Resource Proportional Software Design for Emerging Systems
Resource Proportional Software Design for Emerging Systems

Suparna Bhattacharya
Kanchi Gopinath
Doug Voigt
Contents

Authors xv
Preface xvii
Acknowledgments xxi

I Software Bloat, Lost Throughput, and Wasted Joules 1

1 Introduction 3

1.1 Green Software for the Expanding Digital Universe: Designing with a Sense of Proportion 4
1.2 The Challenge Posed by Emerging Systems: Why Hardware Advancements Are Not Enough 6
  1.2.1 Runtime bloat in framework based software 7
  1.2.2 Software interaction with systems 8
  1.2.3 Impact of non-volatile memory and low-latency fabrics 10
  1.2.4 Large scale connected architectures from edge to cloud 12
  1.2.5 Emerging software models and data centricity 13
1.3 The Heart of the Matter: Why a Plea for Lean Software Is Not Enough 14
  1.3.1 The flexibility, productivity, and efficiency trade-off 14
  1.3.2 Unsustainability of tightly coupled hardware-software abstractions 15
  1.3.3 Traditional performance optimization is not enough 16
  1.3.4 Difficulty in quantifying the opportunity and impact of software bloat reduction 16
1.4 The Resource Proportional Software Design Principle 17
  1.4.1 How to assess the propensity for bloat in a software component? 17
  1.4.2 How to broaden supported features without a runtime overhead? 18
  1.4.3 Can software be designed to cope with changing trade-offs as technology evolves? 19
## Contents

1.4.4 Can we anticipate what proportion of data processed by application is truly useful? .................................................. 20
1.5 Dimensions of Resource Proportional Design ..................... 21
1.5.1 Resource proportional code features ............................ 22
1.5.2 Resource proportional response to technology and system evolution ................................................................. 23
1.5.3 Resource proportional data processing ........................... 25
1.6 Approach in This Book .................................................... 26

2 The Problem of Software Bloat ............................................ 29

2.1 The Notion of Software Bloat ............................................ 31
2.2 Software Bloat: Causes and Consequences ....................... 32
2.2.1 Principal aspects of bloat ........................................... 32
2.2.2 Definitions of software runtime bloat relevant for this book ......................................................................................... 33
2.2.3 Systemic practices in framework based development attributed as causes of bloat ......................................................... 34
2.3 Bloat in Containerized Software ....................................... 37
2.3.1 Application containers vs. virtual machines .................... 37
2.3.2 Container image bloat .................................................. 37
2.3.3 Runtime bloat in serverless computing .......................... 38
2.4 Different Forms of Software Runtime Bloat ....................... 38
2.4.1 Runtime bloat categories in Java applications ................ 38
2.4.2 Relationship between the various runtime manifestations of bloat ................................................................................ 39
2.5 Progress in Bloat Characterization, Measurement, and Mitigation ................................................................. 40
2.5.1 Modeling and measuring bloat ....................................... 40
2.5.2 Mitigating and avoiding bloat ....................................... 44
2.5.2.1 Semi-automated approaches .................................. 44
2.5.2.2 Automated code optimization .................................. 44
2.5.2.3 Can runtime bloat be avoided by construction? .......... 46
2.6 Conclusions ................................................................. 47

3 Does Lean Imply Green? How Bloat in Software Impacts System Power Performance .................................................. 49

3.1 The Effects of Java Runtime Bloat on System Resources ................................................................. 51
3.1.1 Allocation wall effect .................................................. 51
3.1.2 Heap pressure effect ................................................. 52
3.1.3 Object construction computation overhead .................... 52
3.1.4 Influence of system configuration .................................. 52
3.2 Insights from an Experimental Study .................. 53
  3.2.1 Multi-platform experiments and results ............. 54
  3.2.2 Single platform experiment variations: Cache pressure
       and power management .......................... 55
  3.2.3 Key experimental observations .................... 57
3.3 Analyzing the Interplay of Bloat, Energy Proportionality, and
System Bottlenecks .................................. 58
  3.3.1 Power efficiency impact quantified using a simple
       abstract model .................................. 58
  3.3.2 Effect of degrees of energy proportionality ........ 60
  3.3.3 System bottlenecks and bloat: A curious interaction
       3.3.3.1 Bloat at non-bottleneck resource .......... 61
       3.3.3.2 Bloat at bottleneck resource .......... 61
       3.3.3.3 Bloat reduction shifts bottleneck ....... 62
  3.3.4 Summary ........................................ 62
  3.3.5 Model predictions seen in experimental observations 62
3.4 Conclusions ....................................... 64

II The Antidote: Resource Proportional Software Design 65

4 Resource Proportional Software Design Principles to Reduce
Propensity for Bloat 67
  4.1 Insights from Energy Proportional Hardware Design ..... 68
  4.2 Resource Proportional Design of Software Features ..... 68
  4.3 How Software Becomes Non-resource Proportional ..... 70
  4.4 Defining Resource Proportionality with Respect to Feature
       Utilization to Predict Bloat Propensity .......... 74
       4.4.1 Effect of using a generalized component in this scenario
       4.4.2 Weighted RP accounting for scenario distribution ... 76
       4.4.3 Effect of adding features not required for a scenario . 77
       4.4.4 Bloat relative to actual resource consumed by a
       component ........................................ 78
       4.4.5 Computing bloat propensity when $R^{specialized}$ is not
directly available ................................. 78
       4.4.6 Trade-off between feature exploitation and provisioning
       overhead ....................................... 79
       4.4.7 Resource proportionality characteristics ........ 80
       4.4.7.1 Scenarios with monotonically ordered feature
       spaces ........................................ 80
       4.4.7.2 General scenarios (unordered feature spaces) 80
  4.5 Resource Proportional Optimization Control Points for Bloat
       Mitigation ....................................... 81
  4.6 Conclusions ..................................... 83
5 Resource Proportional Design Strategies I: What Component and Tool Developers Can Do

5.1 Strategy 1: Minimize Interactions between Independently Usable Features without Sacrificing Efficient Reuse

5.1.1 Development practice: Abstracting a minimal core of base features – Lessons from the Linux kernel

5.1.2 A formal discipline for labeling feature interactions due to optional features: Insights from FOP (feature oriented programming)

5.1.3 RPD analysis tool: Aid detection of structural interactions using Concern Augmented Program Analysis (CAPA)

5.1.3.1 Computing microslices

5.1.3.2 Computing the microslice interaction graph

5.1.3.3 Computing the Concern Augmented microslice interaction graph

5.1.3.4 Putting it together: The CAPA tool

5.1.3.5 Example: Big endian to little endian conversion

5.1.4 Interactions due to hidden features

5.2 Strategy 2: Reduce Recurring Overheads due to Incidental Sources of Bloat

5.2.1 Object reuse and result caching (amortize data construction overheads)

5.2.2 Adaptive selection and replacement of collection data structures

5.3 Strategy 3: Activate or Deactivate High Overhead Features On-demand

5.3.1 Insights from AOP (aspect oriented programming)

5.3.2 Practical considerations (80-20% rule vs. pure RPD)

5.4 Strategy 4: Design Programming Constructs and Runtimes with Resource Proportionality Awareness

5.4.1 Annotating code with line of sight into sources of overheads

5.4.2 Alternate data and program representations

5.4.2.1 Separating code bloat

5.4.2.2 Separating data structure bloat

5.4.2.3 New programming construct to represent associative pointers

5.4.2.4 Research topic: Content addressable data layout for associative structures

5.5 Summary
6 Resource Proportional Design Strategies II: Refactoring Existing Software for Improved Resource Proportionality 121

6.1 Strategy 1: Whole System Impact Analysis to Identify Candidate Resources and Indicators of Bloat that Are Likely to Matter the Most 122
  6.1.1 Resource utilization, bottleneck analysis and power, performance models 122
  6.1.2 Measure indicators of bloat 124

6.2 Strategy 2: Replacing Entire Components or Features with a Different Implementation 125
  6.2.1 Example: Serialization-Deserialization 125
  6.2.2 Example: Collection replacement 126
  6.2.3 Example: Trimming unused code (bloatware mitigation) 126

6.3 Strategy 3: Reduce Recurring Overheads Due to Incidental Sources of Bloat 126
  6.3.1 Object reuse and memoization 126

6.4 Strategy 4: Refactor Code to Minimize Structural Interactions 127
  6.4.1 Using optional feature indicators 127
  6.4.2 Using concern analysis tools 128

6.5 Summary 128

7 Implications of a Resource Proportional Design 131

7.1 Introduction 131
  7.1.1 Resource proportionality and high level design: Internal vs. external brokering 133
  7.1.2 A simple model for systems resource proportionality 134
    7.1.2.1 A simple regression based model 135
  7.1.3 Steering a system towards RP 136
  7.1.4 Difficulties in realizing $k$-RPD 138
    7.1.4.1 Searching a large RP design space 140
  7.1.5 Summary 141

7.2 RPD over Time 141
  7.2.1 Impact on “optimal” RPDs 141
    7.2.1.1 Impact on “microservices”-based RPDs 142
  7.2.2 RPD in the context of rapid change 143

7.3 RPD and Security: Resource Usage as a Side Channel 145
  7.3.1 RP remediation/countermeasures 148

7.4 RPD and Other Systemwide Concerns 150
  7.4.1 Real time systems 150
  7.4.2 Correctness 152

7.5 Conclusions 154
III  Responding to Emerging Technologies: Designing Resource Proportional Systems 155

8  Resource Proportional Programming for Persistent Memory 157

  8.1 Characteristics of Emerging Persistent Memory Technologies 157
  8.2 System Implications of PM Technology 160
    8.2.1 Data flow implications 160
    8.2.1.1 Marshaling 161
    8.2.1.2 Flushing and fencing 162
    8.2.1.3 Data recoverability 163
    8.2.2 Process flow implications 165
    8.2.2.1 Context switch elimination 165
    8.2.2.2 Perturbation of CPU utilization 167
    8.2.3 Code reuse implications 169
    8.2.3.1 Data access granularity 172
    8.2.3.2 Atomicity 173
  8.3 Storage Stack Bloat in the Dual Stack Scenario 174
    8.3.1 Analytic model of the dual stack scenario 176
    8.3.1.1 PM path 178
    8.3.1.2 RAM disk path 178
    8.3.1.3 Disk path 179
    8.3.1.4 MM disk path 179
    8.3.1.5 Dual stack RP baseline 180
  8.4 Multi-Layer Storage Stack Bloat Related to Atomicity 182
  8.5 Resource Proportional High Availability 184
  8.6 HA and Atomicity Function Deployment Scenarios 186
  8.7 Keeping up with the Evolution of Persistent Memory 188

9  Resource Proportionality in Memory Interconnects 191

  9.1 Characteristics of Memory Interconnects 192
  9.2 Resource Proportionality and the Separation of Media Controllers from Memory Controllers 196
    9.2.1 Cost of asymmetric R/W latency with asynchronous MI 197
  9.3 Efficiency Model of Memory Fabric 201
  9.4 Resource Proportional Capacity Scaling 204
  9.5 PM Related Functionality Placement 206
    9.5.1 PM related functions 206
      9.5.1.1 Multi-phase write 206
      9.5.1.2 Atomic rewrite in place 207
      9.5.1.3 Memory interleave 207
      9.5.1.4 Redundancy 207
    9.5.2 Functionality placement given split vs. monolithic memory controllers 207
9.6 Resource Proportionality and Memory Centric System Architecture .............................................. 211

10 Applying Resource Proportional Design Principles to a Deeply Layered or Complex Software Stack 215

10.1 Introduction .............................................................. 215
10.1.1 Simple examples of RPD in systems design ............... 217
10.1.1.1 Layering costs ............................................. 218
10.1.1.2 Resource rate mismatch costs ....................... 221
10.1.2 Copy elimination in RDMA based storage stacks ... 222
10.1.3 High level RP systems design ................................. 223
10.1.3.1 Proportional design at different levels of the stack ......................................................... 226
10.1.4 RPD by mixing analog and digital components then and now ............................................... 229
10.1.5 Blockchains and the proportional heuristic ............... 231
10.1.6 Anti-RP designs ..................................................... 233
10.1.7 Some newer issues .................................................. 233
10.2 Some High-level Recurring Patterns in RPD .......... 236
10.2.1 Chains ............................................................... 236
10.2.2 Crosslayer optimization ......................................... 237
10.2.3 Memoization, checkpointing, and lazy/deferred designs ......................................................... 238
10.2.4 Managing configuration space ............................... 238
10.2.5 Using virtualization ............................................... 238
10.2.6 Applying the 80% 20% rule where possible ............... 238
10.2.7 Some theoretical insights useful for RPD ............... 239
10.3 General Design Principles and Observations for RPD 240
10.4 What Is Feasible Theoretically? ................................. 245
10.5 Conclusions .............................................................. 248

11 Data Centric Resource Proportional System Design 249

11.1 Characteristics of Data Centric Workloads ................... 250
11.1.1 Data intensive rather than compute intensive ........... 250
11.1.2 Analytics oriented: Emphasis on insight derivation rather than data serving ..................... 251
11.2 Data Centric Frameworks and Stack Evolution ............. 252
11.3 Sources and Impact of Non-resource Proportionality ...... 253
11.3.1 Characteristic resource cost amplifiers in data centric applications ........................................ 253
11.3.1.1 Data movement expense .................................. 254
11.3.1.2 Fault tolerance, check-pointing, and lineage .... 254
Contents

11.3.2 Characteristic sources of overheads in data centric applications ........................................ 255
  11.3.2.1 Impedance mismatch between workload locality patterns and page locality of the system .... 255
  11.3.2.2 Computation on data that does not produce additional insight .............................. 255
  11.3.2.3 Unnecessary synchronization ............................... 257

11.4 Graph Analytics Case Study ................................. 257
  11.4.1 Reducing the size of the input graph data processed .................................................. 259
  11.4.2 Enhancing the proportion of relevant data paged in by the system .................................. 260

11.5 Data Mining Case Study (Map-reduce/Spark) ........... 262
  11.5.1 RPD aware storage systems for insight-centric applications ........................................ 263
    11.5.1.1 Cross-layer insight reuse ................................ 264
    11.5.1.2 Semantic similarity detection ......................... 265
    11.5.1.3 Approximation reuse ................................ 265
    11.5.1.4 Proactive data gradation .......................... 266

11.6 Streaming IoT Analytics Case Study ........................ 266

11.7 Summary .................................................. 269

IV The Road Ahead ................................................. 271

12 Adapting the Systems Software Stack to a Radically Non-Uniform Memory System ....................... 273
  12.1 Resource Proportionality of Memory Resource Allocation .............................................. 274
    12.1.1 The evolution of memory pooling ........................................... 275
    12.1.2 Allocation system model ................................................. 278

12.2 Comparison of Guided vs. Automated Allocation Policies .................................................. 280
  12.2.1 Class of Service driven allocation ........................................ 282
  12.2.2 Automated caching and phase change ........................................ 285

12.3 Resource Proportionality of Waiting, Polling, and Context Switching .................................. 287

12.4 Managing Non-uniformity Using Work Flow Scheduling and Classes of Service ....................... 290

13 Bridging the Gap from What We Know Today to Open Challenges and Research Topics ................. 293
  13.1 Introduction ............................................... 293
  13.2 Approximate Computing ..................................... 294
  13.3 Stateless Designs Revisited, or Reducing “State Spill” .................................................. 298
    13.3.1 “Spill free designs” and failures ........................................ 301
Contents

13.3.2  “Spill free designs” and “serverless” computing/microservices ......... 303
13.3.3  Pointer management ........................................... 303
   13.3.3.1 Memory management, pointer management, and read-copy-update 304
   13.3.3.2 Learned indices ........................................... 306
   13.3.3.3 Systems design vs. ML-based methods ....... 306
13.4 Type Analysis .......................................................... 307
   13.4.1 Information flow ............................................. 309
   13.4.2 Homomorphic computation ................................... 309
   13.4.3 Intermittent computation ..................................... 310
13.5 When Is Resource Proportionality Not Applicable? ............ 310
13.6 Conclusions ............................................................. 312

14 Conclusions .............................................................. 313
   14.1 Applicability to Large Scale Solution Stacks, Rising Complexity, and Software Evolution ............................ 314
   14.2 Resilience to Future Changes as Hardware Technology Advancements Are Raising the Bar ...................... 317
   14.3 RP Impact on Productivity and Other Metrics .................. 318
   14.4 Future Systems Design ........................................... 320

Glossary ................................................................. 323

Bibliography .............................................................. 337

Index ................................................................. 377
Authors

Suparna Bhattacharya is a Distinguished Technologist at Hewlett Packard Enterprise. She has spent most of her career in systems software development and research (5 years at HPE preceded by 21 years at IBM), including several enjoyable years as a well-recognized open source contributor to the Linux kernel. Her recent work advances the use of non-volatile memory technologies and cross-layer optimization in storage and hyper-converged systems for edge to core data services, containers, machine learning, and artificial intelligence. Suparna is an ACM India eminent speaker and has served on program committees for ASPLOS, OOPSLA, MASCOTS, ECOOP, HotStorage, and USENIX FAST. She holds a B.Tech from IIT Kharagpur (1993) and a (late-in-life) PhD with a best thesis award from the Indian Institute of Science (2013).

Kanchi Gopinath is a professor at Indian Institute of Science in the Computer Science and Automation Department. His research interests are primarily in the computer systems area (Operating Systems, Storage Systems, Systems Security, and Systems Verification). He is currently an associate editor of IEEE Computer Society Letters and was earlier an associate editor of ACM Transactions on Storage (2009-2018). His education has been at IIT-Madras (B.Tech’77), University of Wisconsin, Madison (MS’80) and Stanford University (PhD’88). He has also worked at AMD (Sunnyvale) (’80-’82), and as a PostDoc (’88-’89) at Stanford.
Doug Voigt is a retired Distinguished Technologist who worked for HP and HPE storage for his entire 40 year career. He has developed firmware and software for disk controllers, disk arrays, and storage management. He has led HP and HPE virtual array advanced development projects and strategy since 1990. For the last 10 years his focus has been on non-volatile memory systems. Throughout his career Doug was a strong proponent of industry standards. He has been a member the Storage Network Industry Association (SNIA) since 2009. He served on the SNIA board of directors, technical council, and as co-chair of the NVM Programming Technical Working Group. Doug has over 50 patents, mostly in the areas of virtual arrays and persistent memory. Doug’s hobbies include music, photography, and reading science fiction/fantasy.
Preface

This book is an attempt at distilling what good designers instinctively do when confronted with a complex design challenge at the software and systems levels. While it is difficult to state what good designers do concisely, there is one heuristic among the many that stands out and captures this well: the resource proportionality argument (“pay as you go” or “pay for what you get”).¹ C++ design has been stated to be as such by B. Stroustrup and a similar principle appears in many other recent designs such as the concept of energy proportional computing introduced by Barroso and Hölzle. However heuristics can also lead us astray: note the effectiveness of the non-intuitive non-work conserving strategies² that have been employed in some contexts. Thus even when talented designers design systems, multiple heuristics can interact in unusual ways and system behavior can still surprise us all. One such issue is the problem of software bloat, another topic of discussion in this book.

Throughout our careers the authors of this book have been both observers and participants of the incredible progress in computer hardware and software for the past many decades. Collectively we have been involved in many research and industry projects in operating systems, embedded systems, analytics software, software architecture, as well as teaching and mentoring students and younger designers. Given the criticality of the digital world in our everyday lives, we have also observed or participated in interesting social concerns or broad technical community endeavours such as open source development and free software models for the digital economy (as in GNU/Linux systems and many others) as well as sustainable models of digital economy especially from the energy point of view. One example of the rich interplay between technical and social domains is how the Linux kernel evolved in flexibility to support diverse needs because the community of developers and users involved was very diverse in terms of the scenarios they cared about.

Even though the digital economy is constrained by resources in the long term and the advancement of technology generally leads to greater density and higher throughput per unit of energy, the rate of increase in demand in the digital economy has generally continued to exceed the rate of efficiency

¹Some others include the “end to end argument,” “strength of the whole system is that of the weakest link,” “a design is best when nothing can be removed from it,” etc.
²Even when work is available, the system idles for some time to wait for an opportunity to arise to attempt to amortize some cost.
improvements. This is clearly non-sustainable in the long term. Furthermore, the larger economy has often rewarded rapid deployment of inefficient implementations (“roll out updates fast and let customers debug the code”).

There is also the tendency of complexity to increase as the computer industry drives technology to its limits. We have also seen, for example, the advent of persistent memory, the proliferation of machine learning, and the increased importance of distributed data centric solutions. We have repeatedly heard that technology streams in memory and storage have reached their limits only to find that scientists and engineers find ways to circumvent them. Rarely does this result in a net simplification of whole systems. For example, flash technology demands wear leveling which introduces the complexity of managing deferred work such as garbage collection.

We find that Resource Proportional Design (RPD) is an interesting perspective that straddles both the technical and social domains of computing. In this book, we have chosen to use RP as a guiding principle to understand the software and systems problem space with suggestions on ways to navigate this design space. Resource proportionality is an efficiency metric describing how system resources are being used on functionality that is actually needed by the current consumers of the system. Our approach to the book involves the following objectives.

- Encourage the use of Resource Proportional Design (RPD) as a methodology
- Expand on nuances of quantitative RP metrics
- Codify RPD as comprising specific methods for improving RP
- Illustrate the role of RPD in use cases across a wide range of software and hardware scenarios.

The creation of long lasting solutions in the RP design space is primarily the work of researchers and developers who are also the primary audience for this book:

- Software Architecture Researchers and Solution Architects - Evaluate RPD approaches and lay the groundwork for tools and practices that work.
- System Engineers (developers and architects of system software including runtime libraries, data access stacks such as drivers, file systems, databases, and caches, OS’s, and optimizers as well as the underlying hardware) - Create systems that embed RPD and support application RPD through co-design with runtime libraries, data access stacks, OS’s optimizers, and hardware.
- Tool Developers - Create and maintain RPD tools and frameworks that enable developers to efficiently incorporate RP considerations into their designs and implementations.
• Application Developers - Use RPD tools and practices to add this important missing software architecture dimension to their work to make sure that every code feature they develop as programmers is RP by construction.

In addition to researchers and developers, readers may also be interested in specific technology areas such as upper level software, system related software, and distributed applications. Various chapters in the book highlight different technology areas.

The book is divided into four parts with three or four chapters in each. We begin with an overall problem statement and basic description of RPD in Chapter 1. Chapter 2 dives deeper into RPD in the area of software bloat which occurs when resource usage and complexity is out of proportion with utility. Chapter 3 examines the relationship between software bloat and power which is important because power efficiency is the primary goal of RPD. This part of the book should be of general interest to all types of readers. Chapter 1 in particular serves as an important introduction regardless of which subset of the book is of primary interest.

Part 2 focuses on RPD in software development. Chapter 4 elaborates further on RPD principles and definitions while Chapter 5 describes RPD strategies for tool and application developers. Chapters 4 and 5 should be of interest to most readers as they are referenced frequently in subsequent chapters. Chapter 6 speaks to developers responsible for existing (as opposed to new) applications. Chapter 7 follows through with implications of RPD deployments on non-functional requirements such as security.

Part 3 highlights RPD in the context of emerging technologies. Chapter 8 focuses on Persistent Memory while Chapter 9 addresses the related topic of memory interconnects. Chapter 10 addresses RPD in the context of complex, deeply layered software stacks. Chapter 11 applies RPD to data (as opposed to “just” compute) intensive workloads. Reader interest in these chapters is technology specific.

Finally, Part 4 looks forward to managing radically non-uniform systems (Chapter 12), open challenges (Chapter 13), and conclusions (Chapter 14). Part 4 should be of interest to researchers and advanced developers.

We sincerely hope that you, dear reader, take the notion of RPD to heart as an important and interesting perspective. We realize that the current state of RPD tools and frameworks is fragmented, and does not address many of the future looking or advanced concepts described here. Our goal is to further new work in this important area to avoid design bottlenecks and to enable socially responsible and sustainable progress in the digital economy across academia and industry.
Acknowledgments

The perspectives expressed in this book have evolved over decades of lively and enriching discussions with colleagues each of us has worked with throughout our careers. We are deeply grateful to them for shaping our thought process.

Some portions of the book are derived from Suparna’s doctoral dissertation and have appeared in the following papers, which are reused here with permission from the publishers IEEE, ACM, and Springer as appropriate. The thesis was guided by Prof. Kanchi Gopinath from the Indian Institute of Science and Dr. Manish Gupta from IBM. We would like to specially acknowledge them and collaborators Dr. Kathick Rajamani and Dr. Mangala Gowri Nanda for their contributions to these papers:

- Suparna Bhattacharya, Kanchi Gopinath, Karthick Rajamani, and Manish Gupta. Software Bloat and Wasted Joules: Is Modularity a Hurdle to Green Software? in *IEEE Computer*, vol. 44, no. 9, pp. 97-101, Sept. 2011 (the discussion on software bloat in parts of Section 2.1, 2.5)


Acknowledgments

The authors would also like to thank:

- The Indian Institute of Science for providing a good research ambiance
- Milan Shetti, Mark Potter, Joel Lilienkamp, Peter Corbett, and Naresh Shah from Hewlett Packard Enterprise for their support and encouragement in writing this book
- The SNIA for providing a rich and inspiring vendor neutral context for storage technology discussion in pursuit of standardization and education
- Gary Sevitsky, Nick Mitchell, Matt Arnold, Edith Schonberg, and Harry Xu for their work on Java runtime bloat and regular discussions that helped sharpen our understanding of the deep systemic issues that lead to bloat. Also Bob Blainey and members of the IBM Academy of Technology study on “Software Bloat - Causes and Consequences” (2009) for their contributions and rich insights on the problem of software bloat in the context of hardware and software technological trends
- Parthasarathy Ranganathan for insights on requirements-aware energy scale-down and for feedback on our early ideas about resource proportionality of software features
- LakshmiNarasimhan Naveenam for detailed review of Chapter 8
- Muthukumar Murugan and Kalapriya Kannan for their detailed reviews of Chapter 11. Also Sam Adams, Mukthukumar Murugan, Kalapriya Kannan, Madhumita Bhard, and Annmary Justine for collaboration on externally published papers referenced in this chapter
- Aravinda Prasad for feedback on Chapters 7, 10, and 13
- Kushal Arun at the Computer Architecture and Systems Lab, IISc for her help at various times
- Our families for inspiring us and putting up with our absences through the many nights, weekends, holidays, and precious vacation time consumed in writing this book

This is our homage to the Rishi-s (seers) of old, the pioneers and the path makers... Rig Veda 10.14.15.2
Part I

Software Bloat, Lost Throughput, and Wasted Joules
Chapter 1

Introduction

1.1 Green Software for the Expanding Digital Universe: Designing with a Sense of Proportion .............................................. 4
1.2 The Challenge Posed by Emerging Systems: Why Hardware Advancements Are Not Enough ........................................ 6
  1.2.1 Runtime bloat in framework based software .............. 7
  1.2.2 Software interaction with systems ......................... 8
  1.2.3 Impact of non-volatile memory and low-latency fabrics 10
  1.2.4 Large scale connected architectures from edge to cloud 12
  1.2.5 Emerging software models and data centricity .......... 13
1.3 The Heart of the Matter: Why a Plea for Lean Software Is Not Enough ................................................................. 14
  1.3.1 The flexibility, productivity, and efficiency trade-off .... 14
  1.3.2 Unsustainability of tightly coupled hardware-software abstractions ......................................................... 15
  1.3.3 Traditional performance optimization is not enough .... 16
  1.3.4 Difficulty in quantifying the opportunity and impact of software bloat reduction ........................................ 16
1.4 The Resource Proportional Software Design Principle ........ 17
  1.4.1 How to assess the propensity for bloat in a software component? ................................................................. 17
  1.4.2 How to broaden supported features without a runtime overhead? ......................................................... 18
  1.4.3 Can software be designed to cope with changing trade-offs as technology evolves? ................................. 19
  1.4.4 Can we anticipate what proportion of data processed by application is truly useful? .................................. 20
1.5 Dimensions of Resource Proportional Design ................... 21
  1.5.1 Resource proportional code features ...................... 22
  1.5.2 Resource proportional response to technology and system evolution ............................................ 23
  1.5.3 Resource proportional data processing .................... 25
1.6 Approach in This Book ............................................... 26

A smartphone today has a million times the computing power of the systems that NASA used to land on the moon[356] and one hundred thousand times the amount of memory in personal micro-computers we used 30 years ago.
Yet, it does not seem enough to satisfy our needs for long. In cloud-based environments, there is a remarkable growth in the pace of software delivery using continuous integration models where new software is now released every few minutes instead of years. As new applications and data sources proliferate rapidly, we also find that every software update requires more resources than its previous version. Is this as efficient as we can get without slowing down the rate of application innovation or is the resource usage for computing growing out of proportion over the years?

Radical improvements in the ease of gathering data and the artificial intelligence revolution continue to raise the demand for computing resources. The cost of sequencing a human genome has reduced by a factor of a million in less than a decade, out-shadowing Moore’s law[401]. How do we tell if the computing resources we have in any given system are being utilized appropriately for the purpose it serves? If not, then how do we bring in this sense of proportion when designing next generation software and systems solutions?

In this chapter we cover an overview of these questions, why they matter now, our approach to address them, and how the rest of this book will unfold.

1.1 Green Software for the Expanding Digital Universe: Designing with a Sense of Proportion

“Anything that is produced by evolution is bound to be a bit of a mess”

Sydney Brenner, Nobel laureate

In Roger Lewin: Why is Development So Illogical?

Beauty is more important in computing than anywhere else in technology....

Beauty is important in engineering terms because software is so complicated; beauty is the ultimate defense against complexity.

David Gelernter

Machine Beauty: Elegance and the Heart of Technology
Cost, energy efficiency, and performance remain crucial considerations across the entire spectrum of computing systems from the edge to the cloud that collectively serve an ever expanding digital universe. Billions of smartphones, personal devices, and IoT\(^1\) have become indispensable at the edge while data centers gravitate to countries with cold climates\(^2\) and even dive underwater.\(^3\)

The digital expansion is not just a matter of quantity. It is representative of a natural desire for progressively richer and more sophisticated capabilities, as we entrust more and more responsibilities of our lives to the power of software genies residing in the devices we wear or hold in our palms, our homes, our vehicles, our hospitals, our schools, our factories, our farms, spanning both remote edges and the mighty clouds. This unprecedented scale of digitization propels a demand for efficient software and computing systems. Fueled by the constant influx of data to learn from, software is now central to innovation.

To sustain the progressive expansion of software capabilities, the functionality supported by a software system needs to be designed and reused with a sense of proportion about the appropriateness of its resource usage.

However, the inevitable complexity in software stacks (a reflection of the complexity in real world needs) is a barrier to reasoning about efficiency and assessing “appropriateness” with respect to utility. The mass of detail in any software system makes it hard to understand what happens at runtime and how to reuse it proportionally for a given scenario. Building efficient solutions requires some form of elegant co-design across different layers of a stack that contain multiple levels of information transfer (e.g., hardware-software co-optimization in smartphones).\(^4\)

---

1. Internet of Things.
2. E.g., Facebook's data center in Lulea, Sweden and Google's Hamina data centre in Finland (since 2011) or the White data center snowcooling experiment in Japan[11].
3. E.g., Microsoft Project Natick[16].
4. A radical alternative, discussed later in this book, would be to use a deep neural network to infer behavior functions and learn a specialized stack (model) from execution traces of a generalized reusable (but potentially inefficient) implementation. This approach has the drawback of being fragile as it suffers from a loss of explainability, which is problematic from an architect’s point of view.
Analog vs. Digital Design

Analog computing in the ‘50s and ‘60’s provided results at the “speed” of light in some cases but accuracy or controllability was poor. Digital representations and computations based on such representations became necessary to provide some functional guarantees of repeatability and accuracy. However, there are situations where computational or energy requirements are still an issue. Moving from analog to digital fully may be a few orders of magnitude costlier in terms of energy and sometimes also orders of magnitude slower. For example, Spice simulation for circuits is orders of magnitude slower than live electronic speeds.

A system designed with a better sense of proportion about appropriate use of resources may therefore be partly “analog” (for example, to conserve energy) and partly digital (for accuracy). Analog here could refer to systems without quantization, whether electronic, optical, chemical (such as DNA-based computing or storage) or, in the even longer timeframe, quantum-based systems. In communication systems, storage systems, and the like, it is now a widespread practice to use fiber optics that use light (“fast” photons) for transfer of information (“optical”) while using electrons (“slow”) for computation as it needs bistable states (“electronic”). In a recent hybrid design, certain complex computations needed in a digital design are done “simply” optically and without using power, e.g., preprocessing image data with optical filters, resulting in a substantial speedup, after which the rest of the computation is performed electronically[134].

1.2 The Challenge Posed by Emerging Systems: Why Hardware Advancements Are Not Enough

For decades, we have witnessed a near exponential pace of advancements in hardware technologies and systems that has powered this expansion in software capabilities. These advancements have made framework based software development paradigms practical, resulting in order of magnitude improvements in software development productivity through reuse and decoupling from hardware. The cumulative dividend of these hardware developments has made the smartphone a reality and cloud computing affordable. Software defined storage is replacing traditional storage arrays which were built using specialized hardware (ASICs and battery backed NVRAM\textsuperscript{5}). The rise of general purpose GPUs has helped usher in the deep learning revolution

\textsuperscript{5}Non-volatile random access memory.
by unlocking the computational capabilities needed for training multi-layer neural networks.

Any recommendations geared at specific emerging systems, even those based on technologies that are cutting-edge at the time of writing this book, are likely to become obsolete. However, the nature of hardware advancements in recent times is driving a need for changes at many levels in software to unlock the benefits. It is getting increasingly difficult to simply rely on powerful hardware to compensate for inefficiencies that have accumulated in software stacks over previous generations. Adapting to the tension between shifting tradeoffs and opportunities provided by emerging systems is a recurring challenge in software systems design.

This tension is exacerbated by several technology trends embodied in emerging memory centric system architectures [392], [40] which promise new levels of performance and flexibility with lower power consumption, e.g., by leveraging the speed of persistent memory, hardware acceleration, and low-latency fabrics. Such hardware technology trends and operational challenges are now forcing us to pay greater attention to mitigating the impact of any disproportionate resource usage by software on energy consumption and system performance while preserving the undeniable benefits of flexible abstractions.

At the same time it is becoming more and more practical to use machine learning (ML) techniques such as neural networks and alternatives such as approximate computing in response to these challenges. These techniques expend analysis resources to gain runtime efficiency improvements. When is this a good trade-off? Perhaps there are some lessons to be learned from the early days of computation where both analog and digital approaches were in play.

Here are a few examples of the tensions and opportunities we explore in this book:

1.2.1 Runtime bloat in framework based software

The emergence of powerful frameworks for redeployable software has transformed the pace of large scale software development. Framework based software systems are provisioned with a high degree of built-in excess flexibility to support standardization and cope with evolving requirements. An unintended side effect of this trend is that software systems tend to accumulate “runtime bloat” - the resource overhead of excess functions and objects. According to Mitchell, Schonberg and Sevitsky [264], bloat can manifest as a pervasive pattern of excessive memory, processing, and IO overheads in framework based applications. These overheads can cause applications to “miss their scalability goals by an order of magnitude,” resulting in the deployment of additional hardware and complex system infrastructure to meet service level objectives.

\[^{6}\text{We cannot (thankfully) anticipate the future.}\]
Runtime bloat often results from a pile-up of unnecessary consumption of memory and processing resources caused by a lack of effective coupling between layers in a software stack. Specific sources of this, as revealed through research and case studies, include duplication and transformation of data structures across software layers and execution of features in one software layer that are not needed by the layers above. This ineffective coupling is a byproduct of software developer training cornerstones such as encapsulation and code reuse. Thus software stacks grow resource heavy as a side effect of the very paradigm of framework based reuse that revolutionized the pace of information technology. The same flexibility that enables applications to rapidly evolve in scale and function to handle unanticipated changes in demanding environments also tends to encourage the accumulation of excess functionality and data.

Bloat imposes a large and pervasive infrastructure tax which eludes even sophisticated JIT optimizers. This tax consumes excess CPU cycles, clogs memory bandwidth (refer to “Allocation wall” discussion in Chapter 3), increases lock contention, and amplifies memory footprint, IO, and storage costs. The cumulative effect of these power-performance inefficiencies can significantly limit the benefits of emerging systems, especially when adapting to technology advancements that shift tradeoffs.

1.2.2 Software interaction with systems

In general there are multiple ways of implementing a software function, some of which are better suited to particular underlying systems. In many cases, underlying system or technology improvements change the optimal relationship between resources consumed and work accomplished. This raises questions about how software deals with change that originates with underlying systems or technologies in addition to those that arise from functionality required by an application. Often software behavior must change in order to fully realize the potential improvement of a system or technology change.

As an ordinary everyday life example, consider the scenario of walking in the rain to an important meeting. The requirements are to arrive on time and dry. The functional choice to be made is that of rain gear. The system is represented by weather in this example where the unexpected variable is wind. Given the requirements one might choose an umbrella as the solution, ignoring the possibility of wind, that imposes the least transit time overhead. Obviously if wind is a factor the umbrella might be worse than useless relative to both requirements. A raincoat would have been a better choice.

In this example, wind represents a system condition that may not have been accounted for by an otherwise efficient software solution. The ability to adapt the choice of rain gear to wind represents a resource proportional real time response to a system condition. The reader can imagine similar exam-

\footnote{In contrast with localized hotspots.}
ples where choices made with awareness of available technology (e.g., trench coat vs. poncho) could be relevant depending on additional non-functional requirements such as appearance.

More to the point, trade-offs with implications that are inconsequential on one system at one point in time may have dire consequences in a different system or temporal context. Stated differently, a choice that results in an acceptable perturbation from a desired system behavior target in one environment may result in an unacceptable perturbation in another. System behavior, in this case, includes aspects of both performance and resources consumed.

![Diagram of system environments with target and actual behaviors](image)

**FIGURE 1.1**: Perturbation within or beyond acceptable operational bounds.

In Figure 1.1, system environment 1 represents a scenario in which a piece of software runs within acceptable behavior bounds, perhaps represented by a set of service level objectives and a target resource footprint that is nearly optimal. At some point within a conceptual behavioral range there is a target behavior for the function in system environment 1, and an actual behavior point at which a software implementation of a function currently operates. Often, resources consumed or performance are metrics of interest that can be used to create a ratio representing actual behavior relative to target behavior. The acceptable behavior ring represents a range of metric values that is deemed acceptable, perhaps because it is within some bounds of efficiency that accounts for a service level objective that the function must meet. The ratio of actual to target metric values is represented by a distance $k$. 
In system environment 2, a change has occurred that causes the software function to consume more resources or perform more poorly than it did before, even though the implementation has not changed. Thus the behavior of the function in that system has moved a distance of $k'$ from the target. Examples of system environment changes that could cause this might include the following:

(a) Increase in memory pressure causing additional swapping

(b) Degradation in a subsystem such as a network that changes the effectiveness of a retry policy

In the scenario illustrated for system environment 2 the target and acceptable behaviors may not have changed, but the function implementation necessary to achieve them is different. For example, more aggressive garbage collection may be needed to shrink memory footprint, or the choice of network access point may need to change. In the absence of an implementation change, $k'$ in system environment 2 falls outside of acceptable behavior.

In system environment 3, the target behavior and acceptable behavior have changed, perhaps due to new expectations resulting from some underlying change in system implementation or provisioning. If the function implementation is not able to take advantage of this change then the distance from the behavior of the function to the target behavior in that system has changed to $k''$. Examples of system environment changes that could cause this might include the following:

(a) Increase in the amount of memory or the number of cores in the system

(b) Introduction of new types of components such as Persistent Memory (PM) into the system that could assist the function

The scenario of system environment 3 represents the reverse of system environment 2 because the “cheese moved.” The target behavior changed, and moved the acceptable behavior so that $k''$ no longer meets expectations.

These stories illustrate behavior perturbation that originates in system changes rather than variations in the functional requirements of a piece of software. System interaction can create inefficiencies analogous to bloat except that the problem originates outside of the software itself. Let’s look deeper into some specific factors that can perturb system behavior.

1.2.3 Impact of non-volatile memory and low-latency fabrics

The advent of non-volatile memory technologies (also known as storage class memory and henceforth referred to as persistent memory in this book) is driving one such dramatic shift in emerging systems. Traditionally, data used by applications have persisted in high capacity storage systems (based on hard disk technology) that operate at latencies about five orders of magnitude slower than expensive (and volatile) memory systems. Software components at
multiple levels of the stack have therefore typically been designed to operate with the assumption of access to limited amounts of fast memory and large amounts of slow storage.

Figure 1.2 illustrates how this trade-off has shifted over the years. The stacked lines depict the relative latencies of storage, memory interconnect, and software between 2000 and 2020. Solid-state storage technologies such as flash based SSDs reduced this gap by two orders of magnitude. With emerging persistent memory technologies, the gap has shrunk so much that it has created an inflection point where the latencies are close enough to the speed of memory to be accessible using memory semantics (Chapter 8). Further, with the availability of low latency memory interconnect (MI) technology (Chapter 9) it is even possible to access persistent memory attached to a remote node using memory semantics. At these low latencies, path length overheads induced by traditional software stacks may no longer be affordable.

The figure illustrates how software overheads begin to dominate as storage latencies reduce (see “Software considered harmful” discussion by Swanson and Caulfield in [374]). While both storage and network latencies have dropped substantially over the years, software path lengths have actually increased with the rising number of layers present in a typical software defined storage (SDS) stack. The percentage of software contribution is now so high that it
can overshadow the benefits anticipated from switching to persistent memory and low latency interconnects, limiting performance gains. The figure does not include the additional number of application layers beyond the core storage stack that are involved in typical data access operation such as databases and file systems.

Ensuring proportional gains from such significant technology advancements is non-trivial because it requires rethinking assumptions that are deeply embedded in different layers of the software stack.

1.2.4 Large scale connected architectures from edge to cloud

Operating systems and runtimes enable applications to be de-coupled from systems trade-offs. However, emerging computing systems are extending beyond individual (single node) servers to large scale connected infrastructure, where solutions are architected to utilize processing resources distributed in many different ways, e.g., across a cluster, a data center, from mobile elements, across a WAN between IoT edges and a core or across multiple clouds. Examples of such (existing and emerging) systems are very diverse:

- Data analytics clusters (Hadoop, Spark [363])
- High performance computing clusters
- Datacenter as a computer (warehouse scale computers, cloud computing)
- Mobile first internet services
- Scale out storage, object storage
- Edge computing and edge to core analytics
- Distributed deep learning
- Hybrid cloud solutions

In all these cases, it is no longer sufficient to design software stacks with a purely “local” or “single system” scope. The impact of inefficiencies as well as opportunities typically involve a collective view of trade-offs spanning resources from multiple (and potentially heterogeneous) systems and locations. This has necessitated a diffusion of systems software functionality from traditional operating systems (such as resource provisioning, scheduling, fault tolerance) into frameworks that operate at higher levels of the stack. Much innovation in this space has been realized via layers of middleware that step around the gaps between the needs of this emerging application computing environment and the scenarios that established operating system mechanisms such as virtual memory and storage architectures were traditionally designed for.

One implication of this diffusion is duplication of functionality across layers (e.g., a local filesystem and HDFS) and a need to rethink assumptions
and trade-offs about where certain types of functions should be implemented to avoid disproportionate resource usage. For example, in their white paper, “Disks for datacenter” [80] Brewer et al. advocate simplifying individual disk drives by sacrificing a small amount of reliability (e.g., by reducing retries) and instead letting higher level software ensure reliability across a collection of drives, resulting in more consistent performance (reduced tail latency).

In an edge to core analytics workflow, costs and constraints on data that can be moved to the core for analysis and analysis functionality that can be shipped to the edge must be factored in to ensure appropriate resource usage based on an assessment of the utility of computation at each end. Besides data transfer overheads, real time constraints, and security restrictions at the edge, computing resource costs may not be uniform across all the nodes. To compound the situation, software code that is traversed from edge to core to cloud comprises many independently developed components and frameworks with their own CI/CD (continuous integration and deployment) pipelines.

1.2.5 Emerging software models and data centricity

The needs of emerging applications are also driving a shift in software development and deployment models (e.g., cloud native, mobile first, artificial intelligence (AI) driven, anything-as-a-service). Containers and serverless computing (Chapter 2) simplify how software functionality is delivered, deployed, managed, and moved across diverse infrastructure environments. Containers offer the convenience and efficiencies of lightweight virtualization and separation of application data from images which in turn are built as reusable layers. Further, as applications become more data centric and insight driven, building, training, deploying, and updating machine learning models in production across large scale infrastructure is becoming integral to solution workflows and continuous integration and deployment pipelines. Transfer learning techniques that allow pre-trained models to be reused for related tasks are becoming popular especially in deep learning where the amount of training data and computation resources needed for training a model from scratch is huge [2].

Despite the improved efficiencies, designing with a sense of proportion is still essential with emerging software models, which also introduce fresh challenges. Containers can get bloated, microservices have data exchange costs, and much data collected for analysis is often bloated with low value information which can overwhelm storage and processing resources. Proportionality in resource consumption with respect to software utility translates into better economics in terms of lower cloud costs, reduced edge to core constraints, and greater data tiering savings.
1.3 The Heart of the Matter: Why a Plea for Lean Software Is Not Enough

"Software girth has surpassed its functionality largely because hardware advances make this possible. About 25 years ago, an interactive text editor could be designed with as little as 8000 bytes of storage (Modern editors request 100 times that much!). An operating system had to manage with 8000 bytes and a compiler had to fit into 32 Kbytes, whereas their modern descendants require megabytes. Were it not for a thousand times faster hardware, modern software would be utterly unusable."

Niklaus Wirth
"A Plea for Lean Software" [410]

Concern about software code and data bloat is not a new issue - yet for over two decades since the publication of Niklaus Wirth’s article “A Plea for Lean Software” [410], the incidence of bloat has mostly continued unabated. Instead as we observed in the previous section, there are many examples where its proportion has even grown as a consequence of prevalent software and hardware trends.

1.3.1 The flexibility, productivity, and efficiency trade-off

The deep systemic cause of this persistent issue is not one of “bad programmers,” but practical cost-benefit trade-offs that have emerged in reaction to the inherent difficulty in meeting the pressures of simultaneously addressing the need for flexibility, productivity and efficiency\(^8\) of software solutions. Any two of these considerations can typically be achieved at the price of compromising on the third consideration (Figure 1.3). In the case of framework based software the trade-offs have leaned in favor of flexibility and productivity at the cost of efficiency [264].

For example, one explanation for why the problem of software bloat has received very little sustained attention until now is that there have been dramatic improvements in hardware performance over successive CMOS technology generations. Such improvements would tend to obscure the significance

---

\(^8\)Although there are other important non-functional considerations such as security, standardization, RAS (reliability, availability and serviceability), for simplifying the discussion we include those considerations either under the category of flexibility or productivity, as appropriate.
Introduction

Efficiency

Flexibility

Productivity

Framework based software

FIGURE 1.3: Trade-off between software engineering considerations of solution flexibility, runtime efficiency, and development productivity.

of inefficiencies caused by bloat in comparison with the huge gains in software flexibility and development productivity. Efficiency in software is now regaining its importance with the shifting technology trade-offs we discussed in the last section, the gradual reduction in energy efficiency improvements over successive generations of hardware, and the rising operational costs of maintaining overly complex and bulky software stacks. At the same time, both flexibility and productivity continue to be important software engineering considerations.

1.3.2 Unsustainability of tightly coupled hardware-software abstractions

Hence, traditional solutions that tackle code bloat by relying on detailed attention to programming, finely crafted abstractions, tightly coupled hardware software development, or carefully customized feature configuration are unsuitable for the scale and pace of software development required in large flexible IT solutions.

For example, operating system code such as Linux can support a very diverse range of requirements and conditions very efficiently, but require a lot of careful programming and deep expertise, which is a barrier to development productivity as depicted in Figure 1.3. On the other hand, one way to decouple hardware software abstractions from a programmer’s perspective is the use of domain-specific languages or higher level models with automatic
code generators (compilers and runtimes) optimized for a desired hardware system, e.g., using a DSL for graph processing to generate optimized code for GPUs. As illustrated in Figure 1.3, this approach preserves efficiency and productivity at the cost of flexibility.

1.3.3 Traditional performance optimization is not enough

In “Profiling a Warehouse Scale Computer,” Kanev et al. found that data-center workloads show no single hot application to optimize for and no major hotspots within each application. A similar observation has been made in studies of software bloat which finds large applications to be “relatively free of hotspot methods.”[264] In both cases the overheads (bloat or “datacenter tax”) appear to be diffused more pervasively, cutting across methods in an application and across applications in a data center or cutting across machines in a datacenter. Thus traditional performance optimization approaches which are geared at optimizing algorithms in critical methods fail to be useful even though the collective overhead across all methods can lead to disproportionate costs.

1.3.4 Difficulty in quantifying the opportunity and impact of software bloat reduction

Designing software with a sense of proportion requires conducting early cost-benefit assessments to determine where opportunities exist for reducing disproportionately high resource usage. This requires one to have line of sight from software features in code to their relative impact on resource consumption and system power-performance. Distinguishing excess resource utilization from resource utilization due to essential function is non-trivial, as it typically requires a deep knowledge of intended semantics of complex software stacks. Hence software designers only have access to a limited understanding of the circumstances under which bloat arises and the extent to which it affects runtime resource usage and system power-performance. As a result it is difficult to perform a systematic optimization of software to address disproportionate resource consumption.

For example, in order to establish the relationship between bloat and energy efficient design, there is a need to quantitatively relate the overall power-performance impact of bloat in a system to specific sources of bloat and to reason about energy savings expected from alternative strategies for bloat reduction. However, this is a very broad and challenging problem that raises several interesting questions which we attempt to address in this book:

- How can the extent of bloat attributed to a given source be determined?
- How much does bloat matter for power-performance?
• What can be done to mitigate the impact of bloat once we have identified it?

• How do we assess and reduce the propensity for bloat of a given software implementation?

1.4 The Resource Proportional Software Design Principle

Given that simply imploring developers to write efficient code is not particularly effective, is there a principled way to design software with a sense of proportion, in the absence of a free ride on hardware system advancements? In this book, we encourage a shift in software engineering that counters the overhead of deeply layered code by making its resource consumption proportional to situational software utility, defined here as “resource proportional software design.” Resource proportional software design (RPD) introduces a principled approach for developing software components in a way that consciously mitigates the effects of excess resource usage from that contributing to essential function, which would otherwise be highly non-trivial without a deep knowledge of intended semantics of very complex software.

The RPD approach systematically tackles some of the challenges described in previous sections:

1.4.1 How to assess the propensity for bloat in a software component?

If software components are designed to be RP, then it would be easier to extend the methodology to then compose RP stacks using those components. In order to build resource proportional components, we need a way to first assess and quantify whether a given component implementation has propensity for bloat that could cause disproportionate resource consumption with respect to its utility in certain situations. The term “utility” and what is appropriately proportional is subjective and can vary based on deployment environment constraints. We typically assess it specifically for an aspect of interest, e.g.:

1. proportion of code features (functionality) in a program (component) actually utilized or essential for a given scenario

2. proportion of software stack utilized by the underlying hardware operation (e.g., media access latency compared to software level access time observed for data reads and writes)
3. proportion of input data actually utilized or essential for contributing to a useful analysis outcome in data centric software.

A general measure for RP can be devised by comparing the actual resource consumption of a given component under each utilization scenario with the resource consumption of a design of a component that is optimized just for the specific utilized proportion in that scenario. If this ratio is less than \( k \) across all utilization scenarios and inputs, we say that the given component is \( k \)-RPD.

The \( k \)-RPD metric represents an ideal because, in practice, it is unlikely that component designs that are optimized for each utilization level would be available for direct comparison. Further, such comparison may not be directly useful in identifying ways to improve RP of a given component. A more practical approach to assess if the resource usage of a given component is appropriately proportional is to develop line of sight into the resource overhead and power performance cost incurred due to unutilized or non-essential features, capabilities, and data. This overhead is a source of non-resource proportionality. Comparing the overhead with actual resource usage provides an estimate of the percentage bloat, which can then be used to derive estimates of bloat propensity and \( k \). Notice that the overhead and bloat propensity is zero in the ideal perfectly resource proportional component, where \( k = 1 \). If \( k = 2 \), on the other hand, the component is consuming twice the resources of an implementation optimized for the utilization scenario in question.

RP analysis can be broken down into the following questions which will be explored in Chapters 2, 3, 4.

- Why do excess features accumulate in a software stack? (Chapter 2)
- When do excess features lead to runtime bloat? (Chapter 4)
- When is the resource overhead due to bloat most pronounced? (Chapter 4)
- How much does bloat matter for power-performance? (Chapter 3)
- What information is necessary to automatically estimate the extent of bloat? (Chapter 4)

### 1.4.2 How to broaden supported features without a runtime overhead?

A common practice adopted by developers to reduce the burden of excess features in generalized components is to reduce the features or create specialized versions of the component for different scenarios. However, such an approach compromises flexibility, e.g., reduces the situations where the component may be used/reused. This could be visualized as gaps in the RPD utilization scenario space which are not addressed by the component. Such a