Introducing a New Option Processing Mechanism for the OMR Compiler

Nazim Uddin Bhuiyan

github: nbhuiyan

Some motivations behind this rework…

- The OMR::Options class a giant confusing mess doing a lot more than its name would suggest
- Adding a new option was not straightforward, often required making changes in multiple locations
- Many downstream options exist within OMR
- and many more…

Initial contribution: PR #3675

- OMR PR #3675 Introduces the main components of the new option processing mechanism
- Only implements boolean options that were previously using flags with the help of enum CompilationOptions to identify the options
- First of a number of PRs to come that will add more option processing capabilities (eg, setting non-boolean options, option set handling, etc)

Some major design decisions made in the new option processing mechanism …

- Boolean options are not stored in a single array using option flags, instead, each option gets a boolean field
- Members to store options data, member initializer list, as well as the option table are generated by a python script at build time. Data to generate these files are obtained from .json files
- Option table used for matching command-line/env option strings is a hash table that is laid out at build time

The OMR::Options::_options[] field

- OMR::Options uses option flags defined in enum CompilationOptions to operate on the _options[] field
- Every boolean option needs a spot within enum CompilationOptions’s fixed range of values
- Adding a new boolean option would require finding an available spot in the enum
- The current implementation forces downstream projects to add their project-specific options in this enum. This is problematic because there is a maximum limit for the number of options that can exist in this enum (about 830)
- Querying boolean options from this array using bitwise operations is expensive compared to looking up boolean fields

Using boolean fields instead…

- No limit on the number of boolean options a compiler project can have
- Enables a mechanism for allowing project-specific options to stay out of OMR
- 2-3x faster querying of boolean fields vs bitwise operations to extract options from OMR::Options::_options[]. This comparison was made by making 1 billion queries of different options in the 2 different implementations
- Individual boolean fields increase the size of the options object, but this is justified because querying the fields require fewer instructions

Components of the new Option processing framework being introduced in PR #3675

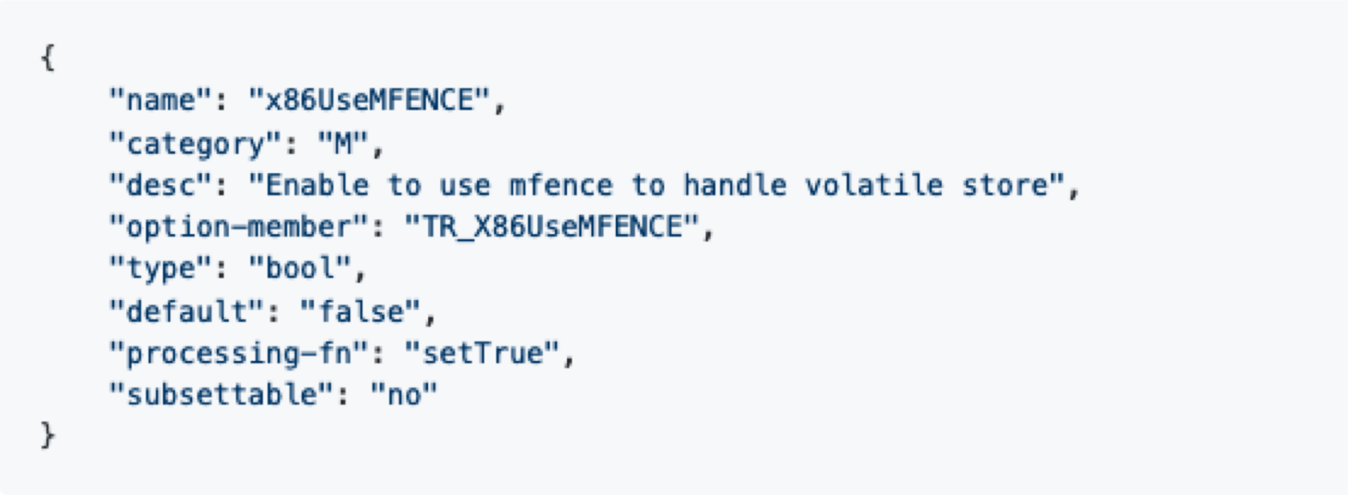
- OMROptions.json
- options-gen
- Generated option entries table
- OptionProcessors
- OptionsBuilder
- CompilerOptions
- CompilerOptionsManager

OMROptions.json

- Contains all the OMR compiler options as an array of JSON objects
- Downstream projects will have their own .json file for their options
- Entries in this file gets processed at build time
- A new option can be added by simply making a new entry in this file and re-building
- Allows us to specify default values for options to reduce the number of options that are always set to the same value at runtime

OMROptions.json

Here is how an entry in OMROptions.json looks like



```
{
    "name": "x86UseMFENCE",
    "category": "M",
    "desc": "Enable to use mfence to handle volatile store",
    "option-member": "TR_X86UseMFENCE",
    "type": "bool",
    "default": "false",
    "processing-fn": "setTrue",
    "subsettable": "no"
}
```

options-gen

- A python script that runs at build time
- Responsible for processing the JSON data
- Generates multiple .inc files that are included in CompilerOptions and CompilerOptionsManager classes:
  - Options.inc
  - OptionInitializerList.inc
  - OptionTableEntries.inc
  - OptionTableProperties.inc
  - OptionCharMap.inc

options-gen

- A python script that runs at build time
- Responsible for processing the JSON data
- Generates multiple .inc files that are included in CompilerOptions and CompilerOptionsManager classes:
  - Options.inc
  - OptionInitializerList.inc
  - OptionTableEntries.inc
  - OptionTableProperties.inc
  - OptionCharMap.inc

Used for building the CompilerOptions class

options-gen

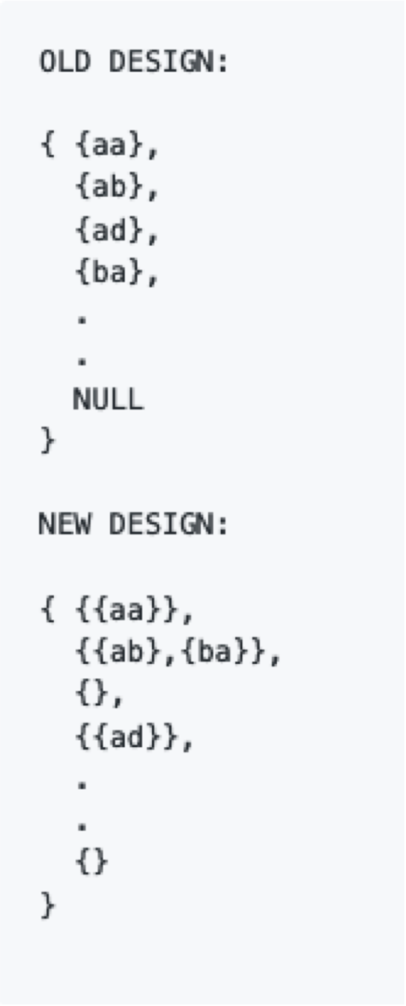
- A python script that runs at build time
- Responsible for processing the JSON data
- Generates multiple .inc files that are included in CompilerOptions and CompilerOptionsManager classes:
  - Options.inc
  - OptionInitializerList.inc
  - OptionTableEntries.inc
  - OptionTableProperties.inc
  - OptionCharMap.inc

Used for building the option entries table

New Option Table

- The new option table design is different from the existing one, where matching option strings to the entries are done by a hash function
- The generated hash table is a 2d array of struct OptionTableItem, with every row representing a bucket in the hash table
- The generated file OptionTableEntries.inc contains braced initializer list of the table entries

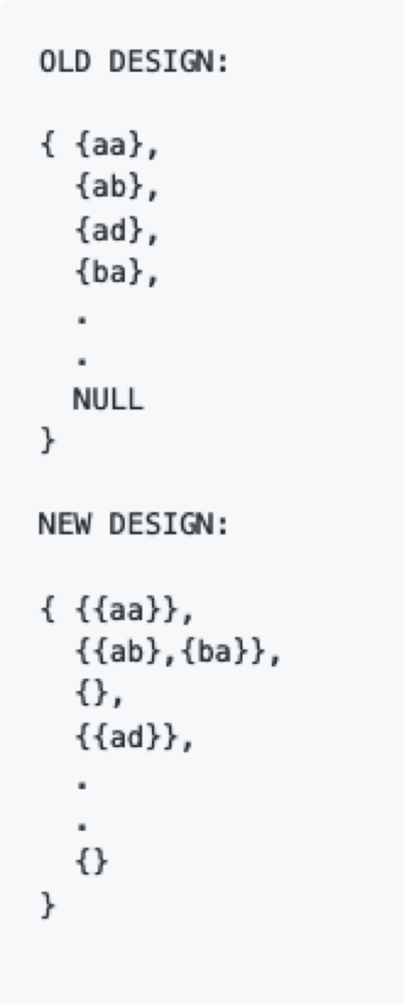
New Option Table



```
OLD DESIGN:

{ {aa},
  {ab},
  {ad},
  {ba},
  .
  .
  NULL
}

NEW DESIGN:

{ {{aa}},
  {{ab},{ba}},
  {},
  {{ad}},
  .
  .
  {}
}
```

- The hash table is not perfect or minimal
- Attempts were made to get close to that, trying to implement gperf’s perfect hash function generating algorithm
- This was abandoned due to the build time added by the script trying to generate a perfect hash table, and the fact that the resulting table was very sparsely populated

New Option Table



```
OLD DESIGN:

{ {aa},
  {ab},
  {ad},
  {ba},
  .
  .
  NULL
}

NEW DESIGN:

{ {{aa}},
  {{ab},{ba}},
  {},
  {{ad}},
  .
  .
  {}
}
```

- The hashed table is indexed into when processing options by hashing the option string, which gets us the bucket
- The bucket could contain multiple options that hash to the same index, so a string comparison is needed
- Theoretically, retrieving an entry in this new table design should be faster than the old design on average

OptionProcessors

- An extensible class containing a set of option processing functions that process options from command-line/env
- OMR will provide a set of basic processing functions (eg, setTrue, setFalse, setInt32, etc.)
- Downstream projects can extend this class to process their options

OptionsBuilder

- This class is used during the initialization of options
- Handles parsing command line options, env options, etc

CompilerOptions

- The class containing all the option members
- It includes the generated Options.inc file

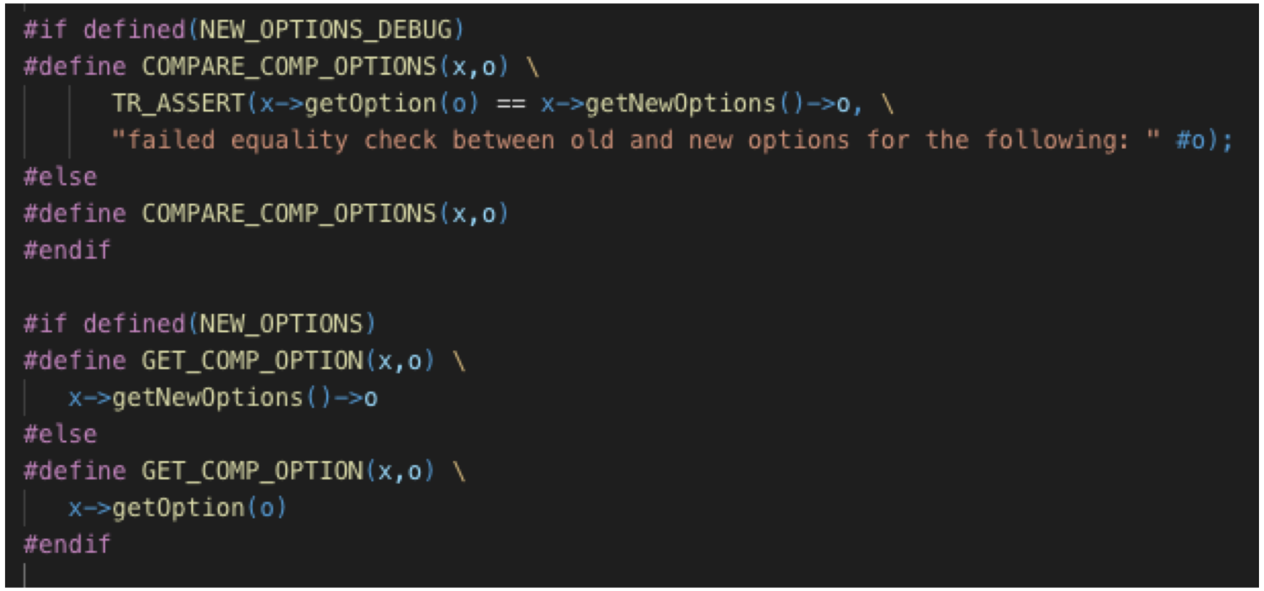
CompilerOptionsManager

- An extensible class managing the different components of options processing
- Handles initialization of options, option table lookup, CompilerOptions object lookup, etc

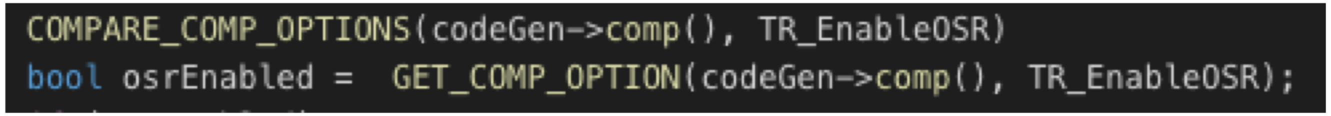
Integrating with OMR

- Unfortunately, replacing the existing OMR::Options class in a single PR is not possible
- The new framework is going to be off by default, and would need to be explicitly turned on
- Macro functions can handle a dual implementation of querying boolean options without any runtime overhead

Integrating with OMR



```
#if defined(NEW_OPTIONS_DEBUG)
#define COMPARE_COMP_OPTIONS(x,o) \
      TR_ASSERT(x->getOption(o) == x->getNewOptions()->o, \
      "failed equality check between old and new options for the following: " #o);
#else
#define COMPARE_COMP_OPTIONS(x,o)
#endif

#if defined(NEW_OPTIONS)
#define GET_COMP_OPTION(x,o) \
   x->getNewOptions()->o
#else
#define GET_COMP_OPTION(x,o) \
   x->getOption(o)
#endif
```



```
COMPARE_COMP_OPTIONS(codeGen->comp(), TR_EnableOSR)
bool osrEnabled =  GET_COMP_OPTION(codeGen->comp(), TR_EnableOSR);
```

Upcoming PRs…

- Support for non-Boolean option types
- Support option sets
- Support setting options in feBase (ie, JitConfig)
- Add support for using the new options in OpenJ9
- Migrate OpenJ9-specific options to Options.json in OpenJ9
- And more…

Questions?