Optimizer Counter Analysis

It is possible by means of providing some special command-line options to ask the Swift compiler to produce different statistics about the optimizer counters. Optimizer counters are most typically counters representing different aspects of a SIL representation for a Swift module being compiled, e.g. the number of SIL basic blocks or instructions. These counters may also reveal some details about transformations and optimizations performed on SIL, e.g. the duration of an optimization or how much memory was consumed by the compiler. Therefore having the information about the changes of optimizer counters over time allows for analysis of changes to the SIL representation during the compilation.

This document describes how you collect and analyze the counters produced by the optimizer of the Swift compiler. This analysis is useful if you need to get a detailed insight into the optimizer passes, their effect on the SIL code, compile times, compiler's memory consumption, etc. For example, you can find out which optimization passes or phases of optimization produce most new SIL instructions or delete most SIL functions.

Table of contents

Which optimizer counters are available for recording

The following statistics can be recorded:

- For SILFunctions: the number of SIL basic blocks for each SILFunction, the number of SIL instructions, the number of SILInstructions of a specific kind (e.g. a number of alloc_ref instructions)
- For SILModules: the number of SIL basic blocks in the SILModule, the number of SIL instructions, the number of SILFunctions, the number of SILInstructions of a specific kind (e.g. a number of alloc_ref instructions) the amount of memory used by the compiler.

How and when the optimizer counters are collected and recorded

The pass manager of the SIL optimizer invokes special hooks before and after it executes any optimization transformation (often called an optimization pass).

The hook checks if there are any changes of counter values since the last time it was invoked. If there are any changes, then the counters are re-computed. They are first re-computed for SILFunctions and then for the SILModule being compiled. The re-computation algorithm is trying to be incremental and fast by re-computing only the minimal amount of counters instead of scanning the whole SILModule every time.

Those counters that are changed are reported if they satisfy different filtering conditions, which are configurable. For example, you may want to see only counters that have changed by more than 50% since last time. Alternatively, you may want to log as many counters as possible, in which case they will be logged on every invocation of the hook.

If there were no changes to the counters that satisfy the filtering conditions, those changes would not be logged. This means that the final set of logged changes may be incomplete, i.e. it would not reflect all changes that happened to those counters.

Differences between different modes of optimizer counters collection

You can collect optimizer counters at different levels of granularity depending on your needs. The granularity also affects the amount of recorded data that will be produced, because the amount of recorded data grows if you decide to collect more fine-grained counters.

Module level counters

The most coarse-grained counters are the module level counters, which are enabled by using `-Xllvm -sil-stats-modules` command-line option. They are usually logged only if a given optimizer counter changed a lot for the whole SILModule, which does not happen that often, because most optimization passes perform just very small transformations on a single function and thus do not significantly change any module-wide counters.

Function level counters

The next level of granularity are SILFunction counters, which are enabled by using `-Xllvm -sil-stats-functions` command-line option. They track statistics for SILFunctions. Every SILFunction has its own set of these counters. Obviously, interesting changes to these counters happen more often than to the module-wide counters.

Instruction level counters

The finest level of granularity are SILInstruction counters, which are enabled by using `-Xllvm -sil-stats-only-instructions` command-line option. You can use them to e.g. collect statistics about how many specific SIL instructions are used by a given SILFunction or a SILModule. For example, you can count how many `alloc_ref` instructions occur in a given SILFunction or SILModule. If you are interested in collecting the stats only for some specific SIL instructions, you can use a comma-separated list of instructions as a value of the option, e.g. `-Xllvm -`

`sil-stats-only-instructions=alloc_ref,alloc_stack`. If you need to collect stats about all kinds of SIL instructions, you can use this syntax: `-Xllvm -sil-stats-only-instructions=all`.

Configuring which counters changes should be recorded

The user has a possibility to configure a number of thresholds, which control what needs to be recorded. Many of those thresholds are formulated for the deltas, e.g. the delta should be more than 50%.

The value of the delta for a given old and new values of a counter are computed using the following simple formula:

```
delta = 100% * (new_value - old_value)/old_value
```

So, a delta basically reflects how big is the change of the counter value when expressed as a percentage of the old value.

TBD Provide more information about different command-line options for configuring the thresholds.

On-line and off-line processing of optimizer counters

As explained above, some of the counters filtering happens on-line at compile-time already. If this is enough for your purposes, you can use them "as is".

But in many cases, you may want to perform more complex analysis of the collected counters, e.g. you may want to aggregate them by the optimization pass or by a stage of the optimization pipeline, etc. This is not directly supported by the on-line filtering mode. Instead, you can record all the interesting counters and then post-process it off-line by first storing them into a SQLite database by means if a special utility and then using the regular SQL queries to perform any kind of analysis and aggregation that you may need.

Specifying where the optimizer counters should be stored

By default, all the collected statistics are written to the standard error.

But it is possible to write into a custom file by specifying the following command-line option:

```
-Xllvm -sil-stats-output-file=your_file_name
```

The format of the recorded optimizer counters

The counters are recorded using a simple CSV (comma separated value) format. Each line represents a single counter value or a counter value change.

For counter value updates, the CSV line looks like this:

- `Kind, CounterName, StageName, TransformName, TransformPassNumber, DeltaValue, OldCounterValue, NewCounterValue, Duration, Symbol`

And for counter stats it looks like this:

- `Kind, CounterName, StageName, TransformName, TransformPassNumber, CounterValue, Duration, Symbol`

where the names used above have the following meaning:

- `Kind` is one of `function`, `module`, `function_history`.
  - `function` and `module` correspond directly to the module-level and function-level counters

    - `function_history` corresponds to the verbose mode of function counters collection, when changes to the SILFunction counters are logged unconditionally, without any on-line filtering.
- `CounterName` is typically one of `block`, `inst`, `function`, `memory`, or `inst_instruction_name` if you collect counters for specific kinds of SIL instructions.
- `Symbol` is e.g. the name of a function
- `StageName` is the name of the current optimizer pipeline stage
- `TransformName` is the name of the current optimizer transformation/pass
- `Duration` is the duration of the transformation
- `TransformPassNumber` is the optimizer pass number. It is useful if you want to reproduce the result later using `-Xllvm -sil-opt-pass-count -Xllvm TransformPassNumber`

Storing the produced statistics into a database

To store the set of produced counters into a database, you can use the following command:

```
utils/optimizer_counters_to_sql.py csv_file_with_counters your_database.db
```

The database schema for counters

The database uses a very simple self-explaining schema:

```
CREATE TABLE Counters(
  Id INTEGER PRIMARY KEY AUTOINCREMENT,
  Stage TEXT NOT NULL,
  Transform TEXT NOT NULL,
  Kind TEXT,
  Counter TEXT NOT NULL,
  PassNum INT NOT NULL,
  Delta NUMBER,
  Old INT,
  New INT,
  Duration INT,
  Symbol TEXT NOT NULL DEFAULT '');
```

Analyzing collected counters using SQL

First, you need to connect to your database, so that you can issue SQL queries. This can be accomplished e.g. by the following command:

```
sqlite3 your_database.db
```

After executing this command, you will be presented with a SQLite command prompt and you can start entering SQL queries. Each query should end with a semicolon.

Examples of interesting SQL queries

SQL gives you a lot of freedom to perform different kinds of analysis. Below you can find some examples of typical queries, which you can use "as is" or as a basis for formulating more complex queries.

Compute aggregate times for each optimization pass/transformation

```
select C.Transform, sum(C.Duration)
from Counters C
where C.counter = 'inst' and C.kind = 'module'
group by C.Transform;
```

Which pass created/deleted most functions?

```
select C.Transform, sum(C.Delta)
from Counters C
where C.counter = 'functions' and C.kind = 'module'
group by C.Transform;
```

Which pass at which optimization pipeline stage created/deleted most functions?

```
select C.Stage, C.Transform, sum(C.Delta)
from Counters C where C.counter = 'functions' and C.kind = 'module'
group by C.Stage, C.Transform;
```

Which pass created/removed most instructions?

```
# Sort by biggest changes
select C.Transform, sum(C.Delta)
from Counters C where C.counter = 'inst' and C.kind = 'module'
group by C.Transform
order by abs(sum(C.Delta));
```

Which pass at which stage created/removed most instructions?

```
# Sort by biggest changes
select C.Stage, C.Transform, sum(C.Delta)
from Counters C where C.counter = 'inst' and C.kind = 'module'
group by C.Stage, C.Transform
order by abs(sum(C.Delta));
```

Which functions were changed most by which stage when it comes to the instruction counts

```
select C.Stage, min(C.Old), max(C.Old), Symbol
from Counters C where C.counter = 'inst' and C.kind = 'function_history'
group by C.Symbol, C.Stage
having min(C.Old) <> max(C.Old)
order by abs(max(C.Old)-min(C.Old));
```

Get the number of instructions at the beginning and at the end of the optimization pipeline for each function

```
select MinOld.Id, MinOld.Old, MaxOld.Id, MaxOld.Old, MinOld.Symbol
from
```

```
(
  select C.Id, C.Old, C.Symbol
  from Counters C where C.counter = 'inst' and C.kind = 'function_history'
  group by C.Symbol
  having C.Id = max(Id)
) as MaxOld,
(select C.Id, C.Old, C.Symbol
   from Counters C
   where C.counter = 'inst' and C.kind = 'function_history'
   group by C.Symbol
   having C.Id = min(Id)
) as MinOld
where MinOld.Symbol == MaxOld.Symbol;
```

Show functions which have the biggest size at the end of the optimization pipeline

```
select MinOld.Id, MinOld.Old, MaxOld.Id, MaxOld.Old, MinOld.Symbol
from
(
  select C.Id, C.Old, C.Symbol
  from Counters C
  where C.counter = 'inst' and C.kind = 'function_history'
  group by C.Symbol
  having C.Id = max(Id)
) as MaxOld,
(
  select C.Id, C.Old, C.Symbol
  from Counters C
  where C.counter = 'inst' and C.kind = 'function_history'
  group by C.Symbol
  having C.Id = min(Id)
) as MinOld
where MinOld.Symbol == MaxOld.Symbol
order by MaxOld.Old;
```

Which optimization pipeline stage took most time?

```
select sum(Duration), Stage
from Counters C
where C.counter = 'inst' and C.kind = 'module'
group by Stage
order by sum(C.Duration);
```

Which stage added/removed most instructions (in term of deltas)?

```
select sum(Delta), Stage
from Counters C where C.counter = 'inst' and C.kind = 'module'
group by Stage
order by sum(C.Delta);
```