LLVM backend for TILE64

The backend in its current state provides the following features:

- Tilera ABI compliance
- Handling variadic functions
- Allocating dynamic local variables
- Generating position-independent code
- Utilizing TILE64 VLIW capabilities
- Generating Tile Processor Assembly Code

It is available on GitHub:
https://github.com/llvm-tilera

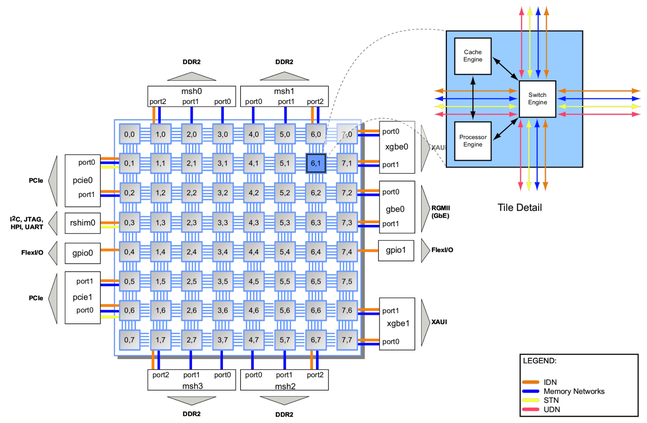
Dávid Juhász, Tamás Kozsik
juhda@caesar.elte.hu, kto@elte.hu

Department of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University
Pázmány Péter sétány 1/C, Budapest 1117, Hungary

What is TILE64?

TILE64 is an energy-efficient massively parallel processor architecture, which was the first commercial product of Tilera Corporation.

- 64 general purpose processor cores (tiles) connected by a mesh-network
- short-pipeline, in-order, three-issue cores
- VLIW instruction set
  - supporting RISC instructions
  - extended with SIMD and DSP-related operations
- the speed of the interconnection between tiles is one hop per tick
- the edges of the mesh are connected to I/O interfaces
  - four DDR2 controllers
  - two 10-gigabit Ethernet interfaces
  - two four-lane PCIe interfaces
  - a software-configured “flexible” I/O interface



Unresolved issues

There are issues seeming to be special ones, but trends in hardware industry make them common problems for new architectures. Such two issues are the followings:

Utilizing special-purpose native instructions

TILE64 has special SIMD and DSP-related instructions that are too complex to be described conveniently with TableGen patterns of actual SelectionDAG nodes.

Exploiting the power of multiple cores

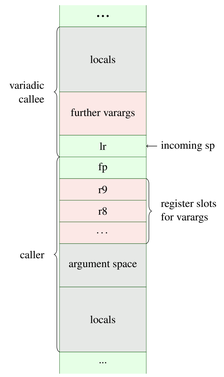
Forced by the lack of a platform-independent model for thread-level parallelism in LLVM, one needs to use architecture-specific library calls to implement parallel applications.

Unchangeable default behaviors

Some aspects of code generation are hard-coded into LLVM CodeGen library. The choices that have been made there may fit mainstream architectures, but just don’t meet the needs of other, more specialized ones.

Argument passing according to the Tilera ABI

- First ten words are passed in registers
- Further parameters are passed in the argument space of the caller
- Alignment is forced by padding on the stack and even in registers
- Padding is conserved inside structs during passing
- All parameters are passed entirely either in registers or on the stack
- If a return value cannot fit in ten registers, then it is passed indirectly



Argument passing by LLVM CodeGen

Some aspects are hard-coded into the library:

- Struct arguments are broken up into component values
- Values are split into register-sized parts

Other ones, like whether to pass a value in register or on the stack, whether a return value must be passed indirectly or not, are to be defined by backends.

Resolving the discrepancy

Hard-coded decisions can be worked around by restricting the way LLVM IR is used. Constraints on the usage of struct values could be forced by a transformation pass preparing for instruction selection.

Transformation for passing a struct value as argument:

```
%struct = type { ... }

declare void @callee(%struct %arg)

define void @caller() {
entry:
    %ptr = alloca %struct
    %val = load %struct* %ptr
    call void @callee(%struct %val)
    ret void
}
```

⇒

```
%struct = type { ... }

declare void @callee(%struct* byval %arg)

define void @caller() {
entry:
    %ptr = alloca %struct
    call void @callee(%struct* byval %ptr)
    ret void
}
```

Transformation for passing an undersize struct value as result:

```
%struct = type { ... }

define %struct @callee() {
entry:
    %ptr = alloca %struct
    ;set struct
    %val = load %struct* %ptr
    ret %struct %val
}

define void @caller() {
entry:
    %val = call %struct @callee()
    ret void
}
```

⇒

```
%struct = type { ... }
%wrapper = type iN ;where N is the size of the struct

define %wrapper @callee() {
entry:
    %ptr = alloca %struct
    ;set struct
    %wrapper_ptr = bitcast %struct* %ptr to %wrapper*
    %wrapped_val = load %wrapper* %wrapper_ptr
    ret %wrapper %wrapped_val
}

define void @caller() {
entry:
    %wrapped_val = call %wrapper @callee()
    %wrapper_ptr = alloca %wrapper
    store %wrapper %wrapped_val, %wrapper* %wrapper_ptr
    %ptr = bitcast %wrapper* %wrapper_ptr to %struct*
    %val = load %struct* %ptr
    ret void
}
```

VLIW packetizing

Utilizing VLIW feature introduced in LLVM 3.2 gives promising results, however the machinery could be improved to fully suit VLIW capabilities of TILE64.

Implementing a VLIW packetizer

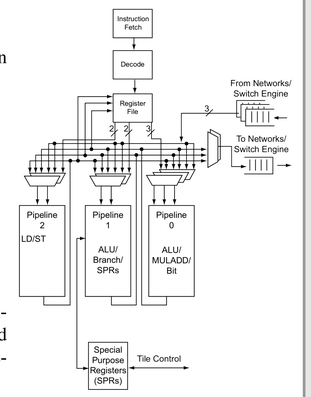
VLIW-scheduler automaton generated from TableGen description:

- Declare functional units
- Classify instructions regarding functional units
- Bundling by looking for a suitable functional unit

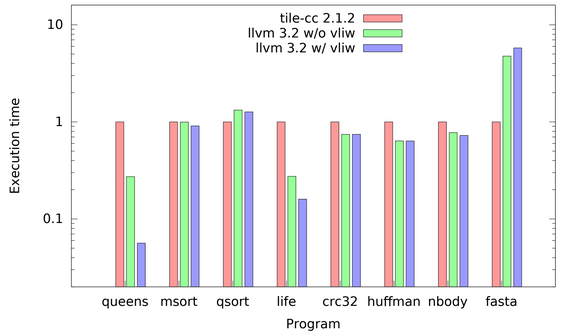
TILE64 has further constraints for valid bundles:

- three functional units
- 64-bit bundles with different sized instruction codes
- therefore, there are two- and three-issue bundles

Constraints beyond availability of functional units cannot be expressed by actual TableGen facilities. A patched VLIWPacketizerList is used for validating bundles created by the scheduler automaton.



Measuring execution times



Nasty technical details

We use pseudo-instructions to postpone generating actual code until proper information is available. Most of these instructions are suitable for fully automatic generation.

However, custom printing mechanisms have to be provided sometimes. One such example is the generation of position indepedent code:

```
GBR gbr:r1
ADDLO_PIC T64GPRF:r0 T64GPRF:r1 picaddr:data
ADDHI_PIC T64GPRF:r0 T64GPRF:r0 picaddr:data
```

⇒

```
    lnk r0
.BASE_POS:
    addli r0 r1 lo16(data - .BASE_POS)
    auli r0 r0 ha16(data - .BASE_POS)
```


The EITKIC_12-1-2012-0001 project is supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund.