Verslag

Koos van Strien (5783437), Tim van Deurzen ()

December 23, 2009

1 Code representatie

De code is in verschillende lagen onder te verdelen.

1.1 Het bestand

Het bestand waar de code zich in bevindt is de hoogste "laag" die binnen de compiler als geheel gezien kan worden. Aangezien een bestand al een geheel is, zou hier geen representatie van hoeven zijn binnen de optimizer. Aangezien het bestand echter ook verzameling en opslagplaats is van basic blocks (de eerstvolgende laag), bestaat binnen de optimizer de klasse "blockBuilder" die codebestanden representeert. Binnen de klasse bevinden zich methoden om basic blocks te herkennen en te achterhalen wat de targets zijn waar vanaf een basic block naartoe gesprongen wordt.

1.2 De basic blocks

Binnen het bestand is de code onder te verdelen in basic blocks. Basic blocks beginnen met een mogelijke jump-target en eindigen met een jump of branch operatie. Binnen de optimizer worden deze gerepresenteerd door instanties van de "basicBlock" klasse. Binnen deze klasse bevinden zich methoden om informatie over basic blocks te verkrijgen en om veranderingen aan te brengen op basic block-niveau.

1.3 Operations en labels

1.3.1 Categorieën

Het laagste niveau zijn de afzonderlijke labels en operations. Binnen de optimizer zijn deze instructies in zes categorieën verdeeld. Samen met de categorie voor labels valt elke regel code in één van de volgende categorieën:

- Control
- Load
- Store
- Integer Arithmetic
- Float Arithmetic

- System
- Label

1.3.2 Klassen

Elke categorie heeft een eigen klasse. Boven alle klassen staat de parent-klasse "operation". Generieke methoden worden in deze klasse geïmplementeerd, specifiekere methoden worden overridden of gespecifieerd binnen de kind-klassen. Naast de "operation" klasse hebben de klassen voor de categorieën Load en Store nog een gezamelijke parent-klasse, aangezien zij sommige acties delen. Binnen de "operation"-klasse is een static methode die bij aanroep automatisch een instantie van de juiste klasse teruggeeft.

1.3.3 Bewerkingen op code

Wijzigingen aan operations worden altijd via methoden gedaan. Aangezien elke operation door een instantiatie wordt vertegenwoordigd, is dit een vrij intuïtieve methode. ((NIET ZEKER OF DIT ERIN MOET: Het is ook een veilige methode, aangezien de mogelijkheden om wijzigingen aan te brengen getest kunnen worden en daarna "gegarandeerd" werken.))

2 Optimalisatie

De optimizer heeft de volgende optimalisaties geïmplementeerd:

- Redundant labels
- Redundant load / stores
- Copy propagation
- Common subexpression elimination

2.1 Structuur

Er is een parent-klasse "optimizationClass". Deze voorziet in de generieke functionaliteit die elke optimalisatie nodig heeft, zoals het opslaan en weergeven van basic blocks.

2.2 Redundant labels

2.2.1 Doel

De redundant labels-optimalisatie kijkt of een basic block a als enige operatie een jump heeft (naar basic block b). In dat geval wordt geheel basic block a verwijderd. Verwijzingen naar basic block a worden vervangen door verwijzingen naar basic block b. Dit scheelt dan een jump-operatie.

2.2.2 Implementatie

Het algoritme geïmplementeerd volgens onderstaand pseudocode. Wanneer er over een "target" van een block gesproken wordt wordt hier altijd een block bedoelt waar een jump-operatie naar toe springt.

```
for each basic block __b:
  if __b has a target __t and __t has exactly two operations

    if the last operation of __t is a CONTROL:
      save the target of __t as __l

    if __l is a label and the last operation __o of __b is a CONTROL:
      set the target of __o to __l

      exclude __t from the code
```

2.3 Redundant load / stores

2.3.1 Doel

Redundant load / store optimalisatie heeft als doel geen waarde uit het geheugen te laden die zich al in een register bevindt (redundant load), en geen waarde terug te schrijven naar het geheugen wanneer deze niet gewijzigd is (redundant store). Aangezien load- en store-operaties door de writeback erg traag zijn, kan dit al snel uitvoertijd schelen, met name wanneer bijvoorbeeld binnen een loop hierdoor geheugen-operaties weggelaten kunnen worden.

Ook staan operaties soms achter een load gescheduled omdat de waarde daarvoor nog niet beschikbaar zou zijn. Wanneer deze load redundant blijkt te zijn is er voor niets gewacht met het uitvoeren van de instructie.

2.3.2 Implementatie

Het algoritme is als volgt geïmplementeerd:

```
for each basic block __b:
  clear list of operations that change a register

  for each operation __op in __b:
    if __op was previously excluded (by another optimized):
      skip operation
    else:
      search for an operation, __op2, in the list that uses the same
      registers as __op

      if __op is a LOAD:
        if __op2 exists:
          if __op2 is a load or a store:
            if the target of __op2 is the target of __op and the address of
              __op2 is the address of __op:

              exclude __op from the code.
```

```
        else if the targets of __op2 and __op are the same but the
          addresses are not:

          remove __op2 from the list.
          add __op to the list.
      else:
        if the target of __op2 and __op are the same:

          remove __op2 from the list.
          add __op to the list.
    else if __op was previously stored:
      exclude __op from the code.

    else:
      add __op to the list of operations

  else if __op is a STORE:
    if __op was previously stored:
      exclude __op from the code.
    else:
      if __op2 exists:
        remove __op2 from the list of operations.
      else:
        add __op to the list of operations.

  else if __op is an ARITHMETIC operation:
    if __op2 exists:
      remove __op2 from the list/

    add __op to the list of operations.
```

2.4 Copy propagation

2.4.1 Doel

Copy propagation zou ook omschreven kunnen worden als "redundant moves elimination". Het gaat hier namelijk om move-instructie die een register kopiëren, maar waar het oude register niet overschreven wordt voordat een operatie het nieuwe register uitleest. Het vervangen van het "doelregister" door het "bronregister" heet copy propagation. Wanneer uiteindelijk het doelregister niet meer gebruikt wordt, of een nieuwe waarde krijgt, terwijl het bronregister tot de laatste aanroep aan toe beschikbaar was, kan de gehele move-operatie verwijderd worden.

2.4.2 Implementatie

```
for each basic block __b:
  clear list of moves
  clear list of dead_moves
```

```
clear list of updatedOperations

for each operation __op in __b:
  if __op was previously excluded (by another optimizer):
    skip operation
  else:
    if __op is a MOVE:
      if __op uses $fp / $sp:
        skip operation

      if there is a reverse move on list:
        remove old operation

      add __op to list of moves


    if __op is a STORE:
      if there is a move in the list of moves that has a target that __op
        uses as source:

        update __op and set the target to the source of the move.
        exclude the move from the code.

        add __op and move to a list of altered operations


    if __op is a LOAD:
      if there is a move in the list of moves that has the same target as
        __op or __op uses the source of the move as target:

        remove move from list
        add move to list of dead_moves

      if the source or destination of the move is used as an offset in __op:

        rollback all edits until the last added move on the list of
        altered operations.

      if there is a move in the list of dead moves:
        if the destination of the move is the target of the load or
          the source of the move is the target of the load:

          remove move from list of dead_moves

        if the source or destination of the move is used as an offset in __op:

          rollback all edits until the last added move on the list of
          altered operations.
```

```
    if __op is an ARITHMETIC or CONTROL operation:
      if there is a move in the list of moves that has a target that is
        the a source of __op:

        update __op by replacing the old register with the source of the
        move.

        exclude the move from the code.

        add __op and move to the list of altered operations.

      else if there is a move in the list of dead moves:
        if the destination or the source of the move is the target
          of __op:

          remove the move from the list of dead moves.
```

2.5 Common subexpression elimination

2.5.1 Doel

Wanneer dezelfde berekening meerdere keren uitgevoerd wordt (met dezelfde argumenten, dus dezelfde uitkomst) heet dit een common subexpression. De tweede berekening kan dan vervangen worden door een move-instructie, of, wanneer zowel de operatie als de parameters hetzelfde zijn, de hele operatie verwijderd worden.

In beginsel lijkt het eerste een vrij zinloze optimalisatie, aangezien er geen direct resultaat is. Een move-operatie is namelijk even duur als een berekening. Echter, het impliceert wel dat registers eerder vrij komen, wat weer kan resulteren in nieuwe redundant stores (die met move-operaties eenvoudig te herkennen zijn), copy propagation en vervolgens dead code elimination.

2.5.2 Implementatie

In de optimizer is alleen het tweede deel van de common subexpression elimination geïmplementeerd. Houd bij het lezen van de pseudocode in gedachten dat er per block een lijst wordt bijgehouden met op dit moment aanwezige expressions.

```
for each basic block in __b:
  for each operation __op in __b:
    if __op was previously excluded (by another optimizer):
      skip operation
    else:

      if __op has arguments:
        look in the list of expressions already passed by if current
          expression is updating an earlier used one.
      else:
        current operation is no expression

      if __op is of an INT_ARITHMETIC or FLOAT_ARITHMETIC
```

```
    commonExpression = self.findCommonExpression(__op )

    if __op has a common expression
      exclude __op from block
    else if __op is not updating another expression:
      add __op to the list of expressions

  if __op is a LOAD and op is updating an expression:
    remove the expression __op is updating from the list
```

3 Resultaten

De versnellingen die er gerealiseerd zijn:

- acron — van 4436540 naar 4360426 cycles — 2.8%
- clinpack — van 1556370 naar 60067 cycles — inconsistent
- pi — van 1715292 naar 1708329 cycles — 2%
- whet — van 2864289 naar 2970553 cycles — 4.6%