Implementação de predição de salto no simulador OrCS

Métodos de predição por BTB e combinado *bimodal*/*gshare*

Gabriel G. de Brito

Universidade Federal do Paraná

21 de maio de 2024

Sumário

OrCS

- Simulador de traços.
- Versão reduzida utilizada: “Micro OrCS”.

A cada ciclo, nossos componentes analisarão a instrução e seu próprio estado interno para computar estatísticas sobre a execução.

OrCS

processor_t::clock()

uOrCS - processor.cpp

```
25 void processor_t::clock() {
26     // Get the next instruction from the trace.
27     opcode_package_t new_instruction;
28     if (!orcs_engine.trace_reader->trace_fetch(&new_instruction)) {
29         // If EOF.
30         orcs_engine.simulator_alive = false;
31         return;
32     }
33
34     orcs_engine.global_cycle += this->btb->check_instruction(
35         &new_instruction, orcs_engine.get_global_cycle());
36     orcs_engine.global_cycle +=
37         this->predictor->check_instruction(&new_instruction);
38 };
```

Branch Target Buffer

- Mecanismo para guardar o resultado de saltos, como um cache.
- Acelera todos os tipos de saltos, exceto retornos.
- No simulador, não precisamos guardar os endereços, como uma implementação real faz.

Branch Target Buffer

Especificação

- 1024 conjuntos associativos, com 12 entradas cada.
- Conjuntos endereçados pelos 10 bits menos significativos do endereço da instrução

Branch Target Buffer

Latência

Saltos incondicionais (*jumps*, chamadas de função e do sistema):

- Nenhuma, caso a BTB possua a entrada da instrução.
- 12 ciclos, caso contrário.

Saltos condicionais:

- Nenhuma, caso a BTB possua a entrada da instrução com a direção correta.
- Nenhuma, caso a BTB não possua a entrada da instrução porém a direção era de salto não tomado.
- 512 ciclos, caso a BTB não possua a entrada da instrução e a direção era de salto tomado.
- 512 ciclos, caso a BTB possua a entrada da instrução, porém a direção esteja errada.

Branch Target Buffer

Implementação

Ao acessar a BTB, acessamos o conjunto endereçado pelos 10 bits menos significativos do endereço da instrução. Procuramos no conjunto uma entrada associada ao endereço específico.

- Caso não exista nenhuma entrada associada, uma entrada sem uso será associada ao endereço.
- Caso todas as entradas estejam em uso, substituímos a entrada acessada a mais tempo. Para isso, guardamos o último ciclo em que cada entrada foi acessada.

Isso é suficiente para instruções de salto incondicional.

Branch Target Buffer

Implementação

Para instruções de salto condicional, precisamos acessar a próxima instrução para computar o resultado do salto.

Guardamos informações sobre a instrução de salto no estado do objeto da BTB e, no próximo ciclo, comparamos as informações em relação ao endereço da instrução executada: caso o endereço da instrução de salto + seu tamanho seja igual ao endereço da próxima instrução, o salto não foi tomado.

Branch Target Buffer

Implementação

uOrCS - btb.hpp

```
21 typedef struct {
22     bool valid;
23     uint64_t address;
24     uint64_t last_access;
25     bool is_conditional;
26     bool is_taken;
27 } btb_entry_t;
28
29 typedef btb_entry_t btb_block_t[btb::BLOCK_SIZE];
```

Branch Target Buffer

Implementação

uOrCS - btb.hpp

```
31  class btb_t {
32      private:
33          btb_block_t buffer[btb::BUFFER_SIZE];
34
35          uint64_t last_address;
36          uint64_t last_size;
37          btb_entry_t *last_entry;
38
```

Preditor

Preditor que combina as técnicas *bimodal* e *gshare*, tentando utilizar a mais eficiente para dado contexto.

Ambas são baseadas em contadores, que também são utilizados para escolher qual preditor será utilizado.

Preditor

Contador

- Um contador guarda o histórico de um evento.
- É incrementado caso um dos eventos ocorra e decrementado caso o outro ocorra.
- O próximo evento é predito de acordo com a proximidade do contador com o 0 (valor mínimo) e o seu valor máximo.
- Na implementação utilizamos 2 bits sempre (i.e., valor máximo de 3).

Preditor

Contador

uOrCS - predictor.hpp

```
10 namespace counter {
11
12 const int BITS = 2;
13 const int MAX = (1 << BITS) - 1;
14 const int MIDDLE_TRUE = (1 << BITS / 2);
15
16 typedef uint8_t counter;
17
18 inline bool is_true(counter counter) { return counter ≥ MIDDLE_TRUE; }
19
20 inline counter increase(counter counter) { return counter + (counter < MAX); }
21
22 inline counter decrease(counter counter) { return counter - (counter > 0); }
23
24 }  // namespace counter
```

Preditor

bimodal

- Semelhante à uma BTB, porém guarda contadores. Endereçados pelos bits menos significativos do endereço.
- Os contadores são incrementados quando o salto é tomado, e decrementados caso contrário.
- Utilizamos uma tabela simples de 2048 entradas. A experiência empírica mostra que tabelas maiores não aumentam significativamente a eficácia.

Preditor

gshare

- Semelhante ao *bimodal*, porém o endereçamento é feito utilizando um *hash* do histórico global e o endereço da instrução.

O histórico global é um conjunto de bits que guarda o histórico dos últimos saltos condicionais. E.g., o histórico 00101 registra a ocorrência dos saltos: não tomado, não tomado, tomado, não tomado, tomado.

O endereço consultado a cada acesso é um OU Exclusivo entre o endereço da instrução e o histórico global do momento.

Preditor

Combinado

O preditor combinado é implementado utilizando um contador para guardar o histórico de acerto de ambos os preditores.

As posições altas do contador indicam que o *gshare* deve ser utilizado.

O contador é atualizado quando um dos preditores erra e o outro acerta.