Generalized Loop-Unrolling: a Method for Program Speed-Up

J. C. Huang and T. Leng
Department of Computer Science
The University of Houston
Houston, TX 77204-3475
jhuang|leng@cs.uh.edu

Abstract - It is well-known that, to optimize a program for speed-up, efforts should be focused on the regions where the payoff will be greatest. Loop constructs in a program represent such regions. In the literature, it has been shown that a certain degree of speed-up can be achieved by loop unrolling. The technique published so far, however, appears to be applicable to FOR-loops only. This paper presents a generalized loop-unrolling method that can be applied to any type of loop construct. Possible complications in its applications, together with some experimental results, are discussed in detail.

Introduction

There has been considerable effort to develop source-to-source transformation methods that restructure loop constructs to expose possibilities for parallelism. Most published loop restructuring methods are designed for countable loops, where the iteration count can be determined without executing the loop. One such method, called *loop unrolling* [2], is designed to unroll FOR loops for parallelizing and optimizing compilers. To illustrate, consider the following loop:

```
for (i = 1; i <= 60; i++) a[i] = a[i] * b + c;
```

This FOR loop can be transformed into the following equivalent loop consisting of multiple copies of the original loop body:

```
for  (i = 1; i <= 60; i+=3)
{
   a[i] = a[i] * b + c;
   a[i+1] = a[i+1] * b + c;
   a[i+2] = a[i+2] * b + c;
}
```

The loop is said to have been unrolled twice, and the unrolled loop should run faster because of reduction in loop overhead.

Loop unrolling was initially developed for reducing loop overhead and for exposing instruction-level parallelism for machines with multiple functional units. More recently it has also been applied in conjunction with instruction scheduling for pipelined and RISC architectures[2]. By increasing the size of the loop body, the instruction scheduler can often produce a shorter schedule for the unrolled loop.

Unlike FOR loops operating on arrays, which can be unrolled simply by modifying the counter and termination condition of the loop as illustrated above, WHILE loops are generally more difficult to unroll and seldom mentioned in the subject area. It is so mainly because of difficulty in determining the termination condition of an unrolled WHILE loop.

In this paper, we present a new method for unrolling a WHILE loop and discuss problems that may be encountered in its application. The method is "generalized" in the sense that it can be applied to any type of loop construct, including implicit one, because any loop construct can be rewritten as a WHILE loop.

Main Results

We assume that loops are written in the form: **while** B **do** S, the semantic of which is defined as usual. This assumption should not cause any loss in generality because any loop of other form can be rewritten in this form. We shall refer to B as the *loop predicate* and S as the *loop body*. By using the analysis method we have developed [1], it is easy to show that the following equivalence relations hold.

while B **do** S; ⇔ **while** B wp(S, B) **do begin** S; S **end; while** B **do** S; (A)

where ⇔ stands for the equivalence relation, and wp(S, B) the weakest precondition of S with respect to postcondition B.

We shall say that the equivalent program on the right-hand side is obtained by *unrolling the loop once.* Observe that if any data will cause the loop on the left-hand side to iterate n times then it will cause the first loop on the right to iterate ën/2û times, and the second 0 or 1 time. Now if we can simplify B wp(S, B) or S; S, as explained below, we can unroll the loop to achieve a certain degree of computational speed-up.

A WHILE loop can be unrolled further. It can be shown that

while B **do** S;
⇔ **while** B wp(S, B) wp(S, wp(S, B)) **do begin** S; S; S **end**; **while** B **do** S; (B)

Expression on the right-hand side shows how to unroll a loop twice. Needless to say, a loop can be unrolled thrice, four times, and so on and so forth.

Thus, given a loop construct of the form: **while** B **do** S; we may be able to speed up its execution by following the steps listed below to unroll it once:

1. Form wp(S, B), the weakest precondition of S with respect to B.
2. Unroll the loop once by replacing it with a sequence of two loops:

   while B   wp(S, B) **do begin** S; S **end**;

   while B **do** S;
3. Simplify the predicate B   wp(S, B) and the loop body S;S to speed up.

The second loop is the original loop. Since it will be iterated no more than once, in theory it can be replaced by the construct: **if** B **then** S. From the software engineering point of view, however, it is more desirable to leave it unmodified because the original loop is easier to understand. Besides, if the loop unrolling becomes no longer desirable for some reason, all we need to do is to delete the first loop. To this end, we suggest that the first loop be always clearly demarcated with appropriate comments. If it needs to be temporarily removed, say, for debugging purposes, it can be readily accomplished by using the host-language facility to turn the first loop into a comment.

The analysis method described in [1] is particularly useful in Step 3.

To illustrate, consider the following example. (NOTE: Throughout this paper, the source code of all example programs are written in C and listed in Courier.)

Example 1. A loop for computing the quotient, q, of dividing b into a:

```
     q=0;
     while (a>=b)
     {
          a=a-b;
          q++;
     }

⇔    q=0;
     while (a>=b && a>=2*b)    //unrolled loop
     {
          a=a-b;
          q++;
          a=a-b;
          q++;
     }                         //end of the unrolled loop
     while (a>=b)
     {
          a=a-b;
          q++;
     }
```

```
⇔    q=0;
     while (a>=2*b)            //unrolled loop
     {
          a=a-2*b;
          q=+2;
     }                         //end of the unrolled loop
     while (a>=b)
     {
          a=a-b;
          q++;
     }

⇔    q=0;
     x=2*b;                    //unrolled loop
     while (a>=x)
     {
          a=a-x;
          q=+2;
     }                         //end of the unrolled loop
     while (a>=b)
     {
          a=a-b;
          q++;
     }
```

Our experimental results show that this unrolled loop is able to achieve a speed-up factor very close to 2, and if we unroll the loop k times, we can achieve a speed-up factor of k. Speed factor is defined to be the ration between the CPU time required to execute the modified program and that required to execute the original program.

Discussions

Although the programs on both sides of ⇔ are logically equivalent in theory, it may not be completely so in practice. Note that in the equivalence relation (A) given above, the loop predicate of the first loop on the right-hand side is B   wp(S, B). Since the logical operation of " " (and) is commutative, it should make no difference in theory if the loop predicate is written as B   wp(S, B) or wp(S, B)   B, or if B or wp(S, B) is evaluated first. In practice, however, B should be evaluated first, and if it is false, wp(S, B) should not be evaluated at all because it determines if the second part of the unrolled loop should be executed.

This would present no problem if (1) components of the loop predicate is always written in the correct order, and (2) the program is compiled and executed in an environment where the components of the loop predicate are evaluated in the order given, and the evaluation is terminated as soon as one is found to be false. Otherwise, the unrolled loop may have run-time errors that would not occur in the original loop. For instance, consider the following program.

Example 2. A loop for finding GCD (Greatest Common Divisor) of two positive integers a and b by using Euclid's algorithm, the final result of the loop, which is the GCD of a and b, is stored in a.

```
     while (b>0)
     {    r = a % b;
          a = b;
          b = r;
      }
```

By unrolling the loop once and simplifying the result we obtain

```
⇔    while (b>0) && (r = a % b > 0) //unrolled loop
     {    a=b;
          b=b % r;
     }                              //end of the unrolled loop
     while (b>0)
     {    r = a % b;
          a = b;
          b = r;
     }
```

Note that the value of a%b can be properly computer only if b>0, and there will be an arithmatic fault in computing a%b if b=0. In order to avoid such fault, we can modify the loop as follows.

```
⇔    while (b>0)
     {    a = a % b;
          if (a>0) b = b % a;
          else
          {    r=a;
               a=b;
               b=r;
               break;
          }
     }
```

The speed-up achieved in this case is rather limited because the gain mostly comes from reduction of instructions in the loop bodies. In an experiment using 10,000 pairs of random integers, the average speed-up factor is approximately 1.05.

A possible alternative is to make use of exception handling. Some programming languages, such as C++, Smalltalk, and Ada, provide mechanism for the programmer to handle exceptions. Although the penalty of exception handling is high, it is deemed a viable solution because exceptions of this sort occur infrequently.

Problems of similar nature may arise if pointers are involved.

Example 3. A loop for traversing a linked list and counting the nodes traversed:

```
count=0;
while (lp != NULL)
{    lp = lp->next;
     count++;
}
```

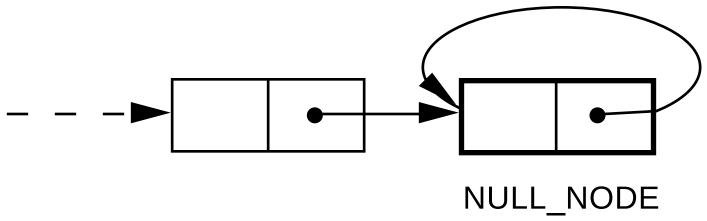
After unrolling the loop twice, it becomes:

```
⇔    count=0;
     lp1=lp->next;
     lp2=lp1->next;
     while (lp2 != NULL)            //unrolled loop
     {    count=+3;
          lp  = lp2->next;
          lp1 = lp ->next;
          lp2 = lp1->next;
     }                              //end of the unrolled loop
     while (lp != NULL)
     {    lp = lp->next;
          count++;
     }
```

Since we do not know apriori how many more items will be in the linked list, computation of lp1 and lp2 may cause rum-time errors, which would not occur in the original loop. To avoid this problem, we need to modify the unrolled loop to prevent the pointer from getting out of range:

```
⇔    count=0;
     while (lp != NULL)
     {    lp = lp->next;
          if (lp != NULL)
          {   lp = lp->next;
              if (lp != NULL)
              count+=3;
              else {count+=2; break;}
          }
          else {count++; break;}
     }
```

Another possible solution is to attach a special sentinel node named NULL_NODE at the end of the list. The link field of this node points to the node itself as depicted below.



Note that a self-pointing sentinel can be used in other applications to unroll a loop whose length is unknown at the beginning of its execution.

Its use allows us to unroll the loop *k* times, and test at the end of every *k* iterations whether the pointer is at the sentinel node:

```
⇔    count=0;
     lp[0]=lp;
     lp[1]=lp->next;
     lp[2]=lp1->next;
     ⋮
     lp[k-1]=lp[k-2] ->next;
     while (lp[k-1] != NULL_LIST)   //unrolled loop
     {    count+=k;
          lp[0] = lp[k-1]->next;
          lp[1] = lp[0]->next;
          lp[2] = lp[1]->next;
          ⋮
          lp[k-1]=lp[k-2] ->next
     }                              //end of the unrolled loop
     while (lp[0] != NULL_LIST)
     {    lp[0] = lp[0]->next;
          count++;
     }
```

We would expect improved performance from the transformation. Since the gains are not only from reducing the loop overhead but also from compacting the computation performed in the loop bodies. In an experiment where the loop is unrolled thrice and the linked lists used are of sizes 100 and 500, the average speed-up factor is approximately 1.19.

Example 4. The power algorithm for computing $r \equiv a^n \pmod{m}$ where a, n, m, and r are integers.

```
     r=1;
     while (n>0)
     {    d = n % 2;
          if (d=1)
          {    r = r * a;
               r = r % m
          }
          a = a * a;
          a = a % m;
          n = (n-d) / 2;
     }
```

After unroll the loop three times, the loop predicate becomes (n>0) && (n>2) && (n>4). This condition can be simplified to (n>4). If the original loop will iterate I times, the unrolled loop will iterate a maximum of (I/3)+2 times (I/3 unrolled iterations and up to 2 iterations of the original loop). In this case because of data dependence between iterations, not much instruction reduction can be achieved. Therefore, the performance gain can only be obtained from the reduction in the number of condition tests.

Example 5. Compute the GCD of two positive integers x and y by using the so-called *binary algorithm*. This algorithm requires no division operations (which may be time-consuming), and relies solely on the operations of subtraction, shifts, and bitwise operations. It has been proved that binary algorithm is about 15% to 20% faster than Euclid's algorithm (Example 2).

```
L1:  while (((x | y) & 1) == 0)
     {    x >>= 1;
          y >>= 1;
          ++common_power_of_two;
     }
     while ((x & 1) == 0) x >>= 1;
L2:  while (y != 0)
L2.1:{   while ((y & 1) == 0) y >>= 1;
         temp = y;
         y = abs(x - y);
         x = temp;
     }
     gcd = x << common_power_of_two;
```

First, we unroll loop L1 three times:

```
⇔    while (((x | y) & 7) == 0)     //unrolled loop
     {    x >>= 3;
          y >>= 3;
          common_power_of_two+=3;
     }                              //end of the unrolled loop
     while (((x | y) & 1) == 0)
     {    x >>= 1;
          y >>= 1;
          ++common_power_of_two;
     }
```

Similarly, we unroll both loops L2 and L2.1 two times as follow:

```
⇔    while ( y!= 0)
     {    while ((y&3) == 0) y >>= 2; //unrolled loop L2.1
          if ((y&1) == 0) y >>= 1;   //end of the unrolled loop
          if (x != y)
          {  temp = y;
             y = abs(x - y);
             while ((y&3) ==0) y >>= 2; //unrolled loop L2.1
             if ((y&1) == 0) y >>= 1;  //end of the unrolled loop
             x = y;
             y = abs(temp - y);
          }
          else
          {    temp = y;
               y = 0;
               break;
          }
     }
```

For this example program, the experimental results show that we are able to achieve a speed-up factor of 1.15 in average.

In conclusion, we have presented a method for speeding up programs by unrolling its loop constructs. Our preliminary investigation reveals that, for most real-world programs, the degree of speed-up that can be achieved is modest. Nevertheless, it can be easily done, and does not require any special hardware to implement -- it works on any platform. Since the degree of speed-up is proportional to the number of times a loop is iterated, it is economically justifiable to apply the method only to programs that will be used often (such as library routines), and to loops that may be iterated a great many number of times during execution.

References

[1] J. C. Huang, "State Constraints and Pathwise Decomposition of Programs", IEEE Tran. on Software Engineering. Vol 16. No. 8. August 1990.

[2] John L. Hennessy; David A. Patterson, "Computer Architecture A Quantitative Approach", 2nd Edition, 1995

[3] L.J. Hendren; G.R. Gao, "Designing Programming Languages for Analyzability: A Fresh Look at Pointer Data Structure", IEEE, 1992.

[4] Michael J. Wolfe, "High Performance Compilers for Parallel Computing", 1996.