Challenges in Operating-Systems Reengineering for Many Cores[*]

[Position Statement]

Michael Gernoth, Daniel Lohmann, Wolfgang Schröder-Preikschat,
Julio Sincero, Reinhard Tartler, Dirk Wischermann
Friedrich-Alexander University Erlangen-Nuremberg
Department of Computer Science 4
Erlangen, Germany
{mg,dl,wosch,js,rt,dw}@cs.fau.de

ABSTRACT

General purpose operating systems such as Linux are reasonably suited for managing massively parallel computing platforms made from many-core processors. However, due to limitations in organization and architecture of the system software, these sorts of operating systems are fairly unsuited for parallel execution in order to better perform on behalf of the (massively) parallel processes needed for running one or more application programs. Regarding many-core support, their functional properties are satisfactorily, however, their nonfunctional properties leave a lot to be desired.

The paper touches on some of the problems discovered in reengineering critical sections of operating systems. It aims at making aware of difficulties, rather than providing solutions, in adapting system software to parallel processing.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Design, Software Architectures, Reusable Software; D.4.7 [**Operating Systems**]: Organization and Design—*hierarchical design*

General Terms

Critical section engineering

Keywords

Synchronization, Scalability, Variability

1. INTRODUCTION

Forthcoming many-core processor technology poses serious challenges not only to software in general but in particular to *system software*, that is, operating systems. Although an operating system is always the epitome of a non-sequential program and developers can fall back on substantial know-how in this domain, contemporary and prominent exponents still suffer from considerable performance handicaps when the number of processors increases significantly. In addition, an effective use of cores for ones own sake in order to improve overall system service is anything but simple. Stumbling block often is the overall design and the software architecture of the operating system, that is to say, the way how the various system functions are implemented and not necessarily what these functions are about.

Being in the fortunate position of building a parallel operating system from scratch, taking care of high performance and dedicated application support is almost straightforward, particularly by experience made with large-scale shared-memory systems [3]. Recent developments set a good example [2, 1, 5]. However, reengineering a contemporary general-purpose operating with the objective of providing roughly similar performance as those special-purpose (parallel) ones is a complete different story—nonetheless indispensible if one is not willing to forswear all the comfortable services general-purpose operating systems typically provide.

2. ROOT OF ALL EVIL

Recent experiments with Linux 2.6 on a 16-core processor [2] revealed, on the one hand, a tremendous decline in performance by a factor of 40 due to nonlocal data accesses issued from "remote" cores and, on the other hand, absolutely no performance increase when getting more than one core in charge of application processing. The case of the latter was manifold:(a) *false sharing* of a central kernel-level data structure, the file descriptor table, (b) too large critical sections, and (c) blocking synchronization. Although deficiencies like these are known, coming up with proper solutions is challenging. Figure 1 gives an idea why this is the case.

Blocking synchronization does not scale, in contrast to *nonblocking synchronization.* However, the former technique is predominant in Linux as almost all of the various synchronization mechanisms are derived therefrom (cf. Fig. 1). A similar picture can be drawn from FreeBSD. The Linux kernel is pervaded by critical sections in the form of the one exemplified in listing 1. Locating critical sections like these, in order to replace samples of blocking synchronization by nonblocking (lock- or wait-free) equivalents, is not always

[*]This work was supported by DFG, grant no. SCHR 603/7-1

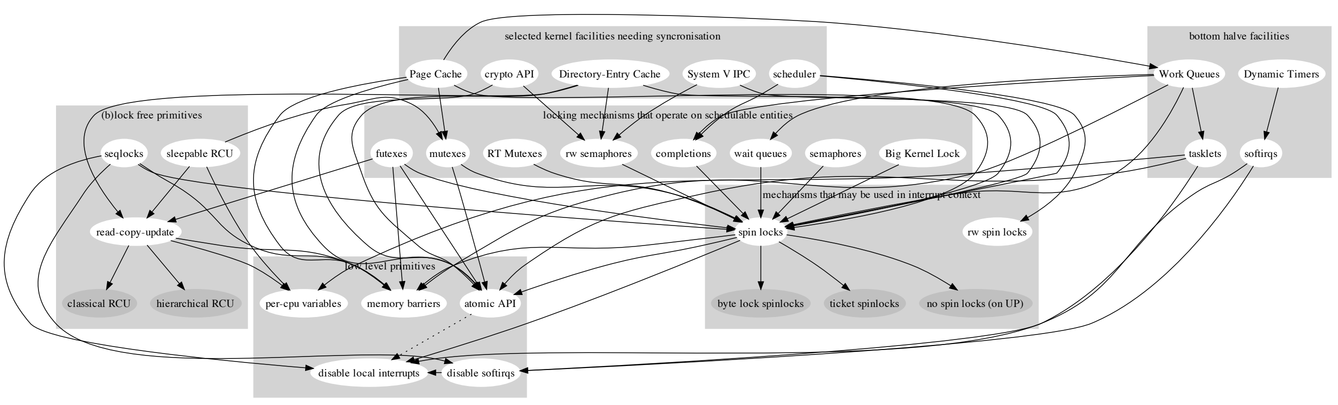

Figure 1: Complexity of design and implementation of synchronization in Linux 2.6.

feasible using pattern matching tools, that is, static program analysis. Even after locating such a section, a complete understanding of the locking protocol in use is needed. When inspecting the sample in listing 1, the use of the *spin-lock* embracing an "atomic" instruction (lines 11 and 13) does not seem necessary, but looking behind the scences as defined by the actual context justifies its use.

Listing 1: Critical section sample (Linux 2.6)

```
1  void enable_mmiotrace(void)
2  {
3      mutex_lock(&mmiotrace_mutex);
4      if (is_enabled())
5          goto out;
6
7      if (nommiotrace)
8          pr_info("MMIO␣tracing␣disabled.\n");
9      kmmio_init();
10     enter_uniprocessor();
11     spin_lock_irq(&trace_lock);
12     atomic_inc(&mmiotrace_enabled);
13     spin_unlock_irq(&trace_lock);
14     pr_info("enabled.\n");
15 out:
16     mutex_unlock(&mmiotrace_mutex);
17 }
```

A major challenge in reengineering operating systems is the reverse engineering part, including the task of understanding existing code. Such code often consists to a great extent of tricky solutions and, for example, also intimates polymorphism in plain C using structures (i.e., records) with function pointers as member variables. Variability is implemented extensively using macros and conditional compilation, or even runtime patching of machine instructions. In the case of abstract interfaces implemented through function pointers, an extraction of the function call stack for code review may even fail. In FreeBSD, for example, a reengineer is faced with a 16-level hierarchy behind a function named lock_mtx once the function pointer embedded in the respective abstract interface has been resolved. In addition to locating the pointer target, the meaning of every single level in this hierarchy needs to be understood in order to cogitate about semantically equivalent nonblocking solutions.

3. SOLUTION STATEMENT

A Linux developer not only has to choose the correct synchronization primitive from a myriad of possibilities (Fig.1), but each of these primitives can be fine-tuned through *configuration options.* We are developing techniques and tools for checking integrity between these options and the corresponding conditional blocks. General basis of this effort is *feature-oriented domain analysis.* First results have shown that keeping the Linux kernel source base consistent with the set of configuration options is hard. We have already found several code blocks that cannot be selected by any valid option combination. In order to confirm our findings we have submitted error corrections to the Linux community, resulting in a dozen patches accepted. Based on these intermediate results we have cause for concern of various inconsistencies regarding the definition/usage of the synchronization primitives. As part of the VAMOS project [4], we aim at further developing tools to detect such "semantic" dependencies to generally aid critical section reengineering.

4. REFERENCES

[1] A. Baumann, P. Birham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, pages 29–44. ACM Press, 2009.

[2] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation*, pages 43–57. USENIX Association, 2008.

[3] H. H. Y. Chang and B. Rosenburg. Experience porting Mach to the RP3 large-scale shared-memory multiprocessor. *FGCS*, 7(2–3):259–267, 1992.

[4] http://www4.cs.fau.de/Research/VAMOS/.

[5] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.