Programming Languages and Systems Literature Review

Emily Herbert

November 22, 2019

Contents

Chapter 1

Introduction

Accumulation of reviews of programming languages and systems papers. This endeavor is for my own personal use, but I figured eh, open science is a good thing. So, a lot of what's written here may just be half-formed ideas, or notes that really only make sense to me.

Chapter 2

Functional Programming

2.1 An Introduction to the Theory of Lists [6]

Formal definitions for lists and list operations. Discusses common list operations (map, filter, reduce, etc.) and offers best use cases. Provides examples of multiple interacting operations and operation equivalences.

- What are infinite lists?

2.2 Functional Pearl: The Zipper [16]

Describes a data structure that is akin to a zipper, for use in situations in which trees need to be modified non-destructively. Handles can be on particular elements of the tree, where locations hold the downward current subtree and the upward path. Functions are given for navigation left, right, up, and down, retrieving the nth element, changing an element in place, inserting elements left, right, and up, and deleting elements. A memoization approach is suggested, where ”scars” hold tree structure for frequently visited elements.

- Scala implementation at https://github.com/stanch/zipper.
- Binary trees are shown, but binary tree example doesn't allow for any data stored in the tree?

Chapter 3

Object Orientated Programming

3.1 The Expression Problem [33]

Proposes a solution to the expression problem - ”goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code and while retaining static type safety.” Solution in GJ creates a series of types that are dependent on one another, and relies on Interfaces for expression evaluation. Type variables can be indexed by any inner class defined in the variable’s bound.

- I’m not sure I know what ”indexing a type variable” is ?

Chapter 4

Machine Learning

4.1 Typesafe Abstractions for Tensor Operations [8]

Typesafe tensors for tensor operations and machine learning. Tensors created with two types, the type (primitives) of the elements, and the phantom types used to label dimensions. This allows operations to be checked at compile time to ensure that all tensor operations are valid. Typesafe tensors are differentiable and can be type-checked. Computation graphs are also typed, and inputs/outputs are also type-checked. Examples given for tensor operations — matrix multiplication, tensor contraction. Examples given for NN layers - FC, conv, recurrent (recursive).

Chapter 5

Simulation

5.1 Multi-agent System Simulation in Scala: An Evaluation of Actors for Parallel Simulation [32]

Proposes multi-agent simulation (MAS) in Scala that utilizes the Actor framework. Satisfies motivations for removing possibility of race conditions. Provides benchmarks comparing Actor framework to threads - Actor framework displays slower results, but the prospect of safer simulations justifies the slow-down.

- This design is outdated. What would a modern design look like?
- Would there be a motivation for simulation for RL agents?
- Scala's delimited continuations library.

5.2 Using Domain Specific Languages for Modeling and Simulation [23]

Gives an overview of the Scalation simulation DSL, implemented in Scala. Discusses Scalation motivations and realizations, and how they make it a good fit for Scala. Mentions use of Actors for parallel simulation, but does not discuss further. System built on event graphs, "where the nodes represent types of events and the edges represent causal links between the events."

- https://github.com/scalation/scalation
- Scala Parser Combinator Library.
- What is the difference between a DSL and a library/ package?
- Hyrid Functional Petri Nets.
- Fortress.

Chapter 6

Software

6.1 Calculation View: multiple-representation editing in spreadsheets [31]

Presents "Calculation View" (CV), a novel alternate view in Microsoft Excel. Allows users to edit spreadsheets in a more high-level way. Introduces ranges, named ranges, pseudocells, and a block detection algorithm. Details further work to expand CV view, including text interface and expanded features.

- Can CV store intermediate values that the user might not need represented on the sheet?
- Can CV be used on its own? Is there a use case for CV being used on its own?

6.2 Elastic Sheet-Defined Functions: Generalising Spreadsheet Functions to Variable-Size Input Arrays [21]

Defines formal syntax and semantics for generalising spreadsheet functions to variable-size input arrays (ranges). Outlines algorithm for identifying generalizations and interpreting most likely generalizations given multiple options.

Chapter 7

Systems History

7.1 Timeline of Papers

1952	The Education of a Computer [15]
1957	The FORTRAN Automatic Coding System [3]
1959	Recursive Functions of Symbolic Expressions and Their Computation by Machine [20]
1964	Architecture of the IBM System/ 360 [2]
1965	Cramming More Components onto Integrated Circuits [24]
1967	Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities [1]
1978	The UNIX Time- Sharing System [30]
1979	Experience with Process and Monitors in Mesa [18]
1980	The Case for the Reduced Instruction Set Computer [27]
	Comments on the Case for RISC [9]
1982	Gprof: A call graph execution profiler [12]
1983	Hints for Computer System Design [19]
1988	Garbage Collection in an Uncooperative Environment [7]
	A Case for Redundant Arrays of Inexpensive Disks (RAID) [28]
1989	Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines [17]
1990	An Empirical Study of the Reliability of UNIX Utilities [22]
2004	Enhancing Server Availability and Security Through Failure-Oblivious Computing [29]
2005	DART: Directed Automated Random Testing [11]
2006	DieHard: Probabilistic Memory Safety for Unsafe Languages [4]
2007	Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation [26]
2008	Amdahls Law in the Multicore Era [14]
	Bitcoin: A Peer-to-Peer Electronic Cash System [25]
2010	A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World [5]
2015	Coz: Finding Code that Counts with Causal Profiling [10]
2017	Bringing the Web Up to Speed with WebAssembly [13]

7.2 Template to Copy

Summary

Context

Discussion Points

Significance

7.3 The Education of a Computer [15]

Summary This paper goes through iterations of "educating" a computer, to where the mathematician using a computer gets the boot, the computer becomes the mathematician, and the programmer becomes an integral part of the computer. A subroutine is a function-like that performs some computation. It has an entry line, exit line, result line, argument lines, and routine lines. Different procedures interact with different lines of the subroutines. The computer is given a bunch of smaller mathematical subroutines that the programmer can use to help construct their programs. It is hypothesised that more "subroutines" could be developed and combined. It is unclear to me what the correlation is between subroutines and modern day functions.

Context I think this work is difficult for modern day programmers to understand/ fathom (myself included). It is difficult to think of what may lie between assembly-style jumps and Haskell, for example, which is the space that this paper explores.

Discussion Points

1. Page 245 discusses turning using programs that contain subroutines as a subroutines itself, which I would consider a profound observation. Yet the "conclusion" focuses just on the arithmetic type mathematical advanecs. Hmm.
2. The proposed UNIVAC is claimed to "not forget" and "not make mistakes." We all know that that is not the case.

7.4 The FORTRAN Automatic Coding System [3]

Summary The paper proposes FORTRAN, a language resulting from a need to reduce computing overhead. The authors report that the goal of FORTRAN is for programmers to be able to use a concise language to specify procedures, rather than writing the granual 704 code directly. FORTRAN achieves this by introducing both a high level language a mechanism of automatic translation to produce optimized 704 code. The language presented uses statements that modern day programmers would recognize - functions, loops, print, etc.

The technical contributions are quite great. The FORTRAN language itself introduces new statements that programmers can use to write pseudo 704 code, with each FORTRAN statement translating to 4 to 20 704 statements. This greatly reduces the time and effort required to write programs and debug. The FORTRAN translator automatically translates FORTRAN statements to 704 statements, and while doing so it makes a number of optimizations that allow for efficient 704 code to be produces. This 704 code may even be more efficient than what a programmer may have wrote directly, and the translator is able to detect patterns and optimizations for different high level ideas. The translator also is able to catch a number of errors and help the programmer debug.

Context This paper's goal was to develop a tool to make programming easier, and it does so by developing a high level representation of frequently used code and introducing a method of translating that high level representation to optimized low level code automatically. Really, that goal hasn't changed much since this time. Obviously languages and toolchains have evolved, but most modern languages employ this automatic translation/ optimization process, and for good reason. It is likely that this paper laid a large portion of the foundation for modern languages.

Discussion Points

1. Not a discussion question, but I belive last class it was said that FORTRAN was originally intended for researchers. In retrospect it is easy to see why FORTRAN-like languages would be appealing for most programmers, not just researchers, but I am interested in how this transition arose at the time.

2. Hmm, I actually keep getting kind of confused by the use of "subroutine" in each of these papers, how is a subroutine different than a function? I read the other paper for this week first, and I was mentally translating "subroutine" to "function", but this paper makes a distinction. Can we clarify the difference?

Significance This paper was definitely significant, as it likely paved the way for most modern languages. Not only does it introduce two major components, the lanaguage representation and the compiler, but it also presents a very compelling case for why each of these is useful. This paper likely introduced a lot of motivation for creating tools that could help programmers be more efficient and more accurate.

7.5 Recursive Functions of Symbolic Expressions and Their Computation by Machine [20]

Summary This paper defines a formalism for defining potentialy recursive functions and proposes S-expressions, S-functions, and S-function apply, which together are a system of theoretical program representation that is independent from its machine implementation. The paper then discusses the representation of S-expressions as lists in IBM 704 memory. This work addressed a need to represent LISP symbolically, and it offered additional tools to be used in mathematical logic and theorem proving.

The key ideas include: S-expressions, symbolic expressions composed of ".", "(", ")", and an infinite set of atomic symbols. Operations upon S-expressions. M-expressions, or functions of S-expressions. S-functions, which are described by M-expressions, and the S-function apply. The implementation of S-expressions in 704 memory as lists, where the list structure is specifically mechanized to implement an early iteration of garbage collection.

Context This paper discusses LISP for the IBM 704. Technologies have changed dramatically, and we no longer use the IBM 704. LISP is still used, and I'm sure its use cases have expanded past those listed in Section 4. S-expressions are still frequently discussed, although I think because of their simplicity they are most discussed in a pedagogical context. I ran into them when I was googling about Rust macros.

Discussion Points

1. The description of the formalism and the methods reminded me of "An Introduction to the Theory of Lists" by Richard S. Bird, in the sense of, they both describe similar operations, one upon S-expressions and the other upon lists. Any relationship there, or no? This potential correlation seems most relavent to the section that discusses representing S-expressions as lists.

2. It seems like the authors of this paper and the FORTRAN paper both naturally extended functions to include higher order functions. Maybe this is just me not understanding things, but I would have expected higher order functions to be some crazy difficult problem, but I guess that is not the case.

3. Section 4 lists that LISP was used to write the compiler to compile LISP programs. I believe this is called "bootstrapping"? Is this the first instance of this? / I forget what the FORTRAN compiler is written in.

Significance This paper primarily proposes a theoretical program representation and secondarily presents the machine implementation. This is in contrast to the FORTRAN paper, which integrated the "theoretical" language design into the physical implementation, and likely had significant impact on mathematical foundations of computer science. This paper also likely laid foundations for garbage collection.

7.6 Garbage Collection in an Uncooperative Environment [7]

Summary This paper discusses a method of garbage collection that does not require any cooperation from the object code generated from the source program. This means that garbage collection can be isolated - allowing for simplier implementations in conventional compilers and for garbage collection overhead to also be isolated such that programs that do not use garbage collection no longer suffer from this incurred overhead.

The garbage collection algorithm presented is the mark-sweep algorithm, which employs two passes over the data to perform collection while gaurenteeing that accessible data is never corrupted. The first pass "marks" all of the data that is accessible by the program and the second pass returns inaccessible objects to a list of free memory.

The authors discuss several issues that arise when thinkng about garbage collection, but the most notable of their observations is that the problem of whether or not a particular item is accessed in any possible iteration of the program is not decidable.

Context Well, garbage collection is still used today, and it will continue to be used for the forseeable future. I don't know enough about modern garbage collection to synthesize how this paper may have influenced modern designs, but the problems discussed in the paper are still problems faced today.

Discussion Points Last class we discussed NVM (spelling?), which was something along the lines of "persistent memory". How does garbage collection work in this environment?

7.7 Bringing the Web Up to Speed with WebAssembly [13]

Summary This paper introduces WebAssembly, a portable low-level "language" (bytecode) intended to be used across most modern web browsers. WebAssembly is designed to be language, hardware, and platform independent, and it is abstract over programming models. The authors introduce WebAssembly semantics, memory access, control flow, and type system. They discuss browser implementations and performance results. WebAssembly is slower than native code but faster than asm.js, but it also is smaller in resulting size.

The control flow and type system are two of the main technical contributions from this paper. WebAssembly uses a structured control flow, where the design bakes in the gaurentee that the code does not contain irreducible loops or "bad" branches. Control flow elements monitor their own local operand stacks, and branching in the code clears an operand stack, such that users don't have to track information about the stack. This design allows the type system to validate the code in a single pass.

The WebAssembly type system is interesting. It is written in such a way that all of the reduction rules are sound. This means that type-correct WebAssembly code contains so invalid calls, illegeal accesses, and is memory safe. The type system is inherently connected to the control flow design - the control flow design allows the type system to validate the code in one pass and the type system implies that the layout of the operand stack is determined statically. This establishes memory and state encapsulation.

Context WebAssembly is recent, this paper is just from 2017. I'm glad we read this, I see lots of people online really excited about the implications of WebAssembly. The control flow design is novel, although the idea of baking into the type system some desired behavior is not novel.

Discussion Points 1) Why is JavaScript the only natively supported language on the Web "by historical accident"? 2) What would be the closest relatives to WebAssembly's type system? 3) It seems like the most recent languages to be put into the spotlight (Rust, WebAssembly) have core features baked into the type systems so that their target behaviors can be statically determined. Should we expect this to be an upcoming trend or is this coincidence?

Significance WebAssembly is very significant, and much needed. It opens up many possibilities for what one might be able to do on the web, and it ensures that those new possibilities are done relatively more safely. I am excited to see more developments in WebAssembly.

7.8 Architecture of the IBM System/ 360 [2]

Summary Amdahl et al. describes the architectural design of the IBM System/ 360 and the rational behind each design choice. At a high level, the authors discuss the goal of the system design - to provide a general purpose machine that would provide an abstract data representation, agnostic from any particular function or application.

Context This paper was published 4 years after the ALGOL 60 paper, another paper working towards the goal of general purpose computing. In addition to being revolutionary at the time of publication (I imagine), this paper also introduced many of the hardware concepts that we know today - I/O, 32 bit and 64 bit processors, etc.

Discussion Points 1) This paper details that one of the requirements of the new IBM System/ 360 is that it would have to be such that ”each individual model and systems configuration in the line would have to be competitive with systems that are specialzed in function, performance level or both.” This is interesting because it is difficult to imagine what the school-of-thought must have been at the time, to where the inherent value of a more generic machine is not recognized. I read this and was suprised that this was a constraint, for this reason. 2) Is it by accident that the IBM System/ 360 ”got right” the concept of 32 -bit and 64 -bit systems? Or is it mearly an informed design choice, that we have continued to use? Could there have been alternative designs that work as well or better? What would the implications be for software developement, for computing schools of thought?

Significance This paper is very significant, it introcued I/O and processor design and likely changed the computing school of thought at the time.

7.9 Cramming More Components onto Integrated Circuits [24]

Summary This paper introduces Moore’s Law, a theory stating the number of components per integrated curcuit should double every year. This is a very well known theory, so it is interesting to read the origin.

The paper also offers detailed motivation for why designers, engineers, or consumers might want Moore’s Law to be in effect - it makes everyone’s lives easier. Moore dispells some false ideas about electricty needs, and offers possible design concepts.

Context This paper has remained quite relavent for some time. It outlived the predictions that Moore himself makes in the paper.

Discussion Points 1) I hear a lot of talk about how Moore’s Law is outdated and is becoming irrelevant to modern design concerns. But a simple Google search suggests that Moore’s Law is still observable. What’s up with that? 2) The last section discusses ”Linear circuitry”. What is that? 3) It is difficult to seperate correlation versus causation. Could it have been the case that

Moore's Law served as a subconcious guide or influence to hardware engineers? As in, perhaps attention would have been focused elsewhere, had engineers not already had the expectation that a 2x improvement was possible.

Significance

7.10 Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities [1]

Summary This paper revisits the original publication that introduced Amdahl's Law, a method of calculating prospective execution speed ups given certain hardware improvements. The paper walks through the rational behind reaching this conclusion.

This paper was short, and quite honestly I had a hard time following it, so I'm struggling with a review. I did enjoy how sassy the editors note is about this very thing: "Interestingly, it has no equations and only a single figure."

Discussion Points 1) What is Amdahl's Law in layman's terms?

Significance I can infer that Amdahl's Law is likely significant, based on the fact that there was a reprint issued. I do not know what the modern perspective is on it though.

7.11 Amdahls Law in the Multicore Era [14]

Summary This paper applied Amdahl's Law to multicore chips, and to do so the authors present a novel cost model that considers the number and performance of cores. They analyze their cost model and demonstrate that it can be used to apply Amdahl's Law to multicore chips. Using this new model, the authors conclude that denser chips increase the likelihood that cores will lose performance. They connect this to Moore's Law and urge researchers to find ways to improve chip performance with denser chips.

They also analyze Amdahl's Law against asymmetric multicore chips, and find that asymmetric chips have potential for greater speedups than symmetric chips. They propose that Amdahl's Law may not be particularly adept to modeling these asymmetric chips.

Discussion Points 1) How does this relate to the "stacked" chip design that was mentioned in class earlier this semester?

7.12 The Case for the Reduced Instruction Set Computer [27]

Summary This paper offers motivation for why exploring reduced instruction set computers (RISC) might be worthwhile or profitable. Patterson and Ditzel claim that the trend towards higher level languages is supported by several nuanced factors. They propose that it has been easier to gain a 10% computation advance by adding 10% more hardware, rather than making code 10% more dense, and this plays into the fact that it is actually more adventageous for companies to market ‘more advanced instruction sets’, basically meaning that there is no pressure on designers to actively reduce instruction sets.

The authors provided several pieces of evidence to offer insight into how more complicated instruction sets might actually be detrimental to compiler writers and assembly-language writers, and they propose why RISC might elleviate some of these burdens.

Context I don’t really know the context. I don’t know what the modern stance is on RISC versus CISC.

Discussion Points

1. Something that strikes me when reading these older papers is that it seems that the general mentality at the time was that there may not be a need or purpose for more complex or general-purpose computers, or maybe even, it seems that there was air of mysticism and uncertainty around how general-purpose computer technologies would unfold or would benefit users. This is demonstrated in this paper in the abstract: “If we review the history of computer families we find that the most common architectural change is the trend toward ever more complex machines.” Do I have a correct assessment of the situation?

2. Given (1) is a correct assessment, it seems that modern technologies might be at a similar empass.

3. Patternson and Ditzel’s complaints about how code compaction interacts with marketing strategies seems to be similar to my own thoughts on machine learning and the use of giant machines for training by ML giants (DeepMind, Google AI, etc).

Significance

7.13 Comments on the Case for RISC [9]

Summary This paper critiqued The Case for the Reduced Instruction Set Computer [27], going through point by point describing by each assertion made as incorrect or misguided. The authors make individual points, but their overall critique seems to be that they feel the original paper is baseless. Clark and Strecker argue that Patternson and Ditzel did not provide sufficient evidence to back up their claims, and in fact offer evidence to the contrary for many of them.

Clark and Strecker seem to believe that the case for RISC versus CISC is on a very case-by-case basis. They state that it could be the case that more specialzed, complex instructions have an advantage over combinations of smaller instructions, in specific scenarios. They believe that instruction set complexity cannot be measured by instruction count, contrary to the foundation of Patternson and Ditzel’s paper.

My favorite line was “Annecdotal accounts of irrational implementations are certainly interesting.”

Context This came out the same year as the original paper that it is critiquing.

Discussion Points

1. So what exactly is RISC versus CISC?
2. What would all four of these authors think of modern lanaguages?
3. Of these two paper, which is the more generally "accepted" one, and why?

Significance It's unclear to me what the significance was of the original paper, but I imagine the situation must be significant enough to warrant a published critique.

7.14 Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines [17]

Summary This paper analyzes the differences between machines with different parallelism mechanisms. It proposes 5 different types of machines: baseline, underpipelined, superscalar, superpipelined, VLIW, and superscalar + superpipelined. The two in focus are superscalar and superpipelined, where superscalar machines executes n instructions in a cycle and superpipelined machines execute 1 instruction every 1/n cycle. It is shown that although the two machines execute the same number of instructions in the same number of cycles, the superpipelined machine has a larger startup latency so the superscalar machine is actually 10% better.

However, these are theoretical machines, and actual instructions are much more complicated and contain multi-cycle instructions.

Discussion Points

1. If a language were to have something like, memory layout/ management/ usage baked into the type system, wouldn't the language be able to maximize automatic parallelisation? Does this exist?
2. I did not understand their explanation of VLIW machines, can we go over this?

7.15 Experience with Process and Monitors in Mesa [18]

Summary Lampson and Redell discuss the integration of processes and monitors into Mesa, and already existing programming language. The work is motivated by a desire to implement parrallelization into Pilot, a program closely linked to Mesa. They authors discuss the mechanism of implementation, the actual implementation in Mesa, and applications that use Mesa.

The notable contributions are the developments made to integrate processes and monitors with Mesa. Processes are integrated by means of 'detach' and 'join'. A processes detaches from its caller and is then joined in later, at a specific point in the code, once it has completed. This allows multiple processes to be executed asynchronously. Monitors are used to protect data, and data gaurded by a monitor can only be accessed within the body of the monitor procedure. A 'notify' system is used, where processes establish conditions upon which some other process *may* be waiting for that condition to hold, and they sends out notifications about these conditions. It is up to the notified process if they perfectly meet that estimated condition or not. This allows one process waiting on a condition to execute, but there also exists a broadcast operation that allows all processes waiting on a condition to execute.

The authors develop this work explicitly without considering security. The "system only supports one user", so the Mesa type system is sufficient. There is still also the possibility of race conditions and deadlocks.

Context This paper came out right before the RISC/ CISC papers, but about 10 years before the "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines" paper.

Discussion Points

1. Did this paper contribute to the modern idea of Futures? The description of Processes sounded eerily similar to Futures, although I don't believe Futures employ the notify system.

7.16 The UNIX Time- Sharing System [30]

Summary Ritchie and Thompson present the design of the UNIX system. They cover in detail the different aspects of the computer's design (file system, I/O, asynchronous processes, the shell) and discuss their importance and contribution to the UNIX system, and discuss

As pointed out by the authors, it is not necessarily that the UNIX system implemented any particularly insane features, its that it does so in a way that is sustainable and abstract. Perhaps most importantly, the users are able to modify the system itself, as all system files are available. This allows users resolve errors quickly. The OS is modular and abstract and handles many processes for the user automatically - address spaces are organized by program, the OS implements I/O so that the user can use one I/O end-point, and others.

Context From the papers we have read, it seems that this paper came out after a bit of a lull. It was published 11 years after Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, but then the MESA and RISC/ CICS papers shortly after.

Discussion Points The paper discusses how severe size constraits on the system lead to innovations. Given what we discussed last week - that how modern hardware is sold by "features" - it seems to me that currently there are not enough imposed pressures upon hardware to foster innovations. This is perhaps akin to natural selection.

1. How might different imposed pressures have lead to different modern hardware?

2. Is our hardware development timeline 'inevitable'?

Significance UNIX is widely used today in different forms. Luckily it does not cost $40,000 per installation.

7.17 Hints for Computer System Design [19]

Summary This paper was pretty straightforward, not a lot of technical detail. In my own words, I would summarize the message into a few key points.

Make components of the system as simple as they need to be, but not more. Components should be modular and "pure". As in, each component should be a dedicated layer of abstraction that is focused on performing one particular task. Artifacting should be reduced across components. Error and faults should be handled modularly.

A good and complete system is better than a great and incomplete system. All components of the system should be fleshed out, reliable, and safe. Components should not be to abstract or general to where they cannot handle particular cases.

Users care about a working system more than they care about the most state-of-the-art technically-advanced system.

Discussion Points

1. Are any of these guidelines particularly important over the others?

Significance These are good guidelines. I suspect I will revist this paper in the future.

7.18 A Case for Redundant Arrays of Inexpensive Disks (RAID) [28]

Summary This paper introduces Redundant Arrays of Inexpensive Disks (RAID), a method of improving I/O by keeping redundant copies of information on multiple hard disks. Because the information can be recovered when a disk read fails, this improves the performance of I/O and provides better fault tolerance. This is achieved through the use of multiple disks, disk mirroring, and disk striping.

Although the premise is ubiquitous, implementations of RAID can be application specific. The others discuss five ”levels” of implementations of RAID with differing performance and reliability. The first level is using disk mirroring without disk striping, where multiple drives duplicate storage. The second level uses disk striping and interleaving storage checks to reduce the overall amount of required space. The third level reduces redundancy by employing an operation that can reconstruct information lost during a read failure. The fourth level improves performance by allowing for multiple I/O operations in parallel, on different disks. The fifth level distributes the data and check information across all disks, so write operations can occur in parallel.

Context Because RAID is a general idea and not a specific implementation, I imagine that it is still relevant today. Hardware has changed, but the desired for increased I/O performance with increased fault tolerance has not.

Discussion Points

1. It isn't clear to me which of these levels is the "best"?

2. Are there any limitations to which level can be used with which application? As in, “don’t use level 3 if your application has this behavior” type thing.

Significance This paper likely improved the performance and fault tolerance of I/O.

7.19 Enhancing Server Availability and Security Through Failure-Oblivious Computing [29]

Summary This paper presents failure-oblivious computing, a computing technique that allows servers to execute through memory errors without memory corruption. The compiler inserts checks that dynamically detect invalid memory accesses. When one of these checks fails, the generated code discards the invalid writes and returns manufactured values. This enables the server to continue normal execution.

Context This paper was published in 2004, before the dawn of serverless computing. I would be interested to see if there have been any updates to this paper tailored towards serverless. Fault tolerance with serverless is more straightforward though - if a serverless function does something bad, either reexecute it or refuse to execute it. Serverless programs are tolerant to reexecution, so this paradigm is compatible with easy fault tolerance.

Discussion Points I have mixed feelings about this paper. On one hand it makes sense, servers should have robust fault tolerance so that they can manage many distinct processes. But on the other hand, I feel uneasy about this claim in the conclusion: *One of the major long-term goals of computer science has been understanding how to build more robust, resilient programs that can flexibly and successfully cope with unanticipated situations. Our research suggests that, remarkably, current systems may already have a substantial capacity for exhibiting this kind of desirable behavior if we only provide a way for them to ignore their errors, protect their data structures from damage, and continue to execute.* Okay ? Yes, if we ignore errors we will no longer have errors. I am likely misinterpreting this, can we clarify?

Significance Servers should have high fault tolerance, as they fulfill many requests and can receive any number of bad actor requests. In particular, this fault tolerance should be invisible and inevitable to programs (as if it wasn’t then the actual problem has not be addressed), and this paper achieves that. The server programming paradigm has shifted this this paper was published. I actually know less about old methods than I do current methods, but currently serverless programs are just run in individualized ”memory safe” virtualized containers. So, some of the motivation for this work (prevent corrupted shared memory) has expired.

7.20 DieHard: Probabilistic Memory Safety for Unsafe Languages [4]

Summary DieHard presents a method of soundly tolerating memory errors in unsafe C and C++ programs. It does so by employing probabilistic memory safety through randomization and replication. The memory management system randomizes the locations of objects in memory to give an approximation of an infinite-sized heap. Replication of this randomized memory can be used to give additional fault tolerance. This randomized allocator operates transparently to programmers, and helps mitigate against buffer overflows, dangling pointers, and reads of uninitialized data.

Making multiple copies of memory can require a lot of space, and the authors specify that DieHard is best suited for systems in which memory footprint is less important than reliability and security. This is a fair trade-off, as in the modern day there seems to be fewer constraints on memory size than the other factors.

Context I know that the authors have since published another paper, DieHarder, so I assume that the work has changed somewhat. I have not read this other paper though so I cannot say to exactly what extent. I believe the ideas presented though are distinct from the implementation. It is likely the case that probabilistic memory safety is still relevant.

Discussion Points

1. It seems that the replication strategy employed by DieHard is, at least somewhat, related to the replication strategy employed by RAID. Is there any formal connection here, or just loosely similar?

2. Could something like this (or some other randomization algorithm) be used to mitigate against Spectre attacks? Say I have 3 copies of memory, each of them randomized. And every access read from a random one of the 3. Of course, this would take up ample space, but systems are large enough now that it may be reasonable.

Significance A lot of important programs are written in C and C++. And a lot of compilers target C and C++. Research that improves memory error tolerance in these programs is significant.

7.21 Gprof: A call graph execution profiler [12]

Summary This paper presents gprof, a program evaluation tool that focuses on presenting information about execution organiztion and execution time. This is done by analyzing program call graphs, where the nodes of the call graphs are sub-routines and the arcs between nodes are the "who called whom." Gprof collects both execution counts and execution times to complete the suite, although both of these present their own unique implementation challenges. The resulting evaluation is presented to the user to help them analyze either components of their own programs or components of unknown programs.

Gprof is designed to be as lightweight as possible. Only minimal logging information is captured during dynamic execution. Once a program completes, gprof completes the data collection and analysis and only then performs the computationally intensive components. The authors discuss how to transform large recursion call graphs into a set of strong connected components, but they note that this fails for *very* large recursion call graphs.

Context The next paper for this week references gprof so it must be important. Also, recently I came across a paper that implemented a tracing JIT for Racket that used call graphs and continuation frames to detect recursion while tracing. I suspect that their work is derived in some capacity from this work.

Discussion Points

1. This evalution tool seems useful. Perhaps this is what the second paper covers, but this tool is not compatible with current technologies and architectural toolchains, so to speak. One of the systems lunch speakers discussed how decreasing the variance in latencies for a component of a serverless system could decrease the average end-to-end latency. That being said, I don't believe the reported metrics from gprof are detailed enough to express these complicated relationships.

Significance I do not know the significance. I suspect that this is an important paper. Although this paper presents an evaluation system, its methodology is application-neutral.

7.22 Coz: Finding Code that Counts with Causal Profiling [10]

Summary This paper present Coz, a causal profiler that uses virtual speedups to better determine possible "entry-points", so to speak, of improving performance. Coz implements virtual speedups through alternative physical slowdowns. By slowing down the remainder of code, it "speeds-up" the non slowed-down code. This information is captured and the processes is repeated. After this is complete, the user is presented with a causal profile with information about LOC and virtual speedups.

Coz can only profile code that is in your main file. It would be useful to expand this system so that it can profile dependencies. However, some of that information is retained, where if the main file calls a function from another file, that information will be captured, just not in granular detail. Using Coz also incurs some amount of overhead, but because this is an evaluation mechanism, the additional overhead is not that important.

Context Coz may or may not still be maintained, but causal profiling is a ubiquitous idea. It would be interesting to have this type of behavior implemented into a compiler of somesort.

Discussion Points

1. I honestly did not understand the intuition behind finding the 0% baseline. Can this be covered briefly?

2. What if one LOC has 100 different computational elements? Does Coz split these up?

7.23 An Empirical Study of the Reliability of UNIX Utilities [22]

Summary This paper describes the first attempt at fuzz testing. Fuzz testing is a method of testing that employs randomly, or pseudo-randomly, generated inputs to detect errors that programmers may have otherwise missed. A big portion of what fuzz testing does is ensuring that I/O operations are fault-tolerant, however there has been recent work to extend fuzz testing to include detecting more complex logical errors in programs.

This paper is the first to enter this domain. The method proposed is one that generates random input strings, tests these input strings on the program to detect crashes, and then reports the types of strings that cause crashes. Aside from time and compute resources, fuzz testing is relatively "cheap" and this is noted by the authors.

Context Fuzz testing is gaining popularity and in my opinion, is super cool. The methodology presented in the paper is the foundation for modern fuzz testing.

Discussion Points

1. Could fuzz testing be used to test other factors of the program? i.e. things that are not input. Does my allocator work if I malloc an array of any size between 0 and 99999999?
2. What about combining fuzz testing, causation analysis, and machine learning? That might be interesting.

Significance I expect fuzz testing to become more significant in upcoming years.

7.24 DART: Directed Automated Random Testing [11]

Summary This paper describes DART, a method of testing that uses a combination of automatically detecting the interface of the system, generating a test suite for this interface, and using the test suite to detect errors. This leads to completing automated testing - one of main research objective of DART. This is accomplished through combining symbolic reasoning when possible and randomization to introduce concrete values everywhere else.

DART is innovative because it strives to understand and explore all paths in an execution tree, which is more robust than just using fuzzing to test random inputs. The symbolic analysis component is able to add soundness and robustness, while the randomization components supplies real values for testing.

Context While its methodology is timeless, I haven't heard anything about it recently.

7.25 A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World [5]

Summary This paper introduces a system of static analysis that can be used to find errors. It can find simple errors (data races) or system-specific errors. It operates under the assumption that programmers tend to write code that is akin to ”programming rules.”

One of their core observations is that you cannot check code that you cannot see, and the authors note that this is particularly difficult when attempting to build an analysis tool for real-world systems, as real-world systems are large and contain many simultaneous working components. They also note that it can be a flaw to separate development environment from production environment, as this introduces new variables.

Context The ideas that they present are still relevant today. Programmers aren't perfect.

Discussion Points

1. I have some contentions about their assumptions. A programmer who is most likely to abide by programming rules is a good programmer already, and thus they are less likely to have bugs. I suppose this is irrelevant though because the tool is just checking for rule disobedience and not the other way around.

2. Are there any languages or compilers that have this built-in? Other than Idris or Coq.

3. I am interested in this idea in terms of fairness. It would be interesting to build an analysis tool for ML that detects fairness.

7.26 Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation [26]

Summary This paper introduces Valgrind, a dynamic binary analysis (DBA) tool that uses dynamic binary instrumentation (DBI). To do this, it implements shadow values with a DBI framework.

This paper discusses the internals of Valgrind, discussing the Valgrind IR and optimizations. These optimizations are necessary because using Valgrind is slow. However, the incurred slowdown is an appropriate tradeoff because it is only used for analysis and can ultimately lead to better systems.

I just skimmed this paper unfortunately, this is a bad review!!

Context People still use valgrind. I haven't noticed much of a slower performance, so maybe there has been more effort in optimizations.

Significance People still use valgrind today.

7.27 Bitcoin: A Peer-to-Peer Electronic Cash System [25]

Summary This paper introduces bitcoin, a peer-to-peer payment system that does not rely on a trusted third party. Coins are constructed by a chain of accepted transactions. Each transaction is publicly broadcast, and actors in the system vote to either accept or reject the transaction. They accept chains by building on them, and they build on chains by computing computationally intensive programs. This means that if any chain were to be tampered with, the bad actors would have to recompute every computationally intensive program in the chain.

Bitcoin only works if the majority of the actors in the system are good actors, for a majority of bad actors could band together to accept inaccurate chains. Unless I missed this in the paper, the authors don't discuss how they could prevent an influx of bad actors in a system. They mention that actors are counted based on CPU's and not IP's which helps, but doesn't completely mitigate.

Context Bitcoin, and other cryptocurrencies, are becoming more popular.

Discussion Points

1. Maybe I am just not understanding, but I am still confused about how the currency enters the system. It makes sense that CPU power and electricity are expended to create the chains, but why is this honored as something of value? I can run 10000 python matrix operations, but this won't be worth anything. Maybe another way of phrasing it is - why have agencies determined that this is a form of currency that they will accept?

2. Facebook wants to start Libra. A bit contention from the US government is this issue of anonymous wallets. But in order for cryptocurrency to work, all payments must be broadcast, ideally with those payments anonymized. So maybe I am misunderstanding, but would a lack of anonymous wallets mean that all transactions would be broadcast un-anonymized?

Significance I believe this is the first paper to introduce bitcoin, so, very significant.

Chapter 8

Relating to Containerless

Bibliography

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[2] Gene M Amdahl, Gerrit A Blaauw, and Frederick P Brooks. Architecture of the ibm system/360. *IBM Journal of Research and Development*, 8(2):87–101, 1964.

[3] John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, L Mitchell Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, H Stern, et al. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198. ACM, 1957.

[4] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[5] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[6] Richard S Bird. An introduction to the theory of lists. In *Logic of programming and calculi of discrete design*, pages 5–42. Springer, 1987.

[7] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[8] Tongfei Chen. Typesafe abstractions for tensor operations (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 45–50, 2017.

[9] Douglas W Clark and William D Strecker. Comments on the case for the reduced instruction set computer, by patterson and ditzel. *ACM SIGARCH Computer Architecture News*, 8(6):34–38, 1980.

[10] Charlie Curtsinger and Emery D Berger. C oz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197. ACM, 2015.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[12] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[13] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM, 2017.

[14] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[15] Grace Murray Hopper. The education of a computer. In *Proceedings of the 1952 ACM national meeting (Pittsburgh)*, pages 243–249. ACM, 1952.

[16] Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[17] Norman P Jouppi and David W Wall. *Available instruction-level parallelism for superscalar and superpipelined machines*, volume 17. ACM, 1989.

[18] Butler Lampson. Experience with processes and monitors in mesa. 1980.

[19] Butler Lampson. Hints for computer system design. 1983.

[20] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. 1959.

[21] Matt McCutchen, Judith Borghouts, Andy Gordon, Simon Peyton Jones, and Advait Sarkar. Elastic sheet-defined functions: Generalising spreadsheet functions to variable-size input arrays. November 2018.

[22] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[23] John A Miller, Jun Han, and Maria Hybinette. Using domain specific language for modeling and simulation: Scalation as a case study. In *Simulation Conference (WSC), Proceedings of the 2010 Winter*, pages 741–752. IEEE, 2010.

[24] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[25] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[26] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[27] David A Patterson and David R Ditzel. The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, 8(6):25–33, 1980.

[28] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[29] Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, and William S Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, volume 4, pages 21–21, 2004.

[30] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[31] Advait Sarkar, Andrew D Gordon, Simon Peyton Jones, and Neil Toronto. Calculation view: multiple-representation editing in spreadsheets. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 85–93. IEEE, 2018.

[32] Aaron B Todd, Amara K Keller, Mark C Lewis, and Martin G Kelly. Multi-agent system simulation in scala: An evaluation of actors for parallel simulation. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2011.

[33] Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.