Fast JavaScript with Data-oriented Design

Lessons from the Firefox Profiler

Markus Stange

FOSDEM 2024

About me

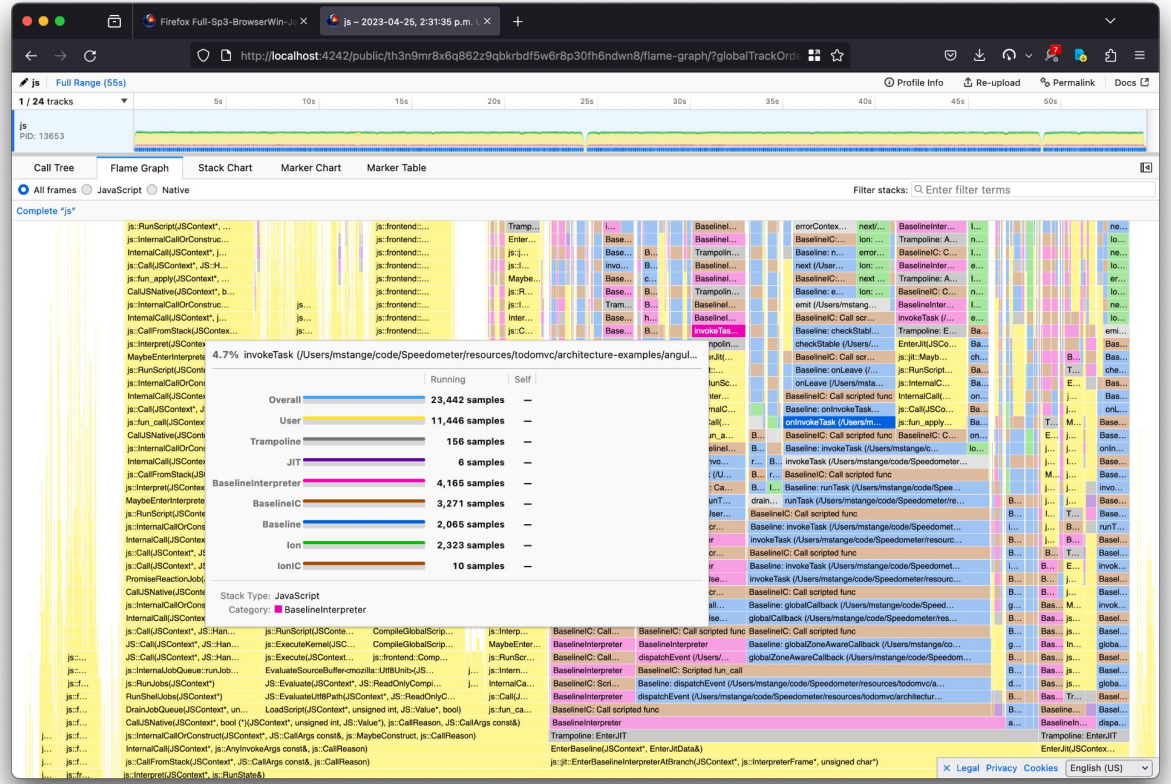
Markus Stange @ Mozilla

Firefox

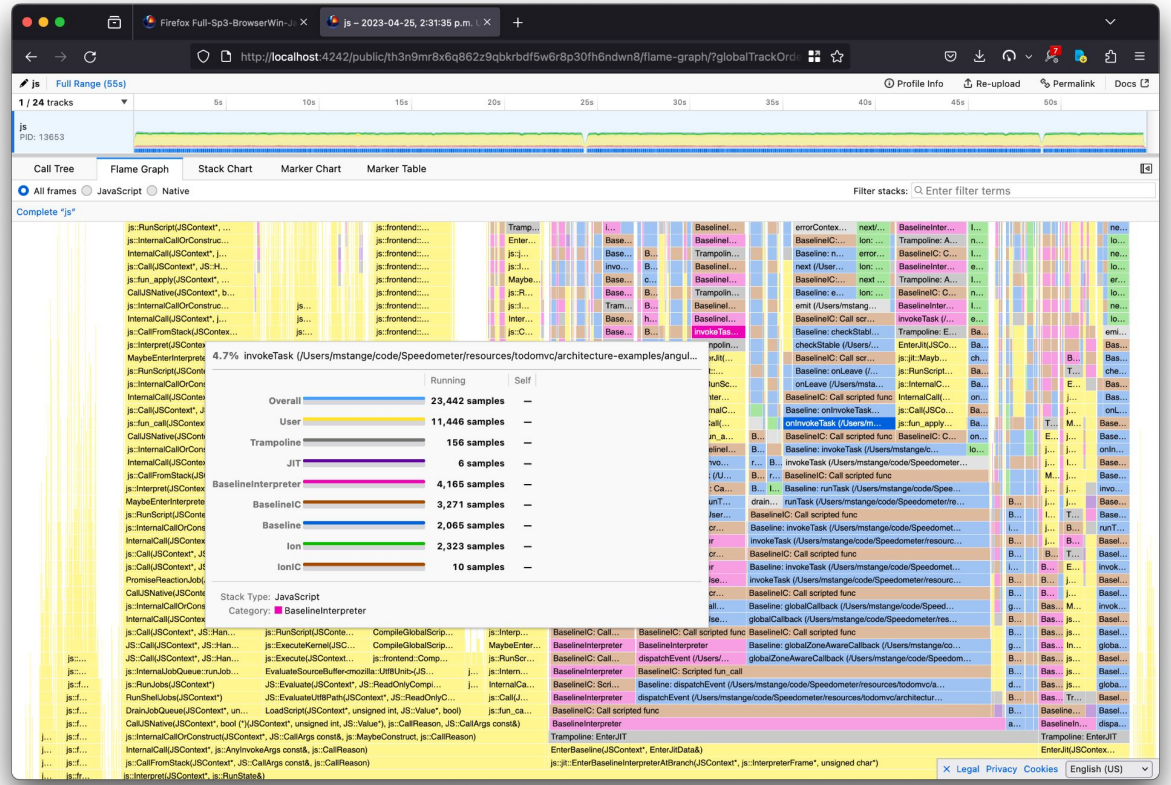
Firefox Profiler: https://profiler.firefox.com/

Samply: https://github.com/mstange/samply

Lots of samples = slow UI?

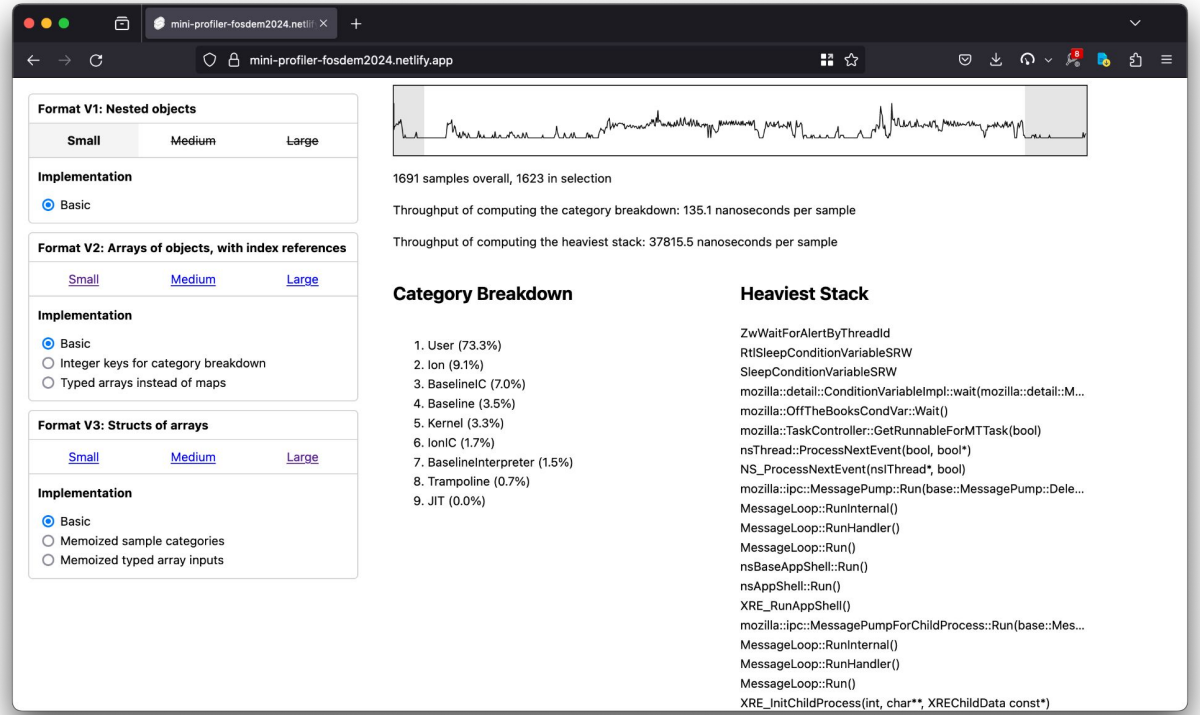


Lots of samples = fast UI! (after a lot of optimization work)



https://share.firefox.dev/3uioexK

Let’s optimize a mini profiler together!



https://github.com/mstange/mini-profiler-fosdem2024

https://mini-profiler-fosdem2024.netlify.app/

Mini profiler features

- Select a range in the graph
- During selection, the following pieces of UI are updated:
  - The category breakdown for the selection
  - The “heaviest stack” for the selection

Profile JSON format:

- List of samples
- Every sample has a time, a weight, and a stack
- Every stack is an array of frames
- Every frame has a name and a category

https://github.com/mstange/mini-profiler-fosdem2024 https://mini-profiler-fosdem2024.netlify.app/

Toy profile format, V1

```
{
  "samples": [
    {
      "time": 17880.8104,
      "stack": [
        { "name": "ZwWaitForAlertByThreadId", "category": "User" },
        { "name": "RtlSleepConditionVariableSRW", "category": "User" },
        { "name": "SleepConditionVariableSRW", "category": "User" },
        { "name": "mozilla::ConditionVariable::wait(...)", "category": "User" },
        { "name": "mozilla::OffTheBooksCondVar::Wait()", "category": "User" },
        ...
      ],
      "weight": 643
    },
    {
      "time": 17721.0814,
      "stack": [
        { "name": "KiSearchForNewThreadOnProcessor", "category": "Kernel" },
        { "name": "KiSwapThread", "category": "Kernel" },
        ...
      ],
      "weight": 1
    },
    ...
  ]
}
```

```
export type Profile = {
  samples: Sample[];
};

export type Sample = {
  time: number;
  stack: Stack;
  weight: number;
};

export type Stack = Frame[];

export type Frame = {
  name: string;
  category: string;
};
```

https://github.com/mstange/mini-profiler-fosdem2024

https://mini-profiler-fosdem2024.netlify.app/

V1: Computing the category breakdown

```ts
export function computeCategoryBreakdown(
  profile: Profile,
  range: SampleIndexRange
): CategoryBreakdown {
  const map = new Map();
  for (let i = range.start; i < range.end; i++) {
    const { stack, weight } = profile.samples[i];
    const topFrame = stack[0];
    const category = topFrame.category;
    map.set(category, (map.get(category) ?? 0) + weight);
  }
  return map;
}
```

```ts
export type Profile = {
  samples: Sample[];
};

export type Sample = {
  time: number;
  stack: Stack;
  weight: number;
};

export type Stack = Frame[];

export type Frame = {
  name: string;
  category: string;
};

export type CategoryBreakdown =
  Map<string, number>;

export type SampleIndexRange =
  { start: number; end: number };
```

V1: Computing the heaviest stack

```ts
export function computeHeaviestStack(
  profile: Profile, range: SampleIndexRange
): Stack {
  const map = new Map();
  let heaviestStackWeight = 0;
  let heaviestStack: Stack = [];
  for (let i = range.start; i < range.end; i++) {
    const { stack, weight } = profile.samples[i];
    const stackJsonString = JSON.stringify(stack);
    const stackWeight = (map.get(stackJsonString) ?? 0) + weight;
    map.set(stackJsonString, stackWeight);
    if (stackWeight > heaviestStackWeight) {
      heaviestStackWeight = stackWeight;
      heaviestStack = stack;
    }
  }
  return heaviestStack;
}
```

```ts
export type Profile = {
  samples: Sample[];
};

export type Sample = {
  time: number;
  stack: Stack;
  weight: number;
};

export type Stack = Frame[];

export type Frame = {
  name: string;
  category: string;
};

export type CategoryBreakdown =
  Map<string, number>;

export type SampleIndexRange =
  { start: number; end: number };
```

Is it fast?

Throughput of computing the category breakdown: 104.9 nanoseconds per sample

Throughput of computing the heaviest stack: 36780.4 nanoseconds per sample

For 100,000 samples: 10.49ms and 3.6 seconds

Reasonable for small profiles, quickly falls down as profiles get bigger.

The JSON file is gigantic and repetitive.

Toy profile format, V2: Lots of indexes

```
{
 "samples": [
    { "time": 17880.8104, "stackIndex": 27, "weight": 643 },
    { "time": 17721.0814, "stackIndex": 87, "weight": 1 },
    ...
  ],
  "stacks": [
    { "frameIndex": 24, "parentStackIndex": null },
    { "frameIndex": 23, "parentStackIndex": 0 },
    { "frameIndex": 22, "parentStackIndex": 1 },
    ...
  ],
  "frames": [
    { "name": "ZwWaitForAlertByThreadId", "categoryIndex": 0 },
    { "name": "RtlSleepConditionVariableSRW", "categoryIndex": 0 },
    { "name": "SleepConditionVariableSRW", "categoryIndex": 0 },
    ...
  ],
  "categories": [
    "User",
    "Kernel",
    "Trampoline",
    "JIT",
    ...
  ]
}
```

```
export type Profile = {
  samples: Sample[];
  stacks: StackNode[];
  frames: Frame[];
  categories: string[];
};

export type Sample = {
  time: number;
  stackIndex: number;
  weight: number;
};

export type StackNode = {
  parentStackIndex: number | null;
  frameIndex: number;
};

export type Frame = {
  name: string;
  categoryIndex: number;
};
```

V2: Computing the heaviest stack

V1:

```
export function computeHeaviestStack(
  profile: Profile, range: SampleIndexRange
): Stack {
  const map = new Map();
  let heaviestStackWeight = 0;
  let heaviestStack: Stack = [];
  for (let i = range.start; i < range.end; i++) {
    const { stack, weight } = profile.samples[i];
    const stackJsonString = JSON.stringify(stack);
    const stackWeight =
      (map.get(stackJsonString) || 0) + weight;
    map.set(stackJsonString, stackWeight);
    if (stackWeight > heaviestStackWeight) {
      heaviestStackWeight = stackWeight;
      heaviestStack = stack;
    }
  }
  return heaviestStack;
}
```

~30000 nanoseconds per sample

V2:

```
export function computeHeaviestStackIndex(
  profile: Profile, range: SampleIndexRange
): number | null {
  const map = new Map();
  let heaviestStackWeight = 0;
  let heaviestStackIndex: number | null = null;
  for (let i = range.start; i < range.end; i++) {
    const { stackIndex, weight } = profile.samples[i];
    const stackWeight =
      (map.get(stackIndex) ?? 0) + weight;
    map.set(stackIndex, stackWeight);
    if (stackWeight > heaviestStackWeight) {
      heaviestStackWeight = stackWeight;
      heaviestStackIndex = stackIndex;
    }
  }
  return heaviestStackIndex;
}
```

103.7 nanoseconds per sample (300x faster)

V2: Computing the category breakdown

```ts
export function computeCategoryBreakdown(
  profile: Profile,
  range: SampleIndexRange
): CategoryBreakdown {
  const map = new Map();
  for (let i = range.start; i < range.end; i++) {
    const { stackIndex, weight } = profile.samples[i];
    const frameIndex = profile.stacks[stackIndex].frameIndex;
    const categoryIndex = profile.frames[frameIndex].categoryIndex;
    const category = profile.categories[categoryIndex];
    map.set(category, (map.get(category) || 0) + weight);
  }
  return map;
}
```

```ts
export type Profile = {
  samples: Sample[];
  stacks: StackNode[];
  frames: Frame[];
  categories: string[];
};

export type Sample = {
  time: number;
  stackIndex: number;
  weight: number;
};

export type StackNode = {
  parentStackIndex: number | null;
  frameIndex: number;
};

export type Frame = {
  name: string;
  categoryIndex: number;
};

export type CategoryBreakdown =
  Map<string, number>;

export type SampleIndexRange =
  { start: number; end: number };
```

V2: Computing the category breakdown

```ts
export function computeCategoryBreakdownWithIndexKeyMap(
  profile: Profile,
  range: SampleIndexRange
): Map<number, number> {
  const map = new Map();
  for (let i = range.start; i < range.end; i++) {
    const { stackIndex, weight } = profile.samples[i];
    const frameIndex = profile.stacks[stackIndex].frameIndex;
    const categoryIndex = profile.frames[frameIndex].categoryIndex;
    map.set(categoryIndex, (map.get(categoryIndex) || 0) + weight);
  }
  return map;
}
```

```ts
export type Profile = {
  samples: Sample[];
  stacks: StackNode[];
  frames: Frame[];
  categories: string[];
};

export type Sample = {
  time: number;
  stackIndex: number;
  weight: number;
};

export type StackNode = {
  parentStackIndex: number | null;
  frameIndex: number;
};

export type Frame = {
  name: string;
  categoryIndex: number;
};

export type CategoryBreakdown =
  Map<string, number>;

export type SampleIndexRange =
  { start: number; end: number };
```

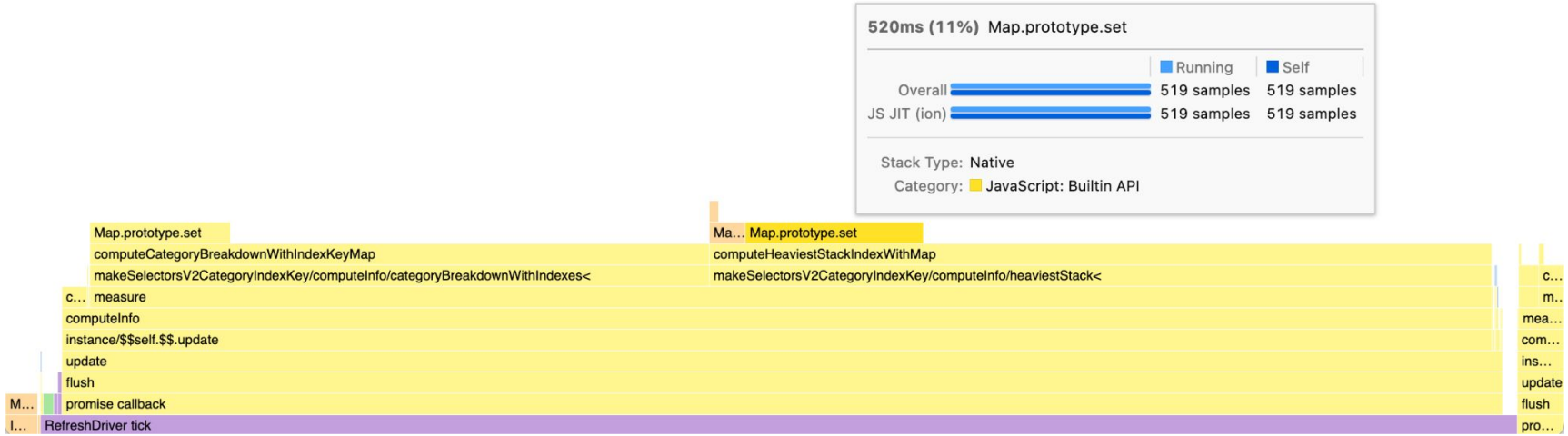
How fast is it now?

Throughput of computing the category breakdown: 47.1 nanoseconds per sample

Throughput of computing the heaviest stack: 51.1 nanoseconds per sample

(on medium-size V2 profile)

Time to use the profiler!


520ms (11%) Map.prototype.set
Running
Self
Overall
519 samples
519 samples
JS JIT (ion)
519 samples
519 samples
Stack Type: Native
Category: JavaScript: Builtin API
Map.prototype.set
Ma…
Map.prototype.set
computeCategoryBreakdownWithIndexKeyMap
computeHeaviestStackIndexWithMap
makeSelectorsV2CategoryIndexKey/computeInfo/categoryBreakdownWithIndexes<
makeSelectorsV2CategoryIndexKey/computeInfo/heaviestStack<
c…
measure
m…
computeInfo
mea…
instance/$$self.$$.update
com…
update
ins…
flush
update
M…
promise callback
flush
I…
RefreshDriver tick
pro…

Category breakdown: Replace Map

```ts
export function computeCategoryBreakdownWithIndexKeyMap(
  profile: Profile,
  range: SampleIndexRange
): Map<number, number> {
  const map = new Map();
  for (let i = range.start; i < range.end; i++) {
    const { stackIndex, weight } = profile.samples[i];
    const frameIndex = profile.stacks[stackIndex].frameIndex;
    const categoryIndex = profile.frames[frameIndex].categoryIndex;
    map.set(categoryIndex, (map.get(categoryIndex) || 0) + weight);
  }
  return map;
}
```

```ts
export type Profile = {
  samples: Sample[];
  stacks: StackNode[];
  frames: Frame[];
  categories: string[];
};

export type Sample = {
  time: number;
  stackIndex: number;
  weight: number;
};

export type StackNode = {
  parentStackIndex: number | null;
  frameIndex: number;
};

export type Frame = {
  name: string;
  categoryIndex: number;
};

export type CategoryBreakdown =
  Map<string, number>;

export type SampleIndexRange =
  { start: number; end: number };
```

47 nanoseconds

Category breakdown: Replace Map with typed array

```ts
export function computeCategoryBreakdownWithTypedArray(
  profile: Profile,
  range: SampleIndexRange
): Float64Array {
  const map = new Float64Array(profile.categories.length);
  for (let i = range.start; i < range.end; i++) {
    const { stackIndex, weight } = profile.samples[i];
    const frameIndex = profile.stacks[stackIndex].frameIndex;
    const categoryIndex = profile.frames[frameIndex].categoryIndex;
    map[categoryIndex] += weight;
  }
  return map;
}
```

```ts
export type Profile = {
  samples: Sample[];
  stacks: StackNode[];
  frames: Frame[];
  categories: string[];
};

export type Sample = {
  time: number;
  stackIndex: number;
  weight: number;
};

export type StackNode = {
  parentStackIndex: number | null;
  frameIndex: number;
};

export type Frame = {
  name: string;
  categoryIndex: number;
};

export type CategoryBreakdown =
  Map<string, number>;

export type SampleIndexRange =
  { start: number; end: number };
```

47 nanoseconds -> 16.3 nanoseconds (~3x faster)

Heaviest stack: Replace Map with typed array

```ts
export function computeHeaviestStackIndexWithTypedArray(
  profile: Profile, range: SampleIndexRange
): number | null {
  const map = new Float64Array(profile.stacks.length);
  let heaviestStackWeight = 0;
  let heaviestStackIndex: number | null = null;
  for (let i = range.start; i < range.end; i++) {
    const { stackIndex, weight } = profile.samples[i];
    const stackWeight = map[stackIndex] + weight;
    map[stackIndex] = stackWeight;
    if (stackWeight > heaviestStackWeight) {
      heaviestStackWeight = stackWeight;
      heaviestStackIndex = stackIndex;
    }
  }
  return heaviestStackIndex;
}
```

51.1 nanoseconds -> 16.3 nanoseconds (~3x faster)

```ts
export type Profile = {
  samples: Sample[];
  stacks: StackNode[];
  frames: Frame[];
  categories: string[];
};

export type Sample = {
  time: number;
  stackIndex: number;
  weight: number;
};

export type StackNode = {
  parentStackIndex: number | null;
  frameIndex: number;
};

export type Frame = {
  name: string;
  categoryIndex: number;
};

export type CategoryBreakdown =
  Map<string, number>;

export type SampleIndexRange =
  { start: number; end: number };
```

Where are we now?

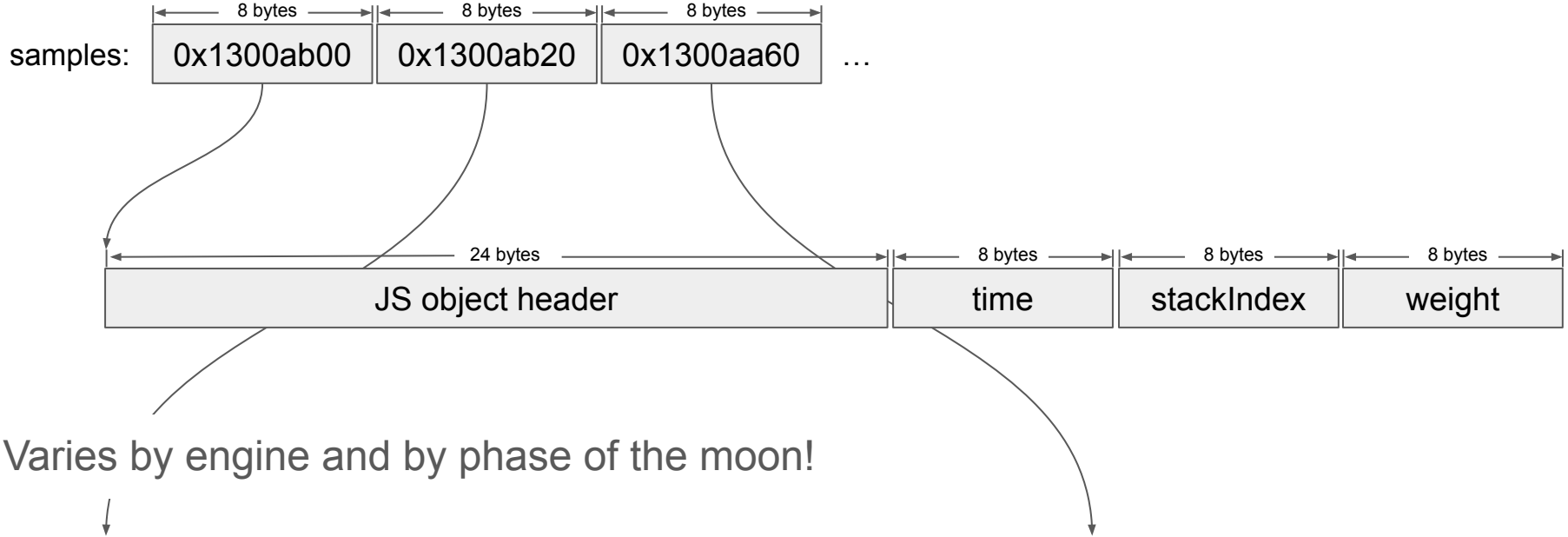
- We’ve addressed the obvious slowdowns:
  - Changed the format so that comparing stacks in cheap
  - Replaced two Maps with typed arrays for a 3x perf boost
    - In the “heaviest stack” case, this came at the cost of higher temporary memory usage
- Impressively fast already
  - 16ns per sample is not bad. Modern computers are beasts.
- Can we do better?

Category breakdown: Thinking about bytes in memory

```
export function computeCategoryBreakdownWithTypedArray(
  profile: Profile,
  range: SampleIndexRange
): Float64Array {
  const map = new Float64Array(profile.categories.length);
  for (let i = range.start; i < range.end; i++) {
    const { stackIndex, weight } = profile.samples[i];
    const frameIndex = profile.stacks[stackIndex].frameIndex;
    const categoryIndex = profile.frames[frameIndex].categoryIndex;
    map[categoryIndex] += weight;
  }
  return map;
}
```

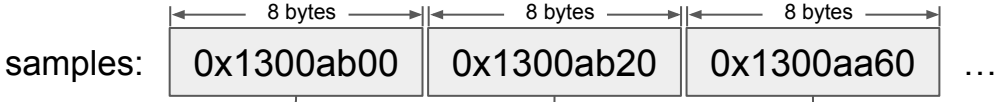
Arrays of objects: memory layout

```
for (let i = range.start; i < range.end; i++) {
  const { stackIndex, weight } = profile.samples[i];
  ...
}
```



Arrays of objects: memory layout

```
for (let i = range.start; i < range.end; i++) {
  const { stackIndex, weight } = profile.samples[i];
  ...
}
```



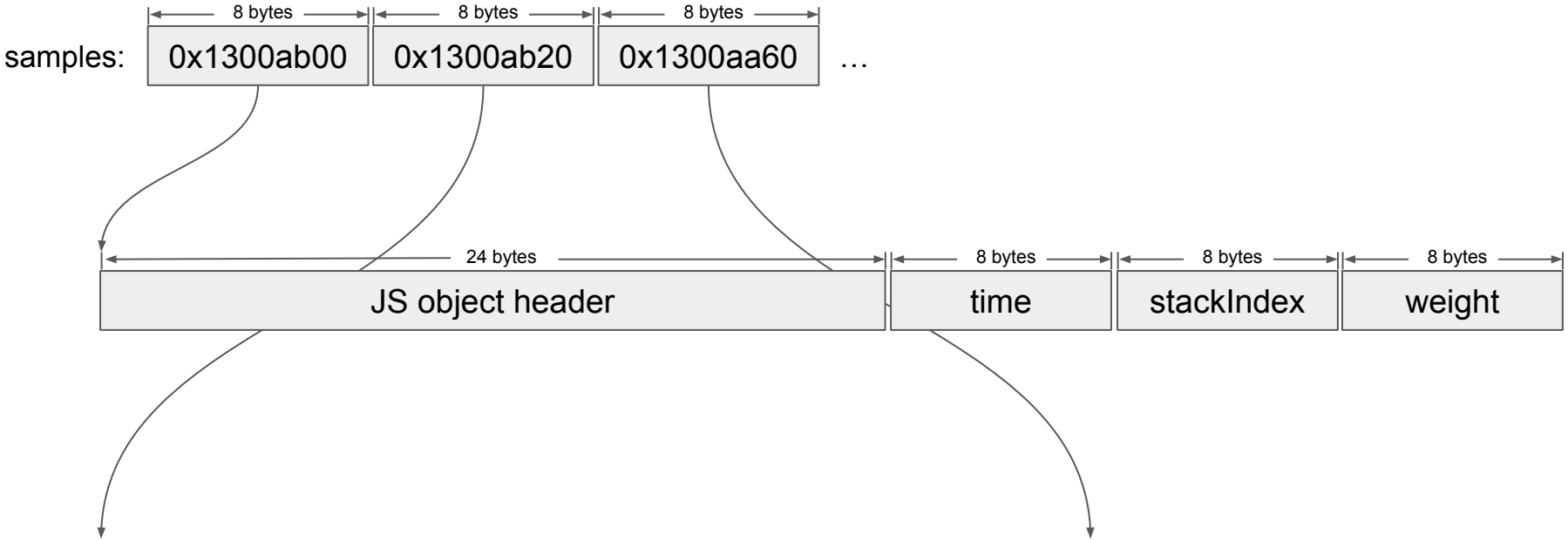
Some sources of variation:

- Pointer compression
- Object header size
- Fields could be inline or out-of-line
- Floating point numbers can be separate heap allocations
- Padding after inline fields

Varies by engine and by phase of the moon!

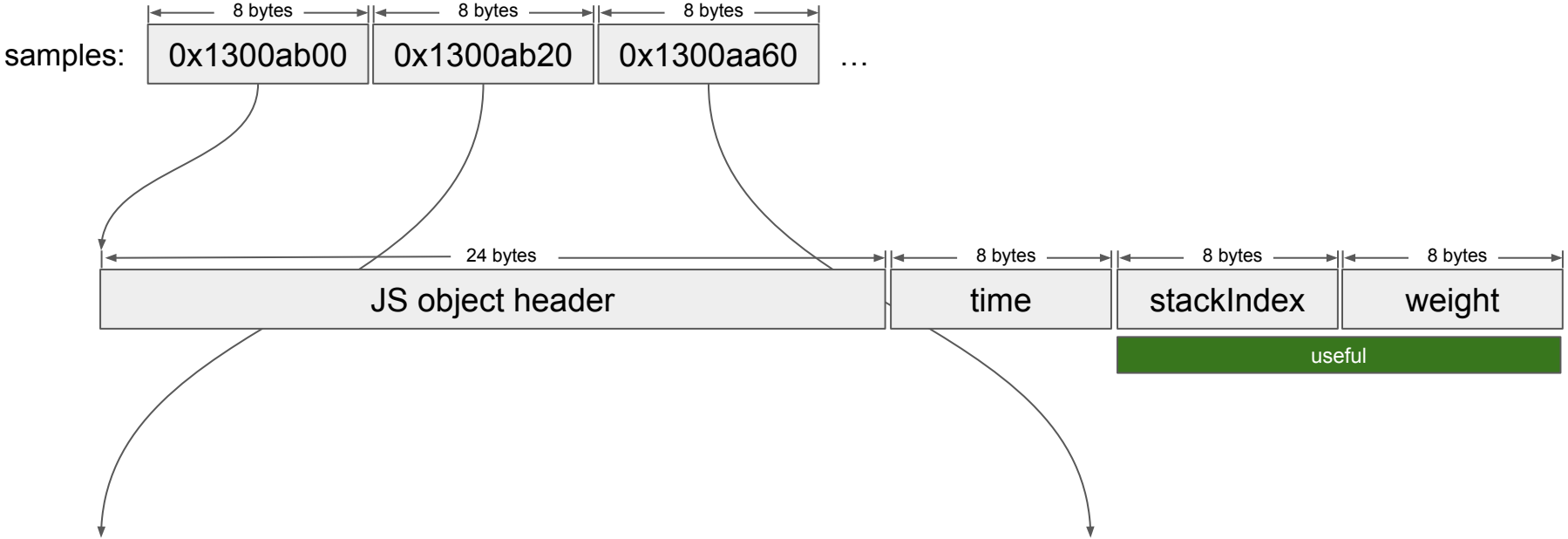
Arrays of objects: memory layout

```
for (let i = range.start; i < range.end; i++) {
  const { stackIndex, weight } = profile.samples[i];
  ...
}
```



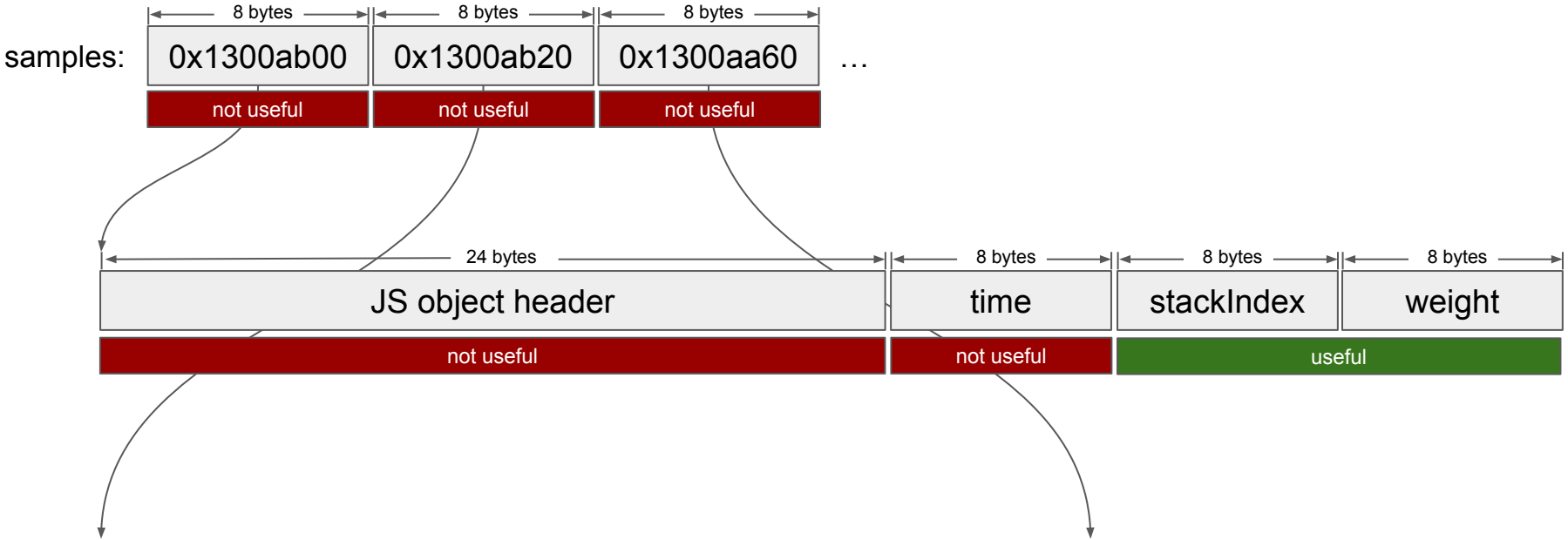
Arrays of objects: memory layout

```
for (let i = range.start; i < range.end; i++) {
  const { stackIndex, weight } = profile.samples[i];
  ...
}
```



Arrays of objects: memory layout

```
for (let i = range.start; i < range.end; i++) {
  const { stackIndex, weight } = profile.samples[i];
  ...
}
```



Arrays of objects: memory layout

- More indirection than we would like (1 extra dependent load)
- Too many wasted bytes: Only 16 useful bytes per 64 byte cache line

It’s time to do something radical.

Turning it on the side

Before:

```
  "samples": [
    { "time": 17880.8104, "stackIndex": 27, "weight": 643 },
    { "time": 17721.0814, "stackIndex": 87, "weight": 1 },
    { "time": 17880.9783, "stackIndex": 93, "weight": 1 },
    { "time": 17881.569, "stackIndex": 102, "weight": 1 },
    { "time": 17881.8177, "stackIndex": 143, "weight": 1 },
    { "time": 17882.0657, "stackIndex": 190, "weight": 1 }
    ...
  ]
```

After:

```
  "sampleTable": {
    "length": 1691,
    "timeColumn": [17880.8104, 17721.0814, 17880.9783, 17881.569, 17881.8177, 17882.0657, ...],
    "stackIndexColumn": [27, 87, 93, 102, 143, 190, ...],
    "weightColumn": [643, 1, 1, 1, 1, 1, 1, ...]
  }
```

Turning it on the side: Everything is backwards

```
Before:  profile.samples[i].weight
After:   profile.sampleTable.weightColumn[i]
```

Toy profile format, V3: Tables everywhere

```
{
  "sampleTable": {
    "length": 1691,
    "timeColumn": [ 17880.8104, 17881.1623, 17881.4107, ... ],
    "stackIndexColumn": [ 27, 27, 27, ... ],
    "weightColumn": [ 643, 1, 1, ... ]
  },
  "stackTable": {
    "length": 25776,
    "parentStackIndexColumn": [ null, 0, 1, ... ],
    "frameIndexColumn": [ 24, 23, 22, ... ]
  },
  "frameTable": {
    "length": 6242,
    "nameColumn": [ "ZwWaitForAlertByThreadId", ... ],
    "categoryIndexColumn": [ 0, 0, 0, ... ]
  },
  "categories": [ "User", "Kernel", "Trampoline", ... ]
}
```

```
export type Profile = {
  sampleTable: SampleTable;
  stackTable: StackTable;
  frameTable: FrameTable;
  categories: string[];
};

export type SampleTable = {
  length: number;
  timeColumn: number[];
  stackIndexColumn: number[];
  weightColumn: number[];
};

export type StackTable = {
  length: number;
  parentStackIndexColumn: Array<number | null>;
  frameIndexColumn: number[];
};

export type FrameTable = {
  length: number;
  nameColumn: string[];
  categoryIndexColumn: number[];
};
```

V3: Computing the heaviest stack

```
function computeHeaviestStackIndexBasic(
  profile: Profile, range: SampleIndexRange
): number | null {
  const map = new Float64Array(profile.stackTable.length);
  let heaviestStackWeight = 0;
  let heaviestStackIndex: number | null = null;
  for (let i = range.start; i < range.end; i++) {
    const stackIndex = profile.sampleTable.stackIndexColumn[i];
    const weight = profile.sampleTable.weightColumn[i];
    const stackWeight = map[stackIndex] + weight;
    map[stackIndex] = stackWeight;
    if (stackWeight > heaviestStackWeight) {
      heaviestStackWeight = stackWeight;
      heaviestStackIndex = stackIndex;
    }
  }
  return heaviestStackIndex;
}
```

16.3 nanoseconds -> 8.7 nanoseconds (~2x faster)

V3: Computing the category breakdown

```
function computeCategoryBreakdownBasic(
  profile: Profile,
  range: SampleIndexRange
): Float64Array {
  const map = new Float64Array(profile.categories.length);
  for (let i = range.start; i < range.end; i++) {
    const stackIndex = profile.sampleTable.stackIndexColumn[i];
    const weight = profile.sampleTable.weightColumn[i];
    const frameIndex = profile.stackTable.frameIndexColumn[stackIndex];
    const categoryIndex = profile.frameTable.categoryIndexColumn[frameIndex];
    map[categoryIndex] += weight;
  }
  return map;
}
```

16.3 nanoseconds -> 4.6 nanoseconds (~3.5x faster!)

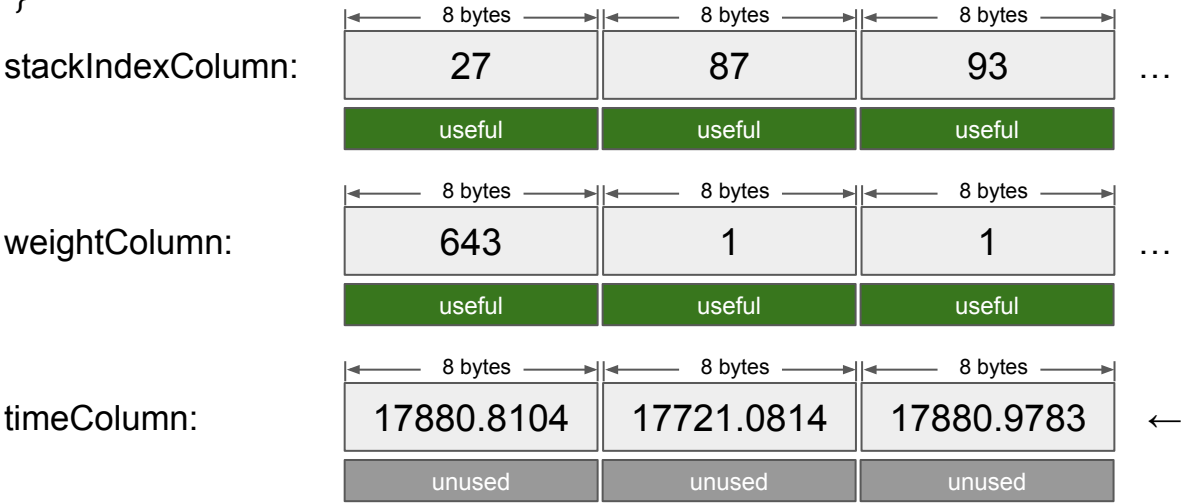
V3: Computing the category breakdown

```
function computeCategoryBreakdownBasic(
  profile: Profile,
  range: SampleIndexRange
): Float64Array {
  const map = new Float64Array(profile.categories.length);
  for (let i = range.start; i < range.end; i++) {
    const stackIndex = profile.sampleTable.stackIndexColumn[i];
    const weight = profile.sampleTable.weightColumn[i];
    const frameIndex = profile.stackTable.frameIndexColumn[stackIndex];
    const categoryIndex = profile.frameTable.categoryIndexColumn[frameIndex];
    map[categoryIndex] += weight;
  }
  return map;
}
```

16.3 nanoseconds -> 4.6 nanoseconds (~3.5x faster!)

Struct of arrays: memory layout

```
for (let i = range.start; i < range.end; i++) {
  const stackIndex = profile.sampleTable.stackIndexColumn[i];
  const weight = profile.sampleTable.weightColumn[i];
  ...
}
```



Cache line utilization went way up!

Recap: struct of arrays

- Also called “structure of arrays” or “parallel arrays”
- Common strategy in game engines, databases, generally high-performance use cases

Recap: struct of arrays

Drawbacks:

- Less familiar “backwards” code
- Sometimes you have to manually materialize objects
- Type system catches fewer mistakes:
  - mismatched lengths
  - index confusion

Benefits:

- Much more cache-efficient
- Easier on the garbage collector (fewer objects to traverse, some engines skip the contents of arrays of numbers)
- Less memory overhead from object headers and padding
- Can change or add individual columns without having to touch other columns
- In JS: Control over integer sizes / float precision with typed arrays

Great work.

… Can we make it even faster?

Category breakdown: Less indirection?

```
function computeCategoryBreakdownBasic(
  profile: Profile,
  range: SampleIndexRange
): Float64Array {
  const map = new Float64Array(profile.categories.length);
  for (let i = range.start; i < range.end; i++) {
    const stackIndex = profile.sampleTable.stackIndexColumn[i];
    const weight = profile.sampleTable.weightColumn[i];
    const frameIndex = profile.stackTable.frameIndexColumn[stackIndex];
    const categoryIndex = profile.frameTable.categoryIndexColumn[frameIndex];
    map[categoryIndex] += weight;
  }
  return map;
}
```

We just want to know the category for a sample. We're not interested in its stack or frame.

Category breakdown: Less indirection!

```
function computeCategoryBreakdownWithPrecomputedSampleCategoriesRegularArray(
  profile: Profile,
  sampleCategories: number[],
  range: SampleIndexRange
): Float64Array {
  const map = new Float64Array(profile.categories.length);
  for (let i = range.start; i < range.end; i++) {
    const weight = profile.sampleTable.weightColumn[i];
    const categoryIndex = sampleCategories[i];
    map[categoryIndex] += weight;
  }
  return map;
}
```

The mapping from samples to categories is independent of the selected range.

Computing the sample categories

```
const getSampleCategories = (profile: Profile): number[] => {
  const sampleCategories = new Array(profile.sampleTable.length);
  for (let i = 0; i < sampleCategories.length; i++) {
    const stackIndex = profile.sampleTable.stackIndexColumn[i];
    const frameIndex = profile.stackTable.frameIndexColumn[stackIndex];
    const categoryIndex = profile.frameTable.categoryIndexColumn[frameIndex];
    sampleCategories[i] = categoryIndex;
  }
  return sampleCategories;
};

... = computeCategoryBreakdownWithPrecomputedSampleCategoriesRegularArray(
  profile,
  getSampleCategories(profile),
  selectedRange
);
```

Computing the sample categories only once

```
import memoize from 'memoize-one';

const getSampleCategories = memoize((profile: Profile): number[] => {
  const sampleCategories = new Array(profile.sampleTable.length);
  for (let i = 0; i < sampleCategories.length; i++) {
    const stackIndex = profile.sampleTable.stackIndexColumn[i];
    const frameIndex = profile.stackTable.frameIndexColumn[stackIndex];
    const categoryIndex = profile.frameTable.categoryIndexColumn[frameIndex];
    sampleCategories[i] = categoryIndex;
  }
  return sampleCategories;
});

... = computeCategoryBreakdownWithPrecomputedSampleCategoriesRegularArray(
  profile,
  getSampleCategories(profile),
  selectedRange
);
```

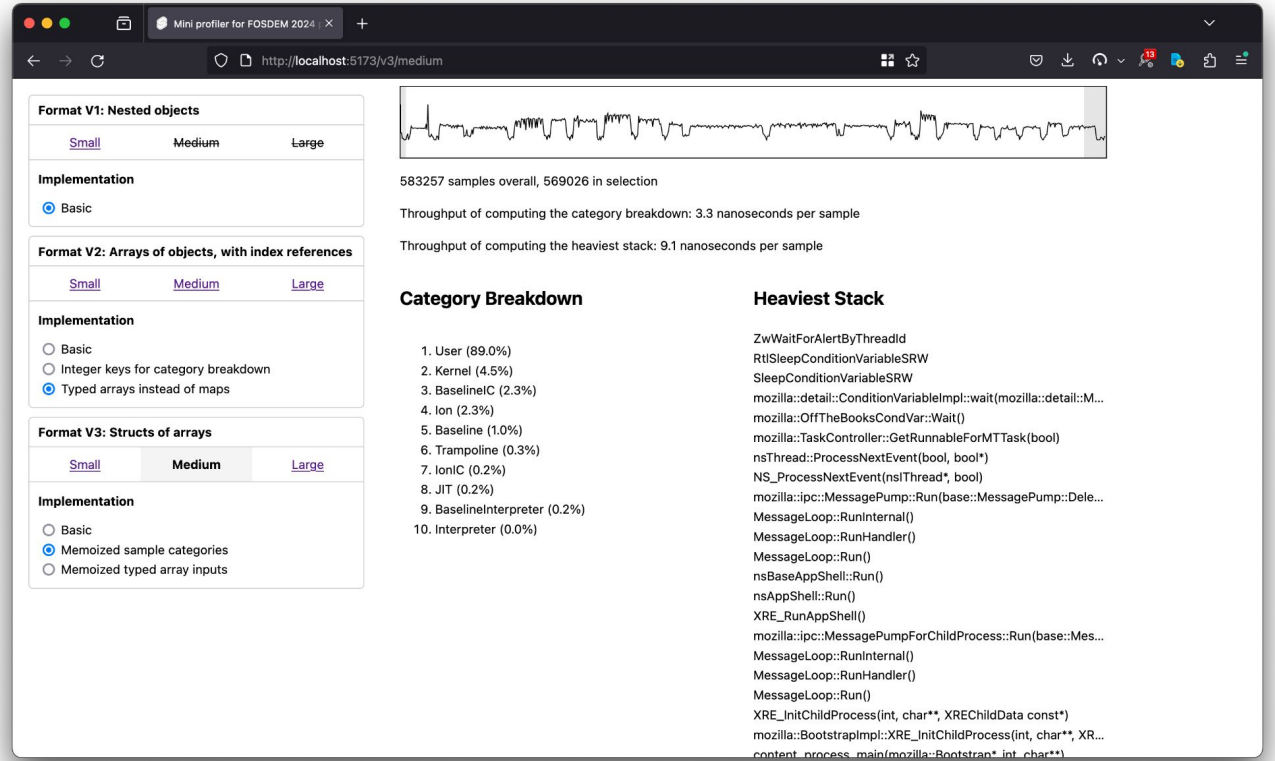
Computing the sample categories only once

```
import memoize from 'memoize-one';

const getSampleCategories = (profile: Profile): number[] => getSampleCategoriesFromColumns(
    profile.sampleTable.stackIndexColumn,
    profile.stackTable.frameIndexColumn,
    profile.frameTable.categoryIndexColumn
  );

const getSampleCategoriesFromColumns = memoize(
  (sampleStacks: number[], stackFrames: number[], frameCategories: number[]): number[] => {
    const sampleCategories = new Array(sampleStacks.length);
    for (let i = 0; i < sampleCategories.length; i++) {
      const stackIndex = sampleStacks[i];
      const frameIndex = stackFrames[stackIndex];
      const categoryIndex = frameCategories[frameIndex];
      sampleCategories[i] = categoryIndex;
    }
    return sampleCategories;
  });
```

… Did it work?


Mini profiler for FOSDEM 2024
http://localhost:5173/v3/medium
Format V1: Nested objects
Small
Medium
Large
Implementation
Basic
Format V2: Arrays of objects, with index references
Small
Medium
Large
Implementation
Basic
Integer keys for category breakdown
Typed arrays instead of maps
Format V3: Structs of arrays
Small
Medium
Large
Implementation
Basic
Memoized sample categories
Memoized typed array inputs
583257 samples overall, 569026 in selection
Throughput of computing the category breakdown: 3.3 nanoseconds per sample
Throughput of computing the heaviest stack: 9.1 nanoseconds per sample
Category Breakdown
1. User (89.0%)
2. Kernel (4.5%)
3. BaselineIC (2.3%)
4. Ion (2.3%)
5. Baseline (1.0%)
6. Trampoline (0.3%)
7. IonIC (0.2%)
8. JIT (0.2%)
9. BaselineInterpreter (0.2%)
10. Interpreter (0.0%)
Heaviest Stack
ZwWaitForAlertByThreadId
RtlSleepConditionVariableSRW
SleepConditionVariableSRW
mozilla::detail::ConditionVariableImpl::wait(mozilla::detail::M...
mozilla::OffTheBooksCondVar::Wait()
mozilla::TaskController::GetRunnableForMTTask(bool)
nsThread::ProcessNextEvent(bool, bool*)
NS_ProcessNextEvent(nsIThread*, bool)
mozilla::ipc::MessagePump::Run(base::MessagePump::Dele...
MessageLoop::RunInternal()
MessageLoop::RunHandler()
MessageLoop::Run()
nsBaseAppShell::Run()
nsAppShell::Run()
XRE_RunAppShell()
mozilla::ipc::MessagePumpForChildProcess::Run(base::Mes...
MessageLoop::RunInternal()
MessageLoop::RunHandler()
MessageLoop::Run()
XRE_InitChildProcess(int, char**, XREChildData const*)
mozilla::BootstrapImpl::XRE_InitChildProcess(int, char**, XR...

It worked!

4.6 nanoseconds → 3.3 nanoseconds (30% faster!)

It seems like we’re no longer memory bound!

The speedup can be higher on machines with slower memory.

Taking a step back

What just happened?

- We noticed that some of the work inside a loop was redundant: looking up the category index for a sample
- We created an additional column as a shortcut: `sampleCategories`
- We identified the “source” columns that the shortcut depends on
- We cached the derived column with precise dependency tracking
- Changing unrelated columns will not invalidate the derived column!

Struct of arrays with immutable columns + memoization = ❤️

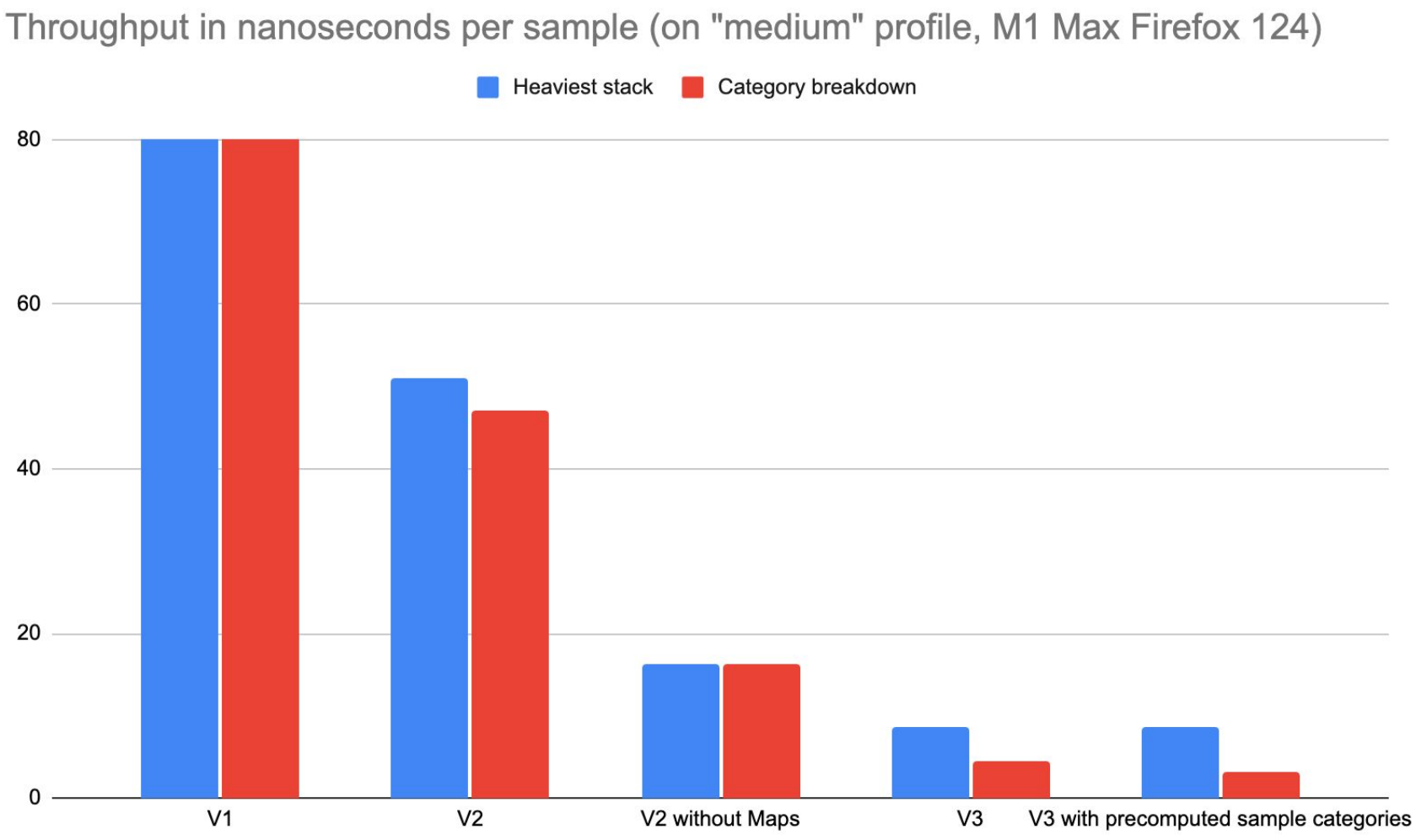
Taking another step back

What is Data-oriented Design?

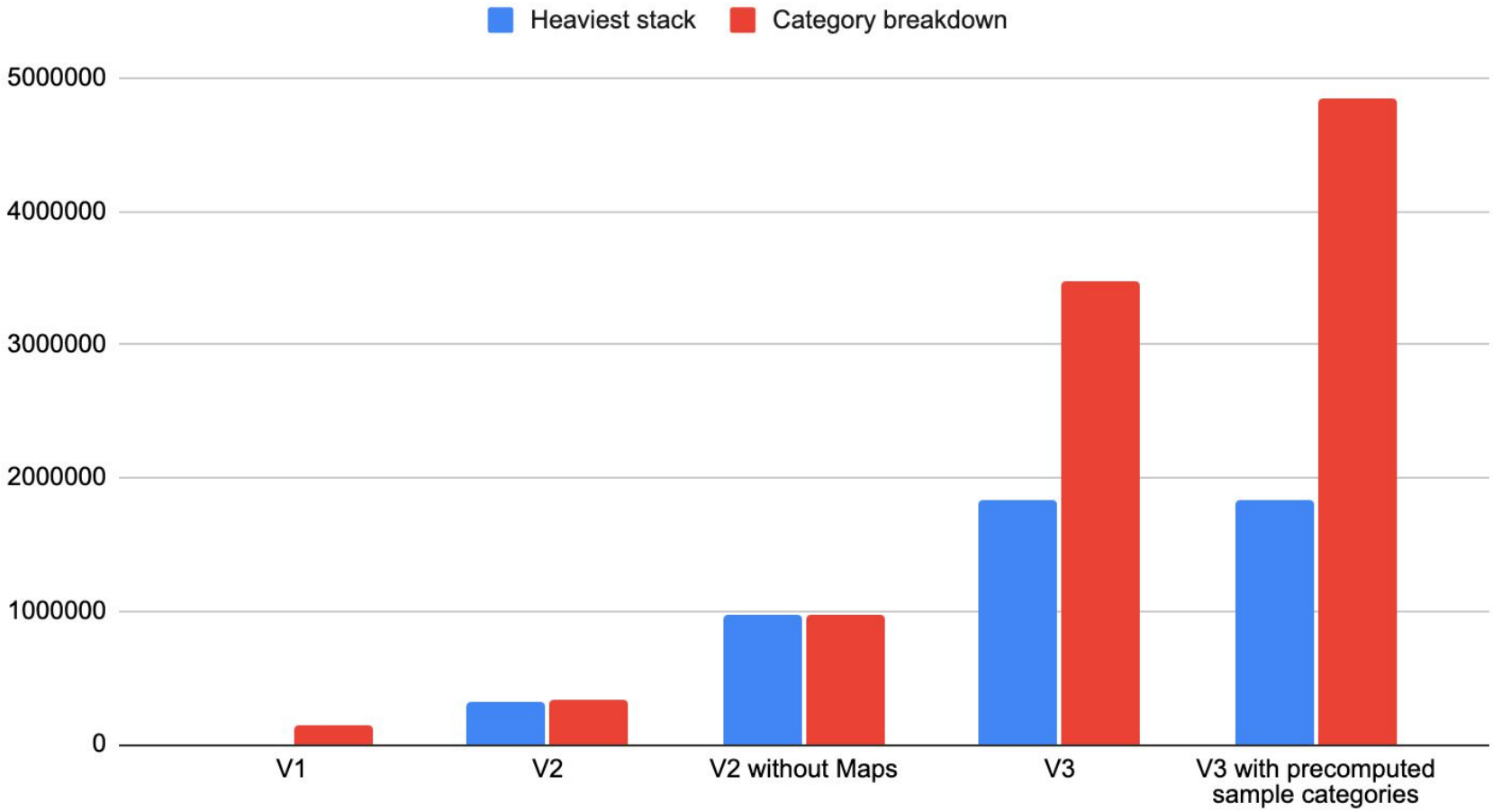
- Mindset:
  - The shape of the data determines the algorithm and its performance.
  - Know where your data is: We usually have 7 of thing A and 7000000 of thing B
  - Keep the in-memory representation in mind, and think about cache line utilization.
- Collection of techniques:
  - Struct of arrays is the main one.
  - Write algorithms as transformations of input columns into output columns.
  - Sometimes consider choosing smaller integer sizes when the domain is restricted.
    - E.g. Int32Array for indexes into another table, if that table can never have more than 2 billion items

General Remarks

- This is a somewhat niche technique.
- Works best with fixed-size data.
- Stress-test your code with large inputs.
  - Generate artificial large inputs if you don’t have real inputs that are large enough.
- Large inputs unveil new bottlenecks and new optimization opportunities.
- There are usually more important things to fix! Do the algorithmic optimizations first.
  - If you have an N^2 algorithm lurking somewhere, your performance is ruined even if that algorithm is very cache-efficient.
- This is another tool in your toolbox.
- The profiler is your friend. Use it as much as you can.


Throughput in nanoseconds per sample (on "medium" profile, M1 Max Firefox 124)
Heaviest stack
Category breakdown
80
60
40
20
0
V1
V2
V2 without Maps
V3
V3 with precomputed sample categories

How many samples can we process in 16ms?



Thank You

@mstange:mozilla.org

https://github.com/mstange/

Install the Firefox Profiler: https://profiler.firefox.com/

Talk to profiler people on https://chat.mozilla.org/#/room/#profiler:mozilla.org

Happy profiling!