Final Report:

20CS02012

Genesis:

In the GCC 2.7.2.3

the compiler unroll mechanism follows 3 routines to determine the suitable unrolling factor by 3 routines absed on the variables MAX_UNROLLED_INSNS(system defined) and loop_iterations.

(i) **UNROLL_COMPLETE** : unrolls the loop completely;

excerpt from gcc-2.7.2.3/unroll.c line 347-352

```
      else if (loop_n_iterations > 0
            && loop_n_iterations * insn_count < MAX_UNROLLED_INSNS)
      {
            unroll_number = loop_n_iterations;
            unroll_type = UNROLL_COMPLETELY;
      }
```

(ii) **UNROLL_MODULO** : unroll till a specific factor

code from unroll.c

```
 for (i = NUM_FACTORS - 1; i >= 0; i--)
      while (temp % factors[i].factor == 0)
        {
          factors[i].count++;
          temp = temp / factors[i].factor;
        }

      /* Start with the larger factors first so that we generally
       get lots of unrolling.  */

      unroll_number = 1;
      temp = insn_count;
      for (i = 3; i >= 0; i--)
      while (factors[i].count--)
        {
          if (temp * factors[i].factor < MAX_UNROLLED_INSNS)
            {
            unroll_number *= factors[i].factor;
            temp *= factors[i].factor;
            }
          else
            break;
        }

      /* If we couldn't find any factors, then unroll as in the normal
       case.  */
      if (unroll_number == 1)
      {
        if (loop_dump_stream)
          fprintf (loop_dump_stream,
                 "Loop unrolling: No factors found.\n");
      }
      else
      unroll_type = UNROLL_MODULO;
    }
```

(iii) **UNROLL_NAIVE** : unrolls the loop until MAX_UNROLLED_INSNS and naively adds a jump instruction if there is a branch in betweeen the block.

```
/* Default case, calculate number of times to unroll loop based on its size.  */
  if (unroll_number == 1)
    {
      if (8 * insn_count < MAX_UNROLLED_INSNS)
      unroll_number = 8;
      else if (4 * insn_count < MAX_UNROLLED_INSNS)
      unroll_number = 4;
      else
      unroll_number = 2;

      unroll_type = UNROLL_NAIVE;
    }
  /* Now we know how many times to unroll the loop.  */
```

Limitations:

(a) The unroll factor is only decided on the software and does not have any optimisation with respect to hardware. The only optimisation is safety barrier that does not fill-up (or) halt the hardware.

```
 /* Limit loop unrolling to 4, since this will make 7 copies of
           the loop body.  */
        if (unroll_number > 4)
          unroll_number = 4;
```

(b) The naive method preconditioning(adding extra jump fuctions to bypass branches in the mid-block) may cause extra overhead in terms of IPC as the block size is consumed by them, therefore modulo is the most preferred method for unrolling.

(c) The complete and modulo methods which are favourable for a performance are not at the run time but are limited to compilation period and only act if the the code size can be calculated before run-time.

Tackling the limitations :

(i) implementing a better hardware concerned unroll factor analyser.

The 4 limiting factors for deciding unroll factors are

- cache,
- Ex units,
- Mem access bandwidth,
- register pressure.

Cache:

spatial behaviour:

The behaviour of the cache for a loop with increasing unrolling factor requires contigious memory locations most of the times. So, the cache block size plays a major role than the total cache size in unrolling as we may need the addresses which are related to general induction variables.

General induction variable is a linear function image of the binary induction variable.

Binary induction variable is the loop iterator which decides the block statements are described.

Temporal behaviour does not play a major role as most of the caches are hybrid LRUs, thus the repeatedly accessed block always stays in the cache.

EX units and Memory acess:

The Ex units and Memory bandwidth are constant for a pipleline, so the increase based on type of instructions plays a crucial role in determining the factor.

Idea: these units can be equally distributed between the new set of indtructions that are added due to unrolling.

Register pressure:

The register pattern is variable for benchmarks we can track the peak register usage for a block and decide based on the number as we have a limited set of register file alloted.

This cant be determined or estimated so we will need a 2-pass compilation for optimal unrolling.

In the first pass, with no unrolling we can track the peak register usage in a block and use it to determine unrolling factor that increases register utillisation(if very utilisation is very low) and avoid register spilling.

Design of the algorithm:

In the first compilation,

we obtain number of number of Memtype, ex unit based typed isntructions and also peak register usage. The max unrolling factor independent of the other constraints can be calculted for every aspect.

REGISTER PRESSURE MAX_FACTOR: 32 / live_registers × multiplier Interpretation.

MEMORY BANDWIDTH MAX_FACTOR: 16 / (mem_accesses × 2) × multiplier .

Assumption: 16 bytes/cycle L1 cache bandwidth; each load/store costs 2 bytes

Interpretation: Scaling factor for how many times memory ops can repeat without saturating L1 bandwidth.

INSTRUCTION CACHE MAX_FACTOR : 16 / loop_size × multiplier

Assumption: 16 instructions fit in I-cache hotspot (64-byte line, 4 bytes/instruction)

Interpretation: How many times loop can expand before exceeding hotspot size.

ARITHMETIC INTENSITY MAX_FACTOR : 8 if (arith_instr / mem_access) > 0.5, else 4

Reasoning:

Compute-bound loops tolerate more unrolling; memory-bound loops are limited by bandwidth.

The factors 8 and 4 are chosen assuming there will be Ex unit and Mem-bancdwidth limitations, if there are better units in the architecture the metrics can be increased.

The MINIMUM of the unrolling factor of all these independent unrolling factors will be the resultant.

PS: Multiplier feature is included for safety as other conditioning procedures like strength reduction, hardware latency fluctuations etc.

The recompoiled file can be unrolled with a new factor !

Future work :

Unifying it with the compiler, to overcome the limitations (b), (c).

Ex units latencies can be added to the algorithm to make note of hazards and adjust the factor accordingly.

Extending the algorithm to find factors also on Superscalar architectures

Kindly, check the future works file for detailed explainations(approaches/inspirations).