Towards a Periodic Table of Computer System Design Principles

JOY ARULRAJ, Georgia Institute of Technology

System design is often taught through domain-specific solutions specific to particular domains, such as databases, operating systems, or computer architecture, each with its own methods and vocabulary. While this diversity is a strength, it can obscure cross-cutting principles that recur across domains. This paper proposes a preliminary "periodic table" of system design principles distilled from several domains in computer systems. The goal is a shared, concise vocabulary that helps students, researchers, and practitioners reason about structure and trade-offs, compare designs across domains, and communicate choices more clearly. For supporting materials and updates, please refer to the repository at: https://github.com/jarulraj/periodic-table.

arXiv:2507.22098v1 [cs.OH] 29 Jul 2025

1 INTRODUCTION

One of the rewards of working in computer systems is the field's sheer diversity, spanning operating systems, databases, computer architecture, distributed systems, programming languages, networking, and more, each with a rich history. For newcomers, it can be challenging to spot connections across different domains due to the diversity of traditions and vocabularies: the same design principle may appear in different guises across domains.

For example, consider the classic paper on database isolation levels by Jim Gray *et al.*. [17]. It offers a careful account of concurrency-control mechanisms and the trade-offs between correctness and performance. Yet without prior exposure to similar issues in operating systems or computer architecture, the ideas can appear narrowly "about databases." In reality, the same design principle, "relaxation of consistency," reappears across systems in different guises, from weakly ordered memory hierarchies to eventual-consistency protocols. When each community uses its own terms and exemplars, newcomers may find it difficult to recognize the underlying design principles. This fragmentation increases cognitive overhead, as the same trade-off must be relearned in each context.

This is a broader pattern: systems research is rich in practical insight but lighter on shared conceptual scaffolding. Across domains, similar challenges recur, managing concurrency, ensuring consistency, and adapting to change, while the framing and vocabulary often differ. As a result, deep connections between seemingly disparate domains can remain relatively obscure.

This article is a small step toward bridging those gaps. Borrowing Mendeleev's metaphor, we present a "periodic table" of recurring system design principles. The goal is not a rigid taxonomy but a working vocabulary: a concise way to annotate papers, lectures, and design documents with the fundamental principles they employ. The aim is to surface structure that already exists in computer systems, so that students can form a more coherent mental map, researchers can situate contributions with precision, and practitioners can discuss design choices across domains with greater clarity.

2 METHODOLOGY

We identified principles by going over 100+ influential papers across operating systems, computer architecture, databases, networking, programming languages, security, and other domains in computer systems. These papers were chosen for historical significance and ongoing relevance, such as classic papers on concurrency control [17] and consensus [25], and more recent work on using machine learning inside systems [22] and designing systems for the cloud [6]. For each paper we asked: what is the underlying high-level design principle? Across domains, independent systems often converged

Group	Focus of the Group
Structure	How to carve and connect parts with clear boundaries and extension points.
Efficiency	Do less work, or do it cheaper by focusing effort where it pays.
Semantics	Specify behavior and interfaces precisely.
Distribution	Coordinate work and data across distributed architectures.
Planning	Select plans automatically from goals, costs, and constraints.
Operability	Observe, adapt, and evolve running systems with minimal disruption.
Reliability	Stay correct under faults, concurrency, and partial failure.
Security	Bound authority and enforce isolation to preserve safety and integrity.

Table 1. Thematic groups of design principles

not on mechanisms but on shared design principles: for example, relaxing consistency to improve performance or lifting abstractions to enhance usability. To qualify as a system design principle, it must satisfy two conditions:

(1) **Abstract.** The principle must be independent of specific technologies or implementations.
(2) **Generality.** The principle must show up across different domains (*e.g.*, database systems, operating systems, programming languages).

This analysis does not aim to catalogue every principle, but to surface many of those with lasting, general-purpose value.

3 THE DESIGN PRINCIPLE TABLE

We have curated a structured set of 40+ general-purpose design principles distilled from the systems literature. As shown in Table 1, they are organised into thematic groups that mirror familiar axes of system design.

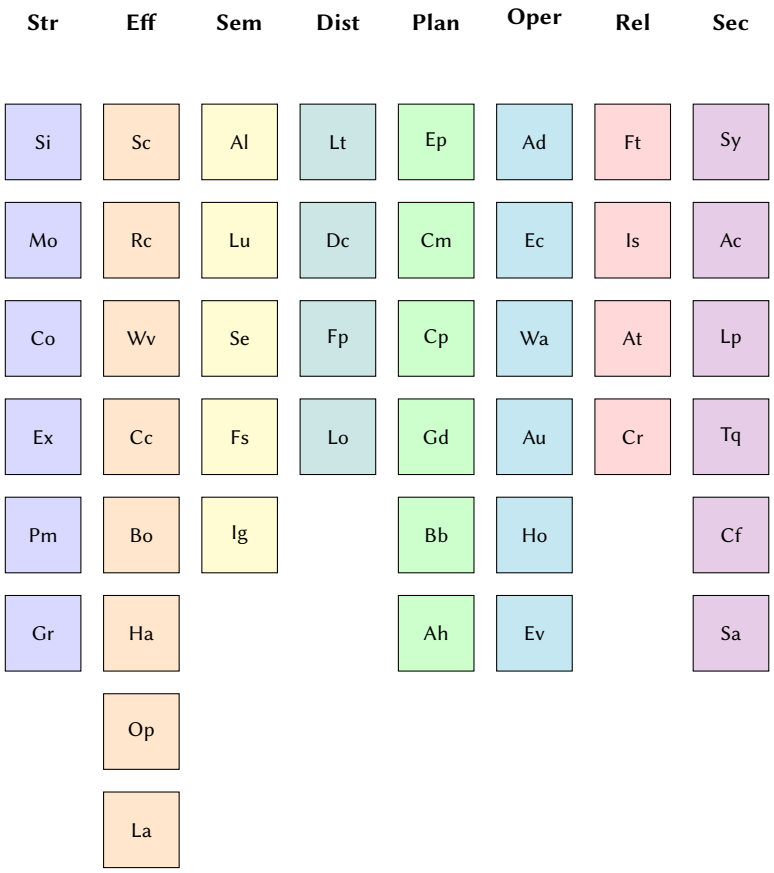
Each principle is tagged with a short symbol (e.g., `Co` for composability, `Op` for optimistic design) for quick reference. We emphasise *design intent* rather than prescribing mechanisms: instead of "use this locking protocol" or "optimise this query plan," the principles state aims such as "preserve correctness under concurrency" or "prioritise the common case," leaving concrete realisations to specific domains.

3.1 Group 1: Structure

Si – Simplicity. Choose the simplest system design that meets current needs; resist complexity, such as additional layers, services, or generality added "just in case", until evidence shows benefit. Avoid premature *architectural* optimisation of the system [23].

Mo – Modularity. Partition the system into cohesive units with minimal interfaces, so that each unit can be reasoned about, replaced, or evolved independently. This principle focuses on decomposition: choosing boundaries to favor clear separation of concerns so that each responsibility sits in one module.
Example: The OSI model decomposes communication into standardised layers with well-defined boundaries that permit independent development and substitution [48].

Co – Composability. Design components that can be safely and flexibly recombined; rely on explicit contracts and type-constrained interfaces so that every legal composition remains correct, letting components be assembled like interchangeable bricks. Unlike modularity, this principle focuses on re-composition: making sure the components can be combined safely and flexibly.



Example: Unix programs (e.g., `grep`, `sort`, `uniq`) read from stdin and write to stdout, letting the user compose complex text processing pipelines [41]

Ex – Extensibility. Design systems to allow safe user-defined extensions, such as plug-ins, without requiring changes to the system core. When extensions come from untrusted parties, isolate them through sandboxing to preserve safety.

Example: Unix also illustrates extensibility: new programs can be added by the user without kernel changes [41].

Pm – Policy/Mechanism Separation. Separate what should be done (policy) from how it is carried out (mechanism) by exposing a common interface through which multiple policies can plug into the same mechanism.

Example: Hydra has a kernel of generic mechanisms (scheduling, paging, protection) and moved resource-allocation policies to user-level modules [32].

Gr – Generalized Design. Design a single core with explicit variation points like types, knobs, or plug-ins, so that it can serve many use cases without duplication, but specialise when doing so yields meaningful gains in performance, accuracy, or clarity.

Example: The C++ Standard Template Library is a collection of containers, iterators, and algorithms parameterized by templates [45]. Postgres allows users to add types and operators to the core database system [46].

Pd – Probabilistic Design. Introduce controlled randomness to gain efficiency, scalability, or simplicity while accepting a small, quantified risk of error or loss.
Example: Routers treat queue length as a probability signal: as the queue grows, they drop incoming packets with increasing probability, proactively signalling congestion [13].

3.2 Group 2: Efficiency

Sc – Scalability. Design the system to handle growth in data, traffic, or nodes with near-linear cost or latency.
Example: MapReduce scales across nodes by dividing work into parallel tasks and aggregating results with minimal coordination [10].

Rc – Reuse of Computation. Avoid redundant work by caching, materializing intermediate results (e.g., indexes), or incrementally updating outputs across repeated or slightly modified inputs, saving computation.
Example: A B+tree reuses its sorted key order: lookups follow the existing search path instead of rescanning the entire data set each time, thereby reusing computation [2].

Wv – Work Avoidance. Skip computation that would not alter externally observable results. Examples include lazy evaluation and predicate short-circuiting.
Example: Lazy evaluation defers work until a value is demanded, eliminating useless computation [19].

Cc – Common-Case Specialization. Detect the execution paths or data items that dominate run-time ("hot spots") and create a streamlined fast path just for them, while a slower, general path still handles every case correctly.
Example: Caching the target method for the receiver class on the first call, so that subsequent calls on that common receiver hit the fast path; uncommon classes fall back to the full method-lookup routine [5].

Bo – Bottleneck-Oriented Optimisation. Profile end-to-end performance, locate the tightest resource constraint, and focus improvement effort there until another stage becomes the limiter.
Example: Rare 99^{th}-percentile stragglers bottleneck latency, and replicated requests help cut tail response times [9].

Ha – Hardware-Aware Design. Shape algorithms and data structures to the latency, bandwidth, parallelism, and persistence properties of underlying hardware (e.g., cache hierarchy, NUMA, SSDs, GPUs).
Example: BLAS defines cache- and vector-tuned kernels so linear-algebra code exploits hardware efficiently [31].

Op – Optimistic Design. Proceed as if the common case will succeed, skipping coordination, and rely on a (possibly expensive) recovery path only when that assumption proves wrong.
Example: Optimistic Concurrency Control runs transactions lock-free, then validates at commit and rolls back only when a conflict is detected [24].

La – Learned Approximation. Replace hand-crafted algorithms with models trained on data, trading bounded inaccuracy for efficiency or flexibility.
Example: The perceptron branch predictor learns weights online to forecast branch outcomes, outperforming fixed two-bit counters without enlarging the table [22].

3.3 Group 3: Semantics

Al – Abstraction Lifting. Wrap low-level operations behind a higher-level interface or domain-specific language that expresses *intent* rather than steps. This enables internal optimization and also allows a single definition to target diverse back-ends.

Example: SQL queries declare the result to retrieve; the DBMS chooses access paths, join orders, and physical operators automatically [44].

Lu – Language Homogeneity. Adopt a single, well-specified intermediate representation (or language) across core components and extensions so semantics align, tools compose, and cross-layer optimisations and reuse happen with minimal effort.

Example: LLVM exposes a typed, SSA-based IR that many front ends target and many back ends share, enabling cross-language optimisation and reuse of the same middle-end passes [30].

Se – Semantically Explicit Interfaces. Specify an interface precisely (covering effect visibility, ordering, durability, etc.) so that users can reason about a call's true externally observable state without guessing about hidden buffering or replication.

Example: SQL isolation levels specify precise anomaly semantics and make visibility guarantees explicit [3].

Fs – Formal Specification. Describe system behaviour using mathematical models or logic to support rigorous reasoning, verification, or synthesis. Mechanisms for realizing this principle include temporal logic, state machines, and other formalisms that make system properties analyzable.

Example: TLA+ shows how to specify and check systems using logic and set theory to catch design errors before coding. [27].

Ig – Invariant-Guided Transformation. Use formally stated invariants to drive safe refactoring, optimisation, or reconfiguration.

Example: In compilers, SSA treats "one definition per name" as an IR invariant; passes rewrite code while preserving semantics and then re-establish SSA [8]. In query optimisers, relational-algebra equivalences (e.g., selection/projection pushdown) preserve result semantics [44].

3.4 Group 4: Distribution

Lt – Location Transparency. Hide the physical whereabouts of resources so clients interact via uniform names or handles.

Example: Programs can call remote procedures as if they were local, masking host location [4].

Dc – Decentralised Control. Distribute decision-making among many nodes to avoid single points of failure or bottlenecks.

Example: Dynamo partitions data via consistent hashing and uses gossip-based membership, avoiding any central coordinator [12].

Fp – Function Placement. Place functionality where the necessary context and resources exist to achieve correctness and efficiency, avoiding redundant work elsewhere.

Example: The end-to-end argument shows that functions like reliability checks achieve correctness only at the endpoints [42].

Lo – Locality of Reference. Place related data and operations close together in time and space to preserve access patterns and minimize separation between computation and state.

Example: The working-set model formalises temporal locality to keep hot pages in memory [11].

3.5 Group 5: Planning

Ep – Equivalence-based Planning. Apply algebraic/logic rewrite rules over a common IR that preserve semantic equivalence; defer final choice to later cost/constraint stages.

Example: Starburst's rule-based rewrite system applies relational equivalences (e.g., predicate pushdown) to generate logically equivalent queries [39].

Cm – Cost-based Planning. When a system must choose among alternative designs, configurations, or execution strategies, use a cost model to guide the search toward low-cost solutions (energy, money, e.t.c.) without needing to enumerate the full space.

Example: The Selinger query optimizer selects the lowest-cost plan under a cost model [44].

Cp – Constraint-based Planning. Encode decisions and hard or soft constraints and rely on a solver (ILP/SMT e.t.c.) to find a feasible or optimal assignment.

Example: Quincy formulates cluster scheduling as a min-cost flow with locality and fairness constraints and solves it to obtain an assignment [21].

Gd – Goal-Directed Planning. Accept a declarative description of the desired end-state and automatically synthesise a concrete sequence of operations to reach it, shielding the user from implementation details.

Example: The Cascades query optimizer turns an SQL query (the goal) into an executable plan via rule-based transformation and cost-guided search [14].

Bb – Black-Box Tuning. When analytic cost models are not available, search the plan/configuration space by *measuring* candidates on the target system, iteratively choosing better ones (e.g., heuristic or Bayesian search), and caching the winner.

Example: ATLAS empirically times candidate BLAS kernel configurations on the target CPU and fixes the best-performing parameters, without an analytic cost model [47].

Ah – Advisory Hinting. Provide non-binding hints that systems may exploit to improve performance, without changing correctness or requiring enforcement.

Example: Lampson advocates optional "hints" that help performance but must not affect correctness if ignored [29].

3.6 Group 6: Operability

Ad – Adaptive Processing. Monitor runtime conditions and automatically adjust parameters or strategy.

Example: Eddies continuously reorder query operators at runtime based on feedback, adapting without stopping execution [1].

Ec – Elasticity. Automatically adjust resource allocation in response to shifting demand and cost goals Examples include predictive autoscaling and load shaping.

Example: Chase *et al.* dynamically provision servers based on load and utility, exemplifying elastic resource management [6].

Wa – Workload-Aware Optimisation. Continuously observe workload shape (skew, locality, access frequency e.t.c.), and adapt data layouts, algorithm choices, or resource allocations to match current patterns.

Example: Database "cracking" incrementally reorganises column data based on query predicates, adapting the data layout continuously to the observed workload [20].

Au – Automation and Autonomy. Let the system perform routine or reactive tasks without human intervention, often by learning from traces or user-provided examples.
Example: AutoAdmin automatically recommends indexes/materialized views from workload traces [7]. Programming-by-example systems automate tasks by generalizing from a few user-provided examples [33].

Ho – Human Observability. Expose internal state of the system, like metrics, traces, plans, to make the system intentionally transparent; that transparency improves observability, debugging, introspection, and control.
Example: Paxson's end-to-end Internet packet dynamics analysis demonstrates how rich measurement and tracing enable informed debugging and tuning. [37]

Ev – Evolvability. Design so the system can change with minimal downtime or rewrites and do so without breaking external contracts or observable behaviour for existing clients. Unlike extensibility that lets outsiders add new behavior via defined hook points without touching the core, evolvability lets the system's internals change over time without breaking existing external contracts.
Example: Parnas presents how a modular design makes system easier to extend without disruptive rewrites [36].

3.7 Group 7: Reliability

Ft – Fault Tolerance. Design the system to continue operating, perhaps in degraded form, despite component failures.
Example: Gray's analysis of why computers stop shows that replication and automatic restart let services keep running through hardware and software faults [15].

Is – Isolation for Correctness. Prevent unintended interference among components so local reasoning remains valid.
Example: Two-phase row-level locking stops one transaction from reading or overwriting another's uncommitted data, preserving isolation guarantees [16].

At – Atomic Execution. Group multiple operations so they appear indivisible, either all take effect or none do.
Example: With Transactional Memory, memory operations inside a transaction speculatively execute, then commit atomically; if any conflict or fault occurs, the entire block aborts and leaves no partial state [18].

Cr – Consistency Relaxation. Deliberately relax strong consistency or ordering constraints, but only within documented bounds, to improve performance, availability, or concurrency.
Example: Bayou lets mobile clients update replicas while disconnected, guaranteeing eventual convergence when replicas reconnect, trading strict consistency for offline availability [38].

3.8 Group 8: Security

Sy – Security via Isolation. Enforce strong boundaries so faults or hostile code cannot affect other components.
Example: A correct virtual machine monitor presents each guest with a complete, isolated machine and intercepts privileged operations, preventing one guest from compromising others or the host [40].

Ac – Access Control and Auditing. Define permissions *and* log every access for accountability.
Example: Lampson's taxonomy of access-control lists, capabilities, and audit trails underpins modern security mechanisms [28].

Lp – Least Privilege. Grant only minimal authority needed for a task, shrinking the blast radius.
Example: The post-mortem on the 1988 Internet Worm shows how excess privilege let the worm spread and spurred widespread adoption of least-privilege daemons [35].

Tq – Trust via Quorum. Rely on agreement from multiple, independent participants rather than a single authority.
Example: Paxos algorithm replicates state across a majority quorum so the service stays correct even if minority nodes crash or act maliciously [26].

Cf – Conservative Defaults. Ship with restrictive, safe settings; let experts opt-in to riskier, faster modes.
Example: With a "default no-access" policy, every protection mechanism should allow access only when explicitly granted [43].

Sa – Safety by Construction. Structure code or data so entire classes of errors become impossible rather than merely detected.
Example: Rust's ownership and borrow checker prevent data races and dangling pointers at compile time [34].

4 CASE STUDY

To illustrate how multiple design principles intersect in practice, consider the mapping from logical to physical operator plans in a relational database system. The database system translates declarative intent into executable steps (Policy/Mechanism Separation). SQL expresses the "what" (Abstraction Lifting) with precise semantics (Semantically Explicit Interfaces). The optimizer first rewrites the query using algebraic equivalences (Equivalence-Based Planning). It then chooses concrete physical operators using a cost model (Cost-based Planning). Physical operators are often optimized for underlying hardware features (Hardware-Aware Design).

Predicate-pushdown illustrates Work Avoidance , while indexes enable Reuse of Computation . Advisory Hints can guide the optimizer, and newer database systems add runtime re-optimization (Adaptive Processing), learned models (Learned Approximation), and sampling (Probabilistic Design). Thus, logical-to-physical operator mapping in database systems exemplifies how several design principles come together to efficiently process declarative SQL queries.

5 LIMITATIONS

Any attempt to organise a field as broad as computer systems involves trade-offs. This table is not a checklist or a universal theory; it is a shared vocabulary that highlights recurring principles and encourages structural reflection. That said, there are several limitations:

- **Orthogonality.** Principles can overlap, reinforce, or partially conflict; design is about balancing such tensions.
- **Subjectivity and granularity.** Deriving and mapping principles involves judgement; boundaries are fuzzy and different readers may tag the same system differently or interpret the same principle differently.
- **Not a formal taxonomy.** This is not a complete or minimal set of design principles. There is no attempt to derive the principles from a minimal core.

Ultimately, this table is a means to help students see recurring design principles more clearly, to assist system designers in communicating tradeoffs more precisely, and to help researchers recognize where their ideas fit into the broader landscape of system design.

6 CONCLUSION

System design spans diverse domains and vocabularies, which can make shared discussion harder. We inherit mechanisms, study trade-offs, and build intuitions, yet concise terms for the underlying ideas are not always available. The "periodic table" of design principles offered here aims to provide a modest common language, naming recurring ideas so they are easier to teach, compare, and build upon.

REFERENCES

[1] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*.

[2] Rudolf Bayer and Edward McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* (1972).

[3] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*.

[4] Andrew D. Birrell and Bruce J. Nelson. 1984. Implementing Remote Procedure Calls. *ACM TOCS* (1984).

[5] Craig Chambers and David Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, a Dynamically Typed Object-Oriented Programming Language. In *PLDI*.

[6] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. 2001. Managing Energy and Server Resources in Hosting Centers. In *SOSP*. Banff, Canada.

[7] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* (1991).

[9] Jeff Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* (2013).

[10] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*.

[11] Peter J. Denning. 1968. The Working Set Model for Program Behavior. *Commun. ACM* (1968).

[12] Giuseppe DeCandia et al. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*.

[13] Sally Floyd and Van Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. In *SIGCOMM*.

[14] Goetz Graefe. 1995. *The Cascades Framework for Query Optimisation.* Technical Report HPL-95-18. Hewlett–Packard Labs.

[15] Jim Gray. 1986. Why Do Computers Stop and What Can Be Done About It? *Tandem Technical Report* (1986).

[16] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann.

[17] J. N. Gray, R. A. Lorie, and G. R. Putzolu. 1975. Granularity of locks in a shared data base. In *VLDB*.

[18] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*. ACM.

[19] John Hughes. 1990. Why Functional Programming Matters. In *Research Topics in Functional Programming*. Addison-Wesley.

[20] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*.

[21] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Christopher Schmitz, and Alexander Totok. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*.

[22] Daniel A. Jiménez and Calvin Lin. 2001. Dynamic Branch Prediction with Perceptrons. In *HPCA*. 197–206.

[23] Donald E. Knuth. 1974. Structured Programming with go to Statements. *Comput. Surveys* (1974).

[24] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6, 2 (1981), 213–226. doi:10.1145/319566.319567

[25] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* (1998).

[26] Leslie Lamport. 1998. The Part-Time Parliament. *ACM TOCS* (1998).

[27] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley.

[28] Butler W. Lampson. 1974. Protection. *ACM Operating Systems Review* (1974).

[29] Butler W. Lampson. 1983. Hints for Computer System Design. *ACM Operating Systems Review* (1983).

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*.

[31] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Software* (1979).

[32] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. 1975. Policy/Mechanism Separation in Hydra. In *SOSP*.

[33] Henry Lieberman. 2001. *Your Wish is My Command: Programming by Example.* Morgan Kaufmann.

[34] Nicholas D. Matsakis and Felix Klock. 2014. The Rust Language. In *ACM SIGAda*.

[35] Robert T. Morris. 1989. A Tour of the Worm. *USENIX* (1989).

[36] David L. Parnas. 1979. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering* (1979).

[37] Vern Paxson. 1999. End-to-End Internet Packet Dynamics. *IEEE/ACM Transactions on Networking* (1999).

[38] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. 1997. Flexible Update Propagation for Weakly Consistent Replication. In *SOSP*.

[39] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*.

[40] Gerald J. Popek and Robert P. Goldberg. 1974. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM* (1974).

[41] Dennis M. Ritchie and Ken Thompson. 1974. The UNIX Time-Sharing System. *Commun. ACM* (1974).

[42] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems* (1984).

[43] Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* (1975).

[44] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*.

[45] Alexander A. Stepanov and Meng Lee. 1994. *The Standard Template Library.* Technical Report. Hewlett-Packard Laboratories. https://stepanovpapers.com/STL/DOC.PDF

[46] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES. In *SIGMOD*.

[47] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *SC*.

[48] Hubert Zimmermann. 1980. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications* (1980).