KB after preprocessing, 227 KB as assembly code, and 40 KB as stripped object code.

It is the process that translates the preprocessed source to assembly code which determines the behavior of this benchmark. The compiler – gcc 3.2.3 – makes over 1000 system calls to map 64 KB chunks of anonymous memory and a few more to allocate up to a couple of megabytes with one call. Eventually, it levels off at somewhat over 80 MB.

Having only one process that matters makes this benchmark very sensitive to memory shortage: Whenever a page fault occurs, the system becomes idle and throughput is immediately affected. The benchmark is immune to unfairness: It is not possible to improve throughput by favoring some processes over others. With only one process, the memory references come always in the exact same order: The run time variance is low.

qsbench

This benchmark neither reads data nor does it write any output files. All disk I/O is due to paging. With arguments as given in figure 3.7, four processes are forked. Each allocates 96 MB of memory, fills that memory with pseudo-random numbers and sorts the resulting array of integers using quicksort – hence the name. Unlike thrash, which was specifically designed to

stress the VM, qsbench uses a significant amount of CPU resources.

Numerous variations of quicksort have been described but due to their divide-and-conquer approach they all exhibit a good reference locality [113].

There are no dependencies between the four processes that constitute the work load. Executing them sequentially in any order will give achieve maximum throughput: Unfairness is an easy way to improve throughput. For this reason we modified the qsbench code to report run times for each of the children separately.

At first glance, qsbench seems to have a low variance. While the CPU is partially idle for every second of the efax benchmark, though, it is used 100% in two out of three 1 second intervals of qsbench. This work load produces bursts of disk I/O: The ratio between average and median data transfer rate is 720 for qsbench compared to 1 for efax. In the most intense second, qsbench transfers more than twice the data that efax or kbuild ever move to or from the disk in the same time. Relative standard deviation and the slowdown factor for qsbench appear too low because they are weighted with the total run time although the system throughput is clearly not constrained by disk I/O most of the time.

3.7. Load Control

3.7.1. A Prototype Implementation

We started from the notion that the way to combat thrashing on a reasonably well tuned VM must be to reduce the combined size of the working sets of the runnable processes by lowering the process load. In order to study the viability of this approach we implemented such a solution for the current Linux development kernel which at the time was 2.6.0-test4 [5].

Design Considerations

Before embarking on such a project, it is helpful to think about goals and constraints, and so we did take some notes:

- The goal was to ease the situation, that is to improve throughput after performance had collapsed. The goal was neither to prevent nor to completely solve the problem.
- The resource usage should be adequate to the problem. Since thrashing is a rare problem, using CPU cycles or polluting the CPU cache with additional data was deemed unacceptable during normal operations. Thus, adding additional fields to frequently used kernel data structures for memory or process resource management was not an option.
- When thrashing occurs, though, resources are at our disposal: The CPU will have most cycles to spare, and even memory or disk I/O usage are easily justified if a thrashing system can be recovered and becomes usable again.
- Load control must never interfere with normal operations. The system should only be manipulated if it is a certain improvement. In other words: The trigger for load control

[5] Appendix H.8.

should reliably fire in cases of extreme thrashing, but it may define and ignore a substantial amount of border cases in order to prevent load control from ever making a situation worse.

- The code itself should be simple and unsophisticated. Load control was going to be rarely used and therefore, plenty of user feedback could not be expected. The implications of a complex solution for real world loads would never be fully understood. Users, however, tend to prefer predictably bad behavior over a system that fails occasionally under an unknown combination of circumstances.

- The code should not be invasive. Ideally, it would be self-contained in one file. Only rarely should changes to the core VM make updates to the load control code necessary.

- With regards to our discussion of decision making in system software in section 3.4, we note that thrashing is rare and typically not planned for. Therefore, it is unlikely that a user space daemon and a policy are set up in such a case. It follows that load control should be fully automatic.

Trigger

The first challenge was to find a reliable indicator for a thrashing situation. The goal was a conservative trigger: It must never hurt performance by changing the default behavior without necessity, but it may ignore some light thrashing cases just as long as it catches heavy thrashing reliably.

The core routine used for memory allocation in Linux 2.6, `__alloc_pages`, first checks the pool of free pages. If the request cannot be satisfied immediately or if doing so would drain the pool below a watermark called `pages_low`, the routine wakes up kernel threads that are responsible for refilling the pool. A second attempt is then made to serve the request from the pool, but this time a lower watermark `pages_min` defines the minimum amount of memory that must remain in the pool.

If the second attempts fails as well, the routine hits what a code comment calls the "*low on memory slow path*": If process flags indicate that the allocation will alleviate memory pressure – for instance because the process is dying – then the pool is tried once again ignoring all watermarks. After this point, atomic allocations have definitely failed already. For the remaining requests, the allocator tries to free pages on its own which means the process that issued the request may block on disk I/O.

We hooked a new function `thrashing` into the allocator to become the first command in the slow path.

The Stunning Cycle

The `thrashing` function returns immediately if it fails to acquire a lock that ensures that only one process is in the stunning phase at any time. This prevents further stunning before the consequences of previous measures are evident. The routine selects a victim to stun and sends a signal to that process. A few small routines walk the process table and assess the "*badness*" of

each process. They are derived from the Linux OOM killer, a mechanism that selects processes to kill if the system is out of memory. Both situations are caused by lack of memory, but they differ in a number of areas. For example, the amount of computation time lost is a concern only for the OOM killer which tries to avoid killing long running, CPU bound processes. The same processes are unlikely to require low latency, though, which makes them prime candidates for stunning.

The process that received a signal from `thrashing` calls the signal handler and is immediately redirected to `stun_me`, which calls `dump_mm` and unlocks the lock taken early in `thrashing` before the process joins a FIFO waiting queue and goes to sleep. Together with its helper functions, `dump_mm` walks all virtual memory regions of the process, writes dirty pages to the backing file or to swap, and uses the regular Linux 2.6 `shrink_list` function to free the memory.

The code around `dump_mm` is fairly complex – with a VM that wants to scale to large NUMA systems Linux 2.6 depends on a fine-grained concurrency control to protect important resources like LRU lists (per memory zone), memory management structures and page tables (per process), the page tables entry chains used for reverse mapping (per page frame), to name a few. Pages cannot be freed indiscriminately: Doing so would be a bug for pages that were previously locked into RAM using the `mlock` system call, for instance.

On the other side of the waiting queue, a modified process scheduler wakes up processes that had been stunned previously. The trade-off between latency and throughput is decided here: The longer applications stay in the queue, the more pronounced the impact on the paging load.

Load Control Options

There are many variable elements in load control and each implementation needs to find answers for many questions.

- We already mentioned some difficulties connected to trigger mechanisms and victim selection. The trigger we used for our prototype would not have worked throughout the 2.5 development series (see section 3.8.2, page 81).

- The scavenging of memory owned by a stunned process can be left to the regular page out routines. Alternatively, those pages can be moved to the end of the priority queue and dirty pages can be written to the disk to prepare for later eviction.

  We reasoned that it might make sense to be even more aggressive and use the knowledge about pages that were not going to be used for a while, regardless of possible recent references, and our limited tests seemed to confirm this to be beneficial. Even so, it may not be optimal to evict all pages of a large process if the remaining processes need only half of them.

- Recalling our discussion about the dangers of freeing shared pages (section 3.3.2, page 61), we wonder if shared pages should get a preferential treatment; for instance, the load control code could be more reluctant to evict shared pages than the regular page out code.

  Another somewhat related question is whether it makes sense to treat code segments differently from data segments.

- The mechanism that maintains the waiting queue plays an important role as well. The growth in latency for stunned applications is not only a function of the rate at which we release processes from the queue – it also depends on the length of the queue. Therefore, we tried a variant that adjusts the release rate to the queue length, increasing the rate dynamically as the queue grows: Whenever the scheduler finds that the waiting queue is not empty, it wakes up a process from the queue provided that enough time has passed since the previous wake up; the time that qualifies as "enough" is calculated based on the queue length. If ten or more processes are in the queue, the maximum release rate is reached and a process is released every half a second.

  Other factors may have significance for an optimal queue management: For instance, after enough processes are stunned to thwart thrashing, comparing the current resident set size (RSS) of a process with its RSS at the time of the stunning can serve as an indicator for the amount of free memory the process will require after waking up, and a sufficient amount of memory may be prepared for freeing prior to waking the process.

  Since the process was stunned under tight memory conditions, it had only pages with a high priority in its RSS, and it may therefore even make sense to memorize the list of pages to fault them back in automatically before allowing the process to run again.

- There is a wide range for the degree to which a load control mechanism is adaptive. As the VM itself does with its priority lists, it may collect data to improve decision making. A very simple addition to our own implementation might sample CPU usage regularly and increase the interval between stunning actions if they are found to further reduce the moving average of that figure.

3.7.2. Load Control in Modern Operating Systems

It took us a while to realize that load control was not what Linux 2.6 needed. Load control fails as a remedy for the problems with the new kernel series in several regards:

- Load control has no benefits if only one relevant process is involved.

- Load control attacks one problem – lack of throughput – and creates another one: Latency. No matter how clever the victim selection, chances are that every now and then the process that gets stunned is an interactive program used by a system administrator trying to fix the problem.

  There is no generic method to automatically and reliably pick the right process. With a carefully tuned selection algorithm and a dynamically managed waiting queue as outlined above it was possible to keep average latency within reasonable limits, but worst case latency remained high: Every now and then, the shell or some other interactive program fell victim to the load control mechanism and froze for several seconds while it moved through the waiting queue.

  In sections 3.2 and 3.5.1, we discussed why nowadays perceived interactivity tends to be more important than throughput even for server operating systems. A mechanism that introduces high latency is not acceptable in most situations.

- Load control can yield impressive results under extreme conditions. The benchmarks we used are examples of high memory overload. Much more common are systems where the combined working set for all running processes is only somewhat higher than the amount of RAM available. Linux 2.6 is slower than 2.4 in these scenarios as well, but load control is hard to tune to be beneficial under such circumstances as Linux 2.4 keeps the CPU busy at nearly all times.

  Moreover, a conflict with one of our design considerations is likely: If load control acts upon small memory overload, chances are it will occasionally make matters worse.

One reason for the popularity of GNU/Linux is its ability to run on low-end hardware. However, that does not necessarily mean that heavy overload situations where load control could be useful are typical even on those machines. Since paging is always slow, the load will be run sequentially if possible at all. The kbuild benchmark would run just fine on the specified hardware had we not explicitly asked for two dozen concurrent processes.

Load control is essentially nothing but a form of concurrency control, with page frames being the scarce resource. Keeping this in mind, we find that for load control to be useful, a system and its work load should have these properties:

1. Massive memory overload.

2. Unattended operation. Latency makes interactive work on that machine rather unpleasant.

3. The work load should *require* that two or more processes run concurrently – otherwise the solution is to reduce concurrency in user space where the relevant knowledge to take an informed decision is more readily available. The likely cause for such a requirement is a dependency between the processes.

   Concurrency in user space can be required or seem desirable for several reasons:

   a) A system is to offer a number of services that are provided by different programs, no dependency exists between them. This problem is usually addressed by a super server that starts the individual programs on demand. If, on the other hand, a number of services are supposed to do actual work, then it can either be serialized or latency makes load control similarly undesirable.

   b) As we pointed out in section 3.2, many modern server applications are written to use threads. In a memory overload situation, though, the application creating threads and not the kernel tends to have all the relevant information about dependencies and the respective importance of each thread. Research is indeed being done for load control in multi-threaded applications rather than in the kernel [39].

   c) Unix programs connected by pipes, Producer-Consumer and similar problems that have interprocess dependencies without a central instance to overlook the whole. In such a case, load control has a good chance of hurting performance, especially when data is processed in small chunks. Stunning one process will in effect bring the whole assembly line down.

d) To increase throughput in a system with I/O bound processes. We discussed in section 3.3.1 that increased system load can improve the throughput if one process can continue while another is blocked waiting for I/O. In a situation with massive memory overload, however, this effect outweighed by adverse effects. We remember that rather then recommending load control, Denning suggested not to start a process unless its working set fits into available memory – the load control operations add to I/O traffic, after all.

   If the processes making up the work load exhibit volatile reference patterns it is at least conceivable, though, that high concurrency combined with load control achieves higher throughput than a pure admission control. A rise in concurrency increases the likelihood that at least one process is and remains CPU bound while others are going through an I/O bound phase, and load control can mitigate the occasional memory overload. In other words, if thrashing brings the system almost to a complete halt, then it is imperative for system based on admission control to anticipate and prevent a work load that will or might, at some point, go through a period with high memory overload. A load controlled system can bear higher load because it will go through an overload phase more gracefully.

   Of course, the prime example for highly volatile and unpredictable reference patterns are interactive applications we ruled out already earlier in this list, but there are certainly others.

While work loads and data sets tend to grow with the hardware, we suggest that a larger machine will rarely run a work load meeting these conditions, although the scenario sketched out in 3d) above is likely to occur in the real world occasionally.

3.7.3. Prototype Performance

Kernel	efax		kbuild		qsbench	
	$\widetilde{x}$	$\hat{s}_r$	$\widetilde{x}$	$\hat{s}_r$	$\widetilde{x}$	$\hat{s}_r$
2.4.15 / 2.5.0	1.0	0.006	1.0	0.099	1.0	0.011
2.6.0-test4	4.6	0.579	3.2	0.132	1.9	0.126
load control	10.1	0.436	1.0	0.051	0.9	0.018

Table 3.1.: Median run time, relative standard deviation for 2.6.0-test4 with load control.

We take a look back at the design considerations set forth in section 3.7.1:

- With the benefit of hindsight, we note that Linux 2.6.0-test4, the kernel we used as a starting point, was not necessarily the best choice. As is evident from a look at figure 3.8, some major regression for the compile benchmarks had just taken place (2.6.0-test4 corresponds to 79 on the x-axis).
- Nevertheless, table 3.1 shows that our load control implementation did improve throughput substantially for two of our benchmarks. The third one, efax, demonstrates an obvious

but severe limitation: If the working set for one process is too large to fit into RAM, no improvement is possible using load control and it takes in fact a smarter trigger or victim selection than ours to prevent a further performance drop.

- CPU usage is limited: One conditional in the signal handler, one call in the slow path of the page allocator, and a block of code in the process scheduler that is entered if the load control waiting queue is active. Per-process state information does not necessarily require extending the process descriptor.

  For instance, information about process state at the time of stunning as mentioned in section 3.7.1 can be stored on the stack of `stun_me`. Obviously, care would have to be taken if potentially voluminous information like page lists were stored so as not to overflow the stack. The circumstances don't make dynamic allocation of additional memory seem advisable, but the page list could be truncated or stored in a compact format using ranges.

- As mentioned before, the simple trigger we used fires even if only one process produces the memory overload. In its current form, our implementation clearly fails the requirement to never make a situation worse.

- The mechanism is simple – the complexity of the code is almost entirely due to the particularities of a sophisticated VM. The interactions with the whole system are non-trivial, though. Most notably, the slow path in the page allocator served as a reasonably robust trigger, but it is an indicator that is very sensitive to changes in the rest of the VM. As we will see in section 3.8.2, it would have been unusable as recently as Linux 2.6.0-test2.

- All the code complexity of load control ended up in one file, but since our implementation manipulates VM data structures to forcibly evict pages the VM wanted to keep, it depends on intimate and accurate knowledge of VM internals. Some of this complexity could be moved out of the load control code if the respective functions in Linux took additional arguments to allow the caller better control over page freeing.

- In retrospect, we are less convinced that load control should be a fully automated mechanism in the kernel: Those situations where load control is a clear improvement are rare, the potential drawbacks severe, the trade-offs dependent on individual preferences.

  The situations which make load control beneficial may not be exactly predictable, but the strong conditions we found suggest that most users will never see them. And those who do will likely experience them regularly, but not under circumstances where context switching and related overhead makes a difference. Therefore, we believe that a good architecture for load control in a general purpose operating system has a privileged background process implement a user defined policy. The process would lock its tiny working set into memory and take decisions on potentially

  Users should at least have a switch to turn the mechanism on and off, and maybe some knobs to tune various aspects of it.

Considering the drawbacks and limitations of load control compared to a good page out mechanism, we finally came to the decision that there was no point in pouring further resources into load control until it was known beyond reasonable doubt that all other means had been exhausted.

3.8. Paging between Linux 2.4 and 2.6: A Case Study

When our project to add load control to Linux started in summer 2003, there was a common assumption among Linux kernel developers that the switch to physical scanning in the page-out code was to blame for the slowdown of 2.6 under tight memory conditions.

After a few samples taken with the kbuild benchmark cast a doubt on that theory, we set out to conduct a systematic, quantitative analysis.

Testing Details

The *development branch* leading to Linux 2.6 *forked* from 2.4.15 – that is, the only difference between 2.4.15 and 2.5.0 is the version number. The series lasted from November 2001 to December 2003 and produced 76 releases from 2.5.0 to 2.5.75, another 11 from 2.6.0-test1 to 2.6.0-test11 and then Linux 2.6.0 proper for a total of 88 releases. This being the development branch, many of the kernels failed to build or crashed early. They had to be patched to allow running any benchmarks. One data point is missing since the kbuild benchmark would not finish on any kernel resembling 2.5.1.

All kernels were compiled with the compiler that has been the officially recommended choice on the x86 architecture for several years: gcc 2.95.3. A later version – gcc 3.2.3 – we used for user space applications and as a work load in the compile benchmarks failed to build early Linux 2.5 kernels.

The test machine is a VIA EPIA system with a Centerhauls CPU running at 533 MHz. One memory bank was filled with 256 MB RAM initially and with 512 MB RAM later. File systems and swap partition reside approximately in the middle of an internal 20 GB IDE hard disk with DMA enabled. Tests were not conducted in chronological order of kernel releases.

The benchmarks were run in single user mode to prevent interference from other programs. Each series of ten runs for one combination of kernel and benchmark was conducted after rebooting the machine.

3.8.1. Overview

Linux 2.5.27, the kernel that merged a rewritten page out code to use physical scanning, brought a noticeable regression for all three benchmarks. However, performance deteriorated further during the rest of the development series, and this problem has yet to be fixed for both compile benchmarks.

The results for three test work loads show that there are different types of work loads with memory overload and that each type can depend on different aspects of the kernel mechanisms. The reference patterns are a key factor in determining the type of a work load: The graphs for efax and kbuild in figure 3.8 look similar despite the fact that efax produces the load with one large process and kbuild with two dozen smaller processes.

It is worth noting, though, that both benchmarks are not equivalent even with regards to qualitative information: Not all major changes for kbuild affected efax – witness the kbuild regression in 2.5.48 and the improvement in 2.5.65. All major changes in efax are reflected in kbuild, but they are sometimes less pronounced, for example between 2.5.32 and 2.5.39, and the results are

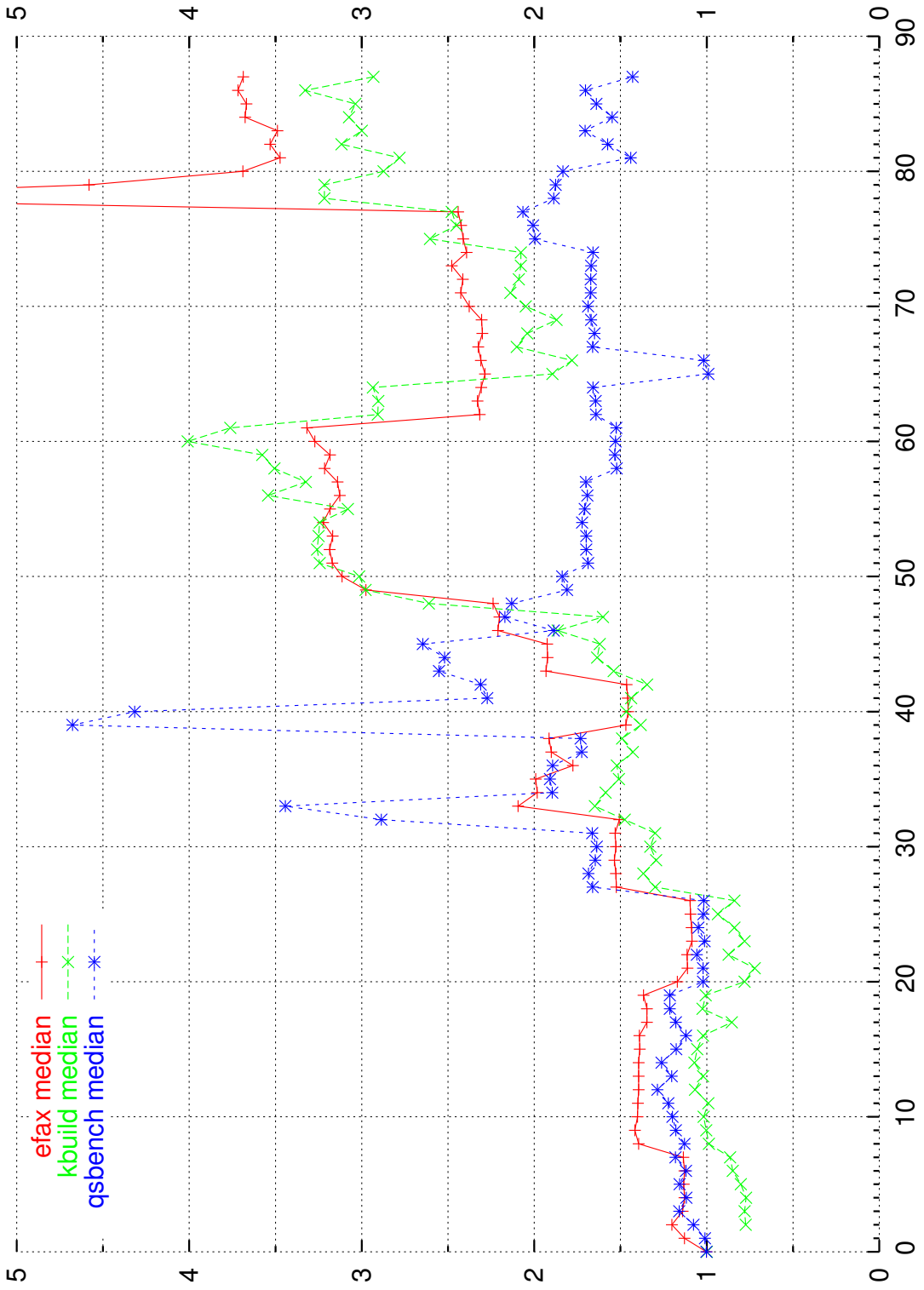

Figure 3.8.: Median benchmark run times Linux 2.5.0 – 2.6.0 (lower is better). A value of 75 on the x-axis corresponds to 2.5.75 and 76 to 2.6.0-test1. Linux 2.6.0 is at 87.

more conclusive with efax due to its lower variance, as can be seen when comparing figures 3.12 and 3.9. On top of the performance loss the 2.5 development series caused a jump in the relative standard deviation for efax which is with 2.6.0 four times that of 2.5.0 (cf. table G.2).

To estimate the influence of I/O scheduling on our results, we compared the default I/O scheduler with the alternatives noop and deadline. The results in table 3.2 support our assumption that the regressions are rather due to the way pages are selected for eviction.

Kernel	efax		kbuild		qsbench	
	$\widetilde{x}$	$\hat{s}_r$	$\widetilde{x}$	$\hat{s}_r$	$\widetilde{x}$	$\hat{s}_r$
2.4.15 / 2.5.0	1.0	0.006	1.0	0.099	1.0	0.011
2.6.0 noop	3.8	0.013	3.8	0.124	1.7	0.067
2.6.0 deadline	3.7	0.020	3.3	0.129	1.6	0.033
2.6.0 as	3.7	0.023	2.9	0.106	1.4	0.167

Table 3.2.: Median run time, relative standard deviation for noop, deadline and anticipatory scheduler.

3.8.2. Identifying a Culprit

The revised plan after we started the systematic benchmarking was to use the graphs as a map to quickly locate regressions which could be studied and hopefully fixed. If we recall from section 2.3.2 that the Linux kernel changes at a rate of 180 changesets per day, a identifying the cause of a regression is not necessarily trivial. A binary search find the culprit soon if the researcher manages to patch working intermediate kernel versions together. Some intermediate versions exist in the form of snapshots for later development kernels already.

At the time we were still debating whether there are in fact significant regressions in Linux 2.6 and if our benchmarks were relevant for the real world. Therefore, we focused on a select few regressions to find out if and demonstrate that our data can be used to identify and fix problems. We picked 2.6.0-test3 for a number of reasons:

- Even after taking into account subsequent improvements it remains the largest regression for the compile benchmarks.
- With a recent kernel the chance is lower that the relevant change has been buried in layers upon layers of later changes that can make it hard to revert in the latest release.

  In addition, a recent kernel makes it less likely that the regression can hide behind another one; this would happen if a number of properties were required for good performance, and one property was removed or damaged earlier than the rest. Reverting that change will not help if other properties have disappeared as well in the meantime.

As contrived as the argument above may sound, this is exactly what we found in 2.6.0-test3: The impact of one seemingly independent change turned out to be entirely dependent on a second change.

The first change added some 40 lines of code to the page out mechanism or, more precisely, to the code for refilling the free memory pools for all memory zones (cf. section 3.7.1). Specialized

The x-axis of each graphs counts kernel releases, the y-axis measures relative median run time.

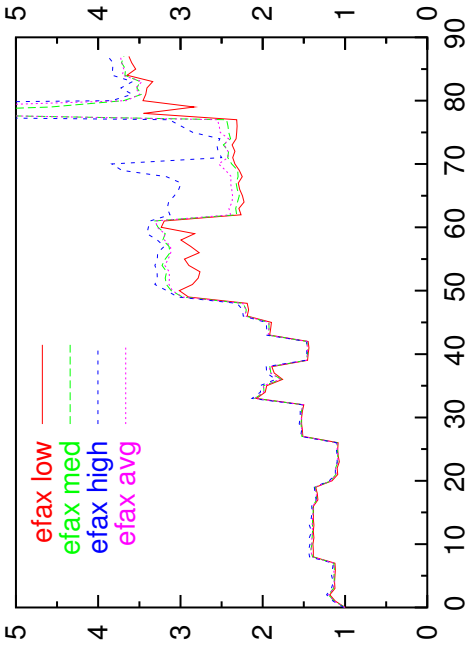

Figure 3.9.: Details for efax.

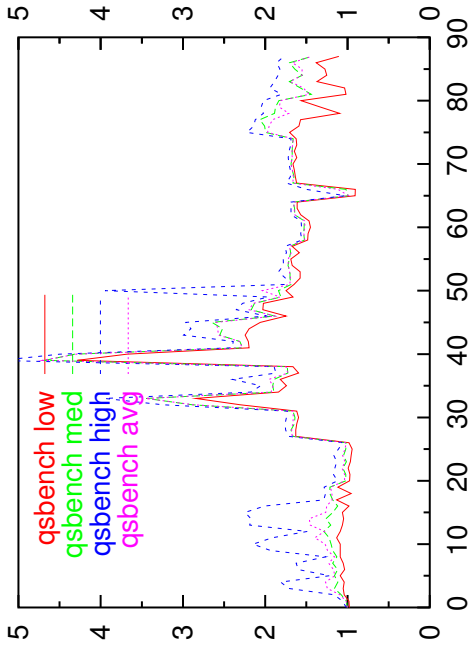

Figure 3.10.: Details for qsbench.

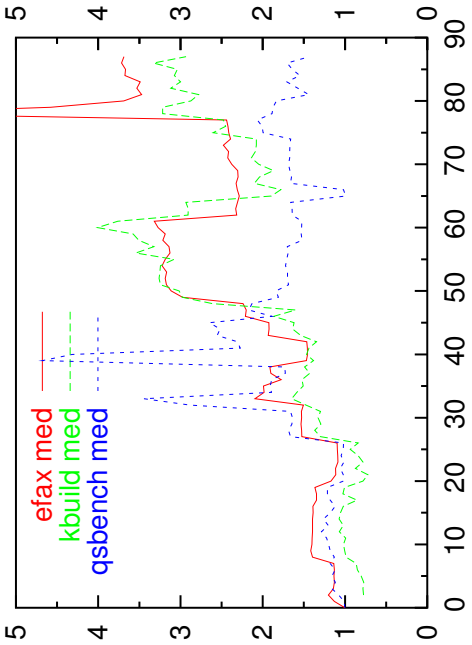

Figure 3.11.: Same as figure 3.8.

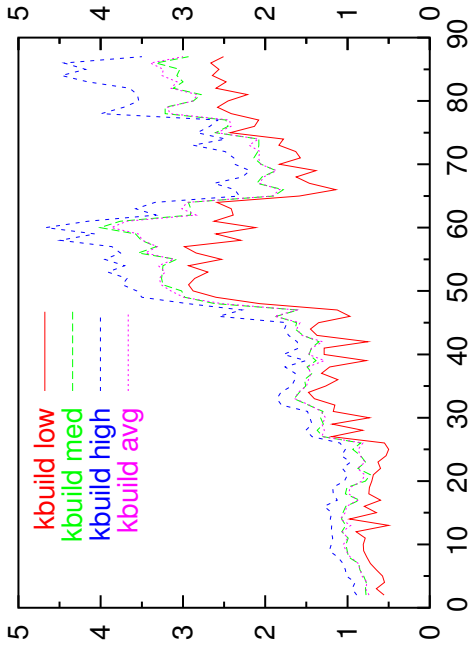

Figure 3.12.: Details for kbuild.

kernel threads called kswapd use the page eviction mechanism with increasing priority if not enough memory to meet the target watermarks could be freed with the previous call. A priority increase of 1 has two effects: The limit for how many pages may be scanned for freeable pages doubles, and pages that are mapped into process page tables are more likely to be evicted. Up to 2.6.0-test2, a typical cycle for kswapd under high memory pressure would free unmapped memory first, then gradually increase priority to eventually evict mapped pages if necessary. The first change introduces code to maintain a decaying average of the highest priority required to satisfy previous requests in a memory zone, making the unmapping decision less volatile: Based on the decaying average, a kswapd cycle will either always or never consider mapped pages for eviction. The 2.6.0 kernel with the first change reverted is called "*priority*" in table 3.3.

Up to 2.6.0-test2, the kswapd waited for up to 0.1 seconds for the write queue of any block device to become uncongested. These breaks under high memory pressure meant that the page allocator would start freeing memory on its own. According to the log entry for the second change, kswapd should "*only throttle if reclaim is not being sufficiently successful*". However, the patch effectively removed throttling altogether unless the system was about to suspend to disk. As kswapd does the paging work in a very aggressive way, the slow path in the page allocator is now rarely executed – rarely enough in fact that it made a decent thrashing indicator when we looked for one to use as a load control trigger in 2.6.0-test4. The 2.6.0 kernel with the second change reverted is called *"throttle"* in table 3.3, while the last line labeled *"both"* refers to 2.6.0 with both changes reverted.

Our results show how the small change that removed throttling masks the effect of the decaying average for our benchmarks. The priority patch makes a significant difference only if the throttle patch is reverted. That both patches were merged into the same kernel was an unusual help – had the priority patch been merged a few releases later, it would have been more difficult to find out why reverting the throttle patch failed to undo the whole regression. This example demonstrates why regular regression testing is important especially in areas where complex side effects and subtle interactions between different parts of the code affect performance frequently and significantly.

Kernel	efax		kbuild		qsbench	
	$\widetilde{x}$	$\hat{s}_r$	$\widetilde{x}$	$\hat{s}_r$	$\widetilde{x}$	$\hat{s}_r$
2.4.15 / 2.5.0	1.0	0.006	1.0	0.099	1.0	0.011
2.6.0-test2	2.4	0.105	2.5	0.062	2.1	0.126
2.6.0	3.7	0.023	2.9	0.106	1.4	0.167
priority	3.8	0.014	3.0	0.132	1.3	0.156
throttle	3.0	0.222	3.1	0.126	1.6	0.093
both	2.5	0.094	2.7	0.097	1.4	0.109

Table 3.3.: Median run time, relative standard deviation for priority and throttle patches.

Each graphs shows the RAM usage for the relevant processes during the full run-time of qsbench. The x-axis is in seconds, the y-axis in megabytes:

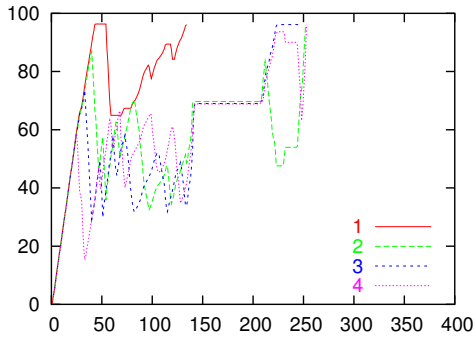

Figure 3.13.: Linux 2.4.15 / 2.5.0.

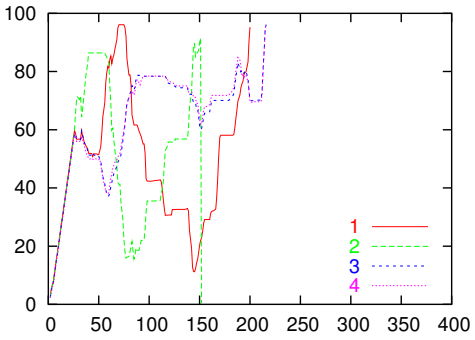

Figure 3.14.: Linux 2.5.65.

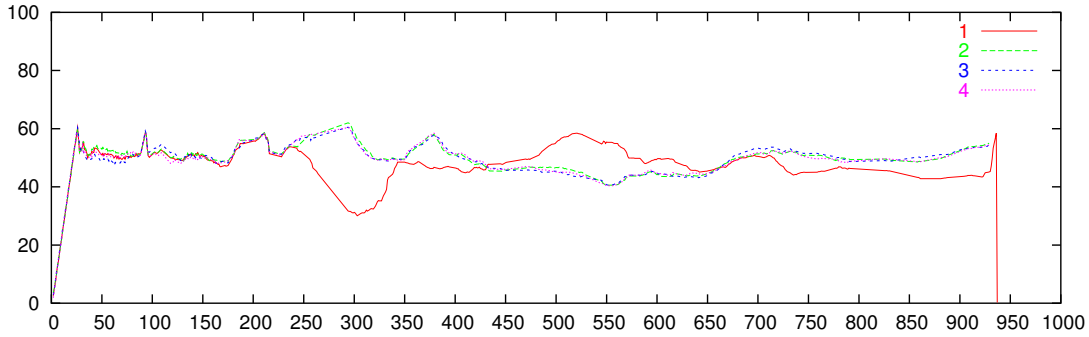

Figure 3.15.: Linux 2.5.39.

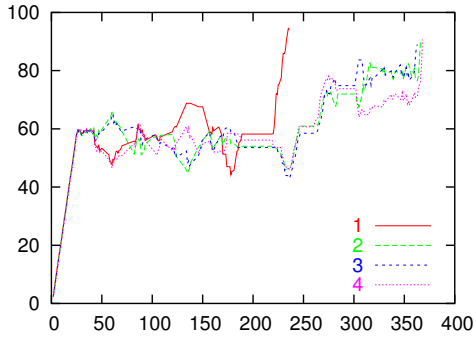

Figure 3.16.: Linux 2.6.0.

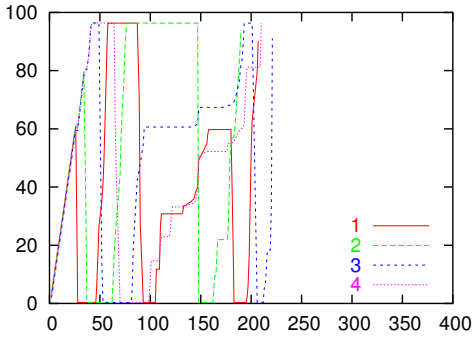

Figure 3.17.: Load control.

3.8.3. Unfairness

In order to improve performance under memory overload, a VM can try to predict future references based on information about the past. Alternatively, a VM can choose to favor one or more processes to make them CPU bound again which improves system throughput. We stressed the importance of unfairness several times in this paper. In this section, we illustrate the concept using qsbench as an example.

The VM in Linux 2.4 allows the first process to allocate all 96 MB of memory in RAM, while the other three processes grow into virtual memory (figure 3.13); the inequality continues until the first process finishes.

Linux 2.5.65 was the first kernel after 2.5.27 to match the performance of 2.5.0, and figure 3.14 highlights the likely reason: Large fluctuations in the amount of memory available to the processes. Figure 3.15 is equally suggestive, but the badness there was not only due to an increase in fairness, as the following graphs show: Figure 3.18 compares data for our standard qsbench runs comprising four processes (*p4 m96*) with a benchmark that has the same program sort 384 MB with only one process (*p1 m384*). As usual, both graphs are normalized relative to their respective run time for Linux 2.5.0. The lone process is less but similarly affected by some of the regressions that occurred during 2.5 development – this includes the one around 2.5.39. Like efax, however, it was not affected at all by the changes in 2.5.65.

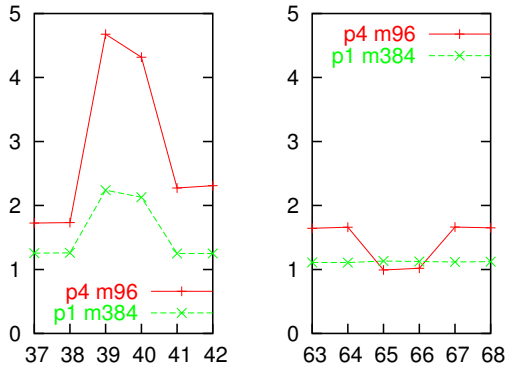

Figure 3.18.: 4 processes at 96 MB, 1 process at 384 MB compared.

Linux 2.6.0 treats all tasks equally for a long time, but one of them finishes early, and yet the overall execution time is substantially longer than for 2.5.0 or with load control. Finally, we know that an extreme kind of unfairness is the reason that graphs in figure 3.17 end after about 220 seconds.

We also note that unfairness comes in different flavors: Under load control, the amount of memory available to each process oscillates wildly, but all of them finish around the same time – on the long run, the unfairness is fair. This long term fairness is due to a victim selection that tends to pick processes with a large RSS.

A sound mechanism for evicting pages remains the cornerstone of good VM behavior under memory overload and neither load control nor unfairness can serve as a substitute. They may, however, be the best choice for some situations where regular page out logic fails: We discussed on page 62 that some popular algorithms have weaknesses of their own and fail to recognize certain memory reference patterns. But even a hypothetical, perfect algorithm is of limited use if reference locality is poor.

Therefore, a controlled form of unfairness may be an alternative to consider for cases where load control seems beneficial. Load control keeps the stunned processes from issuing any I/O

requests and lets an overloaded system come to rest, but it also tends to cause higher latencies,

3.8.4. Notes on Linux Reporting and Monitoring

During our work on load control, we made extensive use of performance data provided by the kernel. We wrote a compact C program to collect snapshot information in the /proc file system; a named pipe accepted requests by appropriately instrumented work load processes to monitor additional files [6]. Another program was written in Perl to process the resulting data files and create data files in a standard format as well as a matching gnuplot script to generate graphs for all data sets [7]. The figures on page 82 present one of 30 to 60 different graphs we routinely pulled from a system running a benchmark – the actual numbers depend on the version of the kernel since many fields were added only during Linux 2.5 development.

There is a lot more consistency in coding style for Linux sources than in formats for files in the /proc file system which come in all kinds and flavors: Files with name/value pairs separated by blank or colon, fixed-width fields, or plain lines with numbers. This makes parsing the files for information more painful than necessary.

A harder problem we hit repeatedly, though, is that the numbers provided by the kernel are sometimes inaccurate, fixed to 0, or utterly bogus. We mentioned in figure 3.1 already that a process may count both as running and blocked – these kind of inaccuracies are sometimes annoying but the effort needed for higher accuracy usually seems not worthwhile.

Especially where files in /proc offer a list of values without labels, it has been common practice to set obsolete fields to 0 in order to leave the position numbers for the following fields unchanged. Unfortunately, there is no way to tell whether a value is in fact 0 or if the field is dead. We would have preferred a standard value like “N/A” to clearly indicate when a field is not updated.

The worst case, however, is the kernel offering wrong information, a behavior we discovered on several occasions: In Linux 2.6.0, the value system calls like `wait4` and `getrusage` return with the `ru_majflt` field of `struct rusage` does not only include the number of major page faults accrued by a program but also the minor faults which did not require any disk I/O. The documentation that comes with the kernel has been confusing field numbers five and six in the process specific file /proc/$PID/statm for many years, which is a minor glitch compared to the fact that several fields in this file contain numbers of a rather dubious nature: The content is completely different between Linux 2.4 and 2.6 or matches neither the description in the documentation nor what the internal variable name implies. It can be argued that another bug we discovered and fixed falls into the same category: Linux 2.6 provides a mechanism to warn about calls from an atomic context to functions which might sleep – until recently, though, the mechanism remained silent for the first five minutes after booting a system, possibly giving a deceptive feeling of security to developers whose code executes exclusively or preferably during the boot phase.

None of these examples affect reliability, stability, or performance of a system. They do, however, make live harder for developers by feeding them misleading information. With promising developments like the new virtual filesystem sysfs, Linux seems to be in the midst of a transition

[6] Appendix H.2.
[7] Appendices H.3, H.4, H.5, and H.6.

period, but for the time being the quality of some reporting facilities in the kernel will likely continue to trail that of other parts of the kernel.

3.9. Conclusions

Thrashing – Who Cares?

For a number of reasons, thrashing has become virtually unsolvable as a problem: The access time gap between RAM and hard disk has grown. The convergence of client and server hardware and software suggests the use of generic mechanisms whenever possible. Typical usage patterns and work loads make traditional cures impractical – the kernel has lost the authority to practice admission control, and high latencies as caused by load control are usually deemed unacceptable.

As thrashing became increasingly difficult to address during the past decades, thrashing prevention became more attractive. Even more so because the price for the thrashing prevention method of choice has been shrinking continually. In our view, thrashing has ceased to be an interesting subject for operating systems research because the economical solution is almost always to add more RAM to a system. In most other cases, userspace – not the kernel – is the correct place to take a decision and action; in section 3.7.2 we presented the unlikely combination of preconditions that make load control in the kernel of a modern operating system seem beneficial.

The Real Problems

The problem with Linux 2.6.0 is not that it lacks a mechanism like control to address thrashing. The problem is the paging behavior of recent Linux kernels under circumstances where no thrashing should occur. In other words, they fail to keep the CPU busy where earlier, older kernels don't. In addition, we documented a higher variance of test results for later kernels. This kind of unpredictability is unpopular with users and it makes a solid evaluation of changes in the kernel even more time-consuming.

We have shown that the new VM introduced in Linux 2.5.27 caused a significant regression indeed. For our compile benchmarks, several and even more severe regressions followed during the later course of 2.5 development. The responsibility of the new VM for all slowdowns turned out to be nothing but a common misconception. This opens the way for coming improvements, because most regressions seem to have been caused by unintended side-effects. Our data also shows that unfairness changed significantly between 2.4 and 2.6, but that increased fairness is not the reason for the performance loss in 2.6.

Several factors make development in this area tedious. Our analysis of the regressions in Linux 2.6 showed that they were often due to subtle interactions that puzzled even the most experienced kernel developers. The same is true for improvements: They are usually based on theory, but they need to be tested in practice. The comparison of Linux 2.4 and 2.6 also indicates that some kernels exhibit better paging performance across the board. However, the results depend heavily on the benchmark: Access patterns and susceptibility to unfairness seem to be the dominating factors.

Future Research

In the immediate future, we believe improvements of paging in Linux 2.6 to be the highest priority task. Our work provides a map that should definitely speed up the process. A deeper understanding of the factors that led to lower performance and higher variance would be valuable; however, a detailed analysis of the causalities is a formidable task compared to the mere fixing of the regressions.

As we noted in appendix G, ten benchmark runs were not sufficient to provide conclusive results in all cases. However, the results are accurate enough to spot the introduction of significant regressions. The Linux development community needs regular regression testing especially in this area: Unintended side-effects are frequent, complex interactions make them hard to fix if the problem is discovered many months and kernel releases later, and users tend to be better at noticing and reporting system crashes than performance regressions. Most of this work is scriptable – even the graphs we presented in this thesis can be produced automatically, and it is often easier to spot trends and patterns in graphs.

We would like to see a systematic study of work loads, resulting in a categorization of distinct access patterns and other factors. This would facilitate the construction of a minimal benchmark to assess the paging behavior of a system.

For those special scenarios where load control might be considered, we suggest to look into a controlled form of unfairness: The upcoming cfq I/O scheduler we mentioned in section 3.5.3 might be a good starting point for a mechanism that gives each process in turn a chance to use a larger portion of the I/O bandwidth for a while. No process would be stunned, which should eliminate the latency excesses of load control. The unfairness caused by the oscillating access to the paging disk should improve throughput while the rotation ensures a long term fairness.

Beyond paging and thrashing, we submit that the Linux kernel needs some serious work as far a reporting is concerned. Based on our experience, we suggest that obsolete fields be clearly marked as such, and that volatile files in the /proc file system contain optional time stamps to identify the time when the snapshot was taken which under high load might differ considerably from the time when a userspace logging program gets a chance to add its own time stamp to the data it just read.

Generic programs for collecting and parsing performance data might be worth some additional work. We have successfully used our tools to process the output of FreeBSD's vmstat, which was rather trivial once we had the code to parse any Linux /proc file [8]. We believe that our basic architecture is sound:

- A small C program to collect raw data has a minimal impact, is easy to customize, and can be audited prior to a possible use in a production environment. When real-time results are not necessary, it can be used as a powerful alternative to tools like vmstat.

- Sophisticated back-end programs process the data – the plot program could and should be split into a log parser to generate standard data files and one or several applications to create graphs and statistics from the data files.

[8] Appendix H.7.

Our logging program receives the paths to process specific files through a named pipe. Instrumented programs can send this information automatically, but if a process forks and is not instrumented, the logging program fails to collect data about the child. There is no efficient, reliable way to find children of a given process in the /proc file system. However, a solution based on the `ptrace` system call should work in most cases.

Common programs like top and vmstat that read information from /proc must be updated whenever the data sources change. Our own code moves the format description into separate configuration files [9]. The next logical step is the most important missing feature in our tool set: The ability of the log processor to smartly guess the raw data format in the absence of configuration files to describe it. This would allow for immediate processing of a large range of numerical log data.

[9] Appendices H.4, H.5, H.6, and H.7

Part III.

Appendices

A. Source Code vs Object Code

This appendix aims to give people with no background in programming an idea of the difference between source code and object code. We refrain from sprinkling the text with footnotes pointing out all the simplifications and inaccuracies that seemed necessary to keep this basic introduction readable.

Computers deal with numbers, and all data is encoded as such. Every instruction to the computer, every letter in a text, every point of a picture, every tone of a piece of music is a number. Not surprisingly, the first step to make programs easier to read was to use place holders, so called *mnemonics*, for computer instructions and have the computer translate them to the number that are the machine instructions. This allowed programmers, for instance, to write the mnemonic `add` instead of the machine instruction `83`.

Since a direct correspondence between the machine instructions and the mnemonics exists, the reverse operation was trivial: The machine instruction `83` became the mnemonic `add` again. The set of mnemonics depended on the specific computer, of course – different computers had different sets of machine instructions.

When computers became more powerful and programs larger and more complex, frequently used code sequences were grouped together so they could be called whenever a programmer needed them. Consequently, the new generation of programs translating the programmer's work gained the ability to translate a single command given by the programmer into a whole sequence of machine instructions.

So a programmer could now write `round` to round a number, which translated to a whole sequence of machine instructions if necessary. Functionality and names of this sequences were standardized to form *programming languages*. This made it possible to write source code that worked on different computers – the only prerequisite was a translator that knew a working sequence for every command in the standard.

The growth of program size and complexity did not end there: Layer upon layer, groups were combined to form even larger sequences. Doing so allowed programmers to handle the ever increasing complexity of their programs: With one command, they could now direct proven sequences containing any number of machine instructions.

What remained virtually unchanged in the past decades, though, is the set of instructions computers offer to the programmer. Basically, they can fetch data from memory and store it back, do basic arithmetics, and decide whether to jump to a different part of the program based on the comparison of two numbers. All modern programs consist of thousands, millions, or more of those primitive machine instructions.

The list of high level commands given by the programmer is called *source code*, the version of a program that is translated to machine instructions *object code*.

A simple example illustrates the differences discussed above. It was written in an old programming language called *C* which remains quite close to the machine instructions (figure A.1).

Modern languages maintain a much higher level of abstraction. The program counts from zero to two and prints the digits 0, 1, and 2 to the screen.

```
/* This is a trivial program which prints "012" to the screen. */
#include <stdio.h>

int main()
{
        int counter = 0;        /* Set the counter to zero. */

        while (counter < 3) {   /* Repeat 2 lines below until counter=3 */
                printf("%d", counter);  /* Print counter value */
                counter = counter + 1;  /* Increment the counter */
        }

        return 0;               /* Program ends here */
}
```

Figure A.1.: Source code for a trivial sample program.

Source code comments are enclosed by `/*` and `*/`. They have no influence on program behavior but help the reader of the source code to understand it [1].

```
#include <stdio.h>

int main()
{
        int counter = 0;

        while (counter < 3) {
                printf("%d", counter);
                counter = counter + 1;
        }

        return 0;
}
```

Figure A.2.: The sample source code with the comments removed.

Translating the program source code above to machine instructions for the most common personal computer architecture results in the object code displayed in figure A.3 [2].

Of course, the object code can be made somewhat more readable again. Figure A.4 contrasts the machine instruction numbers taken from our example object code with the corresponding

[1] The casual reader in this case. In real programs, the comments tend to focus on pointing out *intent* – what is actually being done should be obvious to a reader familiar with the programming language.

[2] The numbers are written in hexadecimal code: Every digit can be in the range of 0 to 15 instead of the 0 to 9 as is the case in the decimal system most people are familiar with. Digits from a through f signify numbers from 10 to 15. Thus, the lines are nothing more than a string of digits. To the computer, though, these numbers mean specific instructions.

```
5589e583ec18895dfc83e4f031db895c240443c7042434840408e809ffffff83fb027eea8b5d
fc89ec31c05dc3
```

Figure A.3.: Object code for our sample program: A string of machine instructions.

```
machine code            mnemonic and arguments
55                      push   %ebp
89 e5                   mov    %esp,%ebp
83 ec 18                sub    $0x18,%esp
89 5d fc                mov    %ebx,0xfffffffc(%ebp)
83 e4 f0                and    $0xfffffff0,%esp
31 db                   xor    %ebx,%ebx
89 5c 24 04             mov    %ebx,0x4(%esp,1)
43                      inc    %ebx
c7 04 24 34 84 04 08    movl   $0x8048434,(%esp,1)
e8 09 ff ff ff          call   0x8048268
83 fb 02                cmp    $0x2,%ebx
7e ea                   jle    0x804834e
8b 5d fc                mov    0xfffffffc(%ebp),%ebx
89 ec                   mov    %ebp,%esp
31 c0                   xor    %eax,%eax
5d                      pop    %ebp
c3                      ret
```

Figure A.4.: Machine instructions and their respective mnemonics.

mnemonics. We note that the comments and descriptive labels (counter, printf) are unrecoverable.

The machine instructions above are only a small part of the whole object code that executes the commands in our source code example, though. Those 17 instructions only call a sequence three times. In order to actually print the three digits to the screen as defined in the source code, the program relies on the hierarchies of sequences we discussed earlier. The complete code for our trivial sample program consists not of 17 but of over 90'000 machine instructions.

Object code is all but useless to learn from which makes software quite different from novels or music records. For any program large enough to be interesting it is extremely hard to change or add functionality if only the object code is available. While short machine instruction sequences can be analyzed and understood, recreating the original source code would be akin to rebuilding a cow from hamburgers. The complexity handling works only as long as the sequence hierarchy is maintained – removing it is for most practical purposes an irreversible operation.

And this effect is a major factor giving closed source software vendors power over their customers and competitors, and last but not least an excellent cloak to hide anti-competitive and even illegal activities [3].

[3] Appendix C.2.

B. Technological Means to Prevent Unauthorized Copying

B.1. Watermarks

Watermarks are changes in an object that don't affect usability and don't prevent copying. They are used to prove the origin of an object.

A watermark hidden in an audio file may indicate the original creator of a record. Watermarks can also be hidden in binary executables, for instance by taking advantage of the redundancy in CPU instruction sets. In a simple example, every choice of an instruction with a functional equivalent may encode one bit of information – the watermark can thus be added without changing the length of the unmarked binary.

Watermarks are a weak form of protection relying on deterrence and are rarely used to protect software.

B.2. Software Activation

Common practice of shareware authors has been for decades to let people freely distribute their proprietary, closed source programs. In order to encourage regular users to pay for the application, many of these programs stop working after a while, or not all features are fully usable in the version that is freely distributable.

Upon registration, users receive a software key that unlocks the full version of the program. In recent years, large proprietary software vendors have resorted to similar measures to protect their own software that – unlike shareware – is sold in shrink-wrapped boxes and cannot be tried before purchase.

One obvious problem with software activation is that the usefulness of a purchased program relies on the continued existence of the activation method. This has been less of a problem when shareware authors used a hash function or something similar to generate a key from the user name: Once the key was obtained, it could always be used to unlock the program again at a later time.

This changed with software activation schemes which have the activation key tied to the hardware: The key would unlock the program only for one specific computer. One consequence: The possible demise of the software vendor leaves the customer in a precarious position: If the hardware breaks or needs an upgrade, there may be no way to activate the software again, although it was fully paid for. Even worse: The access to data stored in a proprietary format is at risk and may become virtually impossible at any time.

B.3. License Manager

Some license managers are third party tools that are sold to help a company keep track of the number of program licenses it purchased, and how many of them are in use.

License managers are also used by some software vendors to enforce license restrictions: Their applications require a software key checked out from a license manager to run. These applications can usually be copied and installed wherever they might be useful in a company, but only a limited number of copies are actually usable at any time: If all licenses in use, the license manager cannot hand out any more keys and attempts to start further copies of the program result in a refusal to run until a software key has been returned to the license manager or additional licenses have been purchased and installed.

Many variations of this theme are used in practice. The potential for additional security holes due to license managers is real, as the CERT Advisory CA-1997-01 illustrates. Depending on the implementation, licenses may be tied to IP addresses or hardware as with software activation. An additional drawback is that if the network goes down, clients cannot contact the license manager and programs may not work because of that.

B.4. Hardware Dongle

Dongles are small devices attached to an external I/O port of a computer – the parallel or a USB port of standard PCs, for example. They are sold together with the program they protect. The associated program queries the dongle regularly for some secret. This prevents running the program on a computer without the proper dongle. This method for ensuring excludability is based on the premise that hardware dongles are harder to copy than software.

Dongles add to the marginal production cost of software and consequently tend to be used for expensive programs. A cheaper, functionally similar method is to require that the installation CD-ROM be in the drive whenever the program is started. Both variants become impractical if a dozen or more programs all demand that their respective dongle or CD-ROM be available.

In contrast to software activation, hardware dongles have the customer hope that the dongle keeps working if the vendor disappears – experience shows that this is not necessarily a safe assumption to make, though.

B.5. Trusted Computing

In 1999, Intel experienced a PR debacle: The new Pentium III CPUs came with a software readable serial number. Privacy and consumer advocates were up in arms and filed a complaint against the company with the US Federal Trade Commission [74], while mainstream media reported prominently about a call for a boycott [70]. IBM announced they would disable the serial number in their machines equipped with that chip [71]. In 2000, Intel announced that they would phase out the CPU serial number with their then upcoming Pentium VI chips, code-named “Willamette” [48].

According to Intel, the serial number had been a “*security building block [added] in order to move the industry forward in developing secure solutions for our customers*” [45] but was not

appreciated as such by consumers. While the feature could have been put to good use in some scenarios, one concern was that it could be used to tie software to hardware as has been common in the days when mainframes reigned supreme – software might end up being tied to the serial number of one CPU.

The same conflicts exist on a higher scale around much more comprehensive plans by the Trusted Computing Group which "*will develop and promote open industry standard specifications for trusted computing hardware building blocks and software interfaces across multiple platforms, including PC's, servers, PDA's, and digital phones*" [35]. This and related initiatives are also known as TCPA (Trusted Computing Platform Alliance), TCG (Trusted Computing Group.), Palladium, and NGSCB (Next Generation Secure Computing Base). Many hardware manufacturers and Microsoft have joined forces with Intel for what has been received with little enthusiasm and much skepticism by consumer advocates, security experts, FOSS exponents, and even the press.

While the TCG explicitly states that "*to enable or embed digital rights management (DRM) technology in computing platforms*" is not their goal, the initiative and the hardware it promotes can take DRM to a new level of perfection, acting as a universal dongle. Many FOSS proponents fear that those "*security building blocks*" will be used to further restrict the rights of users and lock out competition in the software market [98], a concern echoed by consumer advocates [93] and independent security researchers [2]. One paper by leading security researchers and experts warned in 2003 [30]:

> *On the horizon, we see the co-called [sic] Trusted Computing Platform Association (TCPA) and the "Palladium" or "NGSCB" architecture for "trusted computing." [. . .] In the long term, the allure of trusted computing can hardly be underestimated and there can be no more critical duty of government and governments than to ensure that a spread of trusted computers does not blithely create yet more opportunities for lock-in. Given Microsoft's tendencies, however, one can foresee a Trusted Outlook that will refuse to talk to anything but a Trusted Exchange Server, with (Palladium's) strong cryptographic mechanisms for enforcement of that limitation. There can be no greater user-level lock-in than that, and it will cover both local applications and distributed applications, and all in the name of keeping the user safe from viruses and junk. In other words, security will be the claimed goal of mechanisms that will achieve unprecedented user-level lock-in. This verifies the relevance of evaluating the effect of user-level lock-in on security.*

We have seen that all previous technical measures in this appendix amplify the tendency of software consumers to buy from the dominant vendor, because these measures make the negative impact of the worst case scenario with a dying vendor even worse – the software becoming unusable without warning is a real possibility.

And the subject of this last section fails to inspire optimism: The consensus among the majority of those not directly involved with this latest initiative seems to be that its unique potential lies primarily in attacking competitors rather than the real threats it purports to address.

C. Proprietary Software in Practice

C.1. In Defense of Microsoft

When discussing abuse of monopoly power in IT these days, one company is already cast for the role of the villain: Microsoft. There are a number of reasons for that: Highly visible products that have become household names, formidable market capitalization, frequent expeditions in new territories [1] – all these facts make the company stand out.

Of course, there is also the fact that no other company in recent IT history has been exposed to the temptations of monopoly power nearly as intensely as Microsoft, and it may even be the case that this company has yielded to such temptations with undue enthusiasm. There is no reason, however, to believe no other company would have taken that spot had Microsoft never existed.

In fact, IBM had an antitrust suit filed against it as early as 1969 [16]:

> *There were several charges against IBM. The government contended that IBM planned to and did eliminate emerging competition that threatened the erosion of IBM's monopoly power by devising and executing business strategies which were not illegal, but which did not provide users with a better price, a better product or better service. Specifically, it was alleged that IBM had hindered the development of service and peripherals competitors by maintaining a single price policy for its machines, software and support services (bundling); it had granted discounts for universities and other educational institutions and by so doing influenced those places to select IBM computers; and that IBM had introduced underpriced models knowing that they could not be produced on time and did this to prevent the placement of competitors' machines. For example, IBM had prematurely announced new systems such as System/360 claiming that it was a superior product and that its introduction was imminent when in fact, it was several years from completion.*

Bundling, FUD, and vaporware are not Microsoft innovations [2]. They are the classic weapons of a monopolist, and we argue in this paper that regulation helped create an IT market that fosters monopolistic structures and encourages anti-competitive behavior.

Therefore, it is not Microsoft that is the problem; in fact, Microsoft must at least partially be credited for the current separation between hardware and software producers. Traditionally, hardware and an operating system to fit have been controlled by the same manufacturer. Un-

[1] For instance, Microsoft has ventured into the traveling business (Expedia), produced its own keyboards, joysticks, and gaming consoles, became an Internet access provider, and founded a joint venture together with TV broadcaster NBC.

[2] The term FUD was actually coined to describe IBM's behavior when it was fighting up-start mainframe competitor Amdahl. [86].

der such a scenario, the dominating company would have had few problems eliminating FOSS before it became a real threat [3].

The examples in this paper focus on Microsoft simply because those cases tend to be quite recent and well documented due to a number of lawsuits against the company and the rise of the Internet which provides a vast archive of relevant documents.

According to economics, there are no evil firms. Companies exist to maximize their profits by adapting to their environment. Therefore, this is not a paper about evil firms, it is a paper about welfare loss due to regulations that favor inefficient production models and production models that lead to inefficient market structures.

C.2. Vaporware and Sabotage

In 1988, Digital Research released its first version of DR-DOS as an alternative to Microsoft's MS-DOS. DR-DOS was compatible, offered additional features, and was cheaper.

One paper describes Microsoft's reaction to the growing popularity of DR-DOS [116]:

> *In April 1990, DRI introduced DR-DOS 5.0 to critical acclaim. Instantly, it began to make inroads into MS-DOS 4.0's market share. By year-end 1990, DR-DOS's share had increased to 10% of new OS shipments, leaving MS-DOS with 70% and IBM with 18%.*
>
> *Within a month of DR-DOS 5.0's inauguration, Microsoft reported development of MSDOS 5.0. Curiously, it boasted nearly all of the innovative features of the DRI product. Yet MS-DOS 5.0 was not commercially available until July 1991, more than a year after DR-DOS 5.0's release. Anticipation of the new Microsoft product, prolonged by continuous Microsoft statements indicating imminent availability, however, reined in growth of DR-DOS 5.0 sales.*

When it was finally released, MS-DOS 5.0 implemented only a few of the advanced functionality of DR-DOS 5.0 [110]. MS-DOS 5.0 had been a classic vaporware stunt.

For late beta versions of Windows 3.1, a GUI environment operating running on top of DOS, Microsoft hatched a new plan to spread doubts about the viability of DR-DOS as a platform for running Windows. In late 1991, extra code was added to Windows with the sole purpose of throwing a cryptic error message if the Windows beta was started from DR-DOS.

In 1993, Andrew Schulman posted a detailed analysis of the Windows binary in question [94], explaining why he believed that this problem was not a bug but a carefully disguised deliberate incompatibility.

Schulman went on to discuss an article in Columbia Law Review that describes how antitrust laws apply to a monopolist introducing such incompatibilities in order to drive another company out of the market. He quoted a passage that describes the golden opportunity for abuse provided by proprietary software [79]:

[3] Microsoft's gaming console which combines standard PC hardware with some proprietary additions exemplifies one possible method: In order for any program to run, it must be cryptographically signed by the hardware vendor who decides which programs are allowed to run on that platform.

> *[. . .] the plaintiff must bear the burden of proof on this issue. To establish the illegitimacy of R&D expenses by a preponderance of the evidence, the plaintiff would most likely need a "smoking gun" – a document or oral admission that clearly reveals the innovator's culpable state of mind at the time of the R&D decision. Alternatively, the plaintiff could prevail if the innovation involves such trivial design changes that no reasonable man could believe that it had anything but an anticompetitive purpose.*

As usual in cases like this, a smoking gun was not available at the time. However, through an unlikely course of events, one was unearthed after Caldera, the company that had bought the sad remains of DR-DOS, took Microsoft to court for a battle that was going to last from 1996 to 2000. The gun was a message sent in 1991 by David Cole, Microsoft's MS-DOS and Windows program manager, to Brad Silverberg, Microsoft's senior executive responsible for MS-DOS and Windows [25]:

> *It's pretty clear we need to make sure Windows 3.1 only runs on top of MS DOS or an OEM version of it. I checked with legal, and they are working up some text we are suppose to display if someone tries to setup or run Windows on a alien operating system. We are suppose to give the user the option of continuing after the warning. However, we should surely crash at some point shortly later.*
>
> *Now to the point of this mail. How shall we proceed on the issue of making sure Win 3.1 requires MS DOS. We need to have some pretty fancy internal checks to make sure we are on the right one. Maybe there are several very sophisticated checks so that competitors get put on a treadmill. Aaronr [Aaron Reynolds] had some pretty wild ideas after 3 or so beers, earleh has some too. We need to make sure this doesn't distract the team for a couple of reasons 1) the pure distraction factor 2) the less people know about exactly what gets done, the better.*

Microsoft tried and failed to throw the introduction of deliberate incompatibilities and other claims out of court and quietly settled with Caldera in 2000, before the jury trial started.

C.3. Taxing Hardware

In 1994, the USA started an investigation of alleged anti-competitive behavior and abuse of monopoly powers by Microsoft.

After documenting the firm grip Microsoft held on the operating systems market, the Competitive Impact Statement submitted by the Antitrust Division of the US Department of Justice accused the company of unlawfully using its market power to maintain or extend its market share [8]. It stated that Microsoft had pushed per-CPU licenses – having *OEM*s pay royalties for every CPU they sold, regardless of the software on the machine, effectively putting into place a Microsoft tax on hardware sales. The text described incentives the company used to convince major OEMs to sign long-term contracts and the practice of granting privileged early access to information and beta releases for OEMs and *ISV*s only if they complied with Microsoft's requests, protecting the monopoly.

The antitrust investigation ended in 1995 with a consent decree, in which Microsoft agreed to refrain from some of its license practices.

We mentioned in the previous section that Caldera, the new owner of DR-DOS, filed suit against Microsoft in 1996. During the trial, evidence surfaced for a systematic abuse of monopoly powers to drive DR-DOS out of the market, like these quotes from Microsoft internal documents regarding DR-DOS and Digital Research Inc (DRI) [26]:

> *Our DOS gold mine is shrinking and our costs are soaring - primarily due to low prices, IBM share and DR-DOS & I believe people underestimate the impact DR-DOS has had on us in terms of pricing.*
> Bill Gates to Steve Ballmer, May 18, 1989
>
> *This new contract [per processor license] guarantees MS DOS on every processor manufactured and shipped by Budgetron, therefore excluding DRI.*
> Microsoft Canada OEM Sales Monthly Report, dated March 1991
>
> *Hyundai Electronics INC. DRI is still alive. We are pushing them to sign the amendment on a processor-based license. This will block out DR once signed.*
> Joachim Kempin (in charge of Microsoft OEM Sales) Status Report, October, 1990
>
> *It looks like DRI is urging them [Vobis] to focus on DR-DOS & Lieven [Vobis' President] is complaining about the per processor license - he does not want to pay $9 with every computer and thinks about shipping DR-DOS and MS-DOS.*
> Joachim Kempin to Mike Hallman (Microsoft President), Oct. 29, 1991
>
> *I took the opportunity to negotiate with him [Lieven] in German, sign our offer as is [...] Second option - scratch the DOS clause [refuse Microsoft's demand that Vobis sign a per processor license for MS-DOS] and pay $35 for Windows instead of $15 & I have a bet with Jeff that they will sign as is. In my judgment they will hurt if they do not ship WIN and paying $35 for it is out of the question.*
> Kempin to Butler, Mar. 26, 1991

In 2000, Microsoft settled with Caldera out of court under undisclosed terms.

An antitrust trial against Microsoft that took place in the USA starting 1998 found ample evidence for numerous other incidents where the company had abused its monopoly power to fortify a dominant market position [46].

C.4. Fear, Uncertainty, and Doubt

On page 21, we defined FUD as a common industry term which refers "*to any kind of disinformation used as a competitive weapon*" [86]. Vaporware as described in section C.2 is one typical example. FUD tries to scare consumers and drive them away from competitors, usually by insinuating that competing offerings or companies are doomed.

In an internal strategy memorandum [4], Microsoft's Vinod Valloppillil described the relative immunity of FOSS (which he calls OSS) against traditional FUD in 1998 [104]:

[4] Leaked to the public and later confirmed authentic by Microsoft.

> *Long-term credibility*
> Binaries may die but source code lives forever
> *One of the most interesting implications of viable OSS ecosystems is long-term credibility.*
> *Long-Term Credibility Defined*
> *Long term credibility exists if there is no way you can be driven out of business in the near term. This forces change in how competitors deal with you.*
> *[. . .]*
> *Loosely applied to the vernacular of the software industry, a product/process is long-term credible if FUD tactics can not be used to combat it.*
> *OSS is Long-Term Credible*
> *OSS systems are considered credible because the source code is available from potentially millions of places and individuals.*
> *The likelihood that Apache will cease to exist is orders of magnitudes lower than the likelihood that WordPerfect, for example, will disappear. The disappearance of Apache is not tied to the disappearance of binaries (which are affected by purchasing shifts, etc.) but rather to the disappearance of source code and the knowledge base.*
> *Inversely stated, customers know that Apache will be around 5 years from now – provided there exists some minimal sustained interested from its user/development community.*
> *One Apache customer, in discussing his rationale for running his e-commerce site on OSS stated, "because it's open source, I can assign one or two developers to it and maintain it myself indefinitely."*

Despite this insight, FUD has frequently been used to attack FOSS, but it often came in a non-standard flavor. Allegations against FOSS projects and licenses included that they were "un-American", "a cancer", a threat to intellectual property in general, incapable of innovation, business-unfriendly, insecure, not trustworthy, or lacking a roadmap. Debunking FUD has been an important activity of FOSS speakers for many years, and their answers to the latest attacks are readily available on the Internet.

D. Software Market Numbers

The numbers in the table below are based on sales [1]. That likely means that the numbers for FOSS which can be freely copied are too low, even if it can be argued that this effect may be offset by illegal, unauthorized copying of proprietary software.

There are several data sources publicly available for each market. While they sometimes call different winners in close races, they largely agree on market shares. The one exception are Enterprise Application Systems. The distribution depends on the definition of that particular market. Considering all companies selling software for Enterprise Resource Planning (ERP), Supply Chain Management (SCM), *or* Customer Relationship Management (CRM) tends to create an overly optimistic picture of the actual competition. In addition, our observations on proprietary software markets suggest a significant market consolidation is likely to occur in the near future. And indeed, Oracle made a public bid for PeopleSoft in 2003, and Microsoft has added CRM software to its offerings.

Desktop Operating Systems [31]	Microsoft Windows	93.8%
	Apple Mac OS	2.9%
	GNU/Linux	2.8%
	Other	0.5%
Server Operating Systems [31]	Microsoft Windows	55.1%
	GNU/Linux	23.1%
	Unix	11.0%
	Netware	9.9%
	Other	0.9%
Relational Database Management Systems [33]	IBM	36.2%
	Oracle	33.9%
	Microsoft	18.0%
	NCR	2.7%
	Other	9.2%
Enterprise Application Systems [43]	SAP (ERP, SCM, CRM)	19.6%
	Siebel (CRM)	7.1%
	Oracle (ERP, SCM, CRM)	6.1%
	PeopleSoft (ERP, SCM, CRM)	4.9%
	Sage (ERP, CRM)	3.5%
	Microsoft (ERP, SCM)	2.7%
	J.D. Edwards (ERP, SCM, CRM)	2.4%
	Other	53.7%

[1] The exact nature of each number can be gathered from the references

E. Legislation and Overregulation

Uniform Computer Information Transactions Act (UCITA)

The UCITA is a US law initiative that has been hotly debated in the past few years. It goes significantly beyond TRIPS [1] and the WIPO Copyright Treaty of 1996 that require the contracting parties to "*provide effective legal remedies against the circumvention*" of measures taken to prevent unauthorized use of copyrighted works [80] and which were implemented, for instance, through the Digital Millennium Copyright Act (DMCA)[2] and the European Union Copyright Directive.

New laws are not concerned any more with the regulation of copying, which is already perfectly regulated, but with additional use restrictions that copyright holders may impose on their customers.

The UCITA is a large and complex piece of regulation: Including official commentary, the draft published in 2000 weighs in at about 800 KB or over 15'000 lines of text [75]. Over time, numerous amendments and clarifications were introduced to address some of the issues raised by critics who remained mostly unimpressed with those changes. As a proposed uniform law, the UCITA needed enactment by individual US states and therefore had a limited impact in court so far.

So why look at the UCITA at all? – It is an interesting case that we believe is indicative of the future: Stake holders in the software industry were divided into two opposing blocks more clearly than ever before: Proprietary software vendors on one side, consumers, IT professionals, and FOSS supporters on the other.

The leading voices supporting the UCITA's passage were these organizations:

- The Business Software Alliance. According to its web site, the BSA "*is the voice of the world's commercial software industry before governments and in the international marketplace. Its members represent the fastest growing industry in the world. BSA educates consumers on software management and copyright protection, cyber security, trade, e-commerce and other Internet-related issues*".

  BSA members include Adobe, Apple, Autodesk, Borland, Internet Security Systems, Macromedia, Microsoft, Network Associates, and Symantec.

- The Digital Commerce Coalition "*was formed in March 2000 by business entities whose primary focus is to establish workable rules for transactions involving the production, provision and use of computer information - digital information and software products and services. DCC members include companies and trade associations representing the*

[1] Trade-Related Aspects of Intellectual Property Rights, 1995.
[2] In the USA.

leading U.S. producers of online information and Internet services, computer software, and computer hardware".

The DCC's home page is at http://www.ucitayes.org/. DCC members include: AOL Time Warner, the American Electronics Association, Adobe Systems, Autodesk Inc, the Business Software Alliance, Dell, Intel, the Information Technology Association of America, Microsoft, the National Association of Securities Dealers, Novell, and the Software & Information Industry Association.

- The Information Technology Association Of America "*today is the only trade association representing the broad spectrum of the world-leading U.S. IT industry*".

  The membership list seems to confirm that the ITAA represents all major forces of the US IT industry.

The organizations opposing the UCITA in the form it was suggested included:

- The Association for Computing Machinery is "*the world's oldest and largest educational and scientific computing society*". Founded in 1947, the ACM has 75'000 members in 100 countries.

- The Institute for Electrical and Electronics Engineers is better known as IEEE. It is "*a non-profit, technical professional association of more than 380'000 individual members in 150 countries.*"

- "*Established in 1968, the Society for Information Management (SIM) is the premier network for IT leaders comprised of nearly 3000 members, including CIOs, senior IT executives, prominent academicians, consultants, and other IT leaders*".

- SHARE, formed in 1955, "*is a non-profit, voluntary organization whose Member organizations are users of IBM information systems. [...] SHARE now counts more than 2000 of IBM's top enterprise computing customers among its membership ranks. Collectively, these organizations - and SHARE - represent more than 20'000 individual computing specialists. Our constituency includes many of the top international corporations (including the majority of the FORTUNE 500), universities and colleges, municipal through federal government organizations, and industry-leading consultants.*"

- The American Library Association "*is the oldest and largest library association in the world, with more than 64'000 members. Its mission is to promote the highest quality library and information services and public access to information. [...]Libraries annually purchase over $100 million in electronic information products so that the passage of UCITA will have a great impact on the ability of libraries to access and use the information products they purchase*".

- Founded 1936, the Consumers Union "*is an independent, nonprofit testing and information organization serving only consumers.*"

- The Electronic Frontier Foundation "*was created to defend our rights to think, speak, and share our ideas, thoughts, and needs using new technologies, such as the Internet and the World Wide Web. EFF is the first to identify threats to our basic rights online and to advocate on behalf of free expression in the digital age.*"

- The Free Software Foundation, "*founded in 1985, is dedicated to promoting computer users' right to use, study, copy, modify, and redistribute computer programs. The FSF promotes the development and use of free (as in freedom) software – particularly the GNU operating system (used widely today in its GNU/Linux variant) – and free (as in freedom) documentation. The FSF also helps to spread awareness of the ethical and political issues of freedom in the use of software*".

Other organizations opposing the UCITA include the National Writers Union, Computer Professionals for Social Responsibility, and the American Committee for Interoperable Systems.

The UCITA pitted the large software manufacturers and associated IT vendors squarely against computer professionals, enterprise IT executives, academics, FOSS activists, and consumers – society at large.

The core of the UCITA makes "shrink-wrap licenses" enforceable, permitting software publishers to add terms to a contract with a customer *after* a program has been purchased. In addition, it makes a number of provisions enforceable that had been legally dubious before.

Many of the points raised by opponents and discussed below have been disputed by UCITA proponents [11]. That did not change the fact that two unchanged blocks of interested parties remained firmly with their positions, and precisely this dichotomy is the main point of our argument in this appendix.

Software publishers gain a number of additional rights under the UCITA:

- They are allowed to insert backdoors and time bombs into their software to enable them to unilaterally terminate a license and its associated use of a program; the law refers to such practices as "*electronic self-help*".

  If a software company folds without defusing all time bombs, the consequences are bound to be fatal. IEEE-USA points out security implications in a position paper [42] :

  > *The "self-help" provisions of UCITA would allow software publishers to embed security vulnerabilities and other functions in their software that facilitate "denial-of-service" attacks (remote disablement or destruction of the software) while avoiding liability for accidental triggering of the attacks or exploitation of these functions by malicious intruders.*

  The proponents of the law argue that the self-help provision does not interfere with security because "*security hacking can be achieved under any code. Hackers and terrorists do not need any 'backdoor;' they have already demonstrated their ability to create their own. The problem is the ability and sophistication of the hacker or terrorist, not the code*" [11].

  Customers are unlikely to assume such additional risks if given the choice, which suggests that the only beneficiaries of this provision are vendors whose customers are highly dependent already.

- The UCITA makes a software developer liable by default for flaws in a program [3]. Contract terms disclaiming any warranties, however, are declared valid.

  The explicit liability is hardly of use to any buyer of proprietary software, since current practice shows that vendors won't miss out on an opportunity to disclaim everything they can. Quoting IEEE-USA again:

  > *UCITA allows software publishers to disclaim warranties and consequential damages even for software defects known to the publisher prior to sale, undisclosed to the buyer, and having damages that can be reasonably foreseen. For example, under UCITA a software publisher could not only prohibit publication of information on security vulnerabilities that users identify but could avoid responsibility for fixing these vulnerabilities.*

- The UCITA faced stiff opposition also for its codifying of practices that curtail customers rights even further.

  – Transfers of ownership can be prohibited – a program cannot be resold. In their letter, the major library organizations observed [4]:

  > *Many digital licenses are able to – and do – restrict both the resale and lending of digital works and the licensee's ability to use lawfully obtained copies in ways that traditionally have been permitted under fair use, the first sale doctrine and the rules of preservation with regard to analog works.*
  >
  > *[...]*
  >
  > *The replacement of the traditional model of distribution of selling copies of works to the public through the licensing model of distribution of software and information products has substantial, adverse implications for consumers. We would add that, in this regard as well, UCITA hastens the erosion of user rights by codifying recent court decisions enforcing shrink-wrap licenses.*

  It is important to realize the economic impact here. Even a monopolist cannot use arbitrary price discrimination to maximize profits if a product can be easily transferred and stored. After all, if prices differed by a large amount, some customers would start reselling the products they purchased cheaply to those who were expected to pay a higher premium. Unless, of course, contracts restricting resale are found to be enforceable.

  – The publishing of benchmarks and public statements about the quality of a program may become impossible. IEEE-USA explained:

  > *By changing what would otherwise be considered a sale into a licensing transaction, UCITA permits software publishers to enforce contract provisions that may be onerous, burdensome or unreasonable, and places on*

[3] The Free Software Foundation believes FOSS developers to be exempt from the default liability because they never enter a contractual relationship with their users [72].

the purchaser the burden and cost of proving that these provisions are unconscionable or "against fundamental public policy." Examples of these provisions include prohibitions against public criticism of the software and limitations on purchasers' rights to sell or dispose of software. The first provision prohibits the reviews, comparisons, and benchmark testing that are critical for an informed, competitive marketplace. The second issue could legally complicate transactions including corporate mergers/acquisitions, sales of small businesses, the operation of businesses dealing in second-hand software, and even yard sales.

Software publishers share with other copyright owners the desire to weaken the *exhaustion of rights doctrine* and the *first sale doctrine*, two legal principles that limit the powers of copyright owners once a copy of their work has been sold.

But why does the IEEE comment mention corporate mergers? – A firm may have to repurchase (or rather, relicense) software in use at the acquired company, if only that company that just ceased to exist had a license to use the software and it was prohibited from transferring those rights to the purchasing firm.

– Reverse engineering can be prohibited even if it is the only means to provide interoperability. ACM noted [96]:

> *UCITA threatens normal engineering activities, especially reverse engineering. UCITA allows publishers to ban reverse engineering by means of contractual use restrictions. The only limits on these bans require litigation of each and every use that a computer researcher might reasonably pursue to improve a product or correct a flaw in a program. Software developers can freely reverse engineer mass-market products under current law. Without extensive litigation, over a span of many years, this right will be clouded by UCITA.*
>
> *Reverse engineering is a widespread, standard, critically important activity in the software engineering and research communities. How else could we detect and investigate security risks? How else could we develop programs that impede the spread of viruses?*
>
> *How else could we make products interoperable? Many of the Y2K bug fixes have required reverse engineering. It is hard enough to solve the technical problems without the creation of additional legal hurdles. By allowing the establishment of legal restrictions on reverse engineering, UCITA will have real-world effects. It will impede computer research and potentially threaten public safety as the problems with Y2K, computer viruses, and software bugs become more widespread.*

In other words, a software vendor can make it impossible for any competitors to communicate with a program or read its data.

This paper presents overwhelming evidence that the proprietary, closed source production model is inherently and massively imbalanced to favor vendors over consumers and large ven-

dors over smaller ones. Not surprisingly, the largest and most mature software market tend to be vendor markets with a few suppliers dominating both their competition and the consumers.

It is a simple exercise to go through the wish list the UCITA represents and to realize that all provisions have proprietary software vendors as their main beneficiaries – in particular the large, dominant among them. No part is of any use to software users or FOSS developers.

It is painfully obvious that the publishing of benchmarks without vendor approval is not an obstacle on the way to a better, brighter software world. Neither are the transferability of software or interoperability based on the results of reverse engineering. However, it is all too clear why dominant vendors would like to enforce license agreements that prohibit all that.

The organizations behind laws like the UCITA are advocating more regulations in what can only be interpreted as a barely disguised attack on free market forces, as an attempt to eradicate whatever competition is left in the software market. Hoping that laws amplifying the known problems of this market will spur innovation, competition, and choice seems to be an audacious proposition, especially considering the history of IT markets.

However, the UCITA works well to expose the goals of various interest groups and we believe that it delineates the regulations that major parts of the proprietary software industry will keep pushing in the future.

F. Total Cost of Ownership

The TCO concept has been introduced by the Gartner Group in 1987. Initially conceived for desktop systems, it has been extended over the years. A number of research groups offer TCO models for specific scenarios in many other areas of the IT landscape, using surveys and statistics to break down IT related expenses, for instance, into costs for hardware, software, training, and management.

TCO analysis is a tool for enterprises trying to understand their IT cost structure. It is frequently used to assess potential cost savings. In recent years, though, TCO has been increasingly used in marketing, where the term is often supposed to justify selecting and weighing costs to make a product look good.

But even where the standard model of one research group has been impartially applied, such studies hardly warrant headlines touting the superiority of a product over its competitors – the results are valid only for one specific scenario and can rarely be simply adopted for other cases.

This practice can be observed on a little excursion into today's software market. Databases arguably are a prime example of enterprise computing. In the RDBMS space, competition still exists within both proprietary and FOSS products. Task and scope are fairly well defined and limited compared to other areas like ERP. Consequently, one would expect to find first rate TCO information in this very area.

The PostgreSQL web site invites its visitors to "*join the PostgreSQL revolution, and take advantage of Low Total Cost of Ownership (TCO)*", but fails to substantiate its claims with numbers. The MySQL web site seems equally devoid of any useful TCO information[1].

In contrast, finding detailed TCO information requires very little effort for each of the leading proprietary RDBMS vendors.

Microsoft introduces its TCO study [68]:

> *To compete in today's business climate, IT professionals must streamline costs by controlling application delivery and support costs. SQL Server plays a key role in maximizing productivity and lowering total cost of ownership (TCO). NerveWire, a management consulting and system integration firm, recently completed a study of ten companies that use SQL Server as well as a major competitor's database offering. NerveWire explored how each product affected TCO during a three-year period. They assessed costs in the following areas: software, hardware, maintenance, design and development, ongoing activities, and training. The results demonstrate clearly that SQL Server helps firms manage infrastructure costs and generate savings of up to 50 percent throughout the life of an application.*

Oracle provides a document titled "*DB2's 'Low Cost' Conclusion Is a Delusion – Oracle is the TCO Leader*" [78], while IBM exclaims "*Hear it from the customers! Leading customers*

[1] Visited November 2003.

choose DB2 over Oracle and Microsoft for its superior TCO" and "*Don't believe the hype! Oracle's Rauch Report does not reflect the real TCO advantage that DB2 maintains over Oracle*" [41]. Each vendor offers studies praised as "*independent*" only to see them denounced as "*commissioned*" by the competition.

The quotes above give away an open secret of the IT industry: TCO studies are not only dependent on a specific scenario, but also highly controversial and subject to tacit assumption and interpretation.

There are, of course, also TCO studies that show FOSS to be more costly than proprietary software, and they are equally limited in their universality, but they became the weapon of choice in Microsoft's new "*fact-based*" campaign against GNU/Linux [23]. After a commissioned study had been used again to make broad marketing claims, Forrester Research bailed out from what must be a lucrative market in order to save its credibility [66]:

> *Forrester Research Inc. has changed its policy toward vendor-sponsored research following last month's publication of a controversial Microsoft Corp.-funded study that compared the cost of developing applications on Linux and Java to a Microsoft-based approach.*
>
> *The policy change was announced in a letter written by George Colony, the CEO of the Cambridge, Massachusetts, company, and posted to the Forrester Web site late last week.*
>
> *"We will no longer accept paid for, publicized product comparisons," Colony said in an interview. "The best example of that would be the Microsoft report."*
>
> [...]
>
> *"[Microsoft Platforms Strategist] Martin Taylor went out and visited with a bunch of reporters, and he was referring to the study and using it to advance his case that Linux doesn't have a lot of advantages," said [Forrester analyst John] Rymer. "George (Colony) was uncomfortable with this."*

Studying a number of TCO studies can certainly serve an IT decision maker as a starting point, for example by providing a range of criteria for her own TCO study – if she decides that a TCO analysis is required to take a decision. In any case, a forward-looking IT executive may want to consider figuring in costs that traditional TCO models miss, such as those discussed in this paper.

G. A Word on Statistics

The sample size of $n = 10$ we used for benchmarks is clearly too small. It was chosen due to time constraints – the test machine was a scarce resource, and there was no telling which experiments would yield interesting results at the beginning of this project. Even so the data used for figure 3.8 alone represents 2630 successful benchmark runs and a total run time of close to 300 hours.

In order to eliminate the influence of mavericks we favored median ($\widetilde{x}$) over mean ($\bar{x}$) values for most of our analysis. We found mean and median to rarely differ to a significant degree, though, which is also visible in figures 3.9, 3.10, 3.12 and table G.1. Benchmark run times were divided by their respective $\widetilde{x}$ for Linux 2.4.15 / 2.5.0. In other words: for that kernel, each benchmark was assumed to execute in a relative time 1.0.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{G.1}$$

Kernel	efax		kbuild		qsbench	
	$\bar{x}$	$\widetilde{x}$	$\bar{x}$	$\widetilde{x}$	$\bar{x}$	$\widetilde{x}$
Base	3.7	3.7	3.5	3.4	1.7	1.7
2.4.15 / 2.5.0	1.0	1.0	1.0	1.0	1.0	1.0
2.4.23	1.0	1.0	0.9	0.9	1.0	1.0
2.6.0	3.7	3.7	3.5	3.4	1.7	1.7

Table G.1.: Mean and median relative execution times.

We calculated the sample standard deviation $\hat{s}$ as:

$$\hat{s} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} = \sqrt{\frac{1}{9}\sum_{i=1}^{10}(x_i - \bar{x})^2} \tag{G.2}$$

For a meaningful comparison of standard deviation numbers, we used the relative standard deviation $\hat{s}_r$:

$$\hat{s}_r = \frac{\hat{s}}{\bar{x}} \tag{G.3}$$

While $\hat{s}_r$ remained at a high level for the kbuild benchmark, it increased considerably for efax and qsbench between Linux 2.5.0 and 2.6.0. Confidence intervals were equally affected, of course. Under the given circumstances we used the Student's t-distribution with 9 degrees of freedom $T\langle 9\rangle$.

For a modest 90% confidence the interval is:

$$[\bar{x} - t_{0.05}\frac{\hat{s}}{\sqrt{n}} \leq \mu_x \leq \bar{x} + t_{0.05}\frac{\hat{s}}{\sqrt{n}}] \tag{G.4}$$

After inserting $n = 10$ and $t\langle 9\rangle_{0.05} = 1.833$ into (G.4) the confidence interval used for table G.2 is:

$$[\bar{x} - 0.58\hat{s} \leq \mu_x \leq \bar{x} + 0.58\hat{s}] \tag{G.5}$$

	efax			kbuild			qsbench		
Kernel	$\widetilde{x}$	$\hat{s}_r$	90% CI	$\widetilde{x}$	$\hat{s}_r$	90% CI	$\widetilde{x}$	$\hat{s}_r$	90% CI
2.5.0	1.0	0.006	0.997, 1.003	1.0	0.099	0.928, 1.040	1.0	0.011	0.992, 1.005
2.4.23	1.0	0.005	0.998, 1.004	0.9	0.138	0.797, 0.936	1.0	0.018	0.965, 0.985
2.6.0	3.7	0.023	3.674, 3.773	2.9	0.106	2.815, 3.186	1.4	0.167	1.296, 1.574

Table G.2.: Median, relative standard deviation, and 90% confidence interval.

H. Source Code

H.1. thrash.c

Our first attempt at a thrashing load. See section 3.6.

```
/*
 * Author : Roger Luethi <rl@hellgate.ch>, 2003
 * Version: 0.5.2
 * Purpose: Generate thrashing load plus control codes for log program
 * Todo   : Options to control the load that is generated
 */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <sys/wait.h>
#define PAGESIZE        4096
#define MEMSIZE         64*1024*1024
#define WORKSET         MEMSIZE/2
#define CHILDREN        3
#define PASSES          8

static const int SEQ            = 0;    /* Sequential or parallel processing? */

/* Commands for log program */
static const char t_watch[]     = "!watch";     /* Watch this file */
static const char t_unwatch[]   = "!unwatch";

/* Tags for log processor */
static const char t_cover[]     = "#COVER";
static const char t_label[]     = "#LABEL";

                                        /* Per process files in /proc/<pid/ */
const char *files[] = {
        "status",
        "stat",
        NULL    /* Marker */
};

/*
 * Load: allocate a chunk of memory, keep those pages read & dirty
 */
int work() {
        char *buf;
        int ii, jj, tmp;

        buf = (char *)malloc(WORKSET);
        memset(buf,0x01,WORKSET);

        tmp = 0;
        for (jj = 0; jj < PASSES; jj++) {
                for (ii = 0; ii < WORKSET; ii += PAGESIZE)
                {
                        tmp += buf[ii];
                        /* For good measure, make sure pages are always dirty */
                        buf[ii+8] = tmp;
                }
        }
        free(buf);
        /* Careful: removing the statement below will optimize the work
           loop away beginning with -O1! */
        printf("#COVER %d: %d passes @ %d pages (%d)\n", getpid(),
                        PASSES, WORKSET/PAGESIZE, tmp);
        return 0;
}

/*
```

```
 * Wait for all children created so far
 */
int harvest() {
        int cpid, i;

        while ((cpid = wait(NULL)) != -1) {
                i = 0;
                while (files[i]) {
                        printf("%s /proc/%d/%s\n", t_unwatch, cpid,
                                        files[i]);
                        i++;
                }
                printf("%s %d dead\n", t_label, cpid);
        }

        return 0;
}

/*
 * Create a bunch of load processes (sequentially or in parallel, depending
 * on option); time keeping
 */
int main()
{
        int cpid;
        int chi;

        setlinebuf(stdout);
        printf("%s thrash: %s processes: %d ", t_cover,
                        SEQ ? "sequential" : "parallel", CHILDREN);
        printf("(working sets: %d MiB, %d pages)\n", WORKSET / (1024*1024),
                        WORKSET / PAGESIZE);
        for (chi = CHILDREN; chi > 0; chi--) {
                cpid = fork();
                if (cpid) {
                        int i = 0;
                        printf("%s %d forked\n", t_label, cpid);
                        while (files[i]) {
                                printf("%s /proc/%d/%s\n", t_watch, cpid,
                                                files[i]);
                                i++;
                        }
                        if (SEQ) harvest();
                }
                else {
                        int cpu_0;      // CPU: wraps around after 72 minutes
                        struct timeval tv0, tv1;
                        float wall;
                        struct timezone tz0, tz1; // should be merged to tz
                        gettimeofday(&tv0,&tz0);
                        cpu_0 = clock();
                        work();
                        printf("%s %d: CPU time : %fs\n", t_cover, getpid(),
                                        (float)(clock()-cpu_0)/CLOCKS_PER_SEC);
                        gettimeofday(&tv1,&tz1);
                        wall = (float) tv1.tv_sec - tv0.tv_sec +
                                        (tv1.tv_usec - tv0.tv_usec) / 1.0e6;
                        printf("%s %d: Wall time: %fs\n", t_cover, getpid(),
                                        wall);
                        break;
                }
        }
        if (!SEQ) harvest();
        return 0;
}
/* gcc -Wall -Wshadow -Wcast-align -Winline -Wformat=2 -o thrash thrash.c */
```

H.2. log.c

The logging program. See sections 3.8.4 and 3.9.

```
/*
 * Author : Roger Luethi <rl@hellgate.ch>, 2003
 * Version: 0.6.1
 * Purpose: Client controlled data gathering and logging
 * Todo   : Log to network socket instead of stdout
 *          Cmd to fork (e.g. vmstat) and log output into sections
 *          Error handling
 */
```

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <signal.h>
#define DIE { \
        printf("%s:%d: Fatal error.\n", __FILE__, __LINE__); \
        exit(1); \
} while (0)
#define DIEE { \
        printf("%s:%d: Fatal: %s\n", __FILE__, __LINE__, strerror(errno)); \
        exit(1); \
} while (0)


#define INITMAXFILES    4
#define GRABBUFSIZE     1024
#define MSGBUFSIZE      128
#define MAXPATHLEN      64
#define LINEBUFFERED    1                       /* 0: block buffered output */

/* Input tags (from client) */
static const char t_watch[]     = "!watch";     /* Client command code */
static const char t_unwatch[]   = "!unwatch";

/* Output tags (for log processor) */
static const char t_delim[]     = "#STEP";      /* Log entry delimiter */
static const char t_begin[]     = "#BEGIN";     /* Log section tag */
static const char t_end[]       = "#END";

struct file {
        int fd;
        char *path;
};

static struct file ctrl = {
        .path   = "ctrl"        /* Path to control fifo */
};

static struct file *files;
static int maxfiles;

/* System-wide standard files */
//const char *sys[] = {
static const char *std[] = {
        "/proc/meminfo",
        "/proc/stat",
        "/proc/swaps",
        "/proc/vmstat",
        NULL                    /* End of array marker */
};

void sighandler (int sig) {
        unlink(ctrl.path);
        exit(0);        /* Make sure buffers are flushed on SIGINT (Ctrl-C) */
}

int addfile(const char *path) {
        int slot = -1;
        int fd;
        int len;
        int ii;
        char *tmp;

        if ((len = (strlen(path) + 1)) > MAXPATHLEN) {
                /* TODO Print error */
                return -1;
        }
        if ((fd = open(path, O_RDONLY)) == -1) {
                /* TODO Print error */
                return -1;
        }

        for (ii = 0; ii < maxfiles; ii++) {
                if (files[ii].path == NULL) {
                        if (slot == -1) {
                                slot = ii;      /* Remember first empty slot */
                        }
```

```
                        continue;
                }
                if (strcmp(path, files[ii].path) == 0) {
                        /* Already watching this file -- skip */
//                      fprintf(stderr, "WARN Already watching %s\n", path);
                        return -2;
                }
        }
        if (slot == -1) {
                slot = maxfiles;
                maxfiles += INITMAXFILES;
                files = realloc(files, maxfiles * sizeof(struct file));
                /* Clear new buffer memory */
                memset((files + slot), 0, INITMAXFILES * sizeof(struct file));
        }

        tmp = malloc(len);
        strcpy(tmp, path);
        fprintf(stderr, "INFO Adding file %s (slot %d)\n", path, slot);
        files[slot].fd = fd;
        files[slot].path = tmp;

        return 0;
}

int removefile(char *path) {
        int slot = 0;

        while (slot < maxfiles) {
                if (files[slot].path != NULL) {
                        if (strcmp(path, files[slot].path) == 0) {
                                fprintf(stderr, "Removing file %s\n", path);
                                free(files[slot].path);
                                files[slot].path = NULL;
                                close(files[slot].fd);
                                return 0;
                        }
                }
                slot++;
        }
        return -1;
}

int grab(struct file *src) {
        int fd = src->fd;
        int n;
        static char logbuf[GRABBUFSIZE];

        if (lseek(fd, 0, SEEK_SET) < 0) {
                printf("DEBUG Error seeking fd %d: %s\n", fd, strerror(errno));
                removefile(src->path);
                return -1;
        }
        if ((n=read(fd, logbuf, GRABBUFSIZE - 1)) < 0) {
                printf("DEBUG Error reading fd %d: %s\n", fd, strerror(errno));
                if (errno != EAGAIN)
                        removefile(src->path);
                return -1;
        }

        logbuf[n] = '\0';

        printf("%s %s\n", t_begin, src->path);
        printf("%s", logbuf);
        printf("%s %s\n", t_end, src->path);

        return 0;
}

int process_msg(char *buf) {
        static char cmd[32];
        static char section[32];
        int rc = 0;

//      printf("String:%s:\n", buf);

        if (buf[0] == '#') {
                fprintf(stderr, "%s", buf);
                printf("%s", buf);
                return 0;
        }

        sscanf(buf, "%31s %31s", cmd, section);
        if (strcmp(cmd, t_watch) == 0) {
```

```
                addfile(section);
        }
        else if (strcmp(cmd, t_unwatch) == 0) {
                removefile(section);
        }
        else {
                /* TODO XXX */
                printf("ERR Ctrl string %s", buf);
        }

        return rc;
}

int main()
{
        FILE *fctrl;
        int ii;
        char *msgbuf;
        int slot;

        msgbuf = malloc(MSGBUFSIZE);
        files = malloc(INITMAXFILES * sizeof(struct file));
        memset(files, 0, INITMAXFILES * sizeof(struct file));
        maxfiles = INITMAXFILES;

        if (LINEBUFFERED)
                setlinebuf(stdout);
        if (signal(SIGINT, sighandler) == SIG_ERR) { DIE; }
        if (signal(SIGTERM, sighandler) == SIG_ERR) { DIE; }

        /* Open control channel */
        unlink(ctrl.path);
        if (mkfifo(ctrl.path,0600) < 0) { DIEE; }
        if ((ctrl.fd = open (ctrl.path, O_RDONLY | O_NONBLOCK)) < 0) { DIEE; }
        fctrl = fdopen(ctrl.fd, "r");
        fprintf(stderr, "Listening for commands on FIFO \"%s\".\n", ctrl.path);

        for (ii = 0; std[ii] != NULL; ii++) {
                addfile(std[ii]);
        }

        for (;;) {
                printf("%s misc\n", t_begin);
                printf("timestamp %d\n", (int)time(NULL));
                printf("%s misc\n", t_end);

                /* Read and execute commands */
                while (fgets(msgbuf, MSGBUFSIZE, fctrl) > 0) {
                        process_msg(msgbuf);
                };

                /* Gather data */
                for (slot = 0; slot < maxfiles; slot++) {
                        if (files[slot].path == NULL) {
                                continue;       /* Empty slot, next... */
                        }
                        grab(&files[slot]);
                }
                printf("%s\n",t_delim);
                sleep(1);
        }
        /* We can never get here */
        unlink(ctrl.path);
        free(msgbuf);
        return 0;
}
/* gcc -Wall -Wshadow -Wcast-align -Winline -Wformat=2 -o log log.c */
```

H.3. plot

The log processor generates standard data files and scripts to drive gnuplot, a graphing software. See sections 3.8.4 and 3.9.

```
#!/usr/bin/perl -w
# Author : Roger Luethi <rl@hellgate.ch>, 2003
# Version: 0.9.3
# Purpose: Produce statistical raw data files and postscript plots from log
#          files
# Note   : The input log file is a series of ASCII data snapshots (usually
```

```
#          split into sections (e.g. data sources)), separated from each other
#          by a delimiter; each data line contains a number of "name value"
#          pairs (like many /proc files do). Regular expressions can be used
#          where necessary to extract data from more complex source lines.
# Caveat : This program assumes you know what you are doing; in particular, it
#          expects to find a number of executables (see below).
# Todo   : Read section descriptions from file
#          Select appropriate config file automatically
#          Best effort guesses to pull out data if config file absent
# Bugs   : Trying to put a data set / plot on more than one sheet may result
#          in unexpected results (e.g. key is the same for delta bookkeeping)
#          We'd have to use FileCache.pm to print to more files than the
#               system fd limit will allow
#          Hangs if file ends mid-section

use diagnostics;
use strict;
use File::Basename;
use FileHandle;
use English;
#-------------------------------------------------------------------------------
# Load section descriptions from same directory
use lib dirname($0);

# Linux (/proc data gathered with log program)
use linux24;
my @secdescs = @linux24::secdescs;
#use linux26;
#my @secdescs = @linux26::secdescs;

# FreeBSD vmstat output
#use freebsd;
#my @secdescs = @freebsd::secdescs;

# Linux vmstat output
#use linuxvmstat;
#my @secdescs = @linuxvmstat::secdescs;
#-------------------------------------------------------------------------------

unless ($ARGV[0]) {
        print "Usage: $0 <data file>\n";
        exit;
}

# Executable paths
my $a2ps        = "/usr/bin/a2ps";      # package a2ps
my $gnuplot     = "/usr/bin/gnuplot";   # package gnuplot
my $gs          = "/usr/bin/gs";        # package ghostscript
my $psnup       = "/usr/bin/psnup";     # package psutils

# File names
my $flog        = $ARGV[0];
my $base        = basename($flog, '.log');
my $fdat        = "$base.dat";
my $fgnu        = "$base.gnu";
my $fps         = "$base.all.ps";
my $fplot       = "$base.plot.ps";
my $finfo       = "$base.info";
my $fcover      = "$base.cover.ps";

foreach ($a2ps, $gnuplot, $gs, $psnup) {
        die "Can't find $_" if ! -f;
}

# Data stream section tags (data set delimiters)
my $t_begin             = "#BEGIN";
my $t_end               = "#END";
my $t_label             = "#LABEL";
my $t_cover             = "#COVER";
my $t_step              = "#STEP";

# Dump section descriptions
#print "size: $#secdescs\n";
#foreach $secdesc (@secdescs) {
#       print "pattern: $secdesc->{pattern}\n";
#       foreach $ldesc (@{ $secdesc->{ldescs} }) {
#               if (defined $ldesc->{pattern}) {
#                               print " ldescs pattern: $ldesc->{pattern}\n";
#               }
#               foreach $i ('namepat', 'valpat', 'plotnr', 'delta', 'first') {
#                       if (defined $ldesc->{$i}) {
#                               print "         ldescs $i: $ldesc->{$i}\n";
#                       }
#               }
```

```
##              print " ldescs pattern: $ldesc->{pattern}\n";
#       }
#}
#exit;
#-------------------------------------------------------------------------------
my $step = 0;           # Record counter
undef my $has_cover;    # Do we have information to put on cover page?
undef my $xvar;         # Column to use for x axis
my %cols;
my %labels;
undef my $xvarkey;
my @linscale;
my %subtract;
my %prev;
my @colkeys;
my $label;
#-------------------------------------------------------------------------------
# Initialize default .dat file
my %dfiles;
$dfiles{default}{fd} = new FileHandle;
open $dfiles{default}{fd}, ">$fdat";
$dfiles{default}{cnt} = 0;
#-------------------------------------------------------------------------------
sub streq {
        my $var1 = shift;
        my $var2 = shift;
        return 1 if (defined $var1 && defined $var2 && $var1 eq $var2);
        return 0;
}
#-------------------------------------------------------------------------------
sub cover_data {
        my $str = shift;

        unless ($has_cover) {
                open COVER, ">$finfo";
                $has_cover = 1;
        }
        print COVER $str;
}
#-------------------------------------------------------------------------------
sub store {
        my $secname = shift;
        my $ldesc = shift;
        my $line = shift;
        my ($name, $value, $cmd);

        unless (defined $ldesc->{namepat} &&
                        (defined $ldesc->{valpat} ||
                        defined $ldesc->{valcol})) {
                ($name, $value) = split ' ', $line;
#               print "SPLIT $name $value\n";
        }
        if (defined $ldesc->{namepat}) {
                $name = $line;
                # $name must not be evaluated, the namepattern must end up
                # single-quoted
                $cmd = '$name = '."'$ldesc->{namepat}'";
                eval $cmd;
#               print "NAMEPAT $name:\n";
        }
        if (defined $ldesc->{valpat}) {
                $value = $line;
                $cmd = '$value = '."'$ldesc->{valpat}'";
#               eval $cmd;
#               print "VALPAT $value:\n";
        }
        if (defined $ldesc->{valcol}) {
                $value = $line;
                $cmd = '$value =~ s/(\S+\s+){'.$ldesc->{valcol}.'}(\S+).*/$2/';
                eval $cmd;
#               print "VALCOL $value ($ldesc->{valcol}):\n";
        }

        my $key = "$secname.$name";
#       print "KEY:$key:\n";

        # Create new column if we haven't seen this key before
        unless (defined $cols{$key}) {
                my $datf;

                $cols{$key}{ldesc} = $ldesc;

                # Does this column go to a special file?
                if ($ldesc->{dat}) {
```

```
                        $datf = $ldesc->{dat};

                        # Open .dat file (if not open already)
                        unless ($dfiles{$datf}{fd}) {
                                $dfiles{$datf}{fd} = new FileHandle;
                                open $dfiles{$datf}{fd}, ">$fdat.$datf";
                                if (defined $xvarkey) {
                                        $dfiles{$datf}{cnt} = 1;
                                }
                                else {
                                        $dfiles{$datf}{cnt} = 0;
                                }
                        }
                }
                else {
                        $datf = 'default';
                }
                $cols{$key}{dfile} = $dfiles{$datf};

                # Save the per file column number
                $cols{$key}{nr} = $dfiles{$datf}{cnt};
                $dfiles{$datf}{cnt}++;

                # Update per file header string and request printing it
                if ($dfiles{$datf}->{header}) {
                        $dfiles{$datf}->{header} .= " $key";
                }
                else {
                        # Special file with an x variable?
                        if ($ldesc->{dat} && $xvarkey) {
                                $dfiles{$datf}->{header} = "$xvarkey $key";
                        }
                        else {
                                $dfiles{$datf}->{header} = $key;
                        }
                }
                $dfiles{$datf}->{print_header} = 1;
#               print "$datf header: $dfiles{$datf}->{header}\n";

                # Linear plot scale requested?
                if ($ldesc->{linscale}) {
                        $linscale[$cols{$key}{ldesc}{plotnr}] = 1;
                }
                # Is this the variable for the x axis?
                if ((!defined $xvarkey) && $ldesc->{xvar}) {
                        $xvarkey = $key;
#                       print "XVAR $key\n";
                }

#               print "column: file $datf\tcol $cols{$key}{nr}\tkey $key\n";
        }

        # Shave off values relative to first value (if requested)
        if (defined $cols{$key}{ldesc}{first}) {
                unless (defined $subtract{$key}) {
                        # Initial value defines starting level for column
                        $subtract{$key} = $value - $cols{$key}{ldesc}{first};
                }
                $value -= $subtract{$key};
        }

        # Use delta values instead (if requested)
        if ($cols{$key}{ldesc}{delta}) {
                if (defined $prev{$key}) {
                        my $tmp = $value;
                        $value -= $prev{$key};
                        $prev{$key} = $tmp;
                }
                else {
                        $prev{$key} = $value;
                        # The first delta is unknown
                        $value = '?';
                }
        }

        # Ugly workaround to prevent gnuplot from coming down hard if we
        # have a data set consisting entirely of zeroes with logscale.
        # Currently, we disable logscale for this plot.
        if (($value ne 0) && ($value ne '?')) {
                $cols{$key}{non_zero_value} = 1;
        }

        $cols{$key}{value} = $value;
#       print "SAVED ($key|$value)\n";
```

```
}
#-------------------------------------------------------------------------------
sub read_section {
        my $secname = shift;    # Section name
        my $ii = 0;

        # Is it info data (for cover page)?
        if ($secname eq 'cover') {
                while (<>) {
                        last if /^$t_end\s+/;
                        cover_data($_);
                }
        }

        # Do we have a section desc for this?
        my $secdesc;
        while ($ii <= $#secdescs) {
                $secname =~ /^$secdescs[$ii]->{pattern}$/ and do {
#                       print "NOTE Found section: $secname\n";
                        $secdesc = $secdescs[$ii];
                        last;
                };
                $ii++;
        };

        # Unknown section: return after dropping section
        if ($ii > $#secdescs) {
#               print "NOTE Unknown section: $secname\n";
                while (<>) {
                        last if /^$t_end\s+/;
#                       print "discarding: $_";
                }
                return;
        }

        # Known section: read section, looking for known line patterns
        my $lnr = 0;            # Section line number
        while (my $line = <>) {
                my $ldesc;
                last if ($line =~ /^$t_end\s+/);
                if ($secdesc->{linenr}) {
                        $line = "#$lnr $line";
                }
                # Find a line desc matching the current input line
                my $ii = 0;
                while ($ii <= $#{ $secdesc->{ldescs} }) {
                        if ($line =~ /^$secdesc->{ldescs}[$ii]->{pattern}/) {
                                chomp $line;
                                store($secname, $secdesc->{ldescs}[$ii], $line);
                                # Don't skip the remaining entries, we might
                                # have more than one match (e.g. with different
                                # options)
                        }
                        $ii++;
                }
                $lnr++;
        }
}
#-------------------------------------------------------------------------------
# Main loop
#-------------------------------------------------------------------------------
LINE : while (<>) {
        # Section start
        /^$t_begin\s+/ and do {
                chomp;
                s/^$t_begin\s+//;
                read_section($_);
                next LINE;
        };

        # Mere comments go to main .dat file only
        /^#\s/ and do {
                print { $dfiles{default}{fd} } $_;
                next LINE;
        };

        # Text for cover page
        /^$t_cover\s+/ and do {
                s/^$t_cover\s+//;
                cover_data($_);
                next LINE;
        };

        # Label for x axis
```

```
        /^$t_label\s+/ and do {
                chomp;
                print { $dfiles{default}{fd} } $_;
                s/^$t_label\s+//;
                if ($label) {
                        # We cannot plot more than one label per step
                        print { $dfiles{default}{fd} }
                                " (not in plot, shadowed by $label)";
                }
                else {
                        # Store: Only at the end of this timestep will we have
                        # the x variable value for sure (if there is one)
                        $label = $_;
                }
                print { $dfiles{default}{fd} } "\n";
                next LINE;
        };

        # Time step: write one line to .dat file(s)
        /^$t_step$/ and do {
                @colkeys = sort { $cols{$a}{nr} <=> $cols{$b}{nr} } keys %cols;

                foreach my $col (@colkeys) {
                        my $value = $cols{$col}{value};

                        # Make sure we have _some_ value
                        if (defined $value) {
                                undef $cols{$col}{value};
                        }
                        else {
                                # Placeholder for unknown value
                                $value = '?';
                        }

                        if ($xvarkey && $col eq $xvarkey) {
                                # Prepend xvar value to all lines going to
                                # special files
                                foreach my $datf (keys %dfiles) {
                                        next if ($datf eq 'default');
                                        unshift @{ $dfiles{$datf}->{row} },
                                                        $value;
                                }
                                # Store label (if any), with x value
                                if ($label) {
                                        $labels{$value} = $label;
                                        undef $label;
                                }
                        }
                        push @{ $cols{$col}{dfile}->{row} }, $value;
                }
                # Store label, without x value
                if ($label) {
                        $labels{$step} = $label;
                        undef $label;
                }
                foreach my $datf (keys %dfiles) {
                        if ($dfiles{$datf}->{print_header}) {
                                # XXX Should headers contain 'option'?
                                print { $dfiles{$datf}{fd} }
                                                "# $dfiles{$datf}->{header}\n";
                        }
                        undef $dfiles{$datf}->{print_header};

                        print { $dfiles{$datf}{fd} }
                                "@{ $dfiles{$datf}{row} }\n";

                        @{ $dfiles{$datf}{row} } = ();
                }

                # Display a progress meter (unbuffered)
                $|++;
                print "Parsed step $step\r";
                $|--;

                $step++;
                next LINE;
        };
                                        # So it's a data item after all
}
#-------------------------------------------------------------------------------

close COVER if ($has_cover);
foreach my $datf (keys %dfiles) {
        close $dfiles{$datf}{fd};
```

```
}
#-------------------------------------------------------------------------------
# Create plots

open GNU, ">$fgnu";
print GNU "set output \"$fplot\"\n";
# Terminal examples: "latex", "postscript eps enhanced color"
# "postscript enhanced" is required for special characters, super-/subscripts
print GNU "set terminal postscript enhanced color\n";
#print GNU "set terminal postscript enhanced\n";
# Data style: lines, dots, linespoints, points
print GNU "set data style lines\n";
print GNU "set logscale y\n";
#print GNU "set format y \"%.0f\"\n";
print GNU "set format y '10^{%T}'\n";
print GNU "set key left top\n";
#print GNU "set key box\n";
print GNU "set noborder\n";
print GNU "set missing '?'\n";
print GNU "set grid\n";

my $pnum = 0;   # Number of pages, counted for later use

# Calculate bottom margin from maximum label length
my $len = 0;
while ((my $step, my $label) = each %labels) {
        print GNU "set label \"$label\" at $step, graph -0.05 right rotate\n";
        $len = (length $label > length $len) ? length $label : $len;
}
if ($len) { $len /= 1.0; print GNU "set bmargin $len\n"; }

if (defined $xvarkey) { $xvar++; $xvar .= ":"; } else { $xvar = ""; }

# Build the plot command string
my @plot;
foreach my $key (@colkeys) {
        my $title = $key;
        my $using = $xvar;
        $using .= $cols{$key}{nr} + 1;  # gnuplot column numbers start at 1

        # Add delta symbol to title?
        if (defined $cols{$key}{ldesc}{delta}) {
                $title = "{/Symbol D} ".$title;
        }

        # Add phi symbol to title?
        elsif (defined $cols{$key}{ldesc}{first}) {
                $title = "{/Symbol F} ".$title;
        }

        my $sheet = $cols{$key}{ldesc}{plotnr};
        unless (defined $sheet) {
                print "WARN $key has no plotnr. Skipping.\n";
                next;
        }

        # If we have only zero values, disable logscale for this plot
        unless ($cols{$key}{non_zero_value}) {
                $linscale[$sheet] = 1;
        }
        # Replace '_' with '.' ('_' is a subscript indicator)
        $title =~ s/_/./g;
        my $file;
        if ($cols{$key}{ldesc}{dat}) {
                $file = "$fdat.$cols{$key}{ldesc}{dat}";
        }
        else {
                $file = $fdat;
        }
        unless (defined $plot[$sheet]) {
                $plot[$sheet] = "plot \"$file\" using $using title \"$title\"";
                $pnum++;
        }
        else {
                $plot[$sheet] .= ", \"$file\" using $using title '$title'";
        }
}

# Finish the plot file by writing the plot commands
for (my $ii = 0; $ii <= $#plot; $ii++ ) {
        if (defined $plot[$ii]) {
                if (defined $linscale[$ii]) {
                        print GNU "set nologscale y\nset format y '%g'\n";
                }
```

```
                print GNU "$plot[$ii]\n";
                if (defined $linscale[$ii]) {
                        print GNU "set logscale y\nset format y '10^{%T}'\n";
                }
        }
}
close GNU;
system("$gnuplot $fgnu");
#-------------------------------------------------------------------------------
# Create multi-page postscript file (data plots)

if ($has_cover) {
        system("$a2ps -B -o $fcover $finfo > /dev/null 2>&1");
        # Concatenate cover page and plots
        system("$gs -dNOPAUSE -q -dBATCH -sDEVICE=pswrite -sOutputFile=$fps $fcover $fplot");
}
else {
        $fps=$fplot;
}

# Print overview pages
foreach my $pages (1, 2, 4, 6, 8) {
        system("$psnup -l -$pages $fps > ${base}.$pages.ps");
        last if ($pages > $pnum);       # all plots on a single sheet already!
}
#-------------------------------------------------------------------------------
#-------------------------------------------------------------------------------
```

H.4. linuxvmstat.pm

Configuration file for processing Linux vmstat output.

```
#!/usr/bin/perl -w
# Author : Roger Luethi <rl@hellgate.ch>, 2003
# Version: 0.1
# Purpose: plot config for vmstat Linux

package linuxvmstat;

#-------------------------------------------------------------------------------
# Section descriptions

# Most parameters are optional and allow processing bad data material. linenr
# allows identifying section lines by line numbers. valcol picks a column from
# a line, namepat and valpat can be used if even that is not powerful enough.

#       secdesc:        pattern (string: section name)
#                       ldescs[]
#                               pattern (string: identify line within section)
#                               plotnr (number: sheet number to print on)
#                               first (number: first value; e.g. use '2' to
#                                       have data start at '2'; the remaining
#                                       values will be adjusted)
#                               delta (true|false: record diff to prev value?)
#                               xvar (true|false: is this the x variable?)
#                               valcol (column number -> value)
#                               namepat (pattern to use on input line -> name)
#                               valpat (pattern to use on input line -> value)
#                               linscale (true|false: use linear plot scale)
#                               dat (file name: separate dat file)
#                       linenr (true|false: add line numbers to input lines?)

@secdescs = (
        {
                pattern => 'vmstat',
                linenr  => 1,           # Add leading line numbers (#0, #1, ..)
                ldescs  => [
                        {
                                pattern => '#0',
                                namepat => 'running',
                                valcol  => 1,
                                plotnr  => 1,
                                linscale=> 1,
                        },
                        {
                                pattern => '#0',
                                namepat => 'blocked',
                                valcol  => 2,
                                plotnr  => 1,
                                linscale=> 1,
```

```
},
{
        pattern => '#0',
        namepat => 'swapped',
        valcol  => 3,
        plotnr  => 2,
        linscale=> 1,
},
{
        pattern => '#0',
        namepat => 'free',
        valcol  => 4,
        plotnr  => 3,
},
{
        pattern => '#0',
        namepat => 'buff',
        valcol  => 5,
        plotnr  => 3,
        linscale=> 1,
},
{
        pattern => '#0',
        namepat => 'cache',
        valcol  => 6,
        plotnr  => 3,
},
{
        pattern => '#0',
        namepat => 'swapin',
        valcol  => 7,
        plotnr  => 7,
        linscale=> 1,
},
{
        pattern => '#0',
        namepat => 'swapout',
        valcol  => 8,
        plotnr  => 8,
        linscale=> 1,
},
{
        pattern => '#0',
        namepat => 'blocksin',
        valcol  => 9,
        plotnr  => 9,
        linscale=> 1,
},
{
        pattern => '#0',
        namepat => 'blocksout',
        valcol  => 10,
        plotnr  => 10,
        linscale=> 1,
},
{
        pattern => '#0',
        namepat => 'interrupts',
        valcol  => 11,
        plotnr  => 11,
},
{
        pattern => '#0',
        namepat => 'ctxt',
        valcol  => 12,
        plotnr  => 12,
},
{
        pattern => '#0',
        namepat => 'user',
        valcol  => 13,
        plotnr  => 13,
        linscale=> 1,
},
{
        pattern => '#0',
        namepat => 'sys',
        valcol  => 14,
        plotnr  => 14,
        linscale=> 1,
},
{
        pattern => '#0',
        namepat => 'idle',
```

```
                                valcol  => 15,
                                plotnr  => 15,
                                linscale=> 1,
                        },
                        {
                                pattern => '#0',
                                namepat => 'iowait',
                                valcol  => 16,
                                plotnr  => 16,
                                linscale=> 1,
                        },
                ]
        },
);
```

H.5. linux24.pm

Configuration file for processing output collected by log under Linux 2.4 (appendix H.2).

```
#!/usr/bin/perl -w
# Author : Roger Luethi <rl@hellgate.ch>, 2003
# Version: 0.1
# Purpose: plot config for linux 2.4

package linux24;

#-------------------------------------------------------------------------------
# Section descriptions

# Most parameters are optional and allow processing bad data material. linenr
# allows identifying section lines by line numbers. valcol picks a column from
# a line, namepat and valpat can be used if even that is not powerful enough.

#       secdesc:        pattern (string: section name)
#                       ldescs[]
#                               pattern (string: identify line within section)
#                               plotnr (number: sheet number to print on)
#                               first (number: first value; e.g. use '2' to
#                                       have data start at '2'; the remaining
#                                       values will be adjusted)
#                               delta (true|false: record diff to prev value?)
#                               xvar (true|false: is this the x variable?)
#                               valcol (column number -> value)
#                               namepat (pattern to use on input line -> name)
#                               valpat (pattern to use on input line -> value)
#                               linscale (true|false: use linear plot scale)
#                               dat (file name: separate dat file)
#                       linenr (true|false: add line numbers to input lines?)

@secdescs = (
        {
                pattern => 'misc',
                ldescs  => [
                        {
                                pattern => 'timestamp',
                                plotnr  => 0,
                                first   => 0,
                                xvar    => 'y',
                        }
                ]
        },
        {
                pattern => '/proc/stat',
                ldescs  => [
                        {
                                pattern => 'ctxt',
                                delta   => 1,
                                plotnr  => 20,
                        },
                        {
                                pattern => 'procs_running',
                                plotnr  => 21,
                                linscale=> 1,
                        },
                        {
                                pattern => 'procs_blocked',
                                plotnr  => 22,
                                linscale=> 1,
                        },
```

```
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.user',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 1,
                        plotnr  => 23,
                },
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.nice',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 2,
                        plotnr  => 24,
                },
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.system',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 3,
                        plotnr  => 25,
                },
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.idle',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 4,
                        plotnr  => 26,
                },
        ]
},
{
        pattern => '/proc/meminfo',
        ldescs  => [
                {
                        pattern => 'MemTotal:',
                        plotnr  => 100,
                },
                {
                        pattern => 'MemFree:',
                        plotnr  => 100,
                },
                {
                        pattern => 'MemShared:',
                        plotnr  => 100,
                },
                {
                        pattern => 'Buffers:',
                        plotnr  => 102,
                        linscale=> 1,
                },
                {
                        pattern => 'Cached:',
                        plotnr  => 102,
                },
                {
                        pattern => 'SwapCached:',
                        linscale=> 1,
                        plotnr  => 104,
                },
                {
                        pattern => 'Active:',
                        plotnr  => 105,
                        linscale=> 1,
                },
                {
                        pattern => 'Inactive:',
                        plotnr  => 106,
                        linscale=> 1,
                },
                {
                        pattern => 'SwapTotal:',
                        linscale=> 1,
                        plotnr  => 107,
                },
                {
                        pattern => 'SwapFree:',
                        plotnr  => 108,
                        linscale=> 1,
                },
        ]
```

```
        },
        {
                pattern => '/proc/\d+/status',
                ldescs  => [
                        {
                                pattern => 'VmSize:',
                                linscale=> 1,
                                plotnr  => 6,
                        },
                        {
                                pattern => 'VmRSS:',
                                plotnr  => 7,
                                linscale=> 1,
                                dat     => 'vmrss'
                        },
                        {
                                pattern => 'VmData:',
                                linscale=> 1,
                                plotnr  => 8,
                        },
                ]
        },
        {
                pattern => '/proc/\d+/stat',
                linenr  => 'y',         # Add leading line numbers (#0, #1, ..)
                ldescs  => [
#                       {
#                               pattern => '#0',
#                               namepat => 'title',
#                               valpat  => '~ s/(\S+\s+){2}(\S+).*/$2/',
#                               plotnr  => 9,
#                       },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'min_flt',
                                valcol  => 10,
                                plotnr  => 170,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'cmin_fault',
                                valcol  => 11,
                                plotnr  => 171,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'maj_fault',
                                valcol  => 12,
                                plotnr  => 172,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'cmaj_fault',
                                valcol  => 13,
                                plotnr  => 173,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'utime',
                                valcol  => 14,
                                plotnr  => 174,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'stime',
                                valcol  => 15,
                                plotnr  => 175,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'cutime',
                                valcol  => 16,
                                plotnr  => 176,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
```

```
                                namepat => 'cstime',
                                valcol  => 17,
                                plotnr  => 177,
                        },
                        {
                                pattern => '#0',
                                namepat => 'priority',
                                linscale=> 1,
                                valcol  => 18,
                                plotnr  => 178,
                        },
                        {
                                pattern => '#0',
                                namepat => 'nice',
                                linscale=> 1,
                                valcol  => 19,
                                plotnr  => 179,
                        },
                ]
        },
);
```

H.6. linux26.pm

Configuration file for processing output collected by log under Linux 2.6 (appendix H.2).

```
#!/usr/bin/perl -w
# Author : Roger Luethi <rl@hellgate.ch>, 2003
# Version: 0.1
# Purpose: plot config for linux 2.6

package linux26;

#-------------------------------------------------------------------------------
# Section descriptions

# Most parameters are optional and allow processing bad data material. linenr
# allows identifying section lines by line numbers. valcol picks a column from
# a line, namepat and valpat can be used if even that is not powerful enough.

#       secdesc:        pattern (string: section name)
#                       ldescs[]
#                               pattern (string: identify line within section)
#                               plotnr (number: sheet number to print on)
#                               first (number: first value; e.g. use '2' to
#                                       have data start at '2'; the remaining
#                                       values will be adjusted)
#                               delta (true|false: record diff to prev value?)
#                               xvar (true|false: is this the x variable?)
#                               valcol (column number -> value)
#                               namepat (pattern to use on input line -> name)
#                               valpat (pattern to use on input line -> value)
#                               linscale (true|false: use linear plot scale)
#                               dat (file name: separate dat file)
#                       linenr (true|false: add line numbers to input lines?)

@secdescs = (
        {
                pattern => 'misc',
                ldescs  => [
                        {
                                pattern => 'timestamp',
                                plotnr  => 0,
                                first   => 0,
                                xvar    => 'y',
                        }
                ]
        },
        {
                pattern => '/proc/stat',
                ldescs  => [
                        {
                                pattern => 'ctxt',
                                delta   => 1,
                                plotnr  => 20,
                        },
                        {
                                pattern => 'procs_running',
                                plotnr  => 21,
```

```
                        linscale=> 1,
                },
                {
                        pattern => 'procs_blocked',
                        plotnr  => 22,
                        linscale=> 1,
                },
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.user',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 1,
                        plotnr  => 23,
                },
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.nice',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 2,
                        plotnr  => 24,
                },
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.system',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 3,
                        plotnr  => 25,
                },
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.idle',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 4,
                        plotnr  => 26,
                },
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.iowait',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 5,
                        plotnr  => 27,
                },
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.irq',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 6,
                        plotnr  => 28,
                },
                {
                        pattern => 'cpu0',
                        namepat => 'cpu.softirq',
                        delta   => 1,
                        linscale=> 1,
                        valcol  => 7,
                        plotnr  => 29,
                },
        ]
},
{
        pattern => '/proc/vmstat',
        ldescs  => [
                {
                        pattern => 'nr_dirty',
                        plotnr  => 50,
                        linscale=> 1,
                        dat     => 'dirty'
                },
                {
                        pattern => 'nr_writeback',
                        linscale=> 1,
                        plotnr  => 51,
                        #dat    => 'writeback'
                },
                {
                        pattern => 'nr_mapped',
                        linscale=> 1,
                        plotnr  => 52,
```

```
        linscale=> 1,
        #dat    => 'writeback'
},
{
        pattern => 'pgpgin',
        linscale=> 1,
        delta   => 1,
        plotnr  => 53,
},
{
        pattern => 'pgpgout',
        linscale=> 1,
        delta   => 1,
        plotnr  => 54,
        #dat    => 'writeback'
},
{
        pattern => 'pswpin',
        linscale=> 1,
        delta   => 1,
        plotnr  => 55,
        #dat    => 'writeback'
},
{
        pattern => 'pswpout',
        linscale=> 1,
        delta   => 1,
        plotnr  => 56,
},
{
        pattern => 'pgalloc',
        linscale=> 1,
        delta   => 1,
        plotnr  => 57,
},
{
        pattern => 'pgfree',
        linscale=> 1,
        delta   => 1,
        plotnr  => 58,
},
{
        pattern => 'pgactivate',
        linscale=> 1,
        delta   => 1,
        plotnr  => 59,
},
{
        pattern => 'pgdeactivate',
        linscale=> 1,
        delta   => 1,
        plotnr  => 60,
},
{
        pattern => 'pgfault',
        linscale=> 1,
        delta   => 1,
        plotnr  => 61,
},
{
        pattern => 'pgmajfault',
        linscale=> 1,
        delta   => 1,
        plotnr  => 62,
},
{
        pattern => 'pgscan',
        delta   => 1,
        linscale=> 1,
        plotnr  => 63,
},
{
        pattern => 'pgrefill',
        linscale=> 1,
        delta   => 1,
        plotnr  => 64,
},
{
        pattern => 'pgsteal',
        linscale=> 1,
        delta   => 1,
        plotnr  => 65,
},
{
```

```
                        pattern => 'pginodesteal',
                        linscale=> 1,
                        delta   => 1,
                        plotnr  => 66,
                },
                {
                        pattern => 'kswapd_steal',
                        delta   => 1,
                        linscale=> 1,
                        plotnr  => 67,
                },
                {
                        pattern => 'kswapd_inodesteal',
                        linscale=> 1,
                        delta   => 1,
                        plotnr  => 68,
                },
                {
                        pattern => 'pageoutrun',
                        linscale=> 1,
                        delta   => 1,
                        plotnr  => 69,
                },
                {
                        pattern => 'allocstall',
                        linscale=> 1,
                        delta   => 1,
                        plotnr  => 70,
                },
                {
                        pattern => 'pgrotated',
                        linscale=> 1,
                        delta   => 1,
                        plotnr  => 71,
                },
        ]
},
{
        pattern => '/proc/meminfo',
        ldescs  => [
                {
                        pattern => 'MemTotal:',
                        linscale=> 1,
                        plotnr  => 100,
                },
                {
                        pattern => 'MemFree:',
                        plotnr  => 100,
                },
                {
                        pattern => 'Buffers:',
                        plotnr  => 102,
                        linscale=> 1,
                },
                {
                        pattern => 'Cached:',
                        plotnr  => 102,
                },
                {
                        pattern => 'SwapCached:',
                        linscale=> 1,
                        plotnr  => 104,
                },
                {
                        pattern => 'Active:',
                        plotnr  => 105,
                        linscale=> 1,
                },
                {
                        pattern => 'Inactive:',
                        plotnr  => 106,
                        linscale=> 1,
                },
                {
                        pattern => 'SwapTotal:',
                        linscale=> 1,
                        plotnr  => 107,
                },
                {
                        pattern => 'SwapFree:',
                        plotnr  => 108,
                        linscale=> 1,
                },
                {
```

```
                        pattern => 'Committed_AS:',
                        plotnr  => 109,
                        linscale=> 1,
                },
                {
                        pattern => 'VmallocUsed:',
                        linscale=> 1,
                        plotnr  => 110,
                },
                {
                        pattern => 'Dirty:',
                        linscale=> 1,
                        plotnr  => 4,
                },
        ]
},
{
        pattern => '/proc/\d+/status',
        ldescs  => [
                {
                        pattern => 'VmSize:',
                        linscale=> 1,
                        plotnr  => 6,
                },
                {
                        pattern => 'VmRSS:',
                        linscale=> 1,
                        plotnr  => 7,
                        dat     => 'vmrss'
                },
                {
                        pattern => 'VmData:',
                        linscale=> 1,
                        plotnr  => 8,
                },
        ]
},
{
        pattern => '/proc/\d+/stat',
        linenr  => 'y',         # Add leading line numbers (#0, #1, ..)
        ldescs  => [
#               {
#                       pattern => '#0',
#                       namepat => 'title',
#                       valpat  => '~ s/(\S+\s+){2}(\S+).*/$2/',
#                       plotnr  => 9,
#               },
                {
                        pattern => '#0',
                        linscale=> 1,
                        namepat => 'min_flt',
                        valcol  => 10,
                        plotnr  => 170,
                },
                {
                        pattern => '#0',
                        linscale=> 1,
                        namepat => 'cmin_fault',
                        valcol  => 11,
                        plotnr  => 171,
                },
                {
                        pattern => '#0',
                        linscale=> 1,
                        namepat => 'maj_fault',
                        valcol  => 12,
                        plotnr  => 172,
                },
                {
                        pattern => '#0',
                        linscale=> 1,
                        namepat => 'cmaj_fault',
                        valcol  => 13,
                        plotnr  => 173,
                },
                {
                        pattern => '#0',
                        linscale=> 1,
                        namepat => 'utime',
                        valcol  => 14,
                        plotnr  => 174,
                },
                {
                        pattern => '#0',
```

```
                                linscale=> 1,
                                namepat => 'stime',
                                valcol  => 15,
                                plotnr  => 175,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'cutime',
                                valcol  => 16,
                                plotnr  => 176,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'cstime',
                                valcol  => 17,
                                plotnr  => 177,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'priority',
                                linscale=> 1,
                                valcol  => 18,
                                plotnr  => 178,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'nice',
                                linscale=> 1,
                                valcol  => 19,
                                plotnr  => 179,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'rt_priority',
                                linscale=> 1,
                                valcol  => 40,
                                plotnr  => 180,
                        },
                        {
                                pattern => '#0',
                                linscale=> 1,
                                namepat => 'policy',
                                linscale=> 1,
                                valcol  => 41,
                                plotnr  => 181,
                        },
                ]
        },
);
```

H.7. freebsd.pm

Configuration file for processing FreeBSD vmstat output.

```
#!/usr/bin/perl -w
# Author : Roger Luethi <rl@hellgate.ch>, 2003
# Version: 0.1
# Purpose: plot config for vmstat FreeBSD 5.0

package freebsd;

#------------------------------------------------------------------------------
# Section descriptions

# Most parameters are optional and allow processing bad data material. linenr
# allows identifying section lines by line numbers. valcol picks a column from
# a line, namepat and valpat can be used if even that is not powerful enough.

#       secdesc:        pattern (string: section name)
#                       ldescs[]
#                               pattern (string: identify line within section)
#                               plotnr (number: sheet number to print on)
#                               first (number: first value; e.g. use '2' to
#                                       have data start at '2'; the remaining
```

```
#                                       values will be adjusted)
#                                delta (true|false: record diff to prev value?)
#                                xvar (true|false: is this the x variable?)
#                                valcol (column number -> value)
#                                namepat (pattern to use on input line -> name)
#                                valpat (pattern to use on input line -> value)
#                                linscale (true|false: use linear plot scale)
#                                dat (file name: separate dat file)
#                        linenr (true|false: add line numbers to input lines?)

@secdescs = (
        {
                pattern => 'vmstat',
                linenr  => 1,           # Add leading line numbers (#0, #1, ..)
                ldescs  => [
                        {
                                pattern => '#0',
                                namepat => 'running',
                                valcol  => 1,
                                plotnr  => 1,
                                linscale=> 1,
                        },
                        {
                                pattern => '#0',
                                namepat => 'blocked',
                                valcol  => 2,
                                plotnr  => 2,
                                linscale=> 1,
                        },
                        {
                                pattern => '#0',
                                namepat => 'swapped',
                                valcol  => 3,
                                plotnr  => 3,
                                linscale=> 1,
                        },
                        {
                                pattern => '#0',
                                namepat => 'active_pages',
                                valcol  => 4,
                                plotnr  => 4,
                        },
                        {
                                pattern => '#0',
                                namepat => 'free',
                                valcol  => 5,
                                plotnr  => 5,
                        },
                        {
                                pattern => '#0',
                                namepat => 'fault',
                                valcol  => 5,
                                plotnr  => 5,
                        },
                        {
                                pattern => '#0',
                                namepat => 'reclaim',
                                valcol  => 6,
                                plotnr  => 6,
                        },
                        {
                                pattern => '#0',
                                namepat => 'page_in',
                                valcol  => 7,
                                plotnr  => 7,
                        },
                        {
                                pattern => '#0',
                                namepat => 'page_out',
                                valcol  => 8,
                                plotnr  => 7,
                        },
                        {
                                pattern => '#0',
                                namepat => 'page_freed',
                                valcol  => 9,
                                plotnr  => 9,
                        },
                        {
                                pattern => '#0',
                                namepat => 'page_scan',
                                valcol  => 10,
                                plotnr  => 10,
                        },
```

```
                        {
                                pattern => '#0',
                                namepat => 'disk',
                                valcol  => 11,
                                plotnr  => 11,
                        },
                        {
                                pattern => '#0',
                                namepat => 'disk',
                                valcol  => 12,
                                plotnr  => 12,
                        },
                        {
                                pattern => '#0',
                                namepat => 'ctxt',
                                valcol  => 16,
                                plotnr  => 16,
                        },
                        {
                                pattern => '#0',
                                namepat => 'user',
                                valcol  => 17,
                                plotnr  => 17,
                                linscale=> 1,
                        },
                        {
                                pattern => '#0',
                                namepat => 'sys',
                                valcol  => 18,
                                plotnr  => 17,
                        },
                        {
                                pattern => '#0',
                                namepat => 'idle',
                                valcol  => 19,
                                plotnr  => 17,
                        },
                ]
        },
);
```

H.8. loadcontrol.diff

Our implementation of load control. This is a standard unified diff against Linux 2.6.0-test4. See section 3.7.1 for a discussion.

```
diff -ruNp linux-2.6.0-test4/arch/i386/kernel/signal.c linux-2.6.0-test4-lctrl/arch/i386/kernel/signal.c
--- linux-2.6.0-test4/arch/i386/kernel/signal.c 2004-01-19 05:44:04.056893406 +0100
+++ linux-2.6.0-test4-lctrl/arch/i386/kernel/signal.c   2004-01-18 21:52:19.000000000 +0100
@@ -24,6 +24,7 @@
 #include <asm/uaccess.h>
 #include <asm/i387.h>
 #include "sigframe.h"
+#include <linux/thrashing.h>

 #define DEBUG_SIG 0

@@ -568,6 +569,11 @@ int do_signal(struct pt_regs *regs, sigs
                goto no_signal;
        }

+       if (current->flags & PF_STUN) {
+               stun_me();
+               goto no_signal;
+       }
+
        if (!oldset)
                oldset = &current->blocked;

diff -ruNp linux-2.6.0-test4/include/linux/sched.h linux-2.6.0-test4-lctrl/include/linux/sched.h
--- linux-2.6.0-test4/include/linux/sched.h     2004-01-19 05:44:03.000000000 +0100
+++ linux-2.6.0-test4-lctrl/include/linux/sched.h       2004-01-18 21:52:19.000000000 +0100
@@ -488,6 +488,7 @@ do { if (atomic_dec_and_test(&(tsk)->usa
 #define PF_SWAPOFF     0x00080000      /* I am in swapoff */
 #define PF_LESS_THROTTLE 0x00100000    /* Throttle me less: I clean memory */
 #define PF_SYNCWRITE   0x00200000      /* I am doing a sync write */
+#define PF_STUN                0x00400000
```

```
 #ifdef CONFIG_SMP
 extern int set_cpus_allowed(task_t *p, cpumask_t new_mask);
@@ -553,6 +554,7 @@ extern int FASTCALL(wake_up_process(stru
 extern int FASTCALL(wake_up_process_kick(struct task_struct * tsk));
 extern void FASTCALL(wake_up_forked_process(struct task_struct * tsk));
 extern void FASTCALL(sched_exit(task_t * p));
+extern int task_interactive(task_t * p);

 asmlinkage long sys_wait4(pid_t pid,unsigned int * stat_addr, int options, struct rusage * ru);

diff -ruNp linux-2.6.0-test4/include/linux/swap.h linux-2.6.0-test4-lctrl/include/linux/swap.h
--- linux-2.6.0-test4/include/linux/swap.h      2004-01-19 05:44:03.000000000 +0100
+++ linux-2.6.0-test4-lctrl/include/linux/swap.h        2004-01-18 21:52:19.000000000 +0100
@@ -175,6 +175,7 @@ extern void swap_setup(void);

 /* linux/mm/vmscan.c */
 extern int try_to_free_pages(struct zone *, unsigned int, unsigned int);
+extern int shrink_list(struct list_head *, unsigned int, int *, int *);
 extern int shrink_all_memory(int);
 extern int vm_swappiness;

diff -ruNp linux-2.6.0-test4/include/linux/thrashing.h linux-2.6.0-test4-lctrl/include/linux/thrashing.h
--- linux-2.6.0-test4/include/linux/thrashing.h 1970-01-01 01:00:00.000000000 +0100
+++ linux-2.6.0-test4-lctrl/include/linux/thrashing.h   2004-01-19 03:07:43.000000000 +0100
@@ -0,0 +1,15 @@
+#ifndef _LINUX_THRASHING_H
+#define _LINUX_THRASHING_H
+
+#include <linux/wait.h>
+#include <asm/atomic.h>
+#include <linux/sched.h>
+
+extern struct semaphore stun_ser;
+extern struct semaphore unstun_token;
+extern wait_queue_head_t thrashing_wq;
+extern void stun_me(void);
+extern void thrashing(void);
+extern atomic_t waiting;
+
+#endif /* _LINUX_THRASHING_H */
diff -ruNp linux-2.6.0-test4/kernel/Makefile linux-2.6.0-test4-lctrl/kernel/Makefile
--- linux-2.6.0-test4/kernel/Makefile   2004-01-19 05:44:04.000000000 +0100
+++ linux-2.6.0-test4-lctrl/kernel/Makefile     2004-01-18 21:52:19.000000000 +0100
@@ -6,7 +6,8 @@ obj-y     = sched.o fork.o exec_domain.o
             exit.o itimer.o time.o softirq.o resource.o \
             sysctl.o capability.o ptrace.o timer.o user.o \
             signal.o sys.o kmod.o workqueue.o pid.o \
-            rcupdate.o intermodule.o extable.o params.o posix-timers.o
+            rcupdate.o intermodule.o extable.o params.o posix-timers.o \
+            thrashing.o

 obj-$(CONFIG_FUTEX) += futex.o
 obj-$(CONFIG_GENERIC_ISA_DMA) += dma.o
diff -ruNp linux-2.6.0-test4/kernel/sched.c linux-2.6.0-test4-lctrl/kernel/sched.c
--- linux-2.6.0-test4/kernel/sched.c    2004-01-19 05:44:04.000000000 +0100
+++ linux-2.6.0-test4-lctrl/kernel/sched.c      2004-01-19 03:48:09.000000000 +0100
@@ -34,6 +34,7 @@
 #include <linux/rcupdate.h>
 #include <linux/cpu.h>
 #include <linux/percpu.h>
+#include <linux/thrashing.h>

 #ifdef CONFIG_NUMA
 #define cpu_to_node_mask(cpu) node_to_cpumask(cpu_to_node(cpu))
@@ -1276,6 +1277,24 @@ out:
        rebalance_tick(rq, 0);
 }

+#if 0
+static unsigned long stun_time(void) {
+       unsigned long ret;
+       int ql = atomic_read(&waiting);
+       if (ql == 1)
+               ret = 5*HZ;
+       else if (ql == 2)
+               ret = 3*HZ;
+       else if (ql < 5)
+               ret = 2*HZ;
+       else if (ql < 10)
+               ret = 1*HZ;
+       else
+               ret = HZ/2;
+       return ret;
+}
```

```
+#endif
+
 void scheduling_functions_start_here(void) { }

 /*
@@ -1306,6 +1325,22 @@ need_resched:
        prev = current;
        rq = this_rq();

+       if (unlikely(waitqueue_active(&thrashing_wq))) {
+               static unsigned long prev_unstun;
+#if 0
+               unsigned long wait = stun_time();
+               if (time_before(jiffies, prev_unstun + wait) && prev_unstun)
+#endif
+               if (time_before(jiffies, prev_unstun + 5*HZ) && prev_unstun)
+                       goto thrash_done;
+               if (!atomic_read(&stun_ser.count))
+                       goto thrash_done;
+               prev_unstun = jiffies;
+               up(&unstun_token);
+               wake_up(&thrashing_wq);
+       }
+thrash_done:
+
        release_kernel_lock(prev);
        prev->last_run = jiffies;
        spin_lock_irq(&rq->lock);
@@ -1698,6 +1733,16 @@ int task_nice(task_t *p)
 }

 /**
+ * task_nice - return the nice value of a given task.
+ * @p: the task in question.
+ */
+int task_interactive(task_t *p)
+{
+       return TASK_INTERACTIVE(p);
+}
+
+
+/**
  * task_curr - is this task currently executing on a CPU?
  * @p: the task in question.
  */
diff -ruNp linux-2.6.0-test4/kernel/thrashing.c linux-2.6.0-test4-lctrl/kernel/thrashing.c
--- linux-2.6.0-test4/kernel/thrashing.c        1970-01-01 01:00:00.000000000 +0100
+++ linux-2.6.0-test4-lctrl/kernel/thrashing.c  2004-01-19 04:25:30.000000000 +0100
@@ -0,0 +1,352 @@
+#include <linux/thrashing.h>
+#include <linux/mm.h>
+#include <linux/wait.h>
+
+DECLARE_MUTEX(stun_ser);
+DECLARE_MUTEX_LOCKED(unstun_token);
+DECLARE_WAIT_QUEUE_HEAD(thrashing_wq);
+atomic_t waiting;
+
+/*
+ * int_sqrt - oom_kill.c internal function, rough approximation to sqrt
+ * @x: integer of which to calculate the sqrt
+ *
+ * A very broken approximation to the sqrt() function.
+ */
+static unsigned int int_sqrt(unsigned int x)
+{
+       unsigned int out = x;
+       while (x & ~(unsigned int)1) x >>=2, out >>=1;
+       if (x) out -= out >> 2;
+       return (out ? out : 1);
+}
+
+static int stun_badness(struct task_struct *p, int flags)
+{
+       int points, cpu_time, run_time;
+
+       if (!p->mm)
+               return 0;
+
+       if (p->flags & (PF_MEMDIE|flags))
+               return 0;
+
+       /*
+        * The memory size of the process is the basis for the badness.
```

```
+        */
+       points = p->mm->total_vm;
+
+       points += 2*p->mm->rss;
+
+       /*
+        * CPU time is in seconds and run time is in minutes. There is no
+        * particular reason for this other than that it turned out to work
+        * very well in practice.
+        */
+       cpu_time = (p->utime + p->stime) >> (SHIFT_HZ + 3);
+       run_time = (get_jiffies_64() - p->start_time) >> (SHIFT_HZ + 10);
+
+       points *= int_sqrt(cpu_time);
+       points *= int_sqrt(int_sqrt(run_time));
+
+       /*
+        * Niced processes are most likely less important, so double
+        * their badness points.
+        */
+       if (task_nice(p) > 0)
+               points *= 2;
+
+       if (task_interactive(p))
+               points /= 4;
+
+       /*
+        * Superuser processes are usually more important, so we make it
+        * less likely that we kill those.
+        */
+       if (cap_t(p->cap_effective) & CAP_TO_MASK(CAP_SYS_ADMIN) ||
+                               p->uid == 0 || p->euid == 0)
+               points /= 2;
+
+       /*
+        * We don't want to kill a process with direct hardware access.
+        * Not only could that mess up the hardware, but usually users
+        * tend to only have this flag set on applications they think
+        * of as important.
+        */
+       if (cap_t(p->cap_effective) & CAP_TO_MASK(CAP_SYS_RAWIO))
+               points /= 2;
+       points++;
+       return points;
+}
+
+/*
+ * Simple selection loop. We choose the process with the highest
+ * number of 'points'. We expect the caller will lock the tasklist.
+ */
+struct task_struct * pick_bad_process(int flags)
+{
+       int maxpoints = 0;
+       struct task_struct *g, *p;
+       struct task_struct *chosen = NULL;
+
+       do_each_thread(g, p)
+               if (p->pid) {
+                       int points = stun_badness(p, flags);
+                       if (points > maxpoints) {
+                               chosen = p;
+                               maxpoints = points;
+                       }
+                       if (p->flags & PF_SWAPOFF)
+                               return p;
+               }
+       while_each_thread(g, p);
+       return chosen;
+}
+
+
+#include <linux/pagevec.h>
+#include <linux/swap.h>
+#include <linux/mm_inline.h>
+#include <linux/hugetlb.h>
+#include <linux/pagemap.h>
+#include <asm/rmap.h>
+
+/*
+ * Do a quick page-table lookup for a single page.
+ * mm->page_table_lock must be held.
+ */
+struct page *
+follow_page_only(struct mm_struct *mm, unsigned long address, int write)
```

```
+{
+       pgd_t *pgd;
+       pmd_t *pmd;
+       pte_t *ptep, pte;
+       unsigned long pfn;
+       struct vm_area_struct *vma;
+
+       vma = hugepage_vma(mm, address);
+       if (vma)
+               return follow_huge_addr(mm, vma, address, write);
+
+       pgd = pgd_offset(mm, address);
+       if (pgd_none(*pgd) || pgd_bad(*pgd))
+               goto out;
+
+       pmd = pmd_offset(pgd, address);
+       if (pmd_none(*pmd))
+               goto out;
+       if (pmd_huge(*pmd))
+               return follow_huge_pmd(mm, address, pmd, write);
+       if (pmd_bad(*pmd))
+               goto out;
+
+       ptep = pte_offset_map(pmd, address);
+       if (!ptep)
+               goto out;
+
+       pte = *ptep;
+       pte_unmap(ptep);
+       if (pte_present(pte)) {
+               /* If write: only return pages that are writable and dirty */
+               if (!write || (pte_write(pte) && pte_dirty(pte))) {
+                       pfn = pte_pfn(pte);
+                       if (pfn_valid(pfn)) {
+                               struct page *page = pfn_to_page(pfn);
+                               return page;
+                       }
+               }
+       }
+
+out:
+       return NULL;
+}
+
+/* Holding the zone->lru_lock already  */
+int try_to_dump_page(struct page *page, struct list_head *page_list)
+{
+       pte_t *pte;
+       ClearPageReferenced(page);
+
+       /* Make this page old */
+       pte = rmap_ptep_map(page->pte.direct);
+       ptep_test_and_clear_young(pte);
+       rmap_ptep_unmap(pte);
+
+       if (TestClearPageLRU(page)) {
+               struct zone *zone = page_zone(page);
+               list_del(&page->lru);
+
+               if (page_count(page) == 0) {
+                       /* It is currently in pagevec_release(), put it back */
+                       SetPageLRU(page);
+                       if (PageActive(page))
+                               list_add(&page->lru, &zone->active_list);
+                       else
+                               list_add(&page->lru, &zone->inactive_list);
+                       return 0;
+               }
+               if (TestClearPageActive(page)) {
+                       zone->nr_active--;
+               }
+               else {
+                       zone->nr_inactive--;
+               }
+       }
+       else {
+               return 0;
+       }
+       page_cache_get(page); /* Decremented in shrink_list() or
+                                page_vec_release() */
+       list_add(&page->lru, page_list);
+       return 1;
+}
+
```

```
+static void clean_vma(struct vm_area_struct *vma, struct mm_struct *mm)
+{
+       LIST_HEAD(page_list);
+       struct pagevec pvec;
+       unsigned long start;
+       struct zone *zone = NULL;
+
+       start = vma->vm_start;
+       do {
+               struct page *page;
+               /* Call my own follow_page (without activation feature) */
+               if ((page = follow_page_only(vma->vm_mm, start, 0))) {
+                       struct zone *new_zone = page_zone(page);
+                       /* Lazy locking */
+                       if (new_zone != zone) {
+                               if (zone)
+                                       spin_unlock_irq(&zone->lru_lock);
+                               zone = new_zone;
+                               spin_lock_irq(&zone->lru_lock);
+                       }
+
+                       if (PageDirect(page)) {
+                               try_to_dump_page(page, &page_list);
+                       }
+               }
+               start += PAGE_SIZE;
+       } while (start < vma->vm_end);
+
+       if (zone)
+               spin_unlock_irq(&zone->lru_lock);
+       zone = NULL;
+
+       pagevec_init(&pvec, 1);
+       if (!list_empty(&page_list)) {
+               int unused;
+               /* Holding neither zone->lru_lock nor page_table_lock here */
+               spin_unlock(&mm->page_table_lock);
+                       shrink_list(&page_list, __GFP_FS, NULL, &unused);
+               spin_lock(&mm->page_table_lock);
+               while (!list_empty(&page_list)) {
+                       struct page *page;
+                       struct zone *new_zone;
+                       page = list_entry(page_list.prev, struct page, lru);
+                       new_zone = page_zone(page);
+                       if (new_zone != zone) {
+                               if (zone)
+                                       spin_unlock_irq(&zone->lru_lock);
+                               zone = new_zone;
+                               spin_lock_irq(&zone->lru_lock);
+                       }
+                       if (TestSetPageLRU(page))
+                               panic("%s(%d)", __func__, __LINE__);
+                       list_del(&page->lru);
+                       if (PageActive(page))
+                               add_page_to_active_list(zone, page);
+                       else
+                               add_page_to_inactive_list(zone, page);
+                       if (!pagevec_add(&pvec, page)) {
+                               spin_unlock_irq(&zone->lru_lock);
+                               __pagevec_release(&pvec);
+                               spin_lock_irq(&zone->lru_lock);
+                       }
+               }
+       }
+       if (zone)
+               spin_unlock_irq(&zone->lru_lock);
+       pagevec_release(&pvec);
+}
+
+static void dump_mm(void)
+{
+       struct vm_area_struct *vma;
+       struct mm_struct *mm;
+
+       mm = current->mm;
+       if (!mm)
+               goto out;
+       down_read(&mm->mmap_sem);
+       spin_lock(&mm->page_table_lock);
+       lru_add_drain();
+       vma = mm->mmap;
+
+       while (vma) {
+               struct vm_area_struct *next;
```

```
+               next = vma->vm_next;
+               if (irqs_disabled())
+                       panic("%s(%d)", __func__, __LINE__);
+               if (!(vma->vm_flags & (VM_RESERVED | VM_LOCKED)))
+                       clean_vma(vma, mm);
+               vma = next;
+       }
+
+       spin_unlock(&mm->page_table_lock);
+       up_read(&mm->mmap_sem);
+out:
+       ;
+}
+
+void stun_me()
+{
+       DEFINE_WAIT(wait);
+       unsigned long working_set;
+
+       spin_lock_irq(&current->sighand->siglock);
+       spin_unlock_irq(&current->sighand->siglock);
+
+       atomic_inc(&waiting);
+       working_set = current->mm->rss;
+       dump_mm();
+
+       up(&stun_ser);          /* Allow next */
+
+       for (;;) {
+               prepare_to_wait_exclusive(&thrashing_wq, &wait,
+                                       TASK_UNINTERRUPTIBLE);
+               schedule();
+               if (!down_trylock(&unstun_token))
+                       break;  /* Yay. Got unstun token, wake up */
+       }
+       finish_wait(&thrashing_wq, &wait);
+       current->flags &= ~(PF_STUN|PF_MEMALLOC);
+       atomic_dec(&waiting);
+}
+
+void thrashing()
+{
+       struct task_struct *p;
+       unsigned long flags;
+
+       if (down_trylock(&stun_ser))
+               return;
+
+       read_lock(&tasklist_lock);
+       p = pick_bad_process(PF_STUN);
+       if (!p) {
+               up(&stun_ser);
+               goto out_unlock;
+       }
+       if (p->flags & PF_STUN) {
+               goto out_unlock;
+       }
+       p->flags |= PF_STUN|PF_MEMALLOC;
+       p->time_slice = HZ;
+       spin_lock_irqsave(&p->sighand->siglock, flags);
+       signal_wake_up(p, 0);
+       spin_unlock_irqrestore(&p->sighand->siglock, flags);
+out_unlock:
+       read_unlock(&tasklist_lock);
+}
diff -ruNp linux-2.6.0-test4/mm/page_alloc.c linux-2.6.0-test4-lctrl/mm/page_alloc.c
--- linux-2.6.0-test4/mm/page_alloc.c   2004-01-19 05:44:04.000000000 +0100
+++ linux-2.6.0-test4-lctrl/mm/page_alloc.c     2004-01-19 03:49:05.000000000 +0100
@@ -31,6 +31,7 @@
 #include <linux/topology.h>
 #include <linux/sysctl.h>
 #include <linux/cpu.h>
+#include <linux/thrashing.h>

 #include <asm/tlbflush.h>

@@ -593,6 +594,7 @@ __alloc_pages(unsigned int gfp_mask, uns
        }

        /* here we're in the low on memory slow path */
+       thrashing();

 rebalance:
        if ((current->flags & (PF_MEMALLOC | PF_MEMDIE)) && !in_interrupt()) {
```

```
diff -ruNp linux-2.6.0-test4/mm/vmscan.c linux-2.6.0-test4-lctrl/mm/vmscan.c
--- linux-2.6.0-test4/mm/vmscan.c       2004-01-19 05:44:04.000000000 +0100
+++ linux-2.6.0-test4-lctrl/mm/vmscan.c 2004-01-18 21:52:19.000000000 +0100
@@ -35,6 +35,7 @@
 #include <asm/div64.h>

 #include <linux/swapops.h>
+#include <linux/thrashing.h>

 /*
  * The "priority" of VM scanning is how much of the queues we will scan in one
@@ -263,7 +264,7 @@ static void handle_write_error(struct ad
 /*
  * shrink_list returns the number of reclaimed pages
  */
-static int
+int
 shrink_list(struct list_head *page_list, unsigned int gfp_mask,
                int *max_scan, int *nr_mapped)
 {
```

I. Glossary

Bounded rationality Unlike *full rationality*, this concept acknowledges that humans are limited in their ability to obtain, store, and process information. They are still expected to maximize their utility by taking rational decisions within these constraints.

Closed source software Software for which the source code is not publicly available. We use the terms “proprietary software” and “closed source software” to emphasize different aspects of proprietary, closed source software.

Copyleft A provision found in some FOSS licenses. It requires all derived works to be released as FOSS. It prevents all entities other than the copyright holder from using the code in proprietary software.

[28]: More information on copyleft.

Development branch See “Fork”.

Digital convergence The chances and problems arising from the gradual move of information and entertainment to digital storage and distribution have been discussed for many years [10]. In the context of this thesis we are interested in one aspect: Producing perfect copies of digital content – be it text, audio, or video – is trivial and cheap for the owner of any common general-purpose computer. This is in stark contrast to the old world of printed books, vinyl records, or celluloid film. Moreover, the typical computer is connected to the Internet which makes a cheap, global, and possibly anonymous distribution of these perfect copies possible. A strict enforcement of copyrights has become that much more difficult, and thus the costs for ensuring excludability are exploding.

Whether a balance comparable to the old world can be found is unclear. At this point it seems that any solution that restores a high level of copyright enforcement is bound to give copyright holders unprecedented additional means to restrict and control its customers.

[56, 57]: Discussion of copyrights in digital times.

Dual licensing Some company release source code under a FOSS license with copyleft, but sell separate licenses for inclusion of the code in proprietary software. MySQL and Trolltech are two companies generating revenue with dual licensing.

Emulator A program to simulate the behavior of a specific hardware platform. The emulator provides an environment that can be used to run programs written for the emulated hardware.

Exhaustion of Rights This legal principle bars holders of copyrights, trademarks, or patents from using these claims to stop imports of their goods into a country. If the principle is

not applied in a country or for one class of rights, the right owners can prohibit parallel imports and gray markets which makes massive per-country price discrimination possible.

First Sale Doctrine *"First Sale doctrine is an exception to copyright codified in the US Copyright Act, section 109. The doctrine of first sale allows the purchaser to transfer a particular, legally acquired copy of protected work without permission once it has been obtained. That means the distribution rights of a copyright holder end on that particular copy once the copy is sold."* [111] The restrictions copyright holders may impose on their customers with regards to reselling, lending, or destroying a legally purchased copy varies from country to country.

Fork When software is forked, two versions are separately developed from a common starting point. With FOSS, this may happen if developers of a project fail to agree on the future direction. Because the costs of maintaining a separate project are substantial, forks are quite rare, and they often merge back again later.

Some software projects use this in their development model: From a stable version a development branch is forked. The stable version is still maintained and minor releases are published: Bugs are fixed, small features are added, performance is tuned. All experiments and major changes take place in the new branch until it is stabilized to become the new stable version.

Linux uses this model: Since its start, the stable branch 2.4 went from release 2.4.0 to 2.4.23. The development tree 2.5 was forked off 2.4.15 and moved over 88 versions from 2.5.0 to 2.6.0. With the release of 2.6.0, the former development branch 2.5 became the new stable tree 2.6 from which a new development branch 2.7 will fork at some point in the future.

In addition to the "official" Linux versions, several individuals and organizations maintain their own experimental or specialized versions of the Linux kernel. They cater to narrow interests or save as testbeds for new developments.

FOSS Free and Open Source Software (FOSS) refers to both Free Software and Open Source Software. For the scope of this paper, we can ignore the differences between them.

[27, 83]: Definitions of Free Software and Open Source Software, respectively.

[114, 85]: Perspectives on said differences.

FUD Fear, Uncertainty, and Doubt. See appendix C.4, page 98.

Information asymmetry One participant in a transaction has exclusive access to relevant information. Examples of such information include hidden intentions, expert knowledge, and experience.

Nash equilibrium The sum of strategies chosen by all market participants based on their anticipating each other's behavior.

Network effect A good is said to exhibit a network effect if its value to customers grows with the numbers of other people buying it. The network effect is an external effect:

An increase in user numbers benefits existing users and new users alike. However, a monopolist may raise the sales price to seize these external benefits.

The classic example for a network effects are telephones which have become much more useful than they were when hardly anybody had one.

[112]: More information on the network effect.

Open standards The definition of this term is controversial: For some, it refers to the creation of standard bodies – standards that any vendor can buy and implement. The introduction of software patents made it possible that public standards are subject to patent license fees. Standard bodies tend to require that patents be licensed under "*reasonable and non-discriminatory*" (RAND) terms.

Others, most notably prominent members of the FOSS community, argue that the only terms that fit the RAND requirement are royalty-free, because FOSS authors lack the control of their work and the revenue stream to pay license fees.

[82]: Documentation on the controversy on patents and standards.

Price discrimination The practice of charging different prices to different buyers. See page 13.

Proprietary software Software distributed under licenses that leave exclusive control of the software with the owner of the copyright. The owner may even allow third parties or the public to inspect the source code but reserves the exclusive right to copy, modify, and sell the program.

Social dilemma A scenario where individuals maximizing their utility reach a suboptimal solution. "The Prisoner's Dilemma" and "The Tragedy of the Commons" [38] are famous examples of social dilemmas. These problems are usually addressed by introducing some form of accountability, depending on environment and circumstances: One approach assigns exclusive ownership over the former collective good. An alternative solution exerts social control based on information on individual contributions and reputation.

Source Code Escrow The vendor of closed source software may hand over the source code for a program to a trusted third party. A contract defines the incidents that give customers the right to request the code from the third party vault. This may be the demise of the selling company or the discontinuation of the product. Extra care must be taken to ensure that the deposited source code is complete and corresponds to the sold version of the software.

Spin lock A method for mutual exclusion between several processes. Spin locks are based on busy waiting – a process keeps trying to acquire the lock until it succeeds. Other methods for mutual exclusion put processes to sleep and wake them when the lock becomes available.

Transaction costs A large variety of definitions for this term can be found in the literature (cf. [109]). One definition reads [21]:

When information is costly, various activities related to the exchange of property rights between individuals give rise to transaction costs.

These activities include:

1. *The search for information about the distribution of price and quality of commodities and labor inputs, and the search for potential buyers and sellers and for relevant information about their behavior and circumstances*
2. *The bargaining that is needed to find the true position of buyers and sellers when prices are endogenous*
3. *The making of contracts*
4. *The monitoring of contractual partners to see whether they abide by the forms of contract*
5. *The enforcement of a contract and the collection of damages when partners fail to observe their contractual obligations*
6. *The protection of property rights against third-party encroachment for example, protection against pirates or even against the government in the case of illegitimate trade.*

Vaporware The announcement of products which are far from being released, usually to keep a competitor from gaining market share. See page 21.

x86 The dominant computer hardware architecture today. Refers to the names of early Intel CPUs in this line, from 8086 to 80486.

Bibliography

World Wide Web Pages which are likely to change over time are labeled "*WWW Page*". In that case, the associated date marks the most recent check. Files within archives are referenced by URL:FILEPATH.

[1] Greg Aharonian. 17,500 software patents to issue in 1998. *ACM SIGSOFT Software Engineering Notes*, 24:58–62, May 1999.

[2] Ross Anderson. 'Trusted Computing' Frequently Asked Questions. WWW page, visited November 2003. http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html.

[3] Kenneth J. Arrow, Ronald H. Coase, Milton Friedman, et al. Brief as Amici Curiae in Support of Petitioners, In the Supreme Court of the United States, Eric Eldred et al. v. John D. Ashcroft, Attorney General, May 2002. http://cyber.law.harvard.edu/openlaw/eldredvashcroft/supct/amici/economists.pdf.

[4] American Library Association, American Association of Law Libraries, Association of Research Libraries, Medical Library Association, and Special Libraries Association. Re: high-tech warranty project – comment, p994413. Open Letter, September 2000. http://www.arl.org/info/letters/FTC091100.html.

[5] Martha Baer. Immortal Code. Wired Magazine, February 2003. http://www.wired.com/wired/archive/11.02/code.html.

[6] David Bank. 'Open Source' Database Poses Threat to Oracle. Wall Street Journal, July 2003. http://webreprints.djreprints.com/785490482991.html.

[7] Neil Bauman and Doc Searles. Linus Fields Dev Questions On the Future of Linux. Open Enterprise Trends Online Article, October 2003. http://www.oetrends.com/news.php?action=view_record&idnum=277.

[8] Anne K. Bingaman and Donald J. Russell. Competitive Impact Statement, July 1994. http://www.usdoj.gov/atr/cases/f0000/0045.htm.

[9] Carrier Grade Linux Specifications Subgroup. *Carrier Grade Linux Requirements Definition Version 2.0*, September 2003. http://www.osdl.org/docs/carrier_grade_linux_requirements_definition___version_20.pdf.

[10] Denise Caruso. Social Responsibility and the Digital Convergence. Computer Professionals for Social Responsibility, Berkeley Chapter, Newsletter, Fourth Quarter 1992. http://www.caruso.com/Various_Commentaries/Soc_Respon_92.txt.

[11] Digital Commerce Coalition. Ucita Yes — The Issue. WWW page, visited November 2003. http://www.ucitayes.org/issue/.

[12] Ronald H. Coase and H. Demsetz. The Lighthouse in Economics. *Journal of Law and Economics*, 17:357–376, October 1974.

[13] U.S. Federal Trade Commission. To Promote Innovation: The Proper Balance of Competition and Patent Law and Policy. Report, October 2003. http://www.ftc.gov/os/2003/10/innovationrpt.pdf.

[14] Microsoft Corporation. Form 10-Q For the Quarterly Period Ended September 30, 2002. United States Securities And Exchange Commission, November 14, 2002. http://www.sec.gov/Archives/edgar/data/789019/000103221002001614/d10q.htm.

[15] Data Center Linux Technical Working Group. *Proposed Data Center features for Linux in 2004 Draft Version Revision 0.7*, 2003. http://www.osdl.org/docs/dclfeaturesv7.html.

[16] Richard DeLamarter. IBM Antitrust Suit Records. Hagley Museum and Library, May 1991. http://www.hagley.lib.de.us/1980.htm.

[17] Peter J. Denning. Thrashing: Its Causes and Prevention. In *Proceedings AFIPS, 1968 Fall Joint Computer Conference*, volume 33, pages 915–922, 1968.

[18] Peter J. Denning. Virtual Memory. *Computing Surveys*, 2(3):153–189, September 1970.

[19] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux, Draft. Research Paper, January 2003. http://people.redhat.com/drepper/nptl-design.pdf.

[20] The Economist. Patent Wars: Better get yourself armed, everybody else is. *The Economist*, April 8, 2000.

[21] Thrainn Eggertsson. *Economic Behavior and Institutions*, pages 14–15. Cambridge University Press, 1990.

[22] eWeek. Sun's Schwartz Speaks Out on Linux, SCO. Online Article, February 2003. http://www.eweek.com/article2/0,4149,1274614,00.asp.

[23] Mary Jo Foley. Meet Microsoft's 'Joe Friday'. Microsoft Watch, July 2003. http://www.microsoft-watch.com/article2/0,4248,1207677,00.asp.

[24] John Fontana. Microsoft finally publishes secret Kerberos format. InfoWorld Online Article, April 2000. http://archive.infoworld.com/articles/en/xml/00/04/28/000428enkerpub.xml.

[25] Attorneys for Caldera Inc. Caldera Inc.'s Memorandum In Opposition To Defendant's Motion For Partial Summary Judgment On "Product Disparagement" Claims, April 1999. http://www.drdos.com/fullstory/dsprgmnt.html.

[26] Attorneys for Caldera Inc. Caldera, Inc.'s memorandum in support of the San Jose Mercury News, the Salt Lake Tribune, and Bloomberg L.P. motions to intervene and unseal court file, April 1999. http://www.drdos.com/fullstory/M26.pdf.

[27] Free Software Foundation. The Free Software Definition. WWW page, visited November 2003. http://www.gnu.org/philosophy/free-sw.html.

[28] Free Software Foundation. What is Copyleft? WWW page, visited January 2004. http://www.gnu.org/copyleft/copyleft.html.

[29] Robert H. Frank. *Microeconomics and Behavior*. McGraw-Hill, January 1991.

[30] Dan Geer, Rebecca Bace, Peter Gutmann, Perry Metzger, Charles P. Pfleeger, John S. Quarterman, and Bruce Schneier. CyberInsecurity: The Cost of Monopoly. Computer & Communications Industry Association, September 2003. http://www.ccianet.org/papers/cyberinsecurity.pdf.

[31] Al Gillen and Dan Kusnetzky. Worldwide Client and Server Operating Environment Market Forecast and Analysis, 2002-2007. IDC Research Report, October 2003.

[32] Mel Gorman. *Understanding The Linux Virtual Memory Manager*. Book in preparation, 2003. http://www.csn.ul.ie/~mel/projects/vm/.

[33] Colleen Graham and Kevin H. Strange. Gartner Study Shows Worldwide RDBMS Market Declined in 2002. Gartner News Analysis, May 2003. http://www.gartner.com/DisplayDocument?doc_cd=115036.

[34] IBM Software Group. Why DB2 vs Open Source Databases. Sales Guide, October 2003. ftp://ftp.software.ibm.com/software/data/pubs/papers/db2openspace.pdf.

[35] Trusted Computing Group. Trusted Computing Group: Frequently Asked Questions. WWW page, visited November 2003. https://www.trustedcomputinggroup.org/about/faq.

[36] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond Softnet. In *Proceedings of the 5th Annual Linux Showcase & Conference*. USENIX Association, November 2001.

[37] Harald Hagedorn. Patenting Software and Services - stakeholder view -. OECD Conference IPR, innovation and economic performance, Talk slides, August 2003. http://www.oecd.org/dataoecd/48/49/12600939.pdf.

[38] Garrett Hardin. The Tragedy of the Commons. *Science*, 162(3859):1243–1248, December 1968.

[39] Hans-Ulrich Heiss and Roger Wagner. Adaptive Load Control in Transaction Processing Systems. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 47–54, 1991.

[40] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, chapter 7, page 686. Morgan Kaufmann Publishers, third edition, 2003.

[41] IBM. Choose DB2 UDB the TCO leader. WWW page, visited November 2003. http://www.ibm.com/software/data/highlights/db2tco.html.

[42] IEEE-USA. Opposing Adoption of the Uniform Computer Information Transactions Act (UCITA) By the States. Position Paper, February 2000. http://www.ieeeusa.org/forum/POSITIONS/ucita.html.

[43] Gartner Inc. Gartner Says No Other Vendors Likely Candidates for PeopleSoft Merger. Gartner Press Release, July 2003. http://www.gartner.com/5_about/press_releases/pr11july2003a.jsp.

[44] Sun Microsystems Inc. Secure alternative desktop imminently available, says Sun Microsystems SA. Press Release, August 2003. http://www.itweb.co.za/office/sun/0308280734.htm.

[45] Intel. Intel® Pentium® III Processor: Processor Serial Number Questions & Answers. Intel Website, visited November 2003. http://support.intel.com/support/processors/pentiumiii/psqa.htm.

[46] Thomas Penfield Jackson. United States of America, Plaintiff, vs. Microsoft Corporation, Defendant: Court's Findings of Fact, November 1999. http://www.usdoj.gov/atr/cases/f3800/msjudgex.htm.

[47] Maryfran Johnson and Jaikumar Vijayan. Q&A: Sun's McNealy on company plans, role of CIOs. Computerworld, February 2003. http://www.computerworld.com/hardwaretopics/hardware/story/0,10801,78443,00.html.

[48] Michael Kanellos. Intel to phase out serial number feature. CNET News.com Online Article, April 27, 2000. http://news.com.com/2100-1040-239833.html?legacy=cnet.

[49] David A. Kelly. Linux Takes on the Enterprise. *Oracle Technology Network*, July 2003. http://otn.oracle.com/oramag/oracle/03-jul/o43linux.html.

[50] Con Kolivas. [RFC] Orthogonal Interactivity Patches. Linux Kernel Mailing List, August 2003. http://marc.theaimsgroup.com/?l=linux-kernel&m=106178160825835.

[51] Tom Krazit. Desktop Linux advocates to walk before they run. infoWorld, November 2003. http://www.infoworld.com/article/03/11/10/HNdesktopwalk_1.html.

[52] Jan Krikke. Microsoft Loses to Linux in Thailand Struggle. Linux Insider Online Article, November 2003. http://www.linuxinsider.com/perl/story/32110.html.

[53] David Lancashire. The Fading Altruism of Open Source Development. *First Monday*, 6(12), November 2001. http://firstmonday.org/issues/issue6_12/lancashire/index.html.

[54] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS Conference*, 1999. http://ssrnet.snu.ac.kr/~choijm/DEAR/sigmetrics99.ps.

[55] Rick Lehrbaum. Linux, Windows neck-and-neck in embedded. LinuxDevices Online Article, October 2002. http://www.linuxdevices.com/articles/AT7342059167.html.

[56] Lawrence Lessig. Innovating Copyright. *Cardozo Arts & Entertainment Law Journal*, 20(3):611–623, 2002. http://www.lessig.org/content/archives/innovatingcopyright.pdf.

[57] Lawrence Lessig. The Architecture of Innovation. *Duke Law Journal*, 51(6):1783–1801, 2002. http://www.lessig.org/content/archives/architectureofinnovation.pdf.

[58] Davide Libenzi. /dev/epoll Home Page, 2001. http://www.xmailserver.org/linux-patches/nio-improve.html.

[59] Linux Weekly News. *The 2003 Kernel Developers Summit*, July 2003. http://lwn.net/Articles/KernelSummit2003/.

[60] Brian Livingston. Is Microsoft's change in Kerberos security a form of 'embrace, extend, extinguish'? InfoWorld Online Article, May 2000. http://dir.salon.com/tech/log/2000/05/11/slashdot_censor/index.html.

[61] Roger Luethi. Open Source Software for Simulation and Artificial Evolution. Semesterarbeit, Institut für Informatik, Universität Zürich, June 2002.

[62] Roger Luethi. Position Paper for the XP-2003 Workshop: Making Free/Open-Source Software (F/OSS) Work Better. In Brian Fitzgerald and David L. Parnas, editors, *Proceedings of Workshop at XP2003 Conference Genoa*, May 2003.

[63] Bart Massey. Why OSS Folks Think SE Folks Are Clue-Impaired. In Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani, editors, *Taking Stock of the Bazaar: 3rd Workshop on Open Source Software Engineering*. International Conference on Software Engineering (ICSE'03), May 2003. http://opensource.ucc.ie/icse2003/3rd-WS-on-OSS-Engineering.pdf.

[64] P. E. McKenney and J. D. Slingwine, editors. *Read-copy update: using execution history to solve concurrency problems*. International Conference on Parallel and Distributed Computing and Systems, October 1998.

[65] Marshall Kirk McKusick. Twenty Years of Berkeley Unix – From AT&T-Owned to Freely Redistributable. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*, chapter 3. O'Reilly, January 1999.

[66] Robert McMillan. Microsoft report prompts Forrester policy change. IDG News Service, San Francisco Bureau, October 2003. http://www.idg.com.sg/idgwww.nsf/unidlookup/E28B20E2A1DD03E048256DB7002AAA90?OpenDocument.

[67] Larry McVoy. Silly BK statistics. Linux-Kernel mailing list, October 2003. http://marc.theaimsgroup.com/?l=linux-kernel&m=106610041609764.

[68] Microsoft. Microsoft SQL Server: Strategic IT Initiatives: TCO Benefits of SQL Server. WWW page, visited November 2003. http://www.microsoft.com/sql/evaluation/compare/nervewiretco.asp.

[69] Robin Miller. Microsoft Asks Slashdot To Remove Readers' Posts. Posting, May 11, 2000. http://slashdot.org/article.pl?sid=00/05/11/0153247.

[70] Elinor Mills. Boycott widened over new Intel chip ID plan. CNN.com Online Article, January 29, 1999. http://www.cnn.com/TECH/computing/9901/29/intel.boycott.idg/.

[71] Elinor Mills. IBM to disable serial number in Pentium III. CNN.com Online Article, March 1, 1999. http://www.cnn.com/TECH/computing/9903/01/p3disable.idg/.

[72] Eben Moglen. Open Letter, October 2001. http://www.nccusl.org/nccusl/meetings/UCITA_Materials/kunze-ucita.pdf.

[73] Ingo Molnar. [announce] [patch] ultra-scalable O(1) SMP and UP scheduler. Linux-Kernel mailing list, January 2002. http://marc.theaimsgroup.com/?l=linux-kernel&m=101010394225604.

[74] Deirdre K. Mulligan, Ken McEldowney, Beth Givens, and Bob Bullmash. In the Matter of Intel Pentium III Processor Serial Number. Complaint and Request for Injunction, Request for Investigation, and for Other Relief, February 1999. http://www.cdt.org/privacy/issues/pentium3/990226intelcomplaint.shtml.

[75] National Conference of Commissioners on Uniform State Laws Drafting Committee. Uniform Computer Information Transactions Act (2000). Draft, August 2000. http://www.law.uh.edu/ucc2b/082000/082000.html.

[76] Danish Board of Technology. Open-source software - in e-government. Analysis and recommendations., October 2003. http://www.tekno.dk/pdf/projekter/p03_opensource_paper_english.pdf.

[77] The Open Group. *The Open Group Base Specifications Issue 6, IEEE Std POSIX 1003.1-1996*, 2003.

[78] Oracle. DB2's Low CostConclusion Is a Delusion–Oracle is the TCO Leader. WWW page, visited November 2003. http://www.oracle.com/features/insider/index.html?1206_oi_dhbrown.html.

[79] Janusz A. Ordover, Alan O. Sykes, and Robert D. Willig. Predatory Systems Rivalry: A Reply. *Columbia Law Review*, 83:1150–1166, June 1983.

[80] World Intellectual Property Organization. WIPO Copyright Treaty and Agreed statements Concerning the WIPO Copyright Treaty . Treaty, December 1996. http://www.wipo.int/clea/docs/en/wo/wo033en.htm.

[81] Margit Osterloh, Sandra Rota, and Bernhard Kuster. Open Source Software Production: Climbing on the Shoulders of Giants. Working Paper, February 2003. http://www.ifbf.unizh.ch/orga/downloads/publikationen/osterlohrotakuster.pdf.

[82] Cover Pages. Patents and Open Standards. WWW page, visited January 2004. http://xml.coverpages.org/patents.html.

[83] Bruce Perens et al. The Open Source Definition. WWW page, visited November 2003. http://www.opensource.org/docs/definition.html.

[84] Eric S. Raymond. Homesteading the Noosphere. *First Monday*, 3(10), October 1998. http://www.firstmonday.dk/issues/issue3_10/raymond/index.html.

[85] Eric S. Raymond. The Revenge of the Hackers. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*, chapter 15. O'Reilly, January 1999.

[86] Eric S. Raymond, editor. *Jargon File*, 4.3.1, June 2001. http://www.tuxedo.org/~esr/jargon/jarg431.gz.

[87] Eric S. Raymond. *The Cathedral & the Bazaar – Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, January 2001. http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

[88] Eric S. Raymond. *The Art of Unix Programming*, chapter Chapter 16: Reuse. Addison-Wesley, September 2003. http://www.faqs.org/docs/artu/reusechapter.html.

[89] Eric S. Raymond. The Magic Cauldron. WWW page, 2003. http://www.catb.org/~esr/writings/magic-cauldron/.

[90] Gary L. Reback. Patently Absurd. *Forbes*, June 2002. http://www.forbes.com/asap/2002/0624/044_print.html.

[91] Paula Rooney. IBM To Launch Comprehensive Linux Desktop Support Program Next Year. CRN, November 2003. http://www.crn.com/sections/BreakingNews/dailyarchives.asp?ArticleID=45881.

[92] SAP. SAP Entrusts Its Database to MySQL Open Source Community. Press Release, May 2003. http://www.sapdb.org/7.4/pdf/pressrelease_eng.pdf.

[93] Seth Schoen. Trusted Computing: Promise and Risk. EFF Online Article, visited November 2003. http://www.eff.org/Infra/trusted_computing/20031001_tc.php.

[94] Andrew Schulman. Examining the Windows AARD Detection Code. *Dr. Dobb's Journal*, September 1993. http://www.ddj.com/documents/s=1030/ddj9309d/.

[95] Intel Solution Services. Linux Scalability: The Enterprise Question. White Paper, 2000. http://cedar.intel.com/media/pdf/linux/linux_scalability-enterprise_scs.pdf.

[96] Barbara Simons. ACM Letter on UCC2B 10/98. Open Letter, July 1999. http://www.acm.org/usacm/IP/usacm-ucita.html.

[97] Kragen Sitaker. http://www.canonical.org/~kragen/beowulf-faq.txt.

[98] Richard M. Stallman. *Free Software, Free Society: Selected Essays of Richard M. Stallman*, chapter Can you trust your computer? Gnu Press, October 2002.

[99] Jason Stamper. Oracle Developers Switch Allegiance to Linux. *Computer Business Review Online*, October 2003. http://www.cbronline.com/currentnews/04a9a6dadd15336480256dce001e3e51.

[100] Suzanne Taylor and Kathy Shroeder. *Inside Intuit: How the Makers of Quicken Beat Microsoft and Revolutionized an Entire Industry*. Harvard Business School Press, September 2003.

[101] SD Times. Unix Market Consolidates. Online Article, November 2000. http://www.sdtimes.com/news/017/special1.htm.

[102] top500.org. http://www.top500.org/list/2003/11/.

[103] Vinod Valloppillil. Linux OS Competitive Analysis: The Next Java VM? Leaked Microsoft internal Memo, November 1998. http://www.opensource.org/halloween/halloween2.php.

[104] Vinod Valloppillil. Open Source Software – A (New?) Development Methodology. Leaked Microsoft internal Memo, November 1998. http://www.opensource.org/halloween/halloween1.php.

[105] John Viega. The Myth of Open Source Security. Online Article, May 2000. http://www.my-opensource.org/lists/myoss/2000-06/msg00077.html.

[106] Chris Vine. 2.6.0-test9 - poor swap performance on low end machines. Linux Kernel Mailing List, October 2003. http://marc.theaimsgroup.com/?l=linux-kernel&m=106746692622127.

[107] Chris Vine. Re: 2.6.0-test9 - poor swap performance on low end machines. Linux Kernel Mailing List, November 2003. http://marc.theaimsgroup.com/?l=linux-kernel&m=106798381909953.

[108] Paul Vixie. Software Engineering. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*, chapter 7. O'Reilly, January 1999.

[109] Ning Wang. Measuring Transaction Costs: An Incomplete Survey. Research Paper, February 2003. http://coase.org/w-wang2003measuringtransactioncosts.pdf.

[110] wikipedia.org. DR-DOS, visited November 2003. http://en.wikipedia.org/wiki/DR-DOS.

[111] wikipedia.org. First Sale Doctrine, visited January 2004. http://en.wikipedia.org/wiki/First_Sale_Doctrine.

[112] wikipedia.org. Network effect, visited January 2004. http://en.wikipedia.org/wiki/Network_effect.

[113] wikipedia.org. Quicksort, visited January 2004. http://en.wikipedia.org/wiki/Quicksort.

[114] Sam Williams. *Free as in Freedom - Richard Stallman's Crusade for Free Software*, chapter 11. O'Reilly, March 2002.

[115] Oliver E. Williamson. *The Economic Institutions of Capitalism*. New York: The Free Press, 1985.

[116] Glenn A. Woroch, Frederick R. Warren-Boulton, and Kenneth C. Baseman. Exclusionary Behavior in the Market for Operating System Software: the Case of Microsoft. In David Gabel and David Weiman, editors, *Opening Networks To Competition: The Regulation and Pricing of Access*. Kluwer Academic Publishers, 1997. http://elsa.berkeley.edu/~woroch/exclude.pdf.