1. Redundant Expression Elimination

C program:

```
#include <stdio.h>

int ret10()
{
  int i;
  int sum = 0;
  for( i = 0; i < 10; i++ )
    if( i % 2 )
      sum += 2;

  return sum;
}

int ret1()
{
  return 1;
}

int main()
{

  int a = ret10();
  int b = ret1();
  int x;
  int y;

  x = a * b;
  y = ( a * b ) * a + 10;

  printf( "x = %d, y = %d\n", x, y );

  return 0;
}
```

Non−optimized (some parts edited out):

```
.file"p1.c"
gcc2_compiled.:
.section".text"
.align 4
.global main
.type main,#function
.proc04
main:
   !#PROLOGUE# 0
   save %sp,−128,%sp
   !#PROLOGUE# 1
   call ret10,0
   nop
   st %o0,[%fp−20]
   call ret1,0
   nop
   st %o0,[%fp−24]
   ld [%fp−20],%o0
   ld [%fp−24],%o1
   call .umul,0
   nop
   st %o0,[%fp−28]
   ld [%fp−20],%o0
   ld [%fp−24],%o1
   call .umul,0
   nop
   mov %o0,%o1
   mov %o1,%o0
   ld [%fp−20],%o1
   call .umul,0
   nop
   add %o0,10,%o1
   st %o1,[%fp−32]
   sethi %hi(.LLC0),%o1
   or %o1,%lo(.LLC0),%o0
   ld [%fp−28],%o1
   ld [%fp−32],%o2
   call printf,0
   nop
   mov 0,%i0
   b .LL8
   nop
   .LL8:
   ret
   restore
```

Optimized by gcc (some parts edited out):

```
.file"p1.c"
gcc2_compiled.:
.section".text"
.align 4
.global main
.type main,#function
.proc04
main:
   !#PROLOGUE# 0
   save %sp,−112,%sp
   !#PROLOGUE# 1

   call ret10,0   ; %o0 = a = ret10()
   mov 0,%i0

   call ret1,0    ; %o0 = b = ret1()
   mov %o0,%l1    ; %l1 = a

   mov %o0,%o1    ; %o1 = b
   call .umul,0   ; %o0 = a * b
   mov %l1,%o0

   mov %o0,%l0    ; %l0 = a * b
                  ; a * b is reused here
                  ; to compute a * b * a
   call .umul,0   ; %o0 = a * b * a
   mov %l1,%o1

   mov %o0,%o2    ; %o2 = a * b * a
   sethi %hi(.LLC0),%o0
   or %o0,%lo(.LLC0),%o0
   mov %l0,%o1    ; %o1 = a * b
   call printf,0
   add %o2,10,%o2 ; %o2 = a * b * a + 10

   ret
   restore
```

The unoptimized version computes the value a * b twice unnecessarily. The optimized one saves the value in a register and reused it when computing a * b * a, which requires only one multiplication now.

2. Partially Redundant Expression Elimination

C code:

```
#include <stdio.h>

int main()
{

  int x, y, z, u;

  scanf( "%d %d", & x, & y );

  if( x > 0 )
    {
      z = 1;
    }
  else
    {
      printf( "Test." );
      z = x * y + x;
    }
  u = x * y;

  printf( "%d %d", u, z );

  return u + z;
}
```

Non−optimized (some parts edited out):

```
    .file    "p2.c"
gcc2_compiled.:
    [........]
main:
    !#PROLOGUE# 0
    save %sp,−128,%sp
    !#PROLOGUE# 1
    add %fp,−20,%o1
    add %fp,−24,%o2
    [........]
    call scanf,0
    nop
    ld [%fp−20],%o0
    cmp %o0,0
    ble .LL2
    nop
    mov 1,%o0
    st %o0,[%fp−28]
    b .LL3
    nop
.LL2:
    [........]
    ld [%fp−20],%o0
    ld [%fp−24],%o1
    call .umul,0
    nop
    ld [%fp−20],%o1
    add %o0,%o1,%o0
    st %o0,[%fp−28]
.LL3:
    ld [%fp−20],%o0
    ld [%fp−24],%o1
    call .umul,0
    nop
    st %o0,[%fp−32]
    sethi %hi(.LLC0),%o1
    or %o1,%lo(.LLC0),%o0
    ld [%fp−32],%o1
    ld [%fp−28],%o2
    call printf,0
    nop
    ld [%fp−32],%o0
    ld [%fp−28],%o1
    add %o0,%o1,%o0
    mov %o0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
    [........]
```

Optimized by hand (some parts edited out):

```
    .file    "p2.c"
gcc2_compiled.:
    [........]
main:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1
    [........]
    call scanf,0
    add %fp,−24,%o2
    ld [%fp−20],%o0
    cmp %o0,0
    bg .LL3
    mov 1,%l1
    sethi %hi(.LLC1),%o0
    call printf,0
    or %o0,%lo(.LLC1),%o0
    ld [%fp−20],%l0
    ld [%fp−24],%o1
    call .umul,0
    mov %l0,%o0
    b .LL4
    add %o0,%l0,%l1
.LL3:
    ld [%fp−20],%o0
    call .umul,0
    ld [%fp−24],%o1
.LL4:
    mov %o0,%i0
    mov %i0,%o1
    or %l2,%lo(.LLC0),%o0
    call printf,0
    mov %l1,%o2
    ret
    restore %i0,%l1,%o0
    [........]
```

In the unoptimized version, the operation x * y is performed twice on the path .LL2, .LL3 (else branch). After optimization, each branch executes x * y only once.

3. Constant Propagation

C program:

```
#include <stdio.h>

int ret10()
{
  int i;
  int sum = 0;
  for( i = 0; i < 10; i++ )
    if( i % 2 )
      sum += 2;

  return sum;
}
```

```
int main()
{

  int a = 3;
  int b = a * ret10() + 17;

  printf( "a = %d, b = %d\n",  a, b );

  return 0;
}
```

Non−optimized (some parts edited out):

```
.file"p3.c"
gcc2_compiled.:
.section".text"
.align 4
.global main
.type main,#function
.proc04
main:
   !#PROLOGUE# 0
   save %sp,−120,%sp
   !#PROLOGUE# 1

   mov 3,%o0
   st %o0,[%fp−20]  ; %o0 = a = 3

   call ret10,0
   nop; %o0 = ret10()

   mov %o0,%o1
   mov %o1,%o0
   ld [%fp−20],%o1  ; a = 3 is accessed as
                    ; any other variable
   call .umul,0
   nop; %o0 = a * ret10()

   add %o0,17,%o1   ; %o0 = a * ret10() + 17
   st %o1,[%fp−24]

   sethi %hi(.LLC0),%o1
   or %o1,%lo(.LLC0),%o0
   ld [%fp−20],%o1  ; %o1 = a
                    ; a = 3 is accessed as
                    ; any other variable
   ld [%fp−24],%o2  ; %o2 = a * ret10() + 17
   call printf,0
   nop

   mov 0,%i0
   b .LL7
   nop
   .LL7:
   ret
   restore
```

Optimized by gcc (some parts edited out):

```
.file"p3.c"
gcc2_compiled.:
.section".text"
.align 4
.global main
.type main,#function
.proc04
main:
   !#PROLOGUE# 0
   save %sp,−112,%sp
   !#PROLOGUE# 1

   call ret10,0    ; %o0 = ret10()
   mov 0,%i0

   mov %o0,%o1     ; %o1 = ret10()
   call .umul,0    ; %o0 = 3 * ret10()
   mov 3,%o0       ; %o0 = 3

   mov %o0,%o2     ; %o2 = 3
   sethi %hi(.LLC0),%o0
   or %o0,%lo(.LLC0),%o0
   mov 3,%o1       ; %o1 = 3
   call printf,0
   add %o2,17,%o2

   ret
   restore
```

The variable a is given the constant value 3. In the unoptimized version, the value is explicitly loaded from the stack each time the variable a is accessed. The optimized version replaces all accesses to a with its constant value, 3.

4. Copy Propagation

C program:

```
include <stdio.h>

int ret10()
{
  int i;
  int sum = 0;

  for( i = 0; i < 10; i++ )
    if( i % 2 )
      sum += 2;

  return sum;
}
```

```
int main()
{

  int a = ret10();
  int b = a;
  int i;
  int sum = 0;

  for( i = 0; i < 10; i++ )
    if( i % 2 )
      sum += a;
    else
      sum += b;

  printf( "a = %d, b = %d, sum = %d\n",
               a, b, sum );

  return 0;
}
```

Non−optimized (some parts edited out):

```
.file"p4.c"
gcc2_compiled.:
.section".text"
.align 4
.global main
.type main,#function
.proc04
main:
   st %o0,[%fp−20]    ; [%fp−20] = a
   ld [%fp−20],%o0
   st %o0,[%fp−24]    ; [%fp−24] = b
   st %g0,[%fp−32]    ; [%fp−32] = sum
   st %g0,[%fp−28]    ; [%fp−28] = i
   [........]
   .LL11:
   ld [%fp−28],%o0
   and %o0,1,%o1
   cmp %o1,0
   be .LL12
   nop
   ld [%fp−32],%o0
   ld [%fp−20],%o1
   add %o0,%o1,%o0    ; sum = sum + a
   st %o0,[%fp−32]
   b .LL10
   nop
   .LL12:
   ld [%fp−32],%o0
   ld [%fp−24],%o1
   add %o0,%o1,%o0    ; sum = sum + b
   st %o0,[%fp−32]
   [........]
   .LL9:
   [........]
   call printf,0
   nop
   mov 0,%i0
   b .LL7
   nop
   .LL7:
   ret
   restore
```

Optimized by gcc (some parts edited out):

```
    .file    "p4.c"    ; this optimization
                       ; required −O2 level
                       ; for gcc
gcc2_compiled.:
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    call ret10,0
    nop                ; %o0 = a
    mov 0,%o3          ; %o3 = sum
    mov 0,%o2          ; %o2 = i
.LL13:
    andcc %o2,1,%g0
    add %o2,1,%o2      ; i++
    cmp %o2,9          ; i <= 9
    ble .LL13
    add %o3,%o0,%o3    ; sum = sum + a

    mov %o0,%o1        ; %o1 = a
    sethi %hi(.LLC0),%o0
    or %o0,%lo(.LLC0),%o0
    call printf,0
    mov %o1,%o2        ; %o2 = a
                       ; b has been
                       ; completely replaced
                       ; by a in the code

    ret
    restore %g0,0,%o0
```

The value of the variable b is equal to a, but the unoptimized version accesses b when needed instead of accessing a. The optimized version uses only the variable a (even in place of b) throughout the section of the program where the two variables are equal.

5. Constant Folding

C program:

```
#include <stdio.h>

int main()
{

  int x;

  scanf( "%d", & x );

  return x + 3 * 4;
}
```

Non−optimized by hand! (some parts edited out):

```
    .file    "p5.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d"
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1
    add %fp,−20,%o1
    sethi %hi(.LLC0),%o2
    or %o2,%lo(.LLC0),%o0
    call scanf,0
    nop
    ld [%fp−20],%o1
    add %g0,3,%o2
    add %g0,4,%o0
    umul %o0,%o2,%o0
    add %o1,%o0,%o0
    mov %o0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
```

Compiled by gcc without optimizations! (some parts edited out):

```
    .file    "p5.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d"
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1
    add %fp,−20,%o1
    sethi %hi(.LLC0),%o2
    or %o2,%lo(.LLC0),%o0
    call scanf,0
    nop
    ld [%fp−20],%o1
    add %o1,12,%o0
    mov %o0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
```

The expression 3 * 4, used in computing the return value from the main function, is computed each time by the program (at run−time) when not optimized. The optimized version uses the constant value of the expression (as computed at compile time) when using it in computation.

It is interesting to note that gcc, even when run with −O0 (no optimization), insisted on folding the constant expression. Thus, I had to "un−optimize" the code by hand, to illustrate the point of this optimization.

6. Dead Code Elimination

C program:

```
#include <stdio.h>

int main()
{

  int a;
  int b;

  scanf( "%d", & a );
  b = a * 2;

  return a;
}
```

Non−optimized (some parts edited out):

```
    .file    "p6.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d"
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1
    add %fp,−20,%o1
    sethi %hi(.LLC0),%o2
    or %o2,%lo(.LLC0),%o0
    call scanf,0
    nop
    ld [%fp−20],%o0
    mov %o0,%o1
    sll %o1,1,%o0
    st %o0,[%fp−24]
    ld [%fp−20],%o0
    mov %o0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
```

Optimized by gcc (some parts edited out):

```
    .file    "p6.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d"
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0    ; the instruction
                     ; b = a * 2;
                     ; was eliminated
    save %sp,−120,%sp
    !#PROLOGUE# 1
    sethi %hi(.LLC0),%o0
    or %o0,%lo(.LLC0),%o0
    call scanf,0
    add %fp,−20,%o1
    ld [%fp−20],%i0
    ret
    restore
```

The value of the variable b is not used anywhere in the program. The unoptimized version still executes the instructions that just assign to b. The optimized version eliminated the variable completely, along with the instructions that assigned to it, but had not other use.

7. Loop Invariant Code Motion

C program:

```
#include <stdio.h>
#include <math.h>

int main()
{

  int n;
  int i;
  int a;
  int b;
  int sum = 0;

  scanf( "%d", & n );
  scanf( "%d %d", & a, & b );

  for( i = 0; i < n; i++ )
    {
      sum += a * b;
    }

  return sum;
}
```

Non−optimized (some parts edited out):

```
    .file    "p7.c"
gcc2_compiled.:
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−136,%sp
    !#PROLOGUE# 1
    [........]
    call scanf,0
    nop
    add %fp,−28,%o1
    add %fp,−32,%o2
    sethi %hi(.LLC1),%o3
    or %o3,%lo(.LLC1),%o0
    call scanf,0
    nop
    st %g0,[%fp−24]
.LL2:
    ld [%fp−24],%o0
    ld [%fp−20],%o1
    cmp %o0,%o1
    bl .LL5
    nop
    b .LL3
    nop
.LL5:
    ld [%fp−28],%o0
    ld [%fp−32],%o1
    call .umul,0
    nop
    ld [%fp−36],%o1
    add %o1,%o0,%o0
    st %o0,[%fp−36]
.LL4:
    ld [%fp−24],%o0
    add %o0,1,%o1
    st %o1,[%fp−24]
    b .LL2
    nop
.LL3:
    ld [%fp−36],%o0
    mov %o0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
```

Optimized by hand (some parts edited out):

```
    .file    "p7.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d"
    .align 8
.LLC1:
    .asciz    "%d %d"
    .global .umul
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−128,%sp
    !#PROLOGUE# 1
    mov 0,%i0
    sethi %hi(.LLC0),%o0
    or %o0,%lo(.LLC0),%o0
    call scanf,0
    add %fp,−20,%o1
    sethi %hi(.LLC1),%o0
    or %o0,%lo(.LLC1),%o0
    add %fp,−24,%o1
    call scanf,0
    add %fp,−28,%o2
    ld [%fp−20],%o0
    cmp %i0,%o0
    bge .LL3
    mov 0,%l0
    ld [%fp−24],%o0
    call .umul,0
    ld [%fp−28],%o1
    ld [%fp−20],%o1
.LL5:
    add %l0,1,%l0
    cmp %l0,%o1
    bl .LL5
    add %i0,%o0,%i0
.LL3:
    ret
    restore
    [........]
```

The expression a * b, used in the loop, is computed each time the loop is executed, although it does not change. The optimized version computed the expression only once, outside the loop, and reused the computation subsequently.

8. Scalarization

C program:

```
#include <stdio.h>

int main()
{

  int a[ 10 ] = { 1, 2, 4, 8, 16,
                  32, 64, 128, 256, 512 };
  int n, i, sum = 0;

  scanf( "%d", & n );

  for( i = 0; i < 10; i++ )
    {
      sum = sum + a[ n ];
    }

  return sum;
}
```

Non−optimized (some parts edited out):

```
    .file    "p8.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d"
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−168,%sp
    !#PROLOGUE# 1
    [........]
    call scanf,0
    nop
    st %g0,[%fp−64]
.LL2:
    ld [%fp−64],%o0
    cmp %o0,9
    ble .LL5
    nop
    b .LL3
    nop
.LL5:
    ld [%fp−60],%o0
    mov %o0,%o1
    sll %o1,2,%o0
    add %fp,−56,%o1
    ld [%fp−68],%o2
    ld [%o1+%o0],%o0
    add %o2,%o0,%o1
    st %o1,[%fp−68]
.LL4:
    ld [%fp−64],%o0
    add %o0,1,%o1
    st %o1,[%fp−64]
    b .LL2
    nop
.LL3:
    ld [%fp−68],%o0
    mov %o0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
```

Optimized by gcc (some parts edited out):

```
    .file    "p8.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d"
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−160,%sp
    !#PROLOGUE# 1
    [........]
    call scanf,0
    add %fp,−60,%o1
    mov 0,%o2
    ld [%fp−60],%o0
    sll %o0,2,%o0
    add %fp,−56,%o1
    ld [%o1+%o0],%o0
.LL5:
    add %o2,1,%o2
    cmp %o2,9
    ble .LL5
    add %i0,%o0,%i0
    ret
    restore
```

The array element a[n] is accessed in the loop. The unoptimized version computes its address and loads its value each time the loop is executed, although the array element is fixed. After optimization, the address of the element is computed and its value is loaded only once, before the loop starts, and then the value (saved in a register) is reused in the loop.

9. Local Register Allocation

C program:

```
#include <stdio.h>

int main()
{

  int a, b, c;

  scanf( "%d, %d, %d", & a, & b, & c );

  a = b + c;
  b = b + c − 2;
  a = a + c − 1;
  b = b + c − 2;
  a = b + c − a;

  return a + b − c;
}
```

Non−optimized (some parts edited out):

```
    .file    "p9.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d, %d, %d"
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−128,%sp
    !#PROLOGUE# 1
    [........]
    call scanf,0
    nop
    [........]
    [........]
    [........]
    ld [%fp−24],%o0      ; computation of
    ld [%fp−28],%o1      ; a = b + c − 2;
    add %o0,%o1,%o0
    add %o0,−2,%o1
    st %o1,[%fp−24]
    [........]
    ld [%fp−20],%o0      ; computation of
    ld [%fp−24],%o1      ; a + b − c
    add %o0,%o1,%o0      ; for return
    ld [%fp−28],%o1
    sub %o0,%o1,%o0
    mov %o0,%i0

    b .LL1
    nop
.LL1:
    ret
    restore
```

Optimized by gcc (some parts edited out):

```
.LLC0:
    .asciz    "%d, %d, %d"
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−128,%sp
    !#PROLOGUE# 1
    sethi %hi(.LLC0),%o0
    or %o0,%lo(.LLC0),%o0
    add %fp,−20,%o1    ; %o1 = a
    add %fp,−24,%o2    ; %o2 = b
    call scanf,0
    add %fp,−28,%o3    ; %o3 = c

    ld [%fp−28],%i0
    ld [%fp−24],%o1
    add %o1,%i0,%o1
    add %o1,−2,%o5
    add %o5,%i0,%o3
    add %o3,−2,%o3     ; %o3 = b + c − 2
    add %o1,%i0,%o0
    add %o0,−1,%o0
    add %o3,%i0,%o2    ; %i0 = a + b − c
    sub %o2,%o0,%o2
    st %o1,[%fp−20]
    add %o2,%o3,%o4
    st %o5,[%fp−24]
    st %o0,[%fp−20]
    sub %o4,%i0,%i0
    st %o3,[%fp−24]
    st %o2,[%fp−20]
    ret
    restore
```

The section of the main function after the call to scanf() is a basic block (does not contain any jumps or calls that interrupt the sequential flow). The unoptimized version loads and stores each variable for every source code instruction. After optimization, all variables are loaded once in registers, the computations are performed on the registers, and then the values are stored into the respective memory locations.

10. Global Register Allocation

C program:

```
include <stdio.h>

int x;

int compute( int a, int b )
{
  int m, n;

  m = a + b + x;
  n = a − b − x;
  m = n − x;

  return m + n;
}
```

```
int main()
{

  int i, j;

  scanf( "%d", & x );

  for( i = 0; i < x; i++ )
    {
      j = j + i * 2;
    }

  return compute( i, j );
}
```

Non−optimized (some parts edited out):

```
    .file    "p10.c"
gcc2_compiled.:
.section    ".text"
    .align 4
    .global compute
    .type     compute,#function
    .proc    04
compute:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1
    st %i0,[%fp+68]
    st %i1,[%fp+72]
    ld [%fp+68],%o0
    ld [%fp+72],%o1
    add %o0,%o1,%o0
    sethi %hi(x),%o1
    ld [%o1+%lo(x)],%o2
    add %o0,%o2,%o0
    st %o0,[%fp−20]
    ld [%fp+68],%o0
    ld [%fp+72],%o1
    sub %o0,%o1,%o0
    sethi %hi(x),%o1
    ld [%o1+%lo(x)],%o2
    sub %o0,%o2,%o0
    st %o0,[%fp−24]
    sethi %hi(x),%o0
    ld [%fp−24],%o1
    ld [%o0+%lo(x)],%o0
    sub %o1,%o0,%o1
    st %o1,[%fp−20]
    ld [%fp−20],%o0
    ld [%fp−24],%o1
    add %o0,%o1,%o0
    mov %o0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
```

Optimized by gcc (some parts edited out):

```
    .file    "p10.c"
gcc2_compiled.:
.section    ".text"
    .align 4
    .global compute
    .type     compute,#function
    .proc    04
compute:
    !#PROLOGUE# 0
    !#PROLOGUE# 1
    sethi %hi(x),%g2
    ld [%g2+%lo(x)],%g2
    sub %o0,%o1,%o0
    sub %o0,%g2,%o0
    sub %o0,%g2,%g2
    retl
    add %g2,%o0,%o0
```

The code for the compute() function accesses some variables frequently. After optimization all the variables were allocated to registers, while before they were loaded before each operation and stored afterwards. The procedure−wide allocation almost eliminated loads and stores.

11. Inter−procedural Register Allocation

C program:

```
#include <stdio.h>

int x;

int call1()
{
  return x + 2;
}

int call2()
{
  return x * 2;
}

int main()
{

  scanf( "%d", & x );

  printf( "1 = %d, 2 = %d\n",
               call1(), call2() );

  return 0;
}
```

Non−optimized (some parts edited out):

```
    .file    "p11.c"
gcc2_compiled.:
    [........]
call1:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    sethi %hi(x),%o0
    ld [%o0+%lo(x)],%o1
    add %o1,2,%o0
    mov %o0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
    [........]
call2:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    sethi %hi(x),%o0
    ld [%o0+%lo(x)],%o1
    mov %o1,%o0
    sll %o0,1,%o1
    mov %o1,%i0
    b .LL2
    nop
.LL2:
    ret
    restore
    [........]
```

Optimized by hand (some parts edited out):

```
    .file    "p11.c"
gcc2_compiled.:
    [........]
call1:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    add %g1,2,%o0
    mov %o0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
    [........]
call2:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    mov %g1,%o0
    sll %o0,1,%o1
    mov %o1,%i0
    b .LL2
    nop
.LL2:
    ret
    restore
    [........]
main:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    sethi %hi(.LLC0),%o1
    or %o1,%lo(.LLC0),%o0
    sethi %hi(x),%o2
    or %o2,%lo(x),%o1
    call scanf,0
    nop
    sethi %hi(x),%o2
    ld [%o2+%lo(x)],%g1
    call call1,0
    nop
    [........]
```

Three procedures (main, call1, call2) access the same global data (the variable x). In the unoptimized version, the variable was loaded and stored in memory separately in each procedures. The optimization used a global register, that is not affected by shifting register windows, to load that variable once, and have it used by all three procedures directly from the register.

12. Register Targeting

C program:

```
#include <stdio.h>

int main()
{
  int i, sum = 0;

  scanf( "%d", & i );

  for( ; i > 0; i++ )
    sum += i;

  return sum;
}
```

Non−optimized (some parts edited out):

```
    .file    "p12.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d"
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1
    st %g0,[%fp−24]         ; sum
    add %fp,−20,%o1         ; i
    sethi %hi(.LLC0),%o2
    or %o2,%lo(.LLC0),%o0
    call scanf,0
    nop
.LL2:
    ld [%fp−20],%o0         ; i
    cmp %o0,0
    bg .LL5
    nop
    b .LL3
    nop
.LL5:
    ld [%fp−24],%o0         ; sum
    ld [%fp−20],%o1         ; i
    add %o0,%o1,%o0         ; sum = sum + i
    st %o0,[%fp−24]         ; sum
.LL4:
    ld [%fp−20],%o0         ; i
    add %o0,1,%o1           ; i++
    st %o1,[%fp−20]         ; i
    b .LL2
    nop
.LL3:
    ld [%fp−24],%o0
    mov %o0,%i0          ; result is moved
                         ; into correct
                         ; register
    b .LL1
    nop
.LL1:
    ret
    restore
    [........]
```

Optimized by gcc (some parts edited out):

```
    .file    "p12.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d"
.section    ".text"
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1

    mov 0,%i0         ; sum
    ; result register is used from start

    sethi %hi(.LLC0),%o0
    or %o0,%lo(.LLC0),%o0
    call scanf,0
    add %fp,−20,%o1

    ld [%fp−20],%o0   ; i
    cmp %o0,0
    ble .LL3
    mov %o0,%o1
.LL5:
    mov %o1,%o0
    add %i0,%o0,%i0   ; sum += i
    add %o0,1,%o0     ; i++
    orcc %o0,0,%o1
    bg .LL5           ; i > 0
    st %o0,[%fp−20]   ; i
.LL3:
    ret
    restore
    [........]
```

Register %i0 is assigned (by convention) to have the return value from a function, when in the register window of the function (it becomes %o0 in the caller's window). The unoptimized version performs the computation of the function, and then, at the end, copies the value into %i0. When optimized, the function computes the return value directly in %i0, so no copy is needed.

13. Inter−procedural Code Motion

C program:

```
#include <stdio.h>

int x, y, z;

void read_input()
{
  x = 1;
  y = 2;
  z = x + y;
}

int main()
{
  read_input();

  return x ? y : z;
}
```

Non−optimized (some parts edited out):

```
    .file    "p13.c"
gcc2_compiled.:
.section    ".text"
    .align 4
    .global read_input
    .type     read_input,#function
    .proc    020
read_input:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    sethi %hi(x),%o0
    mov 1,%o1
    st %o1,[%o0+%lo(x)]
    sethi %hi(y),%o0
    mov 2,%o1
    st %o1,[%o0+%lo(y)]
    sethi %hi(z),%o0
    sethi %hi(x),%o1
    sethi %hi(y),%o2
    ld [%o1+%lo(x)],%o1
    ld [%o2+%lo(y)],%o2
    add %o1,%o2,%o1
    st %o1,[%o0+%lo(z)]
.LL1:
    ret
    restore
.LLfe1:
    .size     read_input,.LLfe1−read_input
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    call read_input,0
    nop
    [........]
.LL2:
    ret
    restore
    [........]
```

Optimized by hand (some parts edited out):

```
    .file    "p13.c"
gcc2_compiled.:
.section    ".text"
    .align 4
    .global read_input
    .type     read_input,#function
    .proc    020
read_input:
    !#PROLOGUE# 0
    !#PROLOGUE# 1
    sethi %hi(x),%o3   ; no longer reached
                       ; when read_input() is
                       ; called from main()
    mov 1,%o1
    sethi %hi(y),%o2
    mov 2,%g3
    sethi %hi(z),%o0
    mov 3,%g2
    st %o1,[%o3+%lo(x)]
    st %g3,[%o2+%lo(y)]
    retl
    st %g2,[%o0+%lo(z)]
.LLfe1:
    .size     read_input,.LLfe1−read_input
    .align 4
    .global main
    .type     main,#function
    .proc    04
main:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    call read_input+4,0
    sethi %hi(x),%o3
    [........]
.LL4:
    ret
    restore
    [........]
```

The unoptimized version of the program contained a call to read_input(), and wasted the delay slot by filling it with a no−op. Since no other instructions were available to fill the slot from the main function (the caller), the optimization resorted to moving the first instruction from the callee (read_input) into the caller. Accordingly, the call target in main() was modified to point to the next instruction in read_input(), while keeping the first instruction of read_input() in place, for the case when read_input() is called from other places in the program that cannot perform this inter−procedural code motion (i.e., they have their local instructions to fill the delay slots). The move is safe, since the callee uses the same frame as the caller.

14. Call Inlining

C program:

```
#include <stdio.h>

void read_input( int * x, int * y )
{
  scanf( "%d %d", x, y );
}
```

```
int main()
{

  int a, b;

  read_input( & a, & b );

  return a + b;
}
```

Non−optimized (some parts edited out):

```
    .file    "p14.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d %d"
    [........]
read_input:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    st %i0,[%fp+68]
    st %i1,[%fp+72]
    sethi %hi(.LLC0),%o1
    or %o1,%lo(.LLC0),%o0
    ld [%fp+68],%o1
    ld [%fp+72],%o2
    call scanf,0
    nop
.LL1:
    ret
    restore
    [........]
main:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1
    add %fp,−20,%o0
    add %fp,−24,%o1
    call read_input,0
    nop
    ld [%fp−20],%o0
    ld [%fp−24],%o1
    add %o0,%o1,%o0
    mov %o0,%i0
    b .LL2
    nop
.LL2:
    ret
    restore
    [........]
```

Optimized by gcc (some parts edited out):

```
    .file    "p14.c"
gcc2_compiled.:
.section    ".rodata"
    .align 8
.LLC0:
    .asciz    "%d %d"
    [........]
read_input:
    !#PROLOGUE# 0
    save %sp,−112,%sp
    !#PROLOGUE# 1
    sethi %hi(.LLC0),%o0
    or %o0,%lo(.LLC0),%o0
    mov %i0,%o1
    call scanf,0
    mov %i1,%o2
    ret
    restore
    [........]
main:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1
    sethi %hi(.LLC0),%o0
    or %o0,%lo(.LLC0),%o0
    add %fp,−20,%o1
    call scanf,0
    add %fp,−24,%o2
    ld [%fp−20],%o0
    ld [%fp−24],%i0
    ret
    restore %o0,%i0,%o0
    [........]
```

(Relatively) small procedures can be inlined in the caller's body, thus eliminating the overhead of the jump completely. In the above example, the call to the read_input() function, present in the unoptimized version, was completely replaced by the body of read_input(), as seen in the optimized version. Formal parameters were automatically replaced by actual values. A copy of the original read_input() procedure body was kept.

15. Code Hoisting and Sinking

C program:

```
#include <stdio.h>

int main()
{

  int a, b, c;

  scanf( "%d", & a );

  if( a > 0 )
    {
      b = 2 * a;
      c = 1;
      printf( "Path 1\n" );
    }
  else
    {
      b = 2 * a;
      c = a * a;
      printf( "Path 2\n" );
    }

  printf( "a = %d, b = %d, c = %d\n",
              a, b, c );

  return 0;
}
```

Non−optimized (some parts edited out):

```
    .file    "p15.c"
gcc2_compiled.:
    [........]
main:
    !#PROLOGUE# 0
    save %sp,−128,%sp
    !#PROLOGUE# 1
    add %fp,−20,%o1
    [........]
    call scanf,0
    nop
    ld [%fp−20],%o0
    cmp %o0,0
    ble .LL2
    nop
    ld [%fp−20],%o0
    mov %o0,%o1
    sll %o1,1,%o0
    st %o0,[%fp−24]
    mov 1,%o0
    st %o0,[%fp−28]
    [........]
    call printf,0
    nop
    b .LL3
    nop
.LL2:
    ld [%fp−20],%o0
    mov %o0,%o1
    sll %o1,1,%o0
    st %o0,[%fp−24]
    [........]
    call printf,0
    nop
.LL3:
    [........]
    call printf,0
    nop
    mov 0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
    [........]
```

Optimized by gcc (some parts edited out):

```
    .file    "p15.c"
gcc2_compiled.:
    [........]
main:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1
    [........]
    call scanf,0
    add %fp,−20,%o1
    ld [%fp−20],%o1
    cmp %o1,0
    ble .LL5
    sll %o1,1,%l0
    mov 1,%i0
    sethi %hi(.LLC1),%o0
    b .LL4
    or %o0,%lo(.LLC1),%o0
.LL5:
    call .umul,0
    mov %o1,%o0
    mov %o0,%i0
    sethi %hi(.LLC2),%o0
    or %o0,%lo(.LLC2),%o0
.LL4:
    call printf,0
    nop
    [........]
    call printf,0
    mov %i0,%o3
    ret
    restore %g0,0,%o0
    [........]
```

The two branches of the if statement contain some identical code. The unoptimized version just repeats the code twice, without any changes. The optimized version hoisted the b = 2 * a assignment above the if statement, and sunk the printf() call after the if statement. (Quite remarkably, the printf had different arguments for each branch, and it was still moved out!)

16. Loop Unrolling

C program:

```
#include <stdio.h>

int main()
{

  long n;
  long i;
  long sum = 0l;

  scanf( "%ld", & n );

  for( i = 0l; i < n; i++ )
    {
      sum = sum + i * i;
    }

  return sum % 10;
}
```

Non−optimized (some parts edited out):

```
    .file    "p16.c"
gcc2_compiled.:
    [........]
main:
    !#PROLOGUE# 0
    save %sp,−128,%sp
    !#PROLOGUE# 1
    st %g0,[%fp−28]
    add %fp,−20,%o1
    sethi %hi(.LLC0),%o2
    or %o2,%lo(.LLC0),%o0
    call scanf,0
    nop
    st %g0,[%fp−24]
.LL2:
    ld [%fp−24],%o0
    ld [%fp−20],%o1
    cmp %o0,%o1
    bl .LL5
    nop
    b .LL3
    nop
.LL5:
    ld [%fp−24],%o0
    ld [%fp−24],%o1
    call .umul,0
    nop
    ld [%fp−28],%o1
    add %o1,%o0,%o0
    st %o0,[%fp−28]
.LL4:
    ld [%fp−24],%o0
    add %o0,1,%o1
    st %o1,[%fp−24]
    b .LL2
    nop
.LL3:
    ld [%fp−28],%o1
    mov %o1,%o0
    mov 10,%o1
    call .rem,0
    nop
    mov %o0,%i0
    b .LL1
    nop
.LL1:
    ret
    restore
    [........]
```

Optimized by gcc (some parts edited out):

```
    .file    "p16.c"
gcc2_compiled.:
    [........]
main:
    !#PROLOGUE# 0
    save %sp,−120,%sp
    !#PROLOGUE# 1
    mov 0,%l0
    sethi %hi(.LLC0),%o0
    or %o0,%lo(.LLC0),%o0
    call scanf,0
    add %fp,−20,%o1
    [........]
    mov 0,%i0
    mov %o0,%l1
    andcc %l1,3,%o0
    be,a .LL26      ; go to LL26 on n%4=0
    mov %i0,%o0
    cmp %o0,1
    ble .LL9        ; go to LL9 on n%4=1
    cmp %o0,2
    bg,a .LL10      ; go to LL10 on n%4=2
                    ; otherwise, n%4=3
    mov 1,%i0       ; the loop for i = 0 !
                    ; => sum = 0
                    ; => i gets incremented
.LL10:
    mov %i0,%o0
    call .umul,0
    mov %i0,%o1
    add %l0,%o0,%l0    ; sum = sum + i * i
    add %i0,1,%i0   ; i++
.LL9:
    mov %i0,%o0
    call .umul,0
    mov %i0,%o1
    add %i0,1,%i0   ; i++
    cmp %i0,%l1
    bge .LL3
    add %l0,%o0,%l0    ; sum = sum + i * i
.LL5:
    mov %i0,%o0
.LL26:
    call .umul,0    ; 4 multiplies
    mov %i0,%o1
    add %l0,%o0,%l0
    add %i0,1,%o1
    call .umul,0
    [........]
    call .umul,0
    [........]
    call .umul,0
    mov %o1,%o0
    add %i0,4,%i0
    cmp %i0,%l1
    bl .LL5
    add %l0,%o0,%l0
.LL3:
    [........]
    ret
    restore %g0,%o0,%o0
```

gcc unrolled the loop 4 times.

Depending on the loop count, zero or more (up to 3) loop bodies are executed first, in order to make the left loop count a multiple of 4.

Strength Reduction

To find a non−unit Sparc instruction, the following program was used:

find.c

```
#include <stdio.h>

int main()
{
  long i = 10000000000l;
  double sum = 0.0;

  for( ; i > 0l; i-- )
    {
    }

  return ( ( long )sum ) % 10;
}
```

The body of the loop was alternatively filled with one of the following instructions:

1. `sum = i + 5.0;`
2. `sum = i / 5.0;`

The times obtained were as follows (given as simple averages):

`sum = i + 5.0;`	63.52s
`sum = i / 5.0;`	137.2s

Obviously, the floating point division is not unit time, although a theoretical RISC architecture would imply so.

The program to be strength reduced and the result of the strength reduction:

sr.before.c

```
#include <stdio.h>

int main()
{
  long i = 10000000000l;
  double sum = 0.0;

  for( ; i > 0l; i-- )
    {
      sum = sum + i / 5.0;
    }

  return ( ( long )sum ) % 10;
}
```

sr.after.c

```
#include <stdio.h>

int main()
{
  long i = 10000000000l;
  double sum = 0.0;
  double decrement = 1 / 5.0;
  double temp = ( i − 1 ) * decrement;

  for( ; i > 0l; i-- )
    {
      sum = sum + temp;
      temp = temp − decrement;
    }

  return ( ( long )sum ) % 10;
}
```

The strength−reduction improved the time of execution by eliminating the repeated floating point divisions, and replacing them with floating point additions and subtractions, which are much faster.

The run times were as follows:

before strength−reduction: 137.26s
after strength−reduction: 70.29s

Data Cache Locality

The program below, depending on the presence of a third argument, sums up the elements of the matrix in row−major or column−major order.

walksmart.c

```
#include <stdio.h>
#include <string.h>

int main( int argc, char ** argv )
{

  int i, j;
  long sum = 0;
  long k;
  long ** a;

  int dimx;
  int dimy;
  int cols_first = 0;

  sscanf( argv[ 1 ], "%d", & dimx );
  sscanf( argv[ 2 ], "%d", & dimy );
  if( argc > 3 )
    {
      cols_first = 1;
    }

  a = ( long ** )malloc( sizeof( long * ) * dimx );
  for( i = 0; i < dimx; i++ )
    {
      a[ i ] = ( long * )malloc( sizeof( long ) * dimy );
      for( j = 0; j < dimy; j++ )
        {
          a[ i ][ j ] = i * dimx + j;
        }
    }

  /* for( k = 0l; k < 100l; k++ ) */
  if( cols_first )
      for( j = 0; j < dimy; j++ )
    for( i = 0; i < dimx; i++ )
        sum += a[ i ][ j ];
  else
    for( i = 0; i < dimx; i++ )
      for( j = 0; j < dimy; j++ )
        sum += a[ i ][ j ];

  return sum % 10;
}
```

For a 5000 x 5000 matrix, filled as above, the average run times were as follows:

	row−major	***column−major***
nova	00:09.03	00:27.57
sol	04:25.04	09:56.52

The faster processor incurs a much higher penalty when data cache locality is lost. The nova was 300% slower when going through the matrix in column−major order, as opposed to the sol workstation which was only 200% slower.

When walking row−major order, given the way arrays are represented in memory (as a contiguous one−dimensional array), one element access brings into the cache its

surrounding elements. Only at cache line boundaries will new element accesses cause data misses, thus incurring the cost of fetching more data from main memory.

When walking column major order, each element from the same column general a data miss (since each element from a column is a different area of memory). This way the run time is significantly increased.

Data Cache Conflicts

The program below will walk an array of 262144 elements, with a variable skip factor, depending on the command line parameter. The number of elements touched is constant and set to 64. The skip factor can very between 1 and 4096.

cc.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define COUNT 64l
#define MAXSKIP 4096l

int main( int argc, char ** argv )
{

  long * a;
  int increment;
  long sum = 0l;
  long i;
  long k;

  /* we can read at most COUNT elements spaced MAXSKIP positions apart */
  a = ( long * )malloc( sizeof( long ) * ( COUNT + 1 ) * MAXSKIP );
  if( a == 0 )
    {
      printf( "Cannot allocate!\n" );
      return 1;
    }

  sscanf( argv[ 1 ], "%d", & increment );

  for( i = 0l; i < COUNT * MAXSKIP; i++ )
    a[ i ] = i;

  for( k = 0; k < 1000000l; k++ )
    for( i = 0l; i < COUNT; i++ )
      sum += a[ i * increment ];

  return sum % 10;
}
```

The following results were recorded:

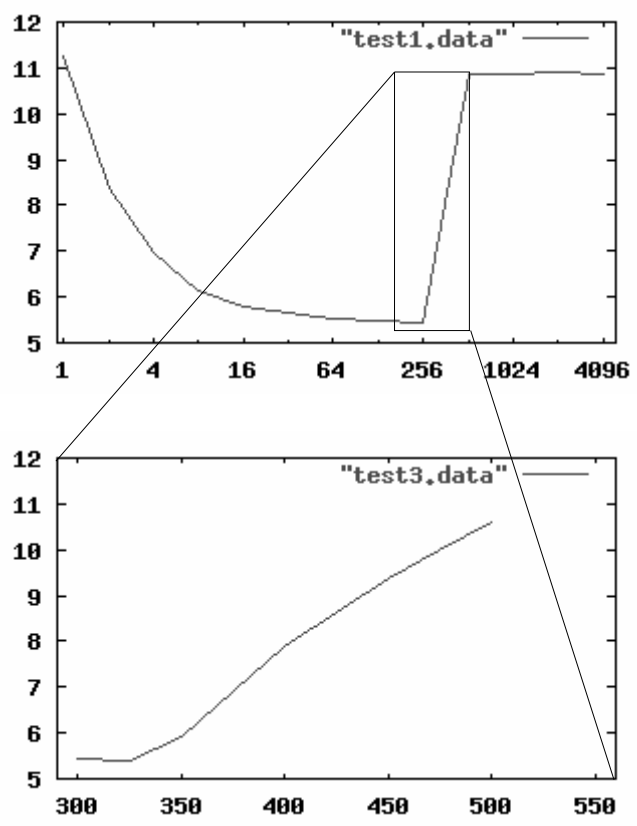
	skip factor = 2	skip factor = 1024
nova	1.15s	2.66s
sol	5.28s	24.25s

It seems that both the novas and the sols use the same algorithm for mapping data into cache lines, that is, <address> modulo <N = some power of 2> (in this case it looks like 1024, but more measurements are required for exact values).

Accessing elements at a skip smaller than N will cause those elements to be cached in different lines in the cache. But if the skip factor is N or a multiple thereof, the elements will be mapped to the same position in the cache, which will cause cache misses at each access, and an increased average execution time.

Instruction Cache Misses

For nova:



Note the scale change
(test1.data is graphed on a logarithmic scale).

The instructions in the loop body were:

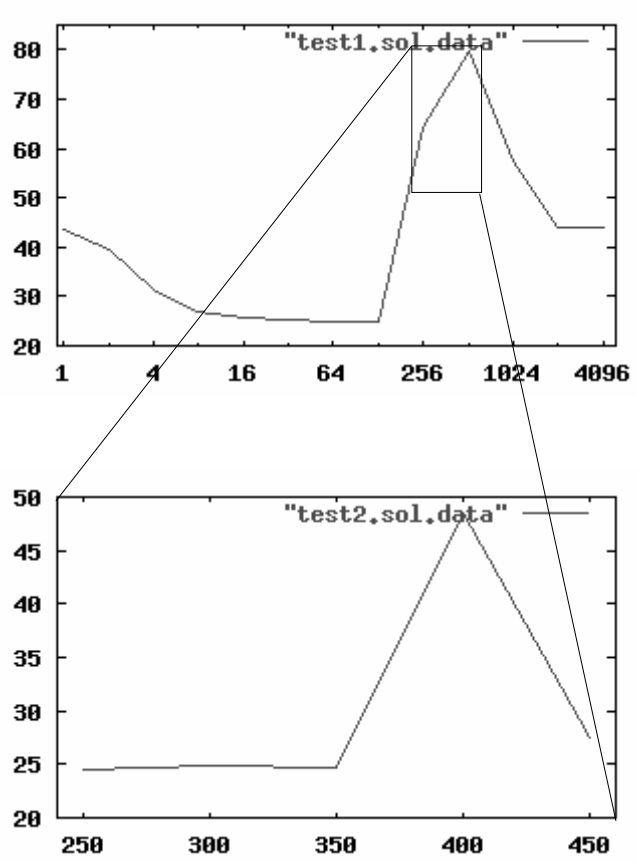
```
sum = sum + 3 * i;
product = 3 * product;
```

which generated the following 12 machine instructions:

```
ld [%fp−20],%o0
mov %o0,%o2
sll %o2,1,%o1
add %o1,%o0,%o1
ld [%fp−24],%o0
add %o0,%o1,%o1
st %o1,[%fp−24]
ld [%fp−28],%o0
mov %o0,%o1
sll %o1,1,%o2
add %o2,%o0,%o0
st %o0,[%fp−28]
```

The instruction cache size can be appreciated as being:
400 (from the graph) x 4 bytes / instr. X 12 instr. = 18.75 kb => 18 kb

For sol:



Note the scale change
(test1.sol.data is on a logarithmic scale)

The similar results (shifted up in time due to difference in processor speed) to the nova lead to the same cache size.