Reproducing System Software Research
A Case Study

Michael Engel
Department of Computer Science
NTNU
https://multicores.org
Motivation

• Reading a paper doesn’t imply you understand it
  • Details might be missing or unclear
  • Details of a paper might be
    • ...unintentionally or intentionally incorrect
    • ...described but never implemented 😞
  • All these things can slip through the usual peer review process for conferences and journals!
• Reproducibility of research results is important
  • Gives confidence that work exists and is useful
  • Can provide a basis to build own research upon
• (Relatively) Recent trend: require reproducibility
  • Delivery of paper + "artifacts" = code, data, …
  • Different levels of artifact evaluation
• What is the situation for systems papers?
# Levels of usefulness

<table>
<thead>
<tr>
<th>Functional</th>
<th>Reusable</th>
<th>Available</th>
<th>Replicated</th>
<th>Reproduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Badge</td>
<td><img src="acm.png" alt="acm" /></td>
<td><img src="acm.png" alt="acm" /></td>
<td><img src="acm.png" alt="acm" /></td>
<td><img src="acm.png" alt="acm" /></td>
</tr>
</tbody>
</table>

**Artifacts documented, consistent, complete, exeriscable, and include appropriate evidence of verification and validation**

- Functional + very carefully documented and well-structured to the extent that reuse and repurposing is facilitated. In particular, norms and standards of the research community for artifacts of this type are strictly adhered to.

- Functional + placed on a publicly accessible archival repository. A DOI or link to this repository along with a unique identifier for the object is provided.

- Available + main results of the paper have been obtained in a subsequent study by a person or team other than the authors, using, in part, artifacts provided by the author.

- Available + the main results of the paper have been independently obtained in a subsequent study by a person or team other than the authors, without the use of author-supplied artifacts.
Example: Persistence from app to hardware

- Persistent operating systems are no new invention
  - "hot" research topic in the 1980s/90s
    - Smalltalk, Lisp (Interlisp, Symbolics), IBM OS/400
    - Eumel/Elan and L3 [1], BirliX [2]

- Implementation of persistence
  - All system state was kept in RAM
  - **Snapshots** generated on non-volatile storage
    - …when shutting down the system
      - crashes ➔ start from initial (boot) state
      - …initiated manually or at regular intervals
    - tradeoff overhead ↔ amount of work lost
What’s the state in 2021?

- Persistent, byte-addressable main memory is available now

- *Can we implement persistent system images on top of persistent main memory?*

- Several *challenges*, e.g.
  - Persistence semantics
  - Ensure consistency
  - Non-persistent state
  - Protection
  - Heap and stack overflows
Persistence challenges: protection

- **Wanted:** protection for small regions of memory
  - e.g. objects on stack and heap
  - Persistent main memory ➔ persistent errors

- Do we really need this?
  - Is language-based protection not safe enough?
  - *Should the operating system trust the compiler?* [3]

Language-based protection has some significant weaknesses: [4]

- the TCB of a system depending on language protection is larger because now we must trust the compilers and code verifiers as well as the "system" TCB objects provided by the system

- language-based protection has its own performance problems and the optimizations to improve performance introduce subtle security flaws
Hardware-based protection today

- Problems with current virtual memory
  - Fixed page size (e.g. 4kB)
  - Trend towards even larger sizes
  - Protection is tied to translation
- Look for existing fine-grained approaches
  - Are we just trying to reinvent the wheel?

- One approach: Liedtke’s *Guarded Page Tables (GPT)* [5]
- GPT properties to investigate for today’s systems
  - Page table depth versus page size?
  - Effects of small pages on TLB miss rate?
  - How can we implement a GPT approach today?
Guarded page tables

- GPTs supplement page table entries (PTEs) by a guard
  - Guard = bit string of variable length

Translation steps:
1. PTE is selected by the highest part of the virtual address
2. Selected entry contains a pointer and the guard $g$
3. If $g$ is prefix of the remaining virtual address
   - Translation continues with remaining postfix
   - ...or terminates with postfix as page offset

\[ v = 0 | 1100101 \ 100101100111 \]
\[ v' = 10 | 0101 \ 100111 \]
\[ v'' = 1 | 0 \ 0111 \]

offset = 0111

data page
How it started…

• Idea: reproduce GPT ideas from Liedtke’s papers [5]
  • Sometimes, papers are as sparse as GPT address spaces 😊

• Can we find details in Liedtke’s PhD thesis [6]?
  • Not available in electronic form
  • A printed copy can be found in the NTNU library!

• Were GPTs ever implemented?
  • Yes, at UNSW in L4/MIPS and L4/Alpha
  • Useful details in the related paper [7] and docs [8]
Finding and compiling L4/MIPS

- Finding the source code
  - not so easy…
  - Finally, found four versions
    - 71, 75, 79, 81

This document is an attempt to document the internal structure of L4 and its operations. It is based on the L4 implementation for the MIPS R4x00 (L4/MIPS), kernel version 79 (February 1999). The document is meant as an aid in teaching operating systems internals, and as a guide for kernel implementors. While the actual code discussed is very specific to the MIPS processor, much of the overall structure and logic of L4 is quite uniform across platforms.

- This used gcc-2.8.1 from 1998…
  - Doesn’t compile on current Linux
  - Set up a Debian 3.0 x86 VM
  - The compiled cross-compiler + L4 tools runs on current Linux!
Hardware platform

- *Problem when reproducing system software*
  - The OS runs directly on the hardware…
  - Special hardware required for implementing GPT
    - Software TLB miss handler instead of hardware PT walk
    - Only implemented on (early) MIPS and Alpha
- What machines did L4/MIPS run on?
  - "The kernel is stable since August 1997, with minor enhancements and bug fixes since. It has been tested on an R4600-based SGI Indy, on the Algorithmics P4000i prototyping board, as well as on the R4700-based U4600 system developed at UNSW as a research and teaching platform." [8]
  - Where can we find specialized 25 year old hardware?
Running L4/MIPS

- NTNU’s "datamuseet" helps: found an SGI Indy!
  - Unfortunately, it has the "wrong" CPU (R5000) [7]:

The kernel code described in this document is for a uniprocessor R4600/R4700 system. There are a number of minor differences between various processors of the R4x00 family. For the purpose of kernel code, no significant differences exist between the R4000 and R10000 will probably run L4/MIPS without major changes. Particularly the R5000’s MMU seems to be similar enough to the R4x00 to allow the code to run virtually unchanged. However, the R5000 and R10000 are multi-issue CPUs, and no attempt has been made in the kernel to
Hardware is hard...

- Emulators are an alternative
  - MAME Indy emulation https://sgi.neocities.org
- MAME runs IRIX… but does not boot L4/MIPS
- Sulima was built to run L4/MIPS – three versions online:
  - sulima-mips-020813, sulima-030910, sulima-src-051124
  - The first one actually works with L4/MIPS!
...how it’s going

- L4/MIPS compiles and can be run in the Sulima emulator
  - Allows \textit{qualitative} analyses
    - e.g. examining page tables structure, TLB content
  - Allows \textit{modifications}

- What’s missing?
  - More precise emulations for \textit{quantitative analyses},
    e.g. timing – Sulima is not cycle-exact, does not emulate the memory hierarchy
  - Application and benchmark code

- Future ideas (for student projects):
  - Run Mungi [10] or some older L4-based example student projects

\textbf{SDIOS06} (T. Bingmann, M. Braun, T. Geiger, A. Maehler - University of Karlsruhe)
This is a toy operating system developed during the “System Design and Implementation” course 2006 at the University of Karlsruhe.

\textbf{SC/OS} (S. Hack, C. Ceelen - University of Karlsruhe)
SC/OS is an experimental multi-server toy operating system using Flick. It was built by two students in the course “System Design and Implementation” in summer 2001.

\textbf{ChacmOS} (A. Haeberlen, C. Schwarz, M. Völp, H. Wenske - University of Karlsruhe)
ChacmOS is an experimental multi-server toy operating system. It was built by four students in the course “System Design and Implementation” in summer 2000.
Takeaways

- Many systems publications from the 1980s/90s are not reproducible
  - No hardware or simulator available
  - No code was published
- The UNSW L4 project already applied good practices
  - Suffered from "bit rot" and unavailability of old web sites
  - Documentation for code in addition to papers [8]
  - Simulator available (no quantitative analyses)
- Compiling the code took some effort (cross-compiler setup)
  - *We need to archive the software source code, compiled binaries and the development environment*
- The OS (source code) alone is not enough
  - Publish binaries of the OS to check if local compilation is equivalent to code used for a publication
  - Also publish application and benchmark code
Future work

• Teach students to work with system code
• Current experiment at NTNU
  • Seminar with system software topics
  • Select small and (relatively) simple papers
• Enable students to understand a paper
  • …by reproducing a central idea from a paper
    • e.g. tickless scheduling, redundancy, new approaches to syscalls, ACLs, …
• Based on MIT’s xv6 OS running on RISC-V
  • qemu or Nezha Allwinner D1 board
  • https://github.com/michaelengel/xv6-d1
  • Alternatives: Raspberry Pi or x86
Surprises…

Chapter 7

Other Stuff (Provisional)

7.1 Scheduling

*Discuss wakeup queue structure.*
Blah blah blah...

"Mut zur Lücke"? [8]

7.2 Interrupts
7.3 Initialisation
7.4 Sigma Zero
References


