System Image Building

Building the Julia system image

Julia ships with a preparsed system image containing the contents of the `Base` module, named `sys.ji`. This file is also precompiled into a shared library called `sys.{so,dll,dylib}` on as many platforms as possible, so as to give vastly improved startup times. On systems that do not ship with a precompiled system image file, one can be generated from the source files shipped in Julia's `DATAROOTDIR/julia/base` folder.

This operation is useful for multiple reasons. A user may:

- Build a precompiled shared library system image on a platform that did not ship with one, thereby improving startup times.
- Modify `Base`, rebuild the system image and use the new `Base` next time Julia is started.
- Include a `userimg.jl` file that includes packages into the system image, thereby creating a system image that has packages embedded into the startup environment.

The `PackageCompiler.jl` package contains convenient wrapper functions to automate this process.

System image optimized for multiple microarchitectures

The system image can be compiled simultaneously for multiple CPU microarchitectures under the same instruction set architecture (ISA). Multiple versions of the same function may be created with minimum dispatch point inserted into shared functions in order to take advantage of different ISA extensions or other microarchitecture features. The version that offers the best performance will be selected automatically at runtime based on available CPU features.

Specifying multiple system image targets

A multi-microarchitecture system image can be enabled by passing multiple targets during system image compilation. This can be done either with the `JULIA_CPU_TARGET` make option or with the `-C` command line option when running the compilation command manually. Multiple targets are separated by `;` in the option string. The syntax for each target is a CPU name followed by multiple features

separated by `,`. All features supported by LLVM are supported and a feature can be disabled with a `-` prefix. (`+` prefix is also allowed and ignored to be consistent with LLVM syntax). Additionally, a few special features are supported to control the function cloning behavior.

1. `clone_all`

   By default, only functions that are the most likely to benefit from the microarchitecture features will be cloned. When `clone_all` is specified for a target, however, all functions in the system image will be cloned for the target. The negative form `-clone_all` can be used to prevent the built-in heuristic from cloning all functions.

2. `base(<n>)`

   Where `<n>` is a placeholder for a non-negative number (e.g. `base(0)`, `base(1)`). By default, a partially cloned (i.e. not `clone_all`) target will use functions from the default target (first one specified) if a function is not cloned. This behavior can be changed by specifying a different base with the `base(<n>)` option. The `n`th target (0-based) will be used as the base target instead of the default (`0`th) one. The base target has to be either `0` or another `clone_all` target. Specifying a non-`clone_all` target as the base target will cause an error.

3. `opt_size`

   This causes the function for the target to be optimized for size when there isn't a significant runtime performance impact. This corresponds to `-Os` GCC and Clang option.

4. `min_size`

   This causes the function for the target to be optimized for size that might have a significant runtime performance impact. This corresponds to `-Oz` Clang option.

As an example, at the time of this writing, the following string is used in the creation of the official `x86_64` Julia binaries downloadable from julialang.org:

```
generic;sandybridge,-xsaveopt,clone_all;haswell,-rdrnd,base(1)
```

This creates a system image with three separate targets; one for a generic `x86_64` processor, one with a `sandybridge` ISA (explicitly excluding `xsaveopt`) that explicitly clones all functions, and one targeting the `haswell` ISA, based off of the `sandybridge` sysimg version, and also excluding `rdrnd`. When a Julia implementation loads the generated sysimg, it will check the host processor for matching CPU capability flags, enabling the highest ISA level possible. Note that the base level (`generic`) requires the `cx16` instruction, which is disabled in some virtualization software and must be enabled for the `generic` target to be loaded. Alternatively, a sysimg could be generated with the target `generic,-cx16` for greater compatibility, however note that this may cause performance and stability problems in some code.

Implementation overview

This is a brief overview of different part involved in the implementation. See code comments for each components for more implementation details.

1. System image compilation

   The parsing and cloning decision are done in `src/processor*`. We currently support cloning of function based on the present of loops, simd instructions, or other math operations (e.g. fastmath, fma, muladd). This information is passed on to `src/llvm-multiversioning.cpp` which does the actual cloning. In addition to doing the cloning and insert dispatch slots (see comments in `MultiVersioning::runOnModule` for how this is done), the pass also generates metadata so that the runtime can load and initialize the system image correctly. A detail description of the metadata is available in `src/processor.h`.

2. System image loading

   The loading and initialization of the system image is done in `src/processor*` by parsing the metadata saved during system image generation. Host feature detection and selection decision are done in `src/processor_*.cpp` depending on the ISA. The target selection will prefer exact CPU name match, larger vector register size, and larget number of features. An overview of this process is in `src/processor.cpp`.

« isbits Union Optimizations Working with LLVM »