Hardware enforced variable scoping

Table of contents

- Hardware enforced variable scoping
  - Introduction
  - Implementation
    - Hardware implementation
  - Load and store
  - Instruction set extension
    - scope block enter sbent
      - Pseudocode description
      - Instruction encoding
    - storage region base srbse
      - Pseudocode description
      - Instruction encoding
    - storage region extend srlmt
      - Pseudocode description
      - Instruction encoding
    - storage region delegate srdlg
      - Pseudocode description
      - Instruction encoding
    - storage region delegate move srdlgm
      - Pseudocode description
      - Instruction encoding
    - scope block exit sbxit
      - Pseudocode description
      - Instruction encoding
    - sub storage region srsub
      - Pseudocode description
      - Instruction encoding

The RISC-V ISA is extended to support enforcement of variable scoping. Idea is to maintain run-time information of variables and scopes. This information is stored in a data structure which is checked on every store or load instruction. System supports dynamic delegation which is needed when pointers are passed in the call stack.

Introduction

A scope block is a continuous block of code which has its own scope. Generally this is a function. A scope block can access its own scope.

A storage region is a continuous piece of address space which is included in a scope. It may be for example a global structure, a stack buffer or string in heap. There are usually many storage regions in a scope.

A function in C has access to locals, globals and arguments. Any of these can be a

pointer or a structure that contains a pointer. A pointer can contain an arbitrary address, but in practice the address points to function local stack, process heap, globals or caller's stack.

Different types of storage regions include

- contents of the current stack frame including
- local variables
- saved return address
- saved frame pointer
- global variables and data structures
- arguments passed via stack
- dynamically allocated variables in heap or stack

Storage Region Stack is a stack-oriented data structure that contains scope entries. Each entry consists of three fields: base, limit. Base is the first address of the region and limit is the last address.

Contents of the stack are split into frames. Each frame contains one scope's entries and every entry belongs to a single frame. Topmost frame includes regions accessible from the current scope.

On every memory access the current frame is checked. If accessed address is not in the current scope, access is denied.

Local variables, arguments and necessary globals are always in the current scope. Dynamic allocation is also supported. Dynamically allocated object is added to the SRS when it is allocated in function like `malloc` and then passed to the scope of the caller by using the delegation mechanism.

Delegation mechanism allows passing storage regions between scopes. Function arguments, return values and others require using delegation.

An example of SRS contents

```
    | base    | limit   | comment
  1 | 0x1000  | 0x100b  | start of main frame, local variables
  2 | 0x1f00  | 0x1fff  | stack buffer
  3 | 0x8100  | 0x81ff  | heap buffer
  4 | 0x2000  | 0x2007  | function arguments
  5 | 0x1f00  | 0x1fff  | delegated stack buffer
  6 | 0x8200  | 0x82ff  | dynamic allocation, will be returned
```

Implementation

The SRS is an abstract data structure and it can be implemented in multiple ways. Implementation should be fast, because it must be checked on every memory access. It will not fit completely into registers and some parts need to be stored in memory.

Hardware implementation

Maximum size of an SRS frame is fixed to 32 entries. Current SRS frame is stored in a set of comparator registers. Each entry consists of base and limit fields. Size of base and limit is one word, 32-bits.

SRS stack is implemented in hardware. Stack operations push and pop save and restore contents of all active base and limit registers. Maximum size of each frame is 32+32+1 = 65 words.

There are two banks of registers to support delegation. Banks are called A and B. A switch happens when the context changes from an execution context to another like on function call or return.

Load and store

All memory accesses are mediated to enforce the scope rules. This requires modifying behaviour of load and store instructions.

Each load and store instruction targets a single memory address. The address is usually calculated by adding a register value to an immediate operand. In RISC-V loads and stores access a word, a halfword or a byte. Target address of the instruction is compared to each entry in the current SRS frame. Access is granted if following conditions are met: (1) for an active entry the address is greater than or equal to base, and (2) address of the last byte accessed is less than or equal to the limit. It is assumed that base is always less than or equal to the limit.

Each active storage region is compared to the accessed address in parallel and an hardware memory access fault is raised in case there are no matches, which results in termination of execution.

Instruction set extension

There are 7 instructions in the extension.

Existing RISC-V instruction format is used to encode the extension instructions. The major opcode is custom-3 (11 110 11) and instruction format is S. This encoding format has funct3-field which is used as minor opcode, and it provides necessary operand fields.

Instructions have zero, one or two operands. In case of one operand, its value is an address that will be added to an SRS entry as base or limit, or a base address of an entry that will be delegated. Value of the operand is calculated by adding the immediate to value stored in registry.

RISC-V S-type instruction format has two source registers and 12 bit immediate in addition to major opcode and minor opcode (funct3):

```
bits  | 31:25       | 24:20 | 19:15 | 14:12  | 11:7     | 6       |
field | imm[11:5]   | rs2   | rs1   | funct3 | imm[4:0] | opcode  |
```

Instruction descriptions use pseudocode with following notations:

- `base_i` is register holding base address of entry number **i**
- `limit_i` is register holding limit value of entry number **i**
- `bankA` is the active bank of registers
- `bankB` is the inactive bank of registers
- `bankX.count` number of active entries in bank **X**
- `find_entry(x)` finds the register holding a rule that matches address **x**
- `bank_copy(x,y)` copy entry `x` to `y` where `x` and `y` are in different banks
- `find_free(x)` finds a free register from bank **x**
- `srs.push` operation pushes active entries to the stack
- `srs.pop` operation pops the topmost entries from the stack to unused registers in bankA
- `bank_switch` changes the active bank

scope block enter `sbent`

Entering a new scope block context. Changes the current context to the next next frame on the stack. Storage blocks that are marked for delegation are copied to the frame. The instruction is executed in the caller before the branching instruction. If function-level granularity is used, `sbent` is only used with branches. The design makes finer-grained enforcement also possible.

Operand: none

Mnemonic: `sbent`

Pseudocode description

```
/* stores the current active entries and switches the bank */

srs.push                          //push registers base_i a and limit_i for each active entry
bank_switch
```

Instruction encoding

```
bits  | 31:25       | 24:20 | 19:15 | 14:12  | 11:7     | 6         |
field | imm[11:5]   | rs2   | rs1   | funct3 | imm[4:0] | opcode    |
value | 000000      | 00000 | 00000 | 000    | 00000    | 11 110 11 |
```

storage region base `srbse`

Sets base address of the topmost entry in SRS.

This can be used for example in

- caller when adding the arguments to callee's scope
- function prologue when adding the local stack variables to the scope
- function prologue when adding globals or static variables to the scope
- allocation functions when the scope is extended with new, dynamic variables

Operand: rs1+immediate

Mnemonic: `srbse imm(rs1)`

Pseudocode description

```
i=find_free(bankA)            //find first free register
base_i=rs1+imm                //set the base
```

Instruction encoding

```
bits  | 31:25       | 24:20 | 19:15 | 14:12  | 11:7     | 6         |
field | imm[11:5]   | rs2   | rs1   | funct3 | imm[4:0] | opcode    |
value |             | 00000 |       | 001    |          | 11 110 11 |
```

storage region extend `srlmt`

Sets limit of or "closes" the topmost storage region in SRS.

The operand is last address of the storage region.

Operand: rs1+immediate

Mnemonic: `srlmt imm(rs1)`

Pseudocode description

```
i=find_free(bankA)            //find first free register
limit_i=rs1+imm               //set the limit
bankA.count++                 //number of active entries is incremented
```

Instruction encoding

```
bits  | 31:25       | 24:20 | 19:15 | 14:12  | 11:7     | 6         |
field | imm[11:5]   | rs2   | rs1   | funct3 | imm[4:0] | opcode    |
value |             | 00000 |       | 010    |          | 11 110 11 |
```

storage region delegate `srdlg`

Marks an entry to be delegated when the scope is changed with either `sbent` or `sbext`. Delegates access to existing storage region to the next execution context. Parameter is an address that belongs to some storage region in the SRS.

Operand: rs1+immediate

Mnemonic: `srdlg imm(rs1)`

Pseudocode description

```
/* find the entry based on an address and copy the other bank */

i = find_entry(imm+rs1)       //find the matching entry in current scope
j = find_free(bankB)          //find a free register in the other bank
copy_to_other_bank(i,j)       //copy the entry to new bank
```

Instruction encoding

```
bits  | 31:25       | 24:20 | 19:15 | 14:12  | 11:7     | 6         |
field | imm[11:5]   | rs2   | rs1   | funct3 | imm[4:0] | opcode    |
value |             | 00000 |       | 011    |          | 11 110 11 |
```

storage region delegate move `srdlgm`

This instruction is a variant of `srdlg`. It is used when the storage region is not needed in the callers scope any more after delegation. For example passing arguments in the stack uses `srdlgm`.

Operand: rs1+immediate

Mnemonic: `srdlgm imm(rs1)`

Pseudocode description

```
/* find the entry based on an address, copy to the other bank, and remove */

i = find_entry(imm+rs1)        //find the matching entry in current scope
j = find_free(bankB)           //find a free register in the other bank
copy_to_other_bank(i,j)        //copy the entry to new bank
bankA.count--                  //number of active entries is decremented
```

Instruction encoding

```
bits  | 31:25       | 24:20 | 19:15 | 14:12  | 11:7     | 6         |
field | imm[11:5]   | rs2   | rs1   | funct3 | imm[4:0] | opcode    |
value |             | 00000 |       | 100    |          | 11 110 11 |
```

scope block exit `sbxit`

Leaving the current scope block context, copies the entries that will be delegated, to the next frame and evicts current frame from the stack. Rolls the SRS pointer back to the previous frame and triggers also delegation. All entries that are marked to be delegated are counted and copied to the bottom of the current stack frame. When the context is then changed, the frame is extended so that delegated storage regions also belong to it.

Operand: none

Mnemonic: `sbxit`

Pseudocode description

```
/* pop the old context and change the active bank */

srs.pop                        //pop the previous context from stack to the empty registers
switch_bank                    //switch activates the new context
```

Instruction encoding

```
bits  | 31:25       | 24:20 | 19:15 | 14:12  | 11:7     | 6         |
field | imm[11:5]   | rs2   | rs1   | funct3 | imm[4:0] | opcode    |
```

```
value | 000000      | 00000 | 00000 | 101    | 00000    | 11 110 11 |
```

sub storage region `srsub`

Creates a new storage region and checks that it is inside the bounds of an existing region.

Operand: rs1, rs2+immediate

Mnemonic: `srsub rs1, imm(rs2)`

Pseudocode description

```
i=find_free(bankA)            //find first free register
base_i=rs1                    //set the base
limit_i=rs2+imm               //set the limit
bankA.count++                 //number of active entries is incremented
```

Instruction encoding

```
bits  | 31:25       | 24:20 | 19:15 | 14:12  | 11:7     | 6         |
field | imm[11:5]   | rs2   | rs1   | funct3 | imm[4:0] | opcode    |
value |             |       |       | 110    |          | 11 110 11 |
```