Beyond Expert-Only Parallel Programming?

Paul E. McKenney
IBM Linux Technology Center
15400 SW Koll Parkway
Beaverton, OR USA
paulmck@linux.vnet.ibm.com

ABSTRACT

My parallel-programming education began in earnest when I joined Sequent Computer Systems in late 1990. This education was both brief and effective: within a few short years, my co-workers and I were breaking new ground [MG92, MS93, MS98].[1] Nor was I alone: Sequent habitually hired new-to-parallelism engineers and had them producing competent parallel code within a few months. Nevertheless, more than two decades later, parallel programming is perceived to be difficult to teach and learn. Is parallel programming an exception to the typical transitioning of technology from impossible to expert-only to routine to unworthy of conscious thought?

1. INTRODUCTION

In 2006, Linus Torvalds noted that in the prior three-year period, the Linux community's grasp of concurrency had improved to the point that patches involving locking were often correct at first submission. In contrast, locking patches submitted in 2003 frequently contained fatal concurrency bugs [Tor06].

What changed between 2003 and 2006 to cause this huge improvement in code quality? It was not the programming language, which was C before, during, and after. It wasn't the synchronization primitives, either: the most common synchronization primitive in Linux for the duration was, by far, locking [McK06].[2]

We are told that parallel programming is a grand challenge impossible to address with today's technology, but the Linux experience argues differently. We must look beyond the programming language and synchronization primitives to solve this mystery. This paper proposes a comforting, but currently underappreciated, answer: Parallel programming is actually becoming easier, as we develop a parallel programming culture and parallel programming tools. To this end, Section 2 investigates acculturation, Section 3 reviews economic changes, Section 4 discusses tooling, Section 5 discusses progress since 2006 along with future directions, and finally, Section 6 presents concluding remarks.

2. ACCULTURATION

My education on parallel programming was typical: I was given a copy of some training materials for Sequent's DYNIX/ptx operating-system kernel,[3] my code was subject to detailed review, and finally, perhaps most important, I was surrounded by accomplished parallel programmers. This apprenticeship-style training had very few "drop outs" and delivered impressive results. For example, in the seven years before I joined, Sequent invalidated the earlier common wisdom that operating systems could not scale beyond two to four CPUs. During that time, Sequent transitioned 30-CPU software scalability from impossible to expert-only to routine.

Acculturation is an underappreciated but powerful force driving technology adoption. This force is not restricted to computer technologies, for example, over the past five centuries, basic arithmetic has moved from a topic of advanced study to an elementary topic taught to all children in grade school [Swe87]. Nor is this an isolated incident: Car driving progressed from a highly skilled task for a chauffeur/mechanic to a simple task taught to almost every American teenager—in less than a century.

To bring the focus back to computing, in less than 30 years, the Internet has gone from a topic for advanced research to an indispensable tool for all ages. Finally, I attended a talk in the late 1970s by none other than Edsger Dijkstra in which he claimed that the typical programmer could not be trusted to correctly code a "while" loop. In contrast, by the turn of the century, it was not unheard of for children to grasp the use of "while" loops well before the age of ten.

Given this weight of evidence from disparate fields of endeavor, along with the speed at which Linux-kernel parallelism moved from expert-only to routine, the burden of

[1] The work reported in the latter McKenney and Slingwine publication [MS98] took place in 1993.

[2] Much as I might hate to admit it, it was not RCU [MS98, McK04]: Although RCU has recently been noted as beneficial to the Linux kernel's scalability [CKZ12, BWCM$^+$10], in 2006, there were fewer than 1,000 uses of RCU in the Linux kernel, and more than 40,000 uses of locking [McK06].

[3] These never were published, but were similar to their user-level counterpart [Seq88] and not too different from material created for the Linux kernel [Rus03]. In addition, many of the key points were alluded to in a pair of 1985 USENIX papers [BK85, Inm85].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
RACES '12 Tucson, Arizona USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

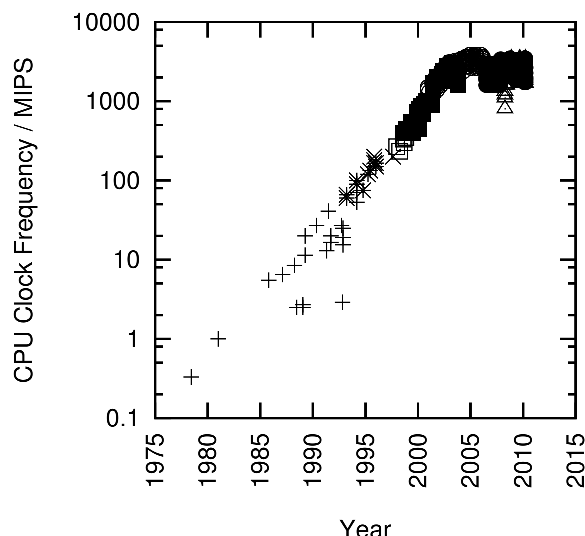

Figure 1: MIPS/Clock-Frequency Trend for Intel CPUs

proof must fall on those claiming that parallel programming will remain difficult.

However, acculturation requires motivation. Although the exact motivation for whole-hearted adoption of parallelism in the Linux kernel is subject to debate, Figure 1 shows a key suspect, namely that during the time period between 2003 and 2006, CPU clock frequencies ceased the exponential climb of the preceding two decades. During this time, it became quite clear that parallelism would pervade the computing industry [Sut05]. Within the Linux community, the questions during code review shifted from whether or not a given level of scalability should be provided to how best to provide it. This motivation helped move parallelism within the Linux kernel community from expert-only to routine.

This industry shift also dramatically changed the economics of parallel systems, as discussed in the next section.

3. ECONOMICS

In the early 1990s, I was assigned to work with Sequent's benchmarking center. They were not achieving their goal: Their CPUs were fully utilized, but there was almost no lock contention and also ample I/O and system-interconnect bandwidth. The solution was simple: Add more CPUs. Thus it was that I found myself carrying five Sequent CPU boards across the parking lot, each equipped with a pair of 80486 CPUs clocked at 50MHz. When I got about halfway across the parking lot, I suddenly realized that I was carrying no less than three times the purchase price of my house in my arms.[4] In the early 1990s, therefore, only a fortunate few had access to parallel systems: (1) People like myself who worked for companies manufacturing them, (2) those working for companies that used them and had spare cycles on test systems, and (3) students attending universities that owned them, perhaps due to earlier research grants. It is no surprise that parallel-programming knowledge and experience remained obscure during the early 1990s.

[4] Yes, I did walk more carefully after that. Why do you ask?

However, by 2006 the price of parallel systems had dropped dramatically, to the point where a graduate student purchased a dual-core system for no better reason than to be able to do a presentation on parallel processing on a parallel processor [HMB06]. In less than 20 years, the cost of a parallel system had dropped from multiples of that of a house to a fraction of that of a used car.

Thus, this economic motivation had a powerful ally, namely, the advent of low-cost dual-core laptops. These laptops dramatically increased the number of Linux developers who could afford parallel hardware. This increased number of developers in turn made possible more capable tooling, as described in the next section.

4. TOOLING

A worker is only as good as his or her tools. To see that this time-honored adage still holds, consider the Internet example from Section 2. Why has Internet become so much easier to use? Although acculturation is one important reason, the fact is that today's users do not spend anywhere as much time crawling around above ceilings adjusting Ethernet transceivers as did users of the early 1980s.

At that time, connecting a system to the Internet was a complex task requiring specialized knowledge and skills. Today, the task is if anything even more complex, especially given the wide variety of wireless protocols. However, almost all of these tasks are now carried out automatically by hardware and software tools, so that in many environments, the user need only power up his or her device.

The development of these hardware and software tools was driven by the economics described in the previous section: The greater the number of users, the more economic sense it makes to save small amounts of their time. This economic force applies equally well to tooling for parallel programming.

To see this, contrast the size of Sequent's DYNIX/ptx development team (at most 40 people) with that of the Linux kernel (numbering in the thousands). Suppose that a software tool saves one percent of each developer's time, but costs one developer-year to construct. It would take the DYNIX/ptx team at least 2.5 years to reach breakeven. In stark contrast, the Linux kernel community would achieve breakeven in less than six weeks. Therefore, all else being equal, the greater the number of developers, the greater the number and variety of software tools, which is in fact one of many advantages of strong software ecosystems.

The Linux experience from 2003 to 2006 bears this out, with three important tools being introduced during this time. The first was "sparse" [Cor04], a static analyzer that (among other things) can detect some cases of lock acquisition and release mismatches. The second is "lockdep" [Cor06], which computes a lock-dependency graph at runtime, automatically detecting and reporting potential deadlock cycles. The third is "coccinelle" [PLM06], which is a source-code analyzer capable of generating patches to fix all occurrences of Linux-kernel bugs expressible in the SmPL domain-specific language, including some concurrency bugs.

Of these three tools, the most successful thus far has been lockdep. The secret of its success is threefold. First, it has relatively few false positives, and most false positives are easily suppressed. In contrast, sparse's output is heavily ridden with false positives, requiring more time and effort to interpret. Second, lockdep is easy to run, in contrast

with coccinelle, which requires specific SmPL patterns to be written to locate specific classes of bugs. Third, lockdep's implementation has proven quite flexible, which has allowed it to detect a number of other classes of concurrency bugs in addition to its original mandate of deadlock cycles [McK10a]. These three properties have acculturated use of lockdep, which is in fact mandated in Linux's patch-submission checklist (`Documentation/SubmitChecklist`). Of course, nothing stops an individual developer from omitting this step of the checklist, but because Linux's maintainers and testers habitually run lockdep, such an omission will be noticed sooner rather than later. The lockdep experience stands in happy contrast to the huge amounts of ink spilled decrying locking's potential for deadlock. One can only hope that future researchers wishing to improve the lot of developers will take to heart the lessons from this contrast.

Please note that assessment of these tools is necessarily subjective and varies with time. For example, the number of coccinelle patterns is constantly growing, and they might well be run on every change to each maintainer's source-code repository.[5] Such a regimen might well eventually cause coccinelle to find as many concurrency bugs than does lockdep, in addition to coccinelle's long list of non-concurrency bugs.

In addition, assessments vary across communities. For example, the DYNIX/ptx community considered deadlock to be neither difficult to avoid or hard to debug. Therefore, the only deadlock tooling in DYNIX/ptx tagged each spinlock with the number of the CPU currently holding it. Something like lockdep would have been considered overkill, especially given the economics of DYNIX/ptx's smaller number of developers. In contrast, in the Linux community, although all three of these tools consumed many developer-years of effort, they also produced large returns, removing numerous bugs from the Linux kernel.

Because good tools greatly ease the task of producing correct parallel code, they accelerate the acculturation process. Greater acculturation results in more developers, which shifts economics further in favor of tooling improvement. When this circle is operating full force, it generates surprisingly large improvements in software quality in very short time periods, as seen in the Linux kernel from 2003 to 2006.

5. PROGRESS AND FUTURE DIRECTIONS

An important contributor to the Linux kernel community's great progress in parallel programming during the time period from 2003 to 2006 was a virtuous circle involving acculturation, economics, and tooling. Those of us who would like to see continued successful adoption of parallelism would therefore be wise to initiate and promote such circles. To that end, the following sections discusses each of the three segments of this circle.

5.1 Progress: Acculturation

Acculturation has been proceeding rapidly since 2006, for example as measured by the number of parallel-programming texts [HS08, Sco06, SSRB00, BHS07, MSM05, But97, Sut08, Rei07, Lea97, GPB+07, CRKH05, McK12a]. In addition, there now are many parallel open-source projects, allowing easy access to production-quality code for study and experimentation. Finally, universities have used open-source projects in their coursework for at least ten years.

That said, there is still a need for better educational materials, particularly surrounding design. My own design education took place in mechanical engineering, but fortunately the lessons carried over to computing. Of course, there are design principles specific to parallel programming, particularly the partitioning and replication techniques required to achieve high performance and scalability, along with sets of transformations to convert broad classes of non-partitionable problems into partitionable form [McK12a, Section 5]. In addition, parallelism is but one performance optimization of many. Therefore, a key component of any parallel design course must include identifying which classes of performance problems are best addressed by parallelism.

Of course, education is not the only way to promote acculturation. To see this, note that the major figures in each project called out in Section 4 was intimately involved in the Linux kernel community. In the case of sparse (kicked off by Linus Torvalds) and lockdep (kicked off by Ingo Molnar), this is unsurprising. The case of coccinelle is more instructive. Julia Lawall is a researcher at INRIA (formerly with Copenhagen University), but regularly contributes code to the Linux kernel. So much so that she sometimes appears on the list of the top 20 contributors to the Linux kernel and that she was invited to the exclusive invitation-only Linux Kernel Summit in 2010. The lesson here is that if you truly wish to help a group of people, there is no substitute for living among them. Please note that this is not to say that each and every researcher should contribute heavily to some open-source community. Far from it—such a policy would be a wasteful failure to apply division of labor. Instead, each researcher should speak regularly with at least one person who has participated fully in some development project, but who also intimately understands the research community.

Similarly, developers would do well to be acquainted with someone who intimately understands research, presumably also by living among researchers. Perhaps greater communication between researchers and developers will help to narrow the gap between these two communities [McK11b].

5.2 Progress: Economics

Although having an significant impact on large-scale economics is beyond the capacity of most individual researchers and developers, it should not be beyond their imaginations. In my case, a course in engineering economics combined with running my own business for several years has been of great help. However, the only reason I took this course was that it was required for my mechanical engineering degree. Perhaps it should be added to the computer science curriculum.

Going forward, the advent of multicore smartphones will push parallel programming even further into mainstream computing, and, more important, greatly increase the unit volumes and decrease the costs of parallel systems and applications. Although this will enable more people to try their hand at parallel programming, it will also require additional work, particularly tooling for validation and reliability.

5.3 Progress: Tooling

Great progress has been made on tooling since 2006. Both sparse and lockdep have added RCU support [McK10b], and lockdep has greatly improved its diagnostics, though additional improvement is possible [Ros11]. Coccinelle has ma-

[5] There is already similar automation that does build-and-boot testing on each change to each maintainer's source-code repository, which has proven quite effective.

tured greatly and is frequently used to find problems in the Linux kernel [PTS+11]. A number of maintainers use combinations of these three tools as part of their validation efforts.

Additional tools have come to light in the years since 2006. One tool that has found a surprising number of concurrency bugs is Trinity, which is an intelligent system-call-level stress test [Jon11]. The University of Cambridge's PPCMEM tool [AMP+11, SSA+11, SMO+12] is also an interesting development, which is capable of validating small algorithms on weakly ordered systems, including some fragments of the Linux kernel. Formalization of new synchronization mechanisms [GRY12] will eventually lead to more powerful validation techniques, while work illuminating the effects of APIs on parallel performance [AGH+11, McK11a] will help guide API design. These types of validation are critically important to acculturation, which relies heavily on use Linux-kernel community code review as part of the learning process. This clearly requires high-quality concurrent code in the Linux kernel, which is promoted by the aggressive validation regimens using advanced tooling.

To complete the cycle, open-source projects serve as an excellent set of test cases for all manner of software tools: Nothing builds credibility for a given tool among practitioners quite like that tool finding a serious problem in their code, particularly if those practitioners spent significant time attempting to track it down. Tooling improves the quality of publicly available code, and also provides test cases for further improvements in tooling, which further improves code quality, forming another virtuous circle.

There is room for many other types of tools, including data-race detectors, whole-program static analyzers that detect additional cases of unbalanced critical-section operations, as well as a long list of full-state-space-search tools [Hol03, SSA+11]. But perhaps the greatest need is for tools that validate the combinations of synchronization operations used in concert by large parallel projects [Bro11, McK12a].

5.4 Progress: Routine

Parallelism is well on its way to becoming routine. Once it does, what will researchers and developers work on?

1. Core counts are still increasing, and there is plenty of work needed to handle systems with hundreds (to say nothing of thousands) of CPUs.

2. The end of clock-frequency scaling ten years ago has renewed interest in special-purpose hardware, which will pose special challenges.

3. The combination of parallelism and real-time response is increasingly important [McK07, McK12c].

4. The combination of parallelism and energy efficiency will only grow in importance with the increasing need for energy conservation [Cor01, McK12b]. In fact, parallelism and energy efficiency are closely linked because decreases in CPU clock frequencies produce quadratic improvements in energy efficiency.

5. The combination of parallelism, real-time response, and energy efficiency will also become increasingly important. I am not aware of much progress in this area, partly due to the real-time habit of automatically disabling all energy-efficiency options, but that will change.

6. The extremely high volumes of multicore embedded systems will place additional stress on parallel-programming methodologies, requiring better validation techniques and tooling, along with modifications to the parallel-programming methodologies themselves.

7. What are now expert-only weakly ordered techniques must become usable by the masses, given the expense of strong ordering [HSW96, AGH+11]. RCU is one effort in this area [MS98, McK04, HMBW07, DMS+12], but there are any number of other expert-only techniques that can be made safe for a broader user base through acculturation, tooling, and componentry, with many more waiting to be discovered.

8. A rigorous theoretical basis is required for all of the above. There is some recent intriguing progress in this area [GRY12], which will hopefully support a new generation of tooling.

Rest assured that there will be no shortage of exciting things to work on for the foreseeable future.

6. CONCLUSION

This paper has called out an important virtuous circle involving acculturation, economics, and tooling, which was a major factor behind the marked increase in quality of parallel code in the Linux kernel from 2003 to 2006. Furthermore, this virtuous circle is still in operation, which is fortunate given the increases in reliability that will be required to support the huge unit volumes of multicore smartphones and other embedded devices. In short, the preponderance of the evidence indicates that parallel programming is no exception to the long standing progression of new technologies: What is impossible today will be expert-only tomorrow, routine the next day, and unworthy of conscious thought the day after that.

This progression will not only bring the benefits of parallel programming to mainstream computing, it will also free up researchers and developers to take on larger challenges, including energy efficiency, real-time response, and yet unforeseen expert-only parallel-programming techniques. Therefore, if teaching parallel programming remains problematic, we will need to raise our teaching game.

Acknowledgments

I owe thanks to Eddie Kohler for many fruitful discussions on this topics, to the RACES referees, and to many colleagues at Sequent, IBM, and in the Linux community. Bob Beck deserves special mention for his leading role in formulating Sequent's parallel-programming methodology [BK85]. I am grateful to Jim Wasko for his support of this effort.

Legal Statement

This work represents the views of the author and does not necessarily represent the views of IBM.
Linux is a registered trademark of Linus Torvalds.
Other company, product, and service names may be trademarks or service marks of such companies.

7. REFERENCES

[AGH+11] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, and Maged M. Michael. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, New York, NY, USA, 2011. ACM.

[AMP+11] Jade Alglave, Luc Maranget, Pankaj Pawan, Susmit Sarkar, Peter Sewell, Derek Williams, and Francesco Zappa Nardelli. PPCMEM/ARMMEM: A tool for exploring the POWER and ARM memory models. June 2011.

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, Chichester, West Sussex, England, 2007.

[BK85] Bob Beck and Bob Kasten. VLSI assist in building a multiprocessor UNIX system. In *USENIX Conference Proceedings*, pages 255–275, Portland, OR, June 1985. USENIX Association.

[Bro11] Neil Brown. Meet the Lockers. Available: `http://lwn.net/Articles/453685/` [Viewed September 2, 2011], August 2011.

[But97] David Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Boston, MA, USA, 1997.

[BWCM+10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *9th USENIX Symposium on Operating System Design and Implementation*, pages 1–16, Vancouver, BC, Canada, October 2010. USENIX.

[CKZ12] Austin Clements, Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, pages @@@–@@@, London, UK, March 2012. ACM.

[Cor01] Jonathan Corbet. No more jiffies? Available: `http://lwn.net/2001/0412/bigpage.php3#kernel` [Viewed August 10, 2012], April 2001.

[Cor04] Jonathan Corbet. Finding kernel problems automatically. Linux Weekly News, June 2004.

[Cor06] Jonathan Corbet. The kernel lock validator. Available: `http://lwn.net/Articles/185666/` [Viewed: March 26, 2010], May 2006.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., third edition, 2005.

[DMS+12] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23:375–382, 2012.

[GPB+07] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java: Concurrency in Practice*. Addison Wesley, Upper Saddle River, NJ, USA, 2007.

[GRY12] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying highly concurrent algorithms with grace (extended version). Available: `http://sites.google.com/site/popl13grace/paper.pdf` [Viewed August 4, 2012], July 2012.

[HMB06] Thomas E. Hart, Paul E. McKenney, and Angela Demke Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *20th IEEE International Parallel and Distributed Processing Symposium*, Rhodes, Greece, April 2006. Available: `http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf` [Viewed April 28, 2008].

[HMBW07] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.

[Hol03] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA, USA, 2008.

[HSW96] Maurice Herlihy, Nir Shavit, and Orli Waarts. Linearizable counting networks. *Distrib. Comput.*, 9:193–203, February 1996.

[Inm85] Jack Inman. Implementing loosely coupled functions on tightly coupled engines. In *USENIX Conference Proceedings*, pages 277–298, Portland, OR, June 1985. USENIX Association.

[Jon11] Dave Jones. Trinity: A system call fuzzer. In *Proceedings of the 13th Ottawa Linux Symposium*, pages ???–???, Ottawa, Canada, June 2011.

[Lea97] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley Longman, Reading, MA, USA, 1997.

[McK04] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf` [Viewed October 15, 2004].

[McK06] Paul E. McKenney. RCU Linux usage. Available: `http://www.rdrop.com/users/paulmck/RCU/linuxusage.html` [Viewed January 14, 2007], October 2006.

[McK07] Paul E. McKenney. SMP and embedded real time. *Linux Journal*, (153):52–57, January 2007. Available: http://www.linuxjournal.com/article/9361 [Viewed May 31, 2007].

[McK10a] Paul E. McKenney. Lockdep-RCU. Available: https://lwn.net/Articles/371986/ [Viewed June 4, 2010], February 2010.

[McK10b] Paul E. McKenney. The RCU API, 2010 edition. Available: http://lwn.net/Articles/418853/ [Viewed December 8, 2010], December 2010.

[McK11a] Paul E. McKenney. Concurrent code and expensive instructions. Available: http://lwn.net/Articles/423994 [Viewed January 28, 2011], January 2011.

[McK11b] Paul E. McKenney. Verifying parallel software: Can theory meet practice? http://www.rdrop.com/users/paulmck/scalability/paper/VericoTheoryPractice.2011.01.28a.pdf, January 2011.

[McK12a] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2012. Available: http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html [Viewed March 28, 2010].

[McK12b] Paul E. McKenney. Making RCU safe for battery-powered devices. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdynticks.2012.02.15b.pdf [Viewed March 1, 2012], February 2012.

[McK12c] Paul E. McKenney. Real-time response on multicore systems: It is bigger than you think. Available: http://www.seas.gwu.edu/~gparmer/ospert12/bigrt.2012.07.10a.pdf [Viewed August 10, 2012], July 2012.

[MG92] Paul E. McKenney and Gary Graunke. Efficient buffer allocation on shared-memory multiprocessors. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, pages 194–199, Tucson, AZ, February 1992. The Institute of Electrical and Electronics Engineers, Inc.

[MS93] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings*, pages 295–306, Berkeley CA, February 1993. USENIX Association. Available: http://www.rdrop.com/users/paulmck/scalability/paper/mpalloc.pdf [Viewed January 30, 2005].

[MS98] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998. Available: http://www.rdrop.com/users/paulmck/RCU/rclockpdcsproof.pdf [Viewed December 3, 2007].

[MSM05] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, Boston, MA, USA, 2005.

[PLM06] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers. In *Proceedings of the ACM SIGOPS EuroSys 2006 Conference*, pages 59–71, Leuven, Belgium, April 2006. ACM.

[PTS+11] Nicolas Palix, Ga el Thomas, Suman Saha, Christophe CalvÃĺs, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, pages 305–318, Newport Beach, California, USA, March 2011. ACM.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA, USA, 2007.

[Ros11] Steven Rostedt. lockdep: How to read its cryptic output. http://www.linuxplumbersconf.org/2011/ocw/sessions/153, September 2011.

[Rus03] Rusty Russell. Unreliable guide to locking. Available: http://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/index.html [Viewed September 10, 2012], 2003.

[Sco06] Michael Scott. *Programming Language Pragmatics*. Morgan Kaufmann, Burlington, MA, USA, 2006.

[Seq88] Sequent Computer Systems, Inc. *Guide to Parallel Programming*, 1988.

[SMO+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronizing C/C++ and POWER. In *Programming Language Design and Implementation (PLDI) 2012*, Beijing, China, June 2012.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Programming Language Design and Implementation (PLDI) 2011*, San Jose, CA, USA, June 2011.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, Chichester, West Sussex, England, 2000.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005. Available: http://www.gotw.ca/publications/concurrency-ddj.htm [Viewed January 1, 2009].

[Sut08] Herb Sutter. Effective concurrency. Series in

Dr. Dobbs Journal, 2008.

[Swe87] Frank J. Swetz. *Capitalism & Arithmetic: The New Math of the 15th Century*. Open Court, 1987.

[Tor06] Linux Torvalds. Open forum on os architecture for multicore and manycore platforms. Panel Discussion, hosted by Intel Research Council Scalable Systems Committee, November 2006.