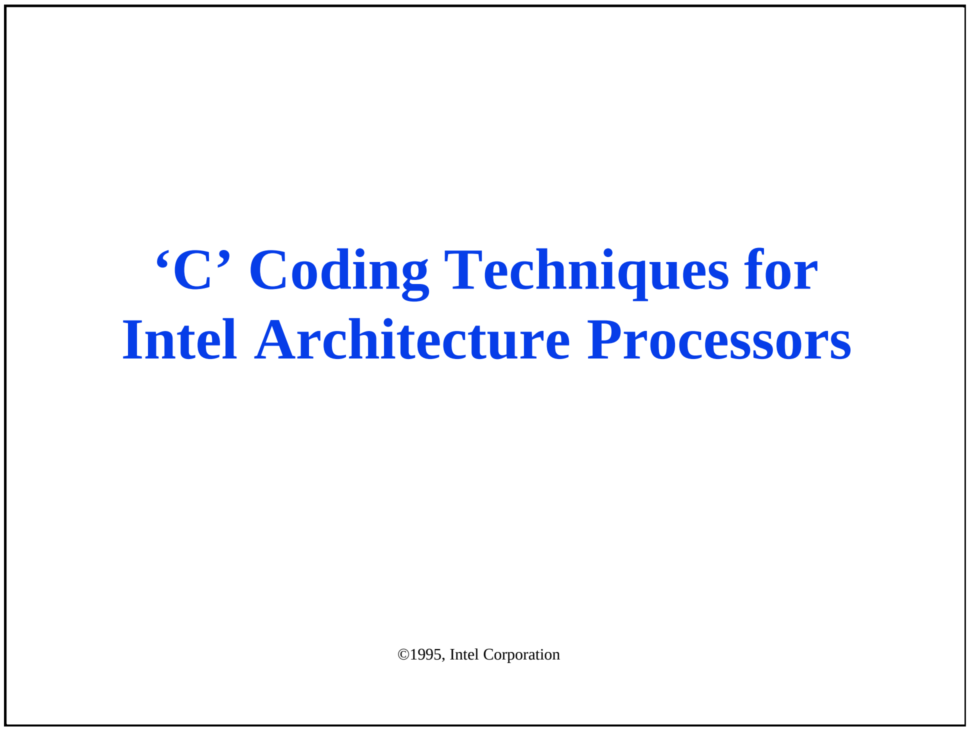
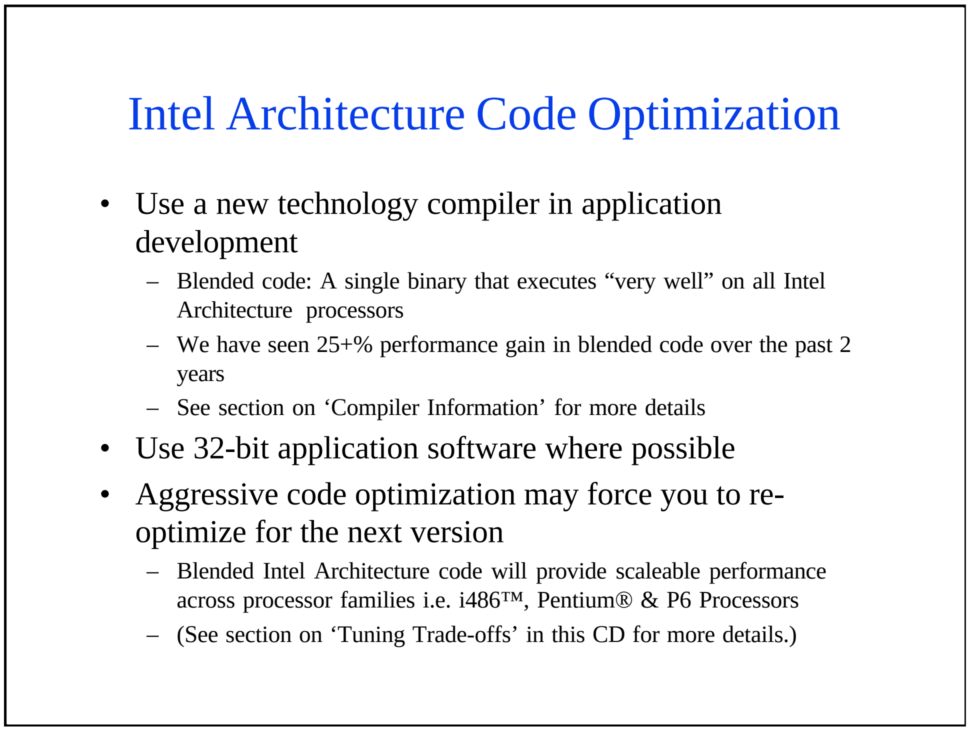


‘C’ Coding Techniques for Intel Architecture Processors

This section includes advice for writing software optimized for the whole Intel Architecture (IA) family of processors, from Intel486™ processors, to Pentium® processors, to P6 processors. We classify this type of IA software as BLENDED.

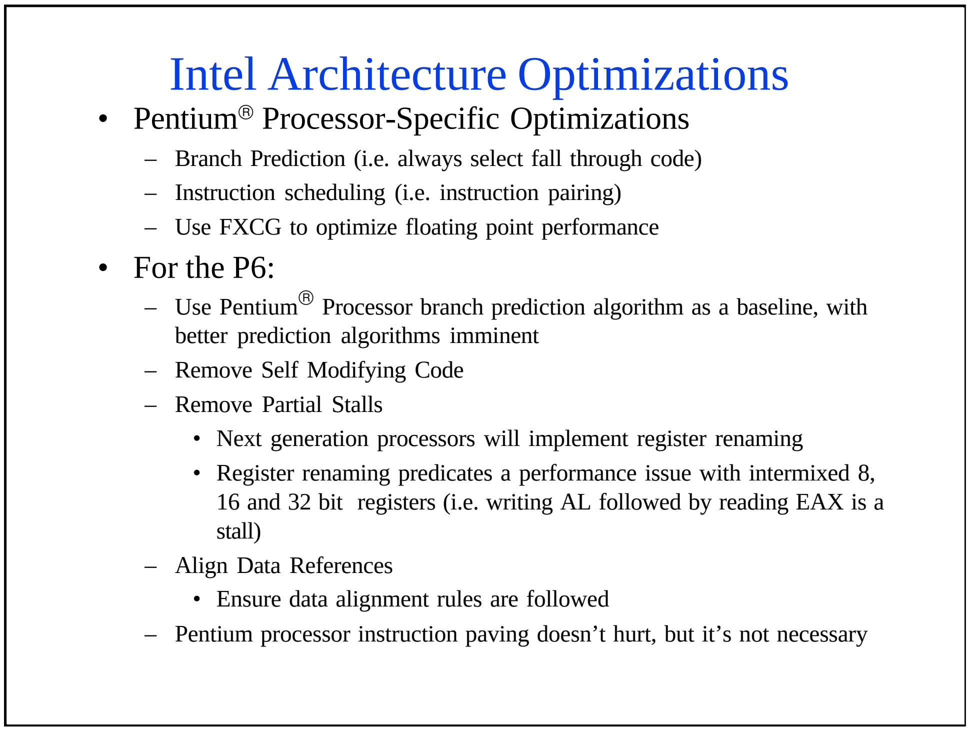


Intel Architecture Code Optimization

- Use a new technology compiler in application development
  - Blended code: A single binary that executes “very well” on all Intel Architecture processors
  - We have seen 25+% performance gain in blended code over the past 2 years
  - See section on ‘Compiler Information’ for more details
- Use 32-bit application software where possible
- Aggressive code optimization may force you to re-optimize for the next version
  - Blended Intel Architecture code will provide scaleable performance across processor families i.e. i486™, Pentium® & P6 Processors
  - (See section on ‘Tuning Trade-offs’ in this CD for more details.)

Compiler techniques are improving all of the time and it is recommended that the latest version of a compiler is always used. Even if you don’t change your code at all, a new compiler can improve the performance of your software.

The 386 processor introduced 32-bit registers and newer generations of IA processors have focussed upon improving this 32-bit software. While 8 and 16 bit software will not run slower on newer CPU generations, it may not speed up as dramatically as 32-bit software. Use 32-bit memory/operations pointers wherever possible.



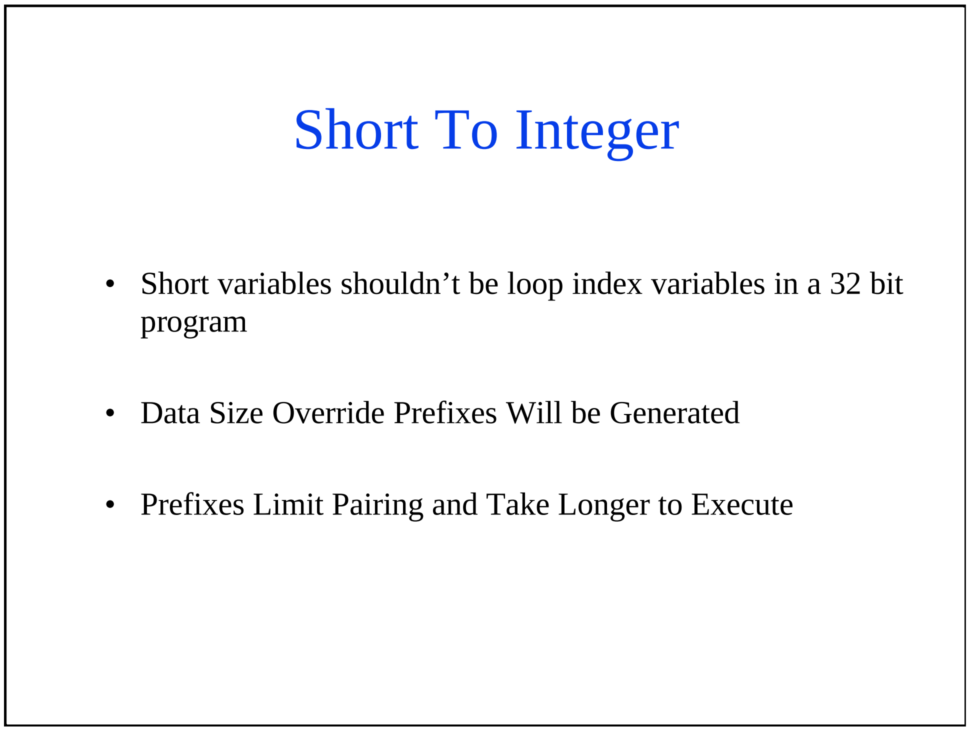
Intel Architecture Optimizations

- Pentium® Processor-Specific Optimizations
  - Branch Prediction (i.e. always select fall through code)
  - Instruction scheduling (i.e. instruction pairing)
  - Use FXCG to optimize floating point performance
- For the P6:
  - Use Pentium® Processor branch prediction algorithm as a baseline, with better prediction algorithms imminent
  - Remove Self Modifying Code
  - Remove Partial Stalls
    - Next generation processors will implement register renaming
    - Register renaming predicates a performance issue with intermixed 8, 16 and 32 bit registers (i.e. writing AL followed by reading EAX is a stall)
  - Align Data References
    - Ensure data alignment rules are followed
  - Pentium processor instruction paving doesn't hurt, but it's not necessary

There are some general optimizations that can improve the execution speed of processors with branch prediction -- important for the Pentium® processor and vital for the P6. Try to make the general flow of the software as a straight path -- divert the flow for exception conditions or other rarely executed code.

Subroutines, or other functions, should have a return statement, not a JMP instruction. This will make them more predictable.

Some instruction pairing can improve the Pentium® processors performance. While this doesn't particularly help the P6 (instruction re-scheduling is done in hardware), it doesn't hurt P6 performance either. The Intel 486™ processor also is not negatively impacted by Pentium processor pairing.

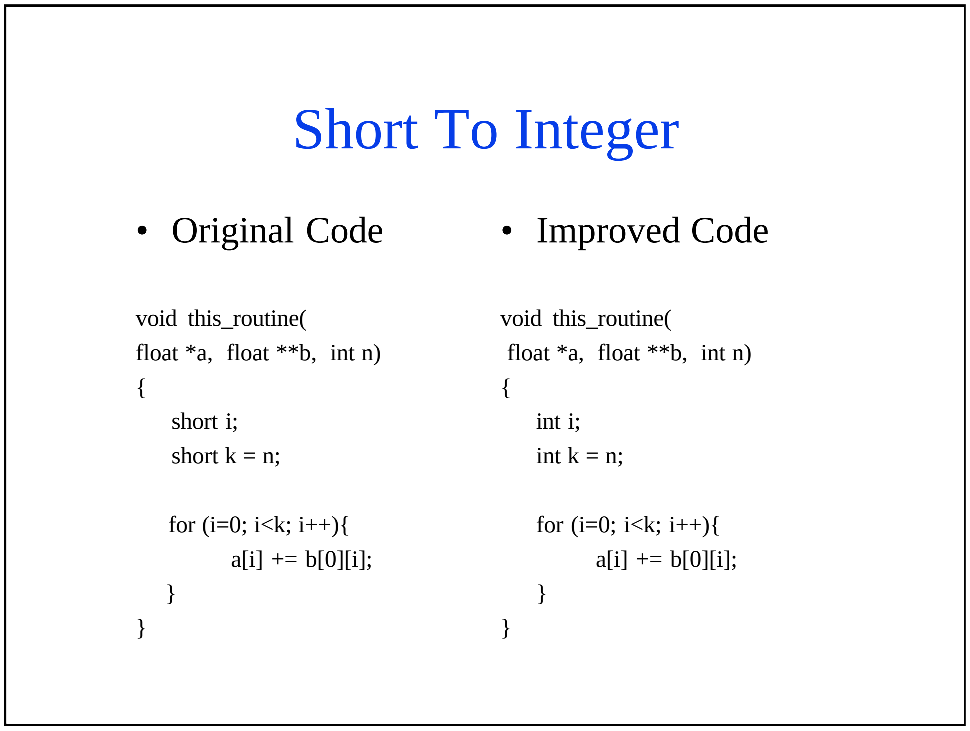


Short To Integer

- Short variables shouldn’t be loop index variables in a 32 bit program
- Data Size Override Prefixes Will be Generated
- Prefixes Limit Pairing and Take Longer to Execute

In a 32-bit applications, using a short as a part of a loop will cause size override prefixes to be incorporated. These overrides take longer to fetch, decode and execute.

ALWAYS use 32 bit integers as loop variables -- your software will run faster.



Short To Integer

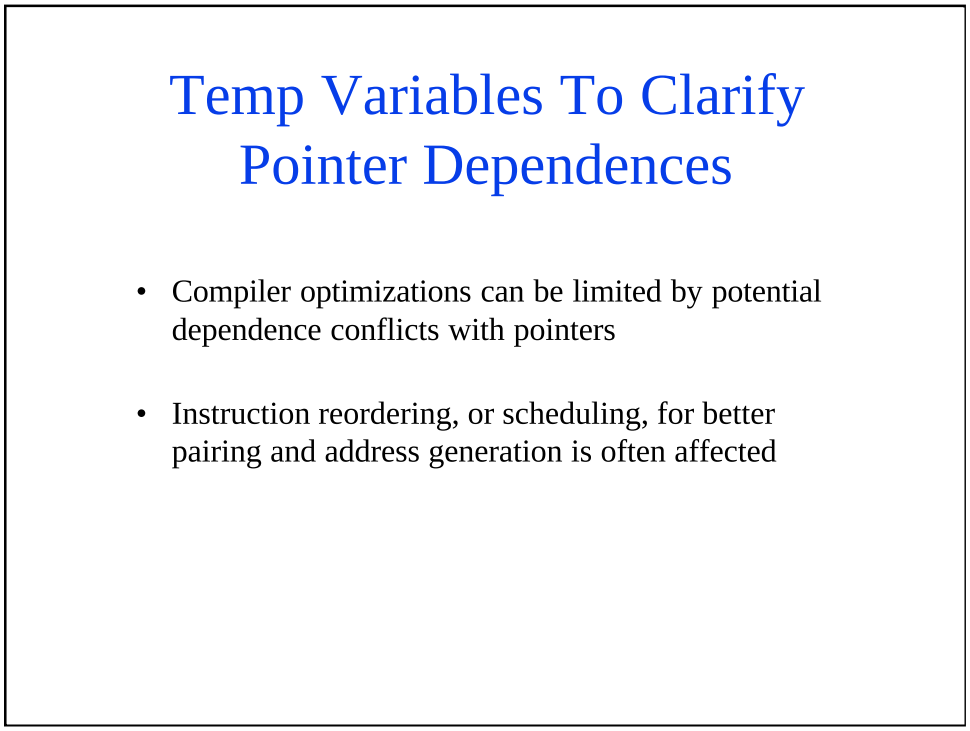
- Original Code

```
void  this_routine(
float *a,  float **b,  int n)
{
     short i;
     short k = n;

     for (i=0; i<k; i++){
              a[i]  += b[0][i];
     }
}
```

- Improved Code

```
void  this_routine(
 float *a,  float **b,  int n)
{
     int i;
     int k = n;

     for (i=0; i<k; i++){
             a[i]  += b[0][i];
     }
}
```

Example for previous page.

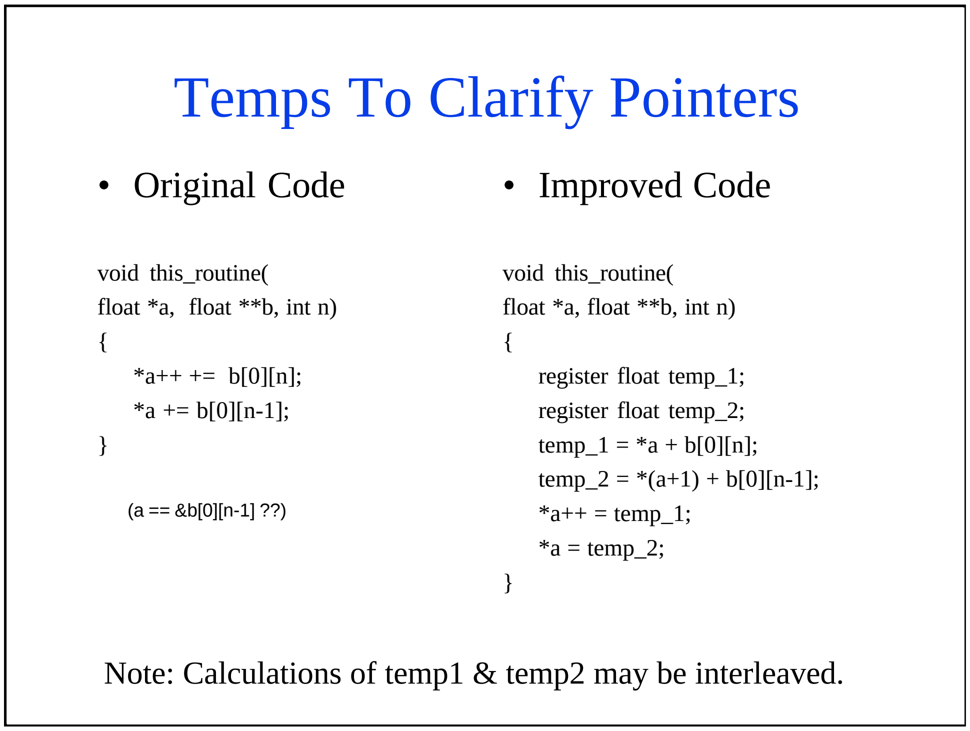


Temp Variables To Clarify Pointer Dependences

- Compiler optimizations can be limited by potential dependence conflicts with pointers
- Instruction reordering, or scheduling, for better pairing and address generation is often affected

Multiple lilnes of ‘C’ code that have the same variables used as pointer references cause the code/data dependency tree to be extended, reduce parallelism and lower performance.

Reorder instructions and introduce temporary pointers where possible.



Temps To Clarify Pointers

- Original Code

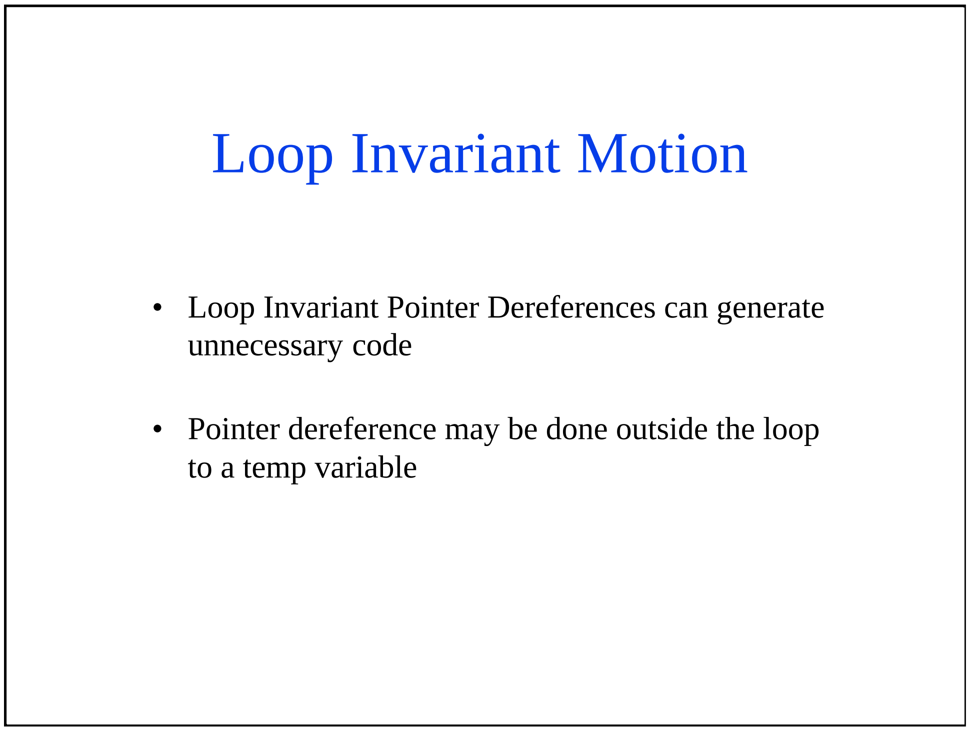
```
void  this_routine(
float *a,  float **b, int n)
{
     *a++ +=  b[0][n];
     *a += b[0][n-1];
}

    (a == &b[0][n-1] ??)
```

- Improved Code

```
void  this_routine(
float *a, float **b, int n)
{
     register float temp_1;
     register float temp_2;
     temp_1 = *a + b[0][n];
     temp_2 = *(a+1) + b[0][n-1];
     *a++ = temp_1;
     *a = temp_2;
}
```

Note: Calculations of temp1 & temp2 may be interleaved.

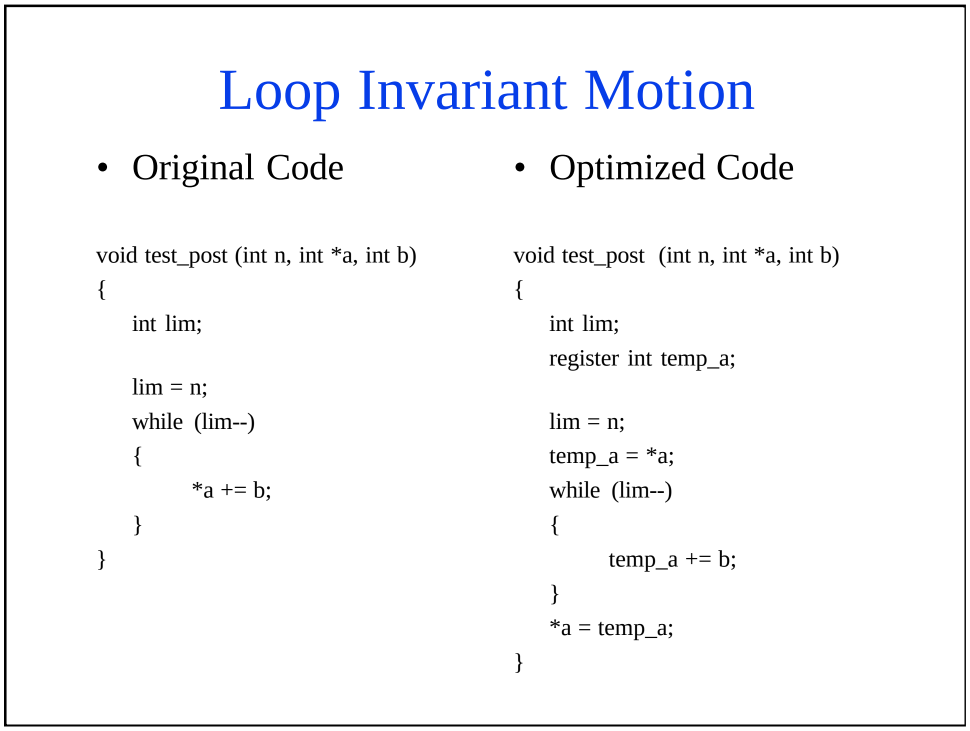
Example for previous page.



Loop Invariant Motion

- Loop Invariant Pointer Dereferences can generate unnecessary code
- Pointer dereference may be done outside the loop to a temp variable

Loop invariant pointer dereferences that are using/accessing normal stack based variables may generate inefficient/slow code. The movement of this code to temporary variables and possibly a register based implementation will improve performance.



Loop Invariant Motion

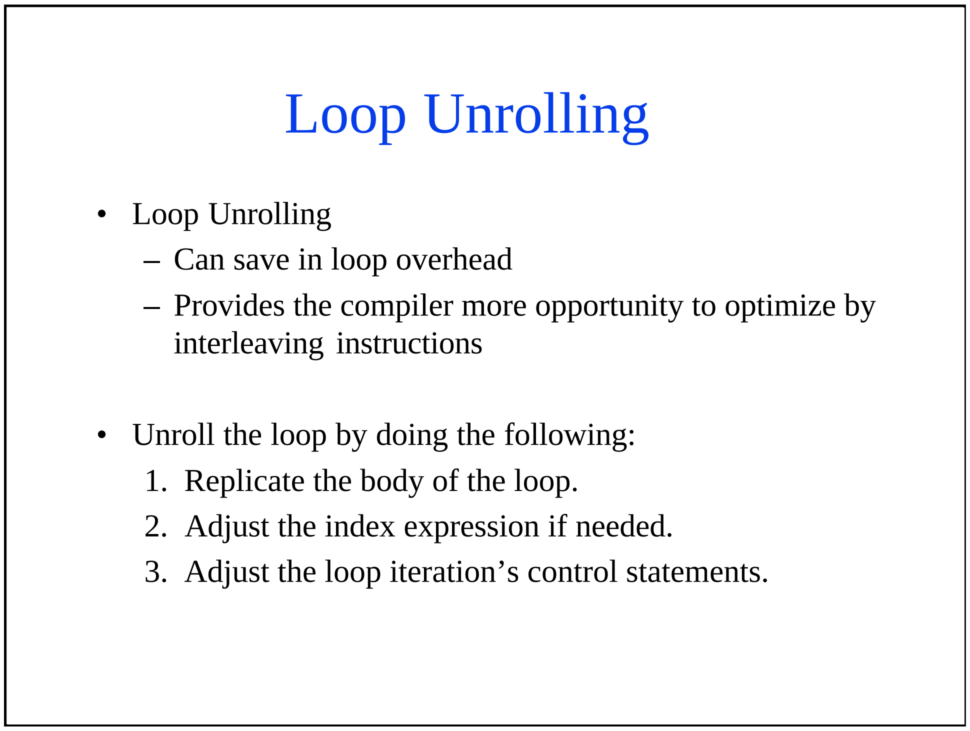
- Original Code

```
void test_post (int n, int *a, int b)
{
    int lim;

    lim = n;
    while (lim--)
    {
        *a += b;
    }
}
```

- Optimized Code

```
void test_post (int n, int *a, int b)
{
    int lim;
    register int temp_a;

    lim = n;
    temp_a = *a;
    while (lim--)
    {
        temp_a += b;
    }
    *a = temp_a;
}
```

Example for previous page.

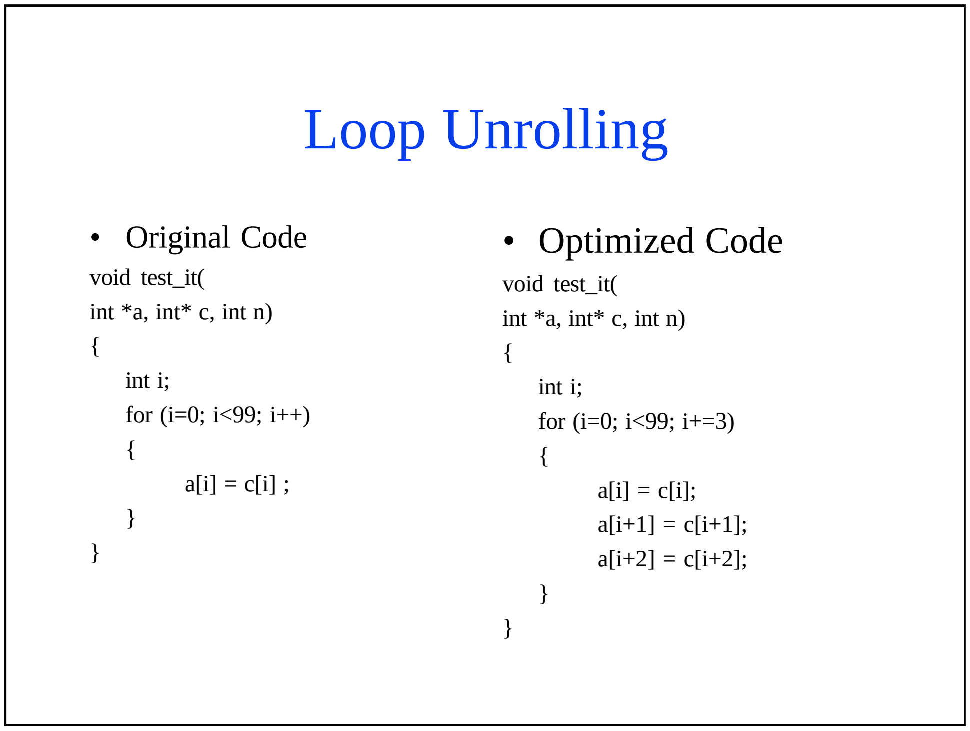


Loop Unrolling

- Loop Unrolling
  - Can save in loop overhead
  - Provides the compiler more opportunity to optimize by interleaving instructions

- Unroll the loop by doing the following:
  1. Replicate the body of the loop.
  2. Adjust the index expression if needed.
  3. Adjust the loop iteration's control statements.

Loop unrolling is a standard compiler technique that provides the opportunity for higher performance by providing the compiler with a larger basic block to optimize and thus more opportunity. This unrolling also allows turning based on cache/memory architecture to be more controllable. Basic rule: the smaller the size of the loop, the higher the priority for unrolling. Consideration: loop unrolling will likely help the Pentium® processor more than the P6.



Loop Unrolling

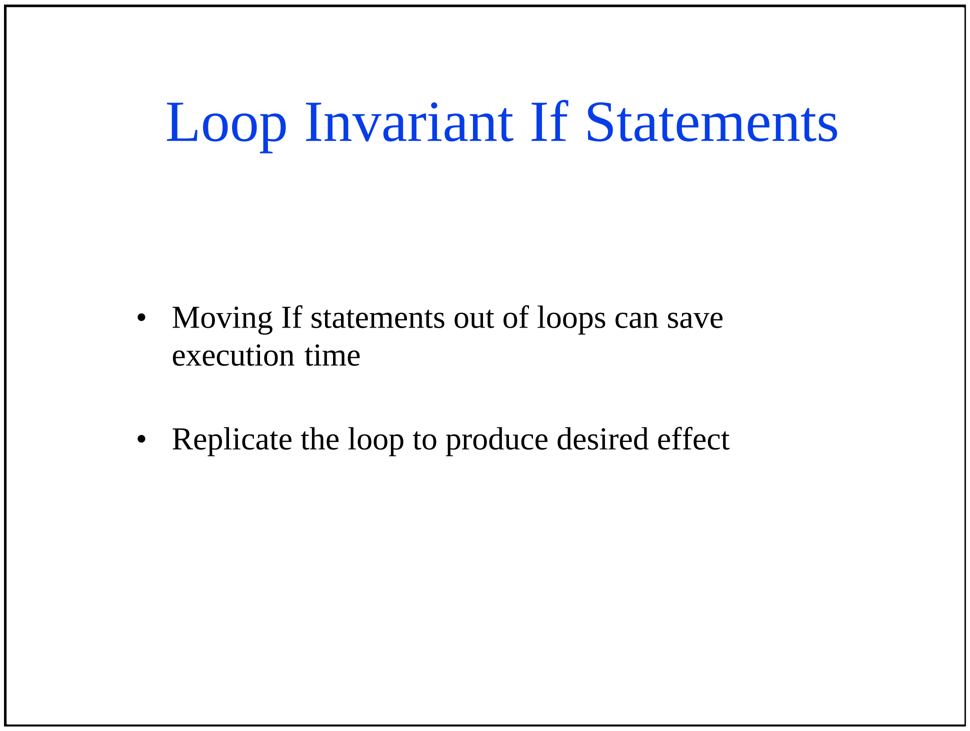
- Original Code

```
void  test_it(
int *a, int* c, int n)
{
    int i;
    for (i=0; i<99; i++)
    {
          a[i] = c[i] ;
    }
}
```

- Optimized Code

```
void  test_it(
int *a, int* c, int n)
{
    int i;
    for (i=0; i<99; i+=3)
    {
          a[i] = c[i];
          a[i+1] = c[i+1];
          a[i+2] = c[i+2];
    }
}
```

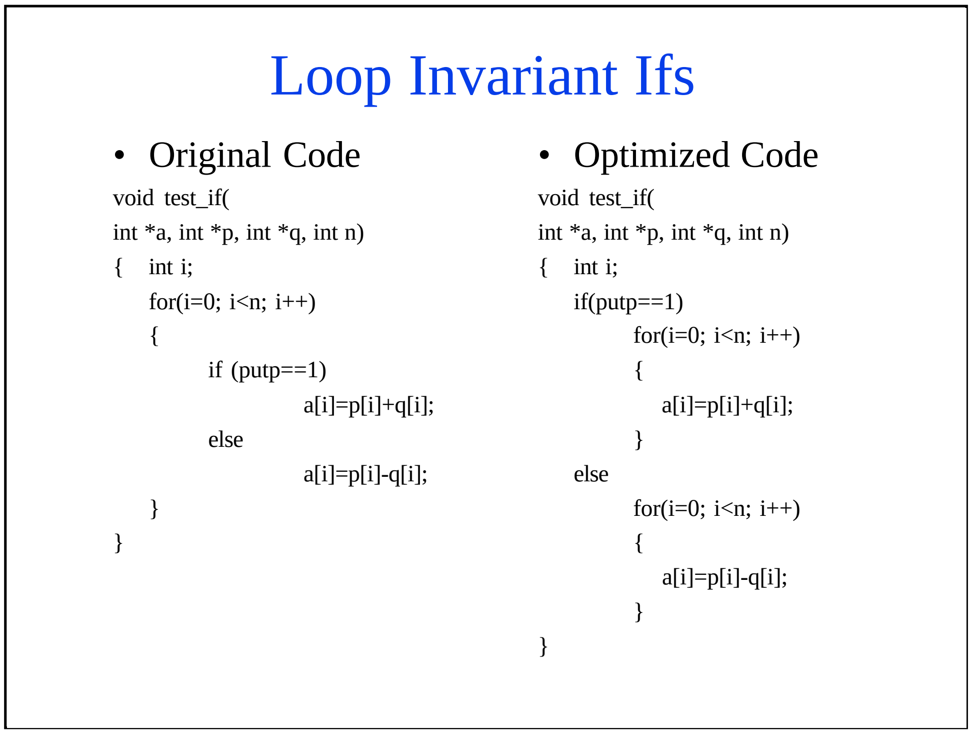
Example for previous page. (While this does not hurt P6 execution time, it is not always necessary since the P6 does many aspects of this operation automatically via its Dynamic Execution core.)



Loop Invariant If Statements

- Moving If statements out of loops can save execution time
- Replicate the loop to produce desired effect

The movement of loop invariants outside the loop core will reduce the unnecessarily repetitive execution of those instructions. This may result in loop repetition or duplication and the resultant larger code size, but execution performance will improve.



Loop Invariant Ifs

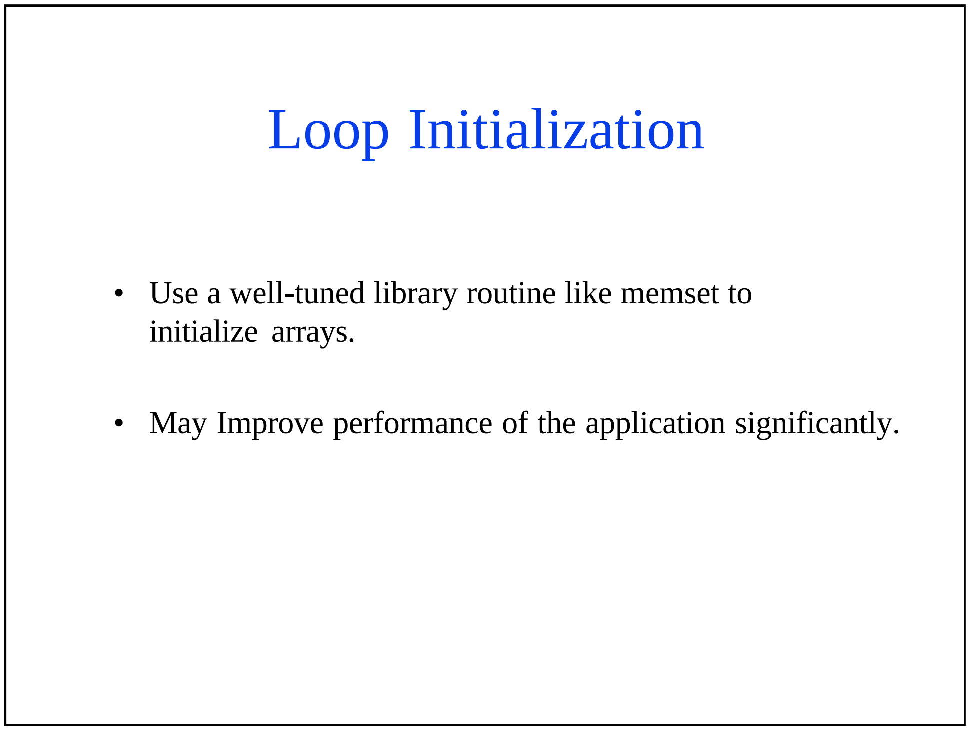
- Original Code

```
void  test_if(
int *a, int *p, int *q, int n)
{    int i;
     for(i=0; i<n; i++)
     {
          if (putp==1)
                    a[i]=p[i]+q[i];
          else
                    a[i]=p[i]-q[i];
     }
}
```

- Optimized Code

```
void  test_if(
int *a, int *p, int *q, int n)
{    int i;
     if(putp==1)
          for(i=0; i<n; i++)
          {
             a[i]=p[i]+q[i];
          }
     else
          for(i=0; i<n; i++)
          {
             a[i]=p[i]-q[i];
          }
}
```

Example for previous page.

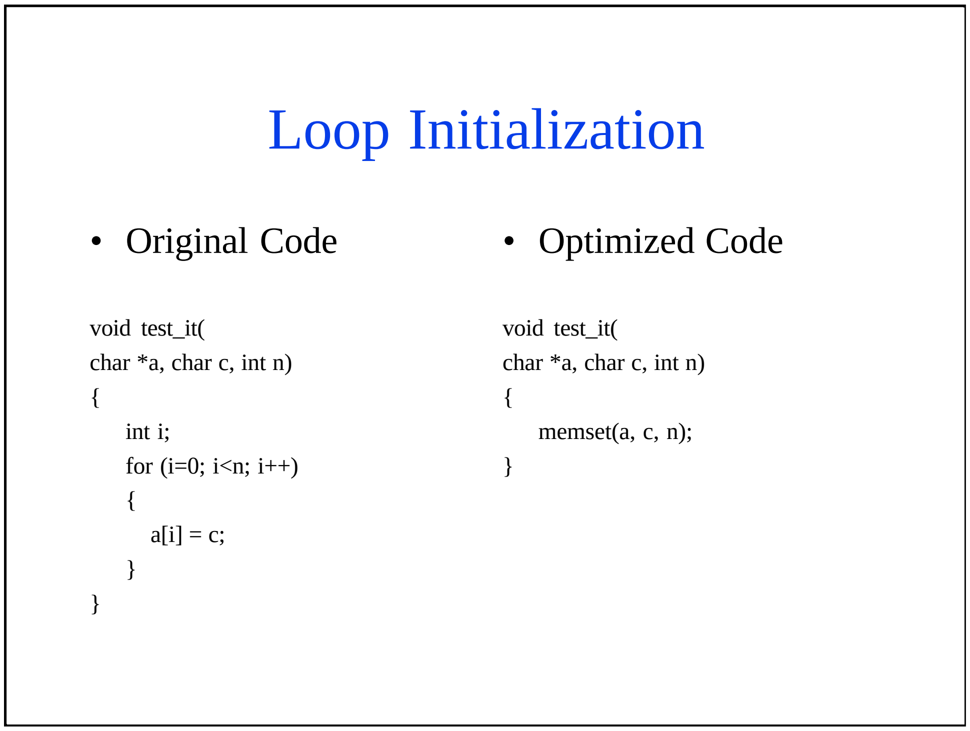


Loop Initialization

- Use a well-tuned library routine like memset to initialize arrays.
- May Improve performance of the application significantly.

Libraries are a very good place for optimization, allowing tuning to be implemented without a complete recompile of the application. Relinking is only necessary for regeneration. The libraries provide a potential isolation of the application from processor/architecture-specific requirements.

Libraries should be scanned for optimal routines that may be incorporated into the normal function required by an application (i.e., memset for array initialization).



Loop Initialization

- Original Code

```
void  test_it(
char *a, char c, int n)
{
     int i;
     for (i=0; i<n; i++)
     {
         a[i] = c;
     }
}
```

- Optimized Code

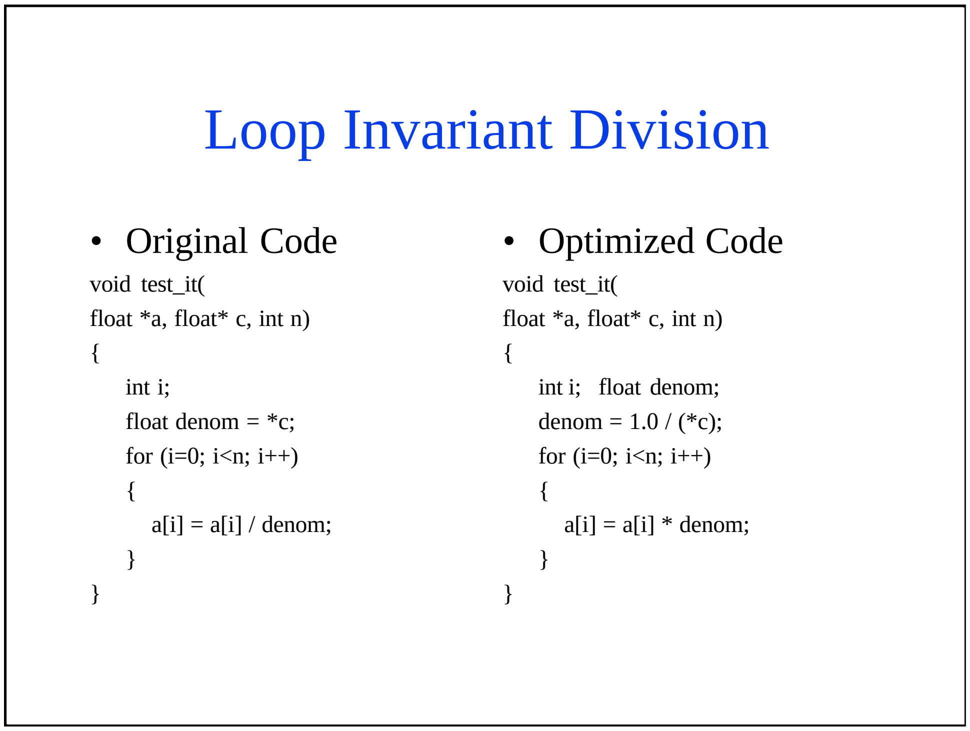
```
void  test_it(
char *a, char c, int n)
{
     memset(a, c, n);
}
```

“Memset” is much faster mechanism for replicating a value through memory.

Loop Invariant Division

- Division is Much Slower than Multiplication
- Calculate Reciprocal outside of loop and use Multiply inside

Another good tip to speed execution: If possible, move any loop-based division to a multiply by reciprocal implementation.



Loop Invariant Division

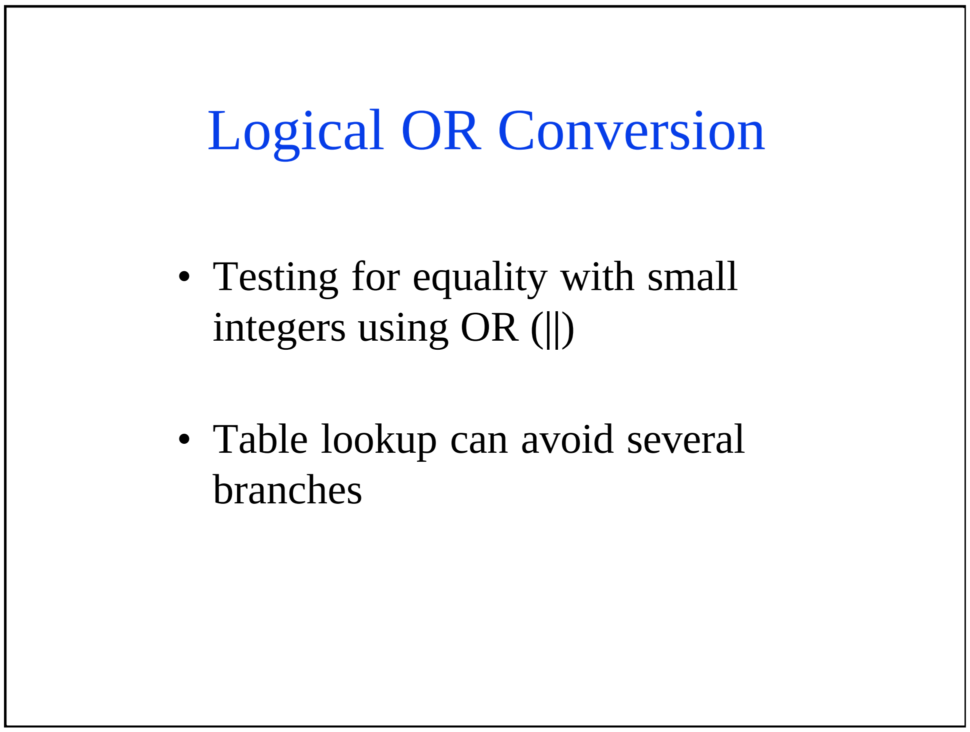
- Original Code

```
void  test_it(
float *a, float* c, int n)
{
    int i;
    float denom = *c;
    for (i=0; i<n; i++)
    {
      a[i] = a[i] / denom;
    }
}
```

- Optimized Code

```
void  test_it(
float *a, float* c, int n)
{
    int i;   float  denom;
    denom = 1.0 / (*c);
    for (i=0; i<n; i++)
    {
      a[i] = a[i] * denom;
    }
}
```

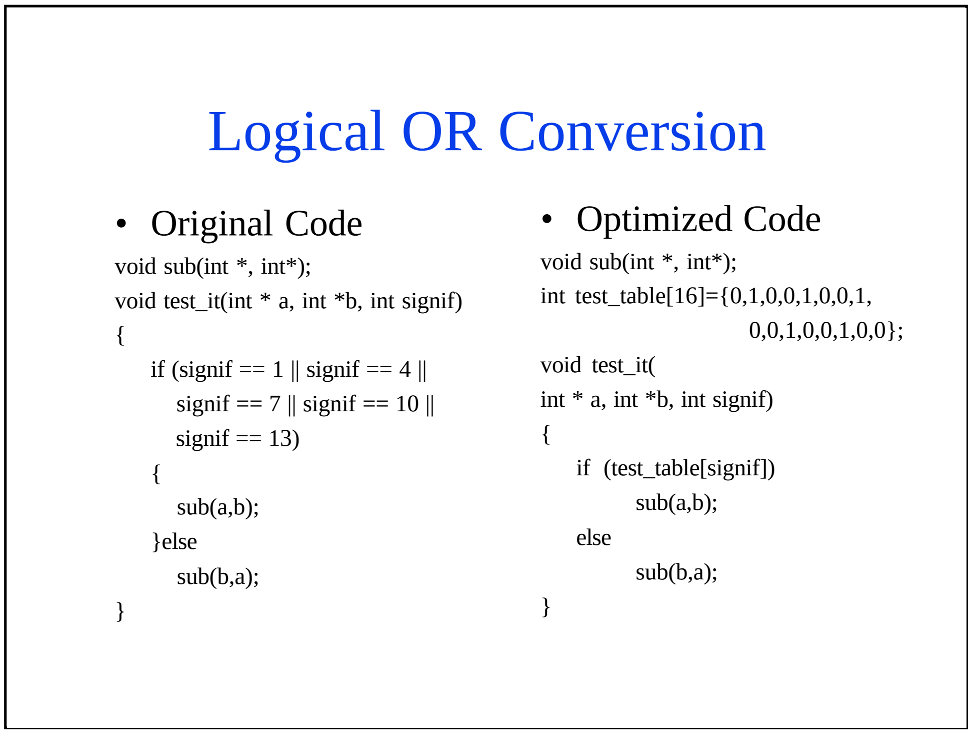
Example for previous page.



Logical OR Conversion

- Testing for equality with small integers using OR (||)
- Table lookup can avoid several branches

Another useful suggestion.



Logical OR Conversion

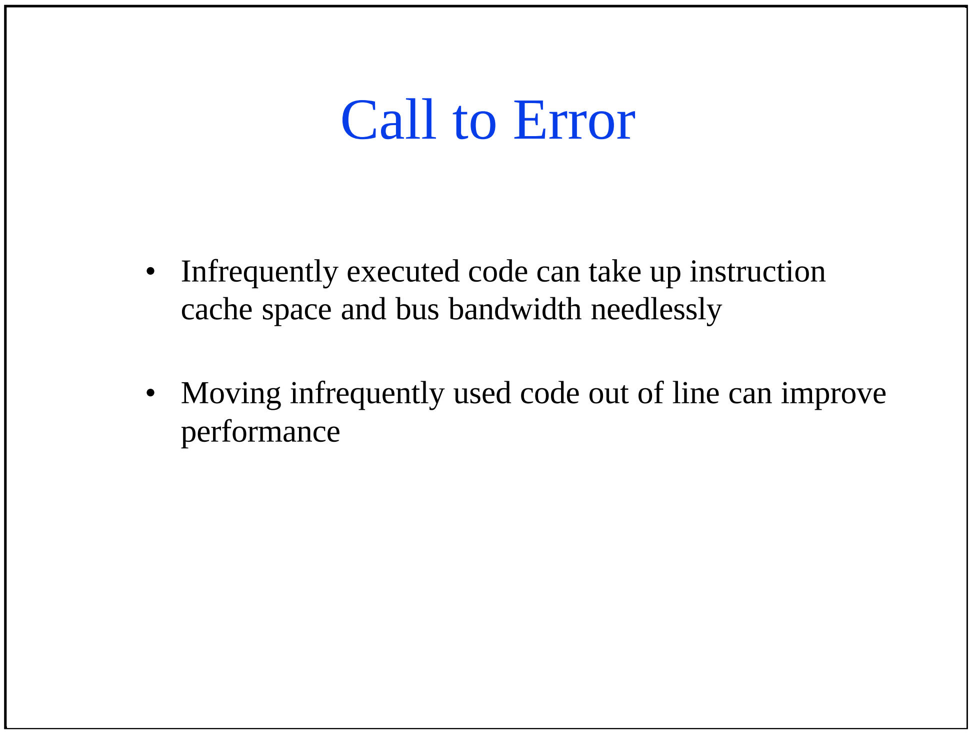
- Original Code

```
void sub(int *, int*);
void test_it(int * a, int *b, int signif)
{
    if (signif == 1 || signif == 4 ||
        signif == 7 || signif == 10 ||
        signif == 13)
    {
        sub(a,b);
    }else
        sub(b,a);
}
```

- Optimized Code

```
void sub(int *, int*);
int test_table[16]={0,1,0,0,1,0,0,1,
                    0,0,1,0,0,1,0,0};
void test_it(
int * a, int *b, int signif)
{
    if (test_table[signif])
        sub(a,b);
    else
        sub(b,a);
}
```

Example for previous page.

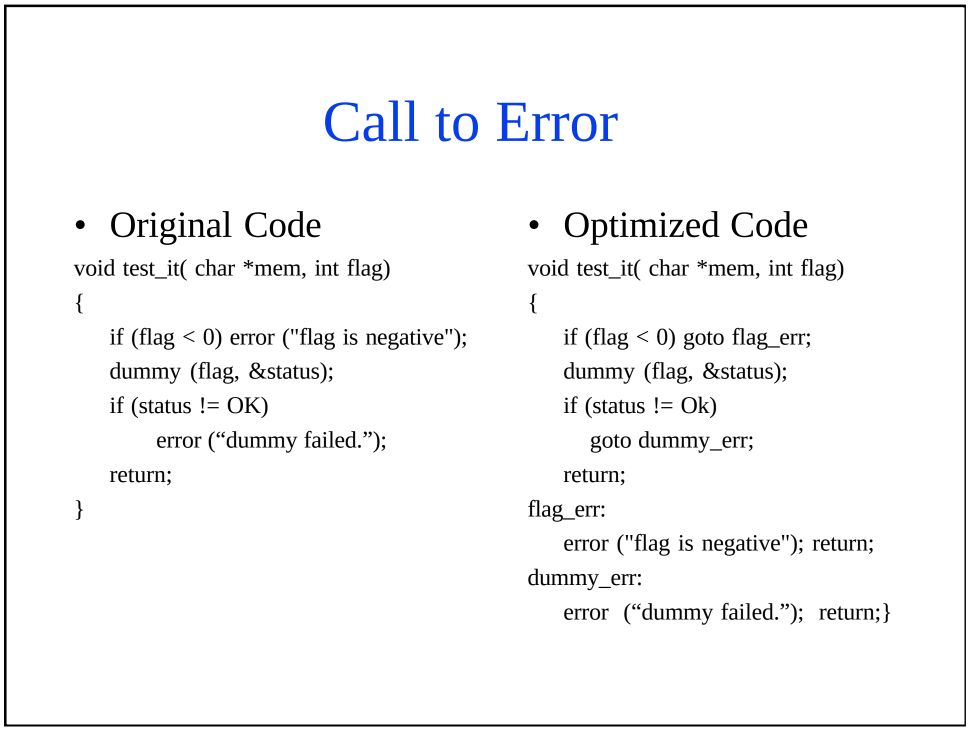


Call to Error

- Infrequently executed code can take up instruction cache space and bus bandwidth needlessly
- Moving infrequently used code out of line can improve performance

A final suggestion: All IA processors perform best when they are allowed to prefetch decode or execute in a straight line with no branches to break the pipeline.

The movement of infrequently executed code (e.g., exception/error handlilng code) will allow the maximum prefetch/decode/execute bandwidth to be exposed.



Call to Error

- Original Code

```
void test_it( char *mem, int flag)
{
    if (flag < 0) error ("flag is negative");
    dummy (flag, &status);
    if (status != OK)
          error (“dummy failed.”);
    return;
}
```

- Optimized Code

```
void test_it( char *mem, int flag)
{
    if (flag < 0) goto flag_err;
    dummy (flag, &status);
    if (status != Ok)
       goto dummy_err;
    return;
flag_err:
    error ("flag is negative"); return;
dummy_err:
    error  (“dummy failed.”);  return;}
```

Example for previous page.

For more information, see the ‘32-bit Optimization Guide’ in this CD.