Efficient IO-Intensive us-scale Applications using eBPF

Kumar Kartikeya Dwivedi Rishabh Iyer Sanidhya Kashyap

EPFL | Meta

Outline

01 Motivation

02 Case Studies

03 Rethinking BPF’s Role

04 Ideas

05 Challenges

01 Motivation

Change is the only constant

- Evolving hardware demands the OS and workloads to adapt.
- Machines are scaling vertically.
  - More core counts, more PCIe bandwidth, more NIC bandwidth, etc.

Change is the only constant

- Evolving hardware demands the OS and workloads to adapt.
- Machines are scaling vertically.
  - More core counts, more PCIe bandwidth, more NIC bandwidth, etc.
- Shifts bottlenecks into the host.
  - Suboptimal scheduling, but we now use sched_ext / ghOSt.
  - Higher resource utilization with isolation is difficult.

Much ink has been spilled

- Tons of people exploring new ideas in industry in academia.
- Only a few make it into production kernels like Linux.
- It’s not just a technical problem, but also a human problem.

“Valley of death” for interesting ideas.

- Interesting academic ideas fail to gain traction due to lack of integration.
- Relegated to prototype systems.

“Valley of death” for interesting ideas

- Interesting academic ideas fail to gain traction due to lack of integration.
- Relegated to prototype systems.
- Others go with user space alternatives since kernel is hard to work with.
- Slower iteration, complex deployment, etc.
- eBPF helps here, but it’s limited in scope.

02 Case Studies

CS 1: Dynamic Scaling of Network Stack

- Spread or join cores doing network processing dynamically.
- Google’s Snap / NetChannel both achieve this.

CS 1: Dynamic Scaling of Network Stack

- Spread or join cores doing network processing dynamically.
- Google's Snap / NetChannel both achieve this.
  - Snap: User space networking stack, single busy-polling thread controls spread or join logic. Engines encapsulate transport layer processing, engines mapped to threads.
  - NetChannel: Decouple packet processing queues from kthreads, then independently scale / compact both queues and threads, and control their assignments.

Why is threaded NAPI / cpumap not enough?

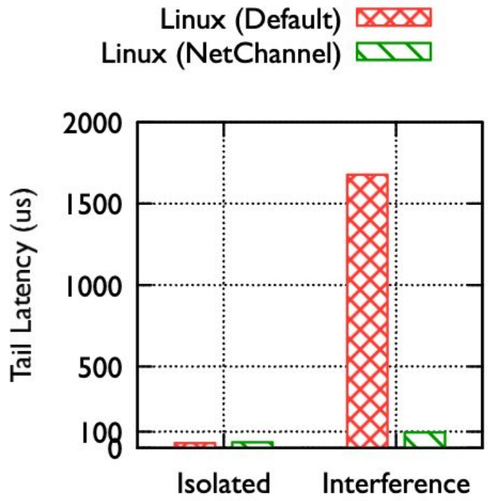
- We still need explicit CPU allocation for each thread.
- Wasted capacity when e.g. 4 threads on 4 CPUs are at 20% utilization each.

Why is threaded NAPI / cpumap not enough?

- We still need explicit CPU allocation for each thread.
- Wasted capacity when e.g. 4 threads on 4 CPUs are at 20% utilization each.
- Cannot co-locate with application threads.
  - Packet processing will suffer ms-scale tail latencies.
- Cannot co-locate with itself to compact work onto same core.
  - Same issue; ms-scale tail latencies.
- See Jakub's TAPI idea.

Efficiency

- Dynamic scaling enables better efficiency in two ways.
- 1) Overcommit and harvest underutilized capacity when relatively idle.
- 2) Reclaim cores proportionally when offered load is high.
- Scaling decisions are a function of many variables:
  - SLOs, target latency / throughput, traffic pattern (bursty vs streaming), etc.



Isolation

- Co-locating ‘latency-critical’ and ‘throughput-intensive’ applications.
- Classic approach is through core-separation for both.
- Compact work onto same core if some latency inflation is tolerable, with prioritization.
- Requires control on packet processing within the worker.
- E.g. Weighted Round Robin when servicing NAPI instances, etc.

Co-design with CPU scheduler

- This is partly a scheduling problem.
- We need soft-partitioning of network processing cores from the rest of the system.
- The network stack then maps work to its soft-partitioned core group.
- The core group grows and shrinks as per the offered load.
- The signal to grow and shrink is given to the CPU scheduler by the network stack.

“Warm” cores for absorbing bursts

- Keep aside a smaller partition of idle cores without going to lower c-state.
- Absorb bursts (translates to “spread through wakeups”) at microsecond-scale.
- Harvestable cores, but can be instantly preempted.
- Hot migration of whatever thread was running on them, to prevent starvation.

Preventing starvation of EEVDF tasks

- Some housekeeping kthreads still need to be run on packet processing CPUs.
- Microsecond-scale time slicing is necessary (to avoid ms-scale tail latencies).
- Google did SCHED_FIFO-like MicroQuanta scheduling class for Snap threads.

Preventing starvation of EEVDF tasks

- Some housekeeping kthreads still need to be run on packet processing CPUs.
- Microsecond-scale time slicing is necessary (to avoid ms-scale tail latencies).
- Google did SCHED_FIFO-like MicroQuanta scheduling class for Snap threads.
- Out of every 1 ms, 0.9 ms dedicated to MicroQuanta task, 0.1 ms to EEVDF tasks.
- Con: Blackout for 0.1 ms / 100 us. Adjust blackout period as per requirement.
- scx_microq? For now I just forward ported Xi Wang's patch set to bpf-next.

CS 2: Dataplane Operating System

- A line of academic work where kernel-bypass libOS link with the application.
- Key idea: Co-design of scheduling and lightweight network data plane.

CS 2: Dataplane Operating System

- A line of academic work where kernel-bypass libOS link with the application.
- Key idea: Co-design of scheduling and lightweight network data plane.
- Central dispatcher on one core.
- Distribute packets for network stack + application processing to workers busy-polling other cores.

CS 2: Dataplane Operating System

- A line of academic work where kernel-bypass libOS link with the application.
- Key idea: Co-design of scheduling and lightweight network data plane.
- Central dispatcher on one core.
- Distribute packets for network stack + application processing to workers busy-polling other cores.
- Different scheduling strategies (FCFS, Processor Sharing) to handle different request latency distributions.

Key differences

- All network processing co-located with application threads.
- Request latency spans (network + application) work.
- Hence inline execution allows scheduling control on both.

Key differences

- All network processing co-located with application threads.
- Request latency spans (network + application) work.
- Hence inline execution allows scheduling control on both.
- Also customize the data path.
  - Zero-copy thin data path doing IP+TCP processing on XDP frames over AF_XDP.

Key differences

- All network processing co-located with application threads.
- Request latency spans (network + application) work.
- Hence inline execution allows scheduling control on both.
- Also customize the data path.
  - Zero-copy thin data path doing IP+TCP processing on XDP frames over AF_XDP.
- Request latency can be 10s of us minimum.

Takeaway

- Dynamic Scaling: Practical, beneficial, already proven to be useful.
- Dataplane OS: Academic, zero traction due to scorched-earth approach.

Takeaway

- Dynamic Scaling: Practical, beneficial, already proven to be useful.
- Dataplane OS: Academic, zero traction due to scorched-earth approach.
- Both need:
  - Close integration of CPU scheduling with the network stack.
  - Fine-grained control on assignment of packet processing work to CPU.
  - Assignment can and will change dynamically.
  - Possible customization of data path.

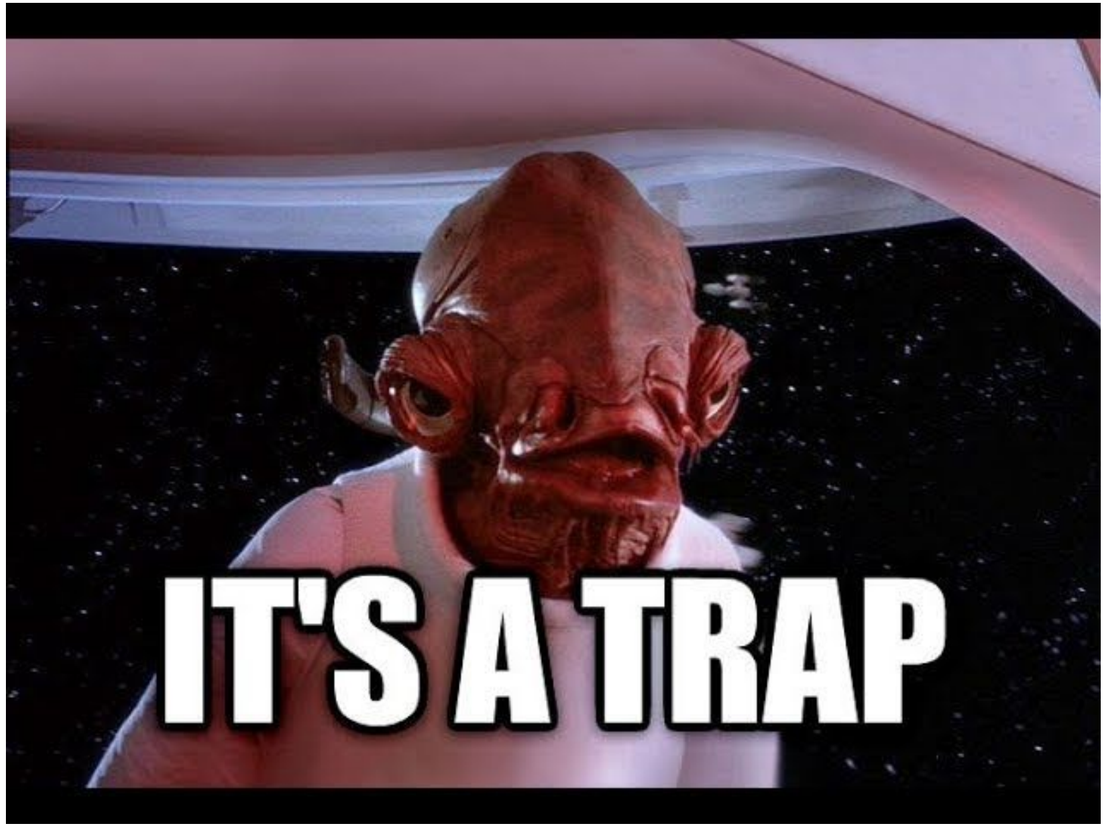
03 Rethinking BPF’s Role

This sounds achievable!

- A lot of necessary pieces are available.
  - sched_ext, AF_XDP, cpumaps, etc.
  - Run BPF programs in kthreads.
  - Expose packet queues?
  - Some glue to tie everything together.
  - Piece together a struct_ops interface?
  - Add more hooks!

This sounds achievable!

- A lot of necessary pieces are available.
  - sched_ext, AF_XDP, cpumaps, etc.
  - Run BPF programs in kthreads.
  - Expose packet queues?
  - Some glue to tie everything together.
  - Piece together a struct_ops interface?
  - Add more hooks!
- Let’s take a step back.



Why did this not happen already?

- Kernel is hard to change.
- Not a pressing business need.
- People equipped to make such changes stuck with other higher-priority work.
- Some scenarios are too deployment-specific.
- A combination of many reasons; social, and technical.

But it did happen outside Linux!

- Academics built their own toy OS.
  - More control, faster iteration.
- Others went the user space kernel-bypass route.
  - Easier development, faster iteration.

But it did happen outside Linux!

- Academics built their own toy OS.
  - More control, faster iteration.
- Others went the user space kernel-bypass route.
  - Easier development, faster iteration.
- What are they getting?
  - Freedom.
  - Freedom to make disruptive design choices people didn't **anticipate**.

But it did happen outside Linux!

- Academics built their own toy OS.
  - More control, faster iteration.
- Others went the user space kernel-bypass route.
  - Easier development, faster iteration.
- What are they getting?
  - Freedom.
  - Freedom to make disruptive design choices people didn’t **anticipate**.
- **We want Linux to be the test bed, and for eBPF to be the means to innovate.**

Anticipation is how extensibility works

- Safety is table stakes.
- So we do need to “anticipate” some sort of usage, and enforce safety around that.
- Safety translates to constraints (static or dynamic checks).
- Constraints limit freedom in design.

What are we trying to **change**?

What are we trying to **change**?

- Trying to change "where" things are executed, and "when".
- Where: Execution context (which kthread, and indirectly, which core).
- When: Time of execution, i.e. scheduling (related: queueing, batching).
- Less concern on "what" is being executed, i.e. we're mostly oblivious to work done.

Not just networking

- A broad set of OS design changes require control over "where" and "when"
  - In addition to "what", which eBPF can address already.
- "What" gets executed encapsulates **functional** logic of the kernel.
- "Where" and "when" encapsulate **structural** properties.

Not just networking

- A broad set of OS design changes require control over "where" and "when"
  - In addition to "what", which eBPF can address already.
- "What" gets executed encapsulates **functional** logic of the kernel.
- "Where" and "when" encapsulate **structural** properties.
- We need **primitives** to make **structural** changes, without anticipating use cases.
- Still safe, still not extreme freedom, but practically more than enough.
- For a correct kernel change, you STILL adhere to safety.

04 Ideas

Approach: Mechanism

- Encapsulate functional logic or computation as an object.
- Expose this object as part of BPF’s programing environment.
- Allow BPF developers to drive these objects to completion (i.e. execute them).

Approach: Mechanism

- Encapsulate functional logic or computation as an object.
- Expose this object as part of BPF's programing environment.
- Allow BPF developers to drive these objects to completion (i.e. execute them).
- Separate execution resources (code + data) from their invocation.
- Hand over control and ownership of invocation to the BPF developer.

Coroutines

- Encapsulation of code + data using coroutines.

Coroutines

- Encapsulation of code + data using coroutines.
  - 1960s tech, so nothing bleeding edge.

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective COBOL except for automatic segmentation and its byproducts, such as those properties of the ALTER verb which are affected by segmentation. The verbs DEFINE, ENTER, USE and INCLUDE are accessible to the design but were not included in the prototype coded at the Case Computing Center.

2. It can be implemented as a true one-pass compiler (with load-time fixup of forward references to procedure names) on a machine with 10,000 to 16,000 words of high-speed storage. In this configuration it processes a source deck as fast as current one-pass algebraic compilers

to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on

Coroutines

- Encapsulation of code + data using coroutines.
  - 1960s tech, so nothing bleeding edge.
- All logic is encapsulated in coroutine body.

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective COBOL except for automatic segmentation and its byproducts, such as those properties of the ALTER verb which are affected by segmentation. The verbs DEFINE, ENTER, USE and INCLUDE are accessible to the design but were not included in the prototype coded at the Case Computing Center.

2. It can be implemented as a true one-pass compiler (with load-time fixup of forward references to procedure names) on a machine with 10,000 to 16,000 words of high-speed storage. In this configuration it processes a source deck as fast as current one-pass algebraic compilers

to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on

Coroutines

- Encapsulation of code + data using coroutines.
  - 1960s tech, so nothing bleeding edge.
- All logic is encapsulated in coroutine body.
- Suspension points indicate “spatial” and “temporal” changes.

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective COBOL except for automatic segmentation and its byproducts, such as those properties of the ALTER verb which are affected by segmentation. The verbs DEFINE, ENTER, USE and INCLUDE are accessible to the design but were not included in the prototype coded at the Case Computing Center.

2. It can be implemented as a true one-pass compiler (with load-time fixup of forward references to procedure names) on a machine with 10,000 to 16,000 words of high-speed storage. In this configuration it processes a source deck as fast as current one-pass algebraic compilers

to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on

Coroutines

- Encapsulation of code + data using coroutines.
  - 1960s tech, so nothing bleeding edge.
- All logic is encapsulated in coroutine body.
- Suspension points indicate “spatial” and “temporal” changes.
  - Execution context / location can be switched.
  - Time until resumption is user-controlled.

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective COBOL except for automatic segmentation and its byproducts, such as those properties of the ALTER verb which are affected by segmentation. The verbs DEFINE, ENTER, USE and INCLUDE are accessible to the design but were not included in the prototype coded at the Case Computing Center.

2. It can be implemented as a true one-pass compiler (with load-time fixup of forward references to procedure names) on a machine with 10,000 to 16,000 words of high-speed storage. In this configuration it processes a source deck as fast as current one-pass algebraic compilers

to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on

Coroutines

- Encapsulation of code + data using coroutines.
  - 1960s tech, so nothing bleeding edge.
- All logic is encapsulated in coroutine body.
- Suspension points indicate “spatial” and “temporal” changes.
  - Execution context / location can be switched.
  - Time until resumption is user-controlled.
- Control over “where” and “when” is achieved.

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective COBOL except for automatic segmentation and its byproducts, such as those properties of the ALTER verb which are affected by segmentation. The verbs DEFINE, ENTER, USE and INCLUDE are accessible to the design but were not included in the prototype coded at the Case Computing Center.

2. It can be implemented as a true one-pass compiler (with load-time fixup of forward references to procedure names) on a machine with 10,000 to 16,000 words of high-speed storage. In this configuration it processes a source deck as fast as current one-pass algebraic compilers

to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on

Coroutines

- Both examples want to maintain queues.
- Control over where and when packets are processed.

```
data_path(struct xdp_md *ctx) {
  co_await queue();
  …
}
```

Coroutines

- Both examples want to maintain queues.
- Control over where and when packets are processed.
- Suspension point provides a natural hook to do so.
- Data path processing on resumption.

```
data_path(struct xdp_md *ctx) {
  co_await queue();
  …
}
```

Coroutines

- Queue function gets access to coroutine handle of suspended coroutine.

```
queue(coro_handle<> h) {
  list = pick_queue(...);
  list_add(&h->node, &list);
}
```

Coroutines

- Queue function gets access to coroutine handle of suspended coroutine.
- Allow stashing coroutines as kptrs into maps, linked lists, RB-trees.

```
queue(coro_handle<> h) {
  list = pick_queue(...);
  list_add(&h->node, &list);
}
```

Coroutines

- Worker kthreads run BPF programs that can pull coroutines out and resume them.

```
worker() {
  while (h = list_pop_front(...))
    h.resume();
}
```

Coroutines

- Worker kthreads run BPF programs that can pull coroutines out and resume them.
- Natural point for batching.

```
worker() {
  int i = 0;
  char coro_handle<> arr[BATCH_COUNT];
  while (arr[i++] = list_pop_front(...))
    if (i == BATCH_COUNT) break;
  while (i--)
    arr[i].resume();
}
```

Coroutines

- Verifier + BPF runtime ensure resource safety and kernel integrity.
- Code + data packaged as an object whose lifetime is controlled by user.
- The same “building block” can be used to control “where” and “when”.

Before you freak out!

I’m not asking us to add coroutines to BPF…

Before you freak out!

I’m not asking us to add coroutines to BPF…**YET**.

Approach: Safety

- We want to apply this mechanism to any part of the kernel.
- The only requirement is to ensure “safety”.
- Safety == constraints enforced by BPF on use of these mechanisms.

Approach: Safety

- We want to apply this mechanism to any part of the kernel.
- The only requirement is to ensure “safety”.
- Safety == constraints enforced by BPF on use of these mechanisms.
- Make the kernel “export” these constraints as **specifications.**
- Not in the verifier, but through **annotation** of kernel code.
- Verifier is simply the consumer of the specification, and it’s enforcer.

Specification

- Network Rx example.
- Lean zero-copy data path for dataplane OS.

```
lean_data_path(struct xdp_md *ctx) {
  iph = ip_hdr(ctx);
  th = tcp_hdr(ctx);

  ip_rcv(ctx, iph);

  sk = sk_lookup(ctx);
  socket_lock(sk);
  // Assume for TCP_ESTABLISHED
  tcp_rcv(ctx, th);

  queue_to_socket(sk, ctx);
  socket_unlock(sk);
  …
}
```

Specification

- Two basic requirements:
  - ip_rcv must happen before tcp_rcv.

```
lean_data_path(struct xdp_md *ctx) {
  iph = ip_hdr(ctx);
  th = tcp_hdr(ctx);

  ip_rcv(ctx, iph);

  sk = sk_lookup(ctx);
  socket_lock(sk);
  // Assume for TCP_ESTABLISHED
  tcp_rcv(ctx, th);

  queue_to_socket(sk, ctx);
  socket_unlock(sk);
  …
}
```

Specification

- Two basic requirements:
  - ip_rcv must happen before tcp_rcv.
  - tcp_rcv must have socket lock held.

```
lean_data_path(struct xdp_md *ctx) {
  iph = ip_hdr(ctx);
  th = tcp_hdr(ctx);

  ip_rcv(ctx, iph);

  sk = sk_lookup(ctx);
  socket_lock(sk);
  // Assume for TCP_ESTABLISHED
  tcp_rcv(ctx, th);

  queue_to_socket(sk, ctx);
  socket_unlock(sk);
  …
}
```

Specification

- Specification through annotations.

```
Enum Stage { Raw, PostIP };

ip_rcv(xdp_md<Raw>, …) -> xdp_md<PostIP>


tcp_rcv(xdp_md<PostIP>, …) -> …
```

Specification

- Specification through annotations.
- Use type states to enforce ordering.
- Distinction only in BPF type system.
- For the user, everything is xdp_md.

```
Enum Stage { Raw, PostIP };

ip_rcv(xdp_md<Raw>, …) -> xdp_md<PostIP>

tcp_rcv(xdp_md<PostIP>, …) -> …
```

Specification

- Need to hold socket lock of socket **tied** to this packet instance.

```
Enum Stage { Raw, PostIP };

ip_rcv(xdp_md<Raw>, …) -> xdp_md<PostIP>


tcp_rcv(xdp_md<PostIP>, …) -> …
```

Specification

- Need to hold socket lock of socket **tied** to this packet instance.
- Needs to be reflected in the type system.
- Represent instances and parameterize constraints on them.

```
Enum Stage { Raw, PostIP };

ip_rcv(xdp_md<Raw, ‘a>, …) ->
xdp_md<PostIP, ‘a>

sk_lookup(xdp_md<PostIP, ‘a>) -> Socket<’a>
__requires_lock(Socket<’a>::lock)
tcp_rcv(xdp_md<PostIP, ‘a>, …) -> …
```

Separate annotation from enforcement

- Encode rules directly in source in a declarative fashion.
- Verifier consumes generic specification constructs and enforces them.
- Rules in the “skulls” of kernel developers are encoded in the source code.
- BPF type system is richer than the C type system w.r.t. the kernel’s safety constraints.

Takeaway

- The core idea is “abstraction”, while preserving kernel’s safety.
- Present the user with tools to manipulate structural properties (spatial, temporal).
- Only impose constraints necessary for safety.
- Hide unnecessary details from the runtime and the user.
- Increase freedom, flexibility, accessibility.

05 Challenges

Specification complexity

- We rely on the ingenuity of kernel developers to define a complete specification of safety properties.
- If they’re too strict, they restrict the design space.
- If they’re too relaxed, they open up potential safety holes.

Specification complexity

- We rely on the ingenuity of kernel developers to define a complete specification of safety properties.
- If they’re too strict, they restrict the design space.
- If they’re too relaxed, they open up potential safety holes.
- This is a problem today as well, to some extent. Safety rules can be incomplete.
- Can we do better?
  - Exhaustive runtime checking of the specification for violations?
  - Eliminate incompleteness by construction?
  - If so, how to define completeness (what properties must be fulfilled)?

Strict specifications restrict freedom

- Kernel built in a way such that its safety specification doesn’t allow for much freedom.
- E.g. no point in parallel processing if all code needs a lock around it.

Strict specifications restrict freedom

- Kernel built in a way such that its safety specification doesn’t allow for much freedom.
- E.g. no point in parallel processing if all code needs a lock around it.
- Path 1: Relax constraints by changing kernel implementation.
- Path 2: Allow BPF to provide alternative, unconstrained implementation.