Code Optimization Techniques

By:
Hardik Devani
Rockwell Collins, IDC
May 2011

“Code Less… Create More.. !!“

Compiler Optimization Techniques.

It describes some of the
optimizations the compiler/programmer may use to improve the performance of their code.
So….

Compiler/ Programmer Optimization Techniques.

“The more information a programmer gives the compiler the better
job it can do at optimizing the code.”

Agenda

- Introduction
- CPU & Memory Organization
- Pipelining Architectures
- Optimization criterions
- 16 Compiler Optimizations Techniques
- Discussions
- Take away

What is not included here ??

- How a compiler used with the proper optimization flags can reduce the runtime of a code.
  - **http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html**
- Machine Code Optimizations
- Implementing Optimization in the Hardware

Introduction

- Modern CPU's are **Super-scaler** and **Super-pipelined**
- It is not only the clock speed of a CPU that defines how fast it can execute code.
- CPUs now have clock speeds approaching 1000Mhz while the bus that moves data between memory and the CPU may only run at 100Mhz.

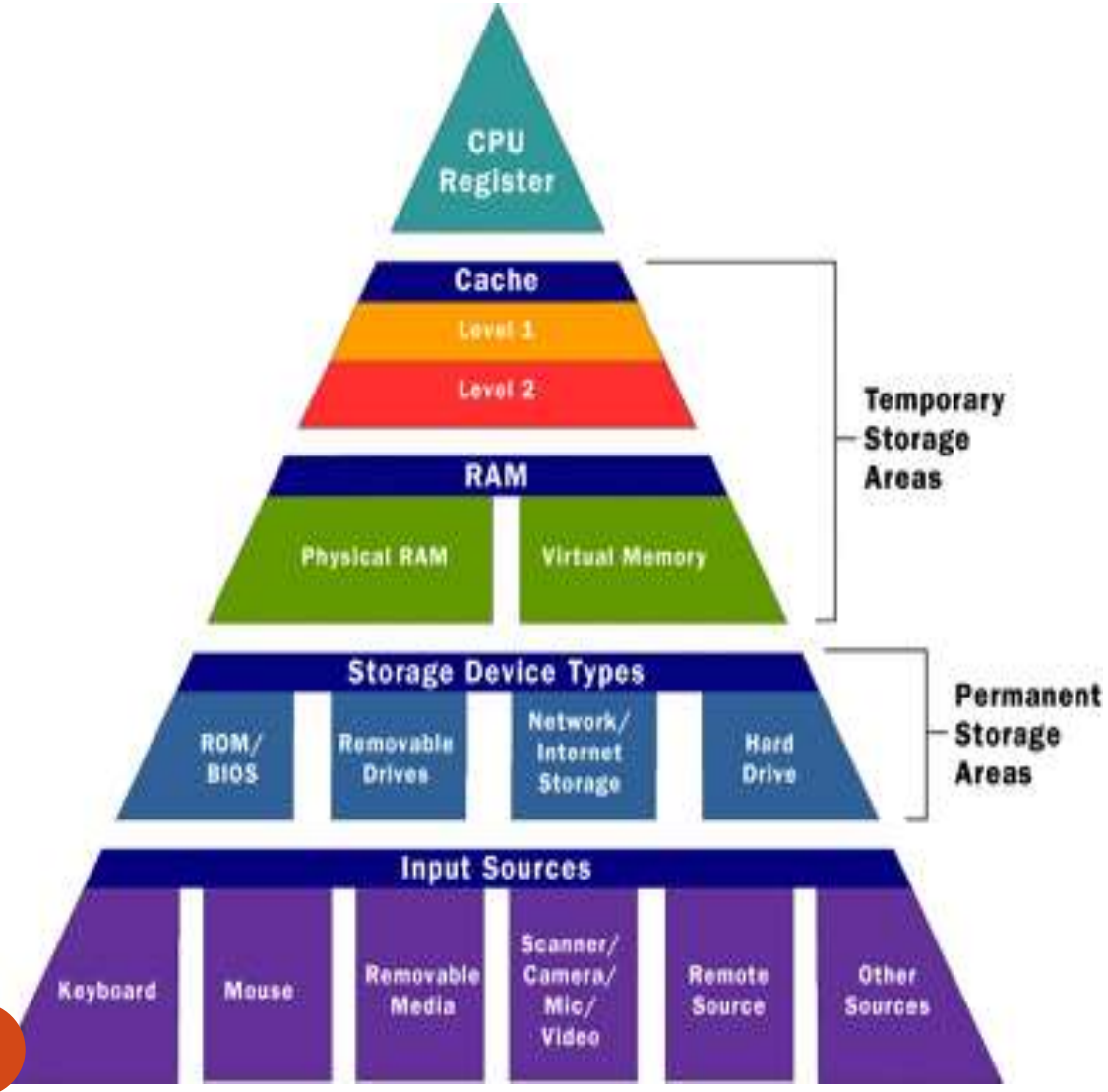
Contd..

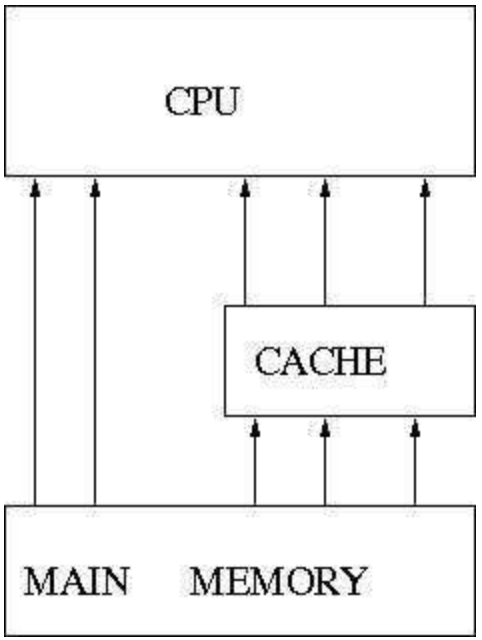
- *In Order / Out of Order Instruction Execution*

- By breaking complex operations into smaller stages the CPU designer can remove any bottle necks in the pipeline and speed up throughput.

- If a CPU has multiple floating point units and integer units the problem becomes keeping them all busy to achieve maximum efficiency.

CPU & Memory Organization





Cache Memory

- The cache is a small amount of very fast memory that is used to store frequently used data or instructions.
  - Level 1 cache is small, 8k-128k, runs at the same speed as the CPU
  - Level 2 cache will be larger, 128k-8MB, generally run slower, maybe half the speed of the CPU.

- Also See ***"Role of Cache in pipelined architecture"***

What can we think for Optimization??

- Think in Binary Powers
  - Shift is easy in/then Multiply and divide.
- Think Independent
  - instruction execution for multi processing or multiple ALU capabilities.
- Keep Pipelined architecture in mind.
- Its Not a “MUST” to optimize..
  - Mind it..!!

Factors affecting optimization

- The machine
  - GCC Target ?
  - Embedded Sys
  - Debugging Build
  - Special Purpose
- Architecture of the CPU
  - No. Of CPU registers
  - RISC / CISC
  - Pipelines (Pipeline Stalls)
  - No. Of Functional Units
- Machine Architecture
  - Cache size

Optimizing Techniques

1. Copy Propagation
2. Constant Folding
3. Dead Code Removal
4. Strength Reduction
5. Induction Variable Simplification
6. Function In-Lining
7. Loop invariant Conditionals
8. Variable Renaming
9. Common Sub-Expression Elimination
10. Loop Invariant Code Motion
11. Loop Fusion
12. Pushing Loops Inside Subroutine Calls
13. Loop Index Dependent Conditionals
14. Loop Unrolling
15. Loop stride Size
16. Float Point Optimizations

Copy Propagation

- Consider

  X=Y
  Z=1.0+X

- The compiler might change this to

  X=Y
  Z=1.0+Y

✓ Modern CPUs can sometimes execute a number of instructions at the same time. (Multi-Core / Multi-ALU)

✓ Remove as many dependencies as possible so that instructions can be executed independently.

✓ Reuse outer scope variables.

Constant Folding

- Consider
  - const int J=100;
  - const int K=200;
  - int M= J+K;

- The compiler might compute M at compile time as a const rather than at run time.

✓ Programmer can help the compiler to make the optimization by defining any variables that are constant as constant using either the keywords const or PARAMETER.

✓ This will also help prevent the programmer from accidently changing the value of a variable that should be constant.

✓ Define Scope and access specifiers to optimization.

Dead Code Removal

- The compiler will search out pieces of code that have no effect and remove them.

✓ examples of dead code are ***functions that are never called*** and ***variables that are calculated but never used*** again.

✓ When the compiler compiles a piece of code it makes a number of passes, on each pass it tries to optimize the code, on some passes it may remove code it created previously as it is no longer needed.

⌗ When a piece of code is being developed or modified the programmer often adds code for testing purposes that should be removed when the testing is complete.

Strength Reduction

- Replace complex or difficult or expensive operations with simpler ones.
  - replacing division by a constant with multiplication by its reciprocal

- Consider

  Y=X**2

  ✓ Raising a Number to an exponent
    - first X is converted to a logarithm and then multiplied by two and then converted back.

- Y=X*X Is much more efficient.
  - Multiplication is Easier than raising a number to a power.

- Consider Y=X**2.0
  - This can not be optimized as in the case above as

    (Float) 2.0 != 2 (integer) in general.

  ✓ This is an optimization the programmer can use when raising a number to the power of a small integer.

- Similarly Y=2*X to Y=X+X. Also see ***other Implied inferences***
- if the CPU can perform the addition faster than the multiplication.
- Some CPU's are much slower at performing division compared to addition/subtraction/multiplication so divisions should be avoided were possible.

Induction Variable Simplification

- Consider

```
 for(i=1; i<=N; i++)
{
        K=i*4+M;
        C=2*A[K];
        .....
}
```

- Optimized as

```
K=M;
for(i=1; i<=N; i++)
{
        K=K+4;
        C=2*A[K];
        .......
}
```

✓ Facilitate better memory access patterns, on each iteration through the loop it can predicate which element of the array it will require.

Function In-lining & “inline” directive

- Consider

```
for(i=0; i<=N ; i++)
{
    ke[i]=kinetic_energy(mass[i], velocity[i]);
}
```

- Optimized as

```
float kinetic_energy(float m, float v)
{
    return 0.5*m*v*v;
}
```

✓ Calling a function is associated with overhead. (Stack switching, clear registers etc.).

✓ Huge functions been “inlined” = > Code Bloat => difficult to cache the large *.exe file

Loop Invariant Conditionals

- Consider

```
DO I=1,K
        IF ( N.EQ.0 ) THEN
                A(I)=A(I)+B(I)*C
        ELSE
                A(I)=0
        ENDIF
ENDDO
```

- Optimized as

```
IF ( N.EQ.0 ) then
        DO I=1,k
                A(I)=A(I)+B(I)*C
        ENDDO
ELSE
        DO I=1,K
                A(I)=0
        ENDDO
ENDIF
```

✓ Condition checking is done once so,, **LOOP UNROLLING** can be done.

Variable Renaming

- Consider

```
X=Y*Z
Q=R+X+X
X=A+B
```

- Optimized as

```
X_=Y*Z
q=R + X_ + X_
X= A+B
```

✓ The third line of code can not be executed until the second line has completed. This inhibits the flexibility of the CPU. The compiler might avoid this by creating a temporary variable, _X

✓ The second and third lines are now independent of each other and can be executed in any order and possibly at the same time.

Common Sub-Expression Elimination

- Consider

  A=C*(F+G)

  D=(F+G)/N

- Optimized as

  temp=F+G

  A=C*temp

  D=temp/N

✓ if the expression gets more complicated then it can be more difficult to perform and so it is up to the programmer to implement. As with most optimizations there is some penalty for the increase in speed, in this case the penalty for the computer is storing an extra variable and for the programmer it is readability.

Loop Invariant Code Motion

- Consider

```
for(i=0; i<=N; i++)
{
        A[i]=F[i] + C*D;
        E=G[K];
}
```

- Optimized as

```
temp=C*D;
for(i=0; i<=N; i++)
{
        A[i]=F[i] + temp;
}
E=G[K];
```

- ✓ For each iteration of the loop the value of C*D, E and G[K] does not change.
- ✓ C*D is only calculated once rather than N times and E is only assigned once.
- ✓ compiler may not be able to move a complicated expression out of a loop and it will be up to the programmer to perform this optimization.

Loop Fusion

- Consider

```
DO i=1,n
        x(i) = a(i) + b(i)
ENDDO
DO i=1,n
        y(i) = a(i) * c(i)
ENDDO
```

- Optimized as

```
DO i=1,n
    x(i) = a(i) + b(i)
    y(i) = a(i) * c(i)
ENDDO
```

✓ It has better data reuse, a(i) will only have to be loaded from memory once. Also the CPU has a greater chance to perform some of the operations in parallel.

Pushing Loops inside Subroutine Calls

- Consider

```
 DO i=1,n
                CALL add(x(i),y(i),z(i))
ENDDO

SUBROUTINE add(x,y,z)
        REAL*8 x,y,z
        z = x + y
END
```

- Optimized as

```
CALL add(x(i),y(i),z(i),n)

SUBROUTINE add(x,y,z,n)
        REAL*8 x(n),y(n),z(n)
        INTEGER i
        DO i=1,n
        z(i)=x(i)+y(i)
        ENDDO
END
```

✓ The subroutine add was called n times, adding the overhead of n subroutine calls to the run time of the code. After the loop is moved inside the body of the subroutine, there will only be one subroutine call.

Loop Index Dependent Conditionals

- Consider

```
DO I=1,N
    DO J=1,N
        IF( J.LT.I ) THEN
            A(J,I)=A(J,I) + B(J,I)*C
        ELSE
            A(J,I)=0.0
        ENDIF
    ENDDO
ENDDO
```

- Optimized as

```
DO I=1,N
    DO J=1,I-1
        A(J,I)=A(J,I) + B(J,I)*C
    ENDDO
    DO J=I,N
        A(J,I)=0.0
    ENDDO
ENDDO
```

- ✓ It has an if-block that must be tested on each iteration of the loop. The loop could be rewritten as follows to remove if-block

Loop unrolling

- Consider

```
DO I=1,N
      A[I]=B[I]+C[I]
ENDDO
```

  - There are two loads for B[i] and C[i], one store for A[i], one addition B[i]+C[i], another addition for I=I+1 and a test i<=N. A total of six operations.

- Consider

```
DO I=1,N,2
      A(i)=B(i)+C(i)
      A(i+1)=B(i+1)+C(i+1)
ENDDO
```

  - four loads, two stores, three additions and one test for a total of ten.

✓ The second form is more efficient with ten operations for effectively two iterations against six operations for one iteration of the first loop. This is called loop unrolling

Contd..

- if the loop is unrolled further it becomes even more efficient. It should also be noted that if the CPU can perform more that one operation at a time the payoff may be even greater.
- Originally programmers had to perform this optimization themselves but now it is best left to the compiler.

- What happens when there is to be an odd number of iterations of the loop????

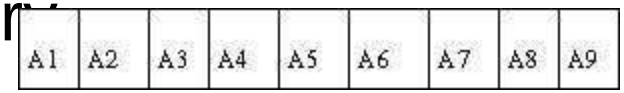
The compiler adds a pre-conditioning loop to fill in the extra iterations

```
NUM=IMOD(N,4)
DO I=1,NUM
      A(I)=B(I)+C(I)
ENDDO
DO I=NUM,N,4
      A(I)=B(I)+C(I)
      A(I+1)=B(I+1)+C(I+1)
      A(I+2)=B(I+2)+C(I+2)
      A(I+3)=B(I+3)+C(I+3)
ENDDO
```

- ✓ The second loop has been unrolled four times, the first loop computes the "odd" number of iterations. IMOD(N,4) returns the remainder of dividing N by 4.

Loop Stride Size

- Arrays are stored in sequential parts of main memory



- When the CPU requests an element from an array it not only grabs the specified element but a number of adjacent elements to it. It stores these extra elements in a cache, the cache is a small area of very fast memory
- In the above diagram if the CPU requests A1 it will grab A1 through to A9 and store them in the cache.

Contd..

- Hopefully the next variable it requests will be one of these elements stored in the cache saving time by not accessing the slower main memory. (Cache Hit)
- If not, all the elements stored in the cache are useless and we have to make the long trip to main memory. (Cache Miss)
- The elements are loaded into what is called a cache line, the length of the cache line defines how many elements can be stored. The cache line size differs from CPU to CPU.
- Using the cache in the most efficient manner is the Objective.

Contd..

- The size of the step that a loop takes through an array is called the stride.
  - Unit Stride (max Cache Hits => Most Efficient Utilization)
  - Stride of Two
- lay out of Multi Dimensional arrays are different in memory depending on language.
  - Viz. Implementation in FORTRAN vs. C
    - In FORTRAN unit stride is achieved by iterating over the left most subscript first.
    - To achieve unit stride for multi-dimensional array in C you should iterate over the rightmost subscript

Float-Point optimizations

- Compilers follow a IEEE Floating-Point Standards (viz. IEEE_754) to perform floating point operations.
- This rules can be governed and by using a set of less restrictive set of rules the compiler may be able to speed up the computation.
- This less restrictive rules rarely give the results that may not be same as those for non-optimized ones or the difference is insignificant.

✓ Optimizations done should be checked for results with the non-optimized ones.

Discussions ??

Take Away

- Reorder operations to allow multiple computations to happen in parallel, either at the instruction, memory, or thread level.
- Code and data that are accessed closely together in time should be placed close together in memory to increase spatial ***locality of reference***. (Temporal and Spatial)
- Accesses to memory are increasingly more expensive for each level of the memory hierarchy, so place the most commonly used items in registers first, then caches, then main memory, before going to disk

Notes

- The more information a programmer gives the compiler the better job it can do at optimizing the code.

- “Code Less.. Create More..!! “
- The more precise the information the compiler has, the better it can employ any or all of these optimization techniques.

Glossary

- **Super Scalar** :

  microprocessor architectures that enable more than one instruction to be executed per clock cycle. Nearly all modern microprocessors, including the Pentium, PowerPC, Alpha, and SPARC microprocessors are superscalar.

Bibliography

- http://en.wikipedia.org/wiki/Compiler_optimization

Thank you..!!

Replace Multiply by Shift

- A := A * 4;
  - Can be replaced by 2-bit left shift (signed/unsigned)
  - But must worry about overflow if language does
- A := A / 4;
  - If unsigned, can replace with shift right
  - But shift right arithmetic is a well-known problem
  - Language may allow it anyway (traditional C)

Addition chains for multiplication

- If multiply is very slow (or on a machine with no multiply instruction like the original SPARC), decomposing a constant operand into sum of powers of two can be effective:

$$X * 125 = x * 128 - x*4 + x$$

  - **two shifts**, **one subtract** and **one add**, which may be faster than one multiply

Note: similarity with efficient exponentiation method

Implementation of Multi Dimensional Arrays

- FORTRAN (Unit Stride)

```
DO J=1,N
    DO I=1,N
        A(I,J)= B(I,J)*C(I,J)
    ENDDO
ENDDO
```

- FOTRAN (Stride N)

```
DO J=1,N
    DO I=1,N
        A(J,I)= B(J,I)*C(J,I)
    ENDDO
ENDDO
```

- C (Unit Stride)

```
for(i=0; i<=n; i++)
    for(j=0; j<=n; j++)
        a[i][j]=b[i][j]*c[i][j];
```

Resume

Instruction Pipelining in CPU

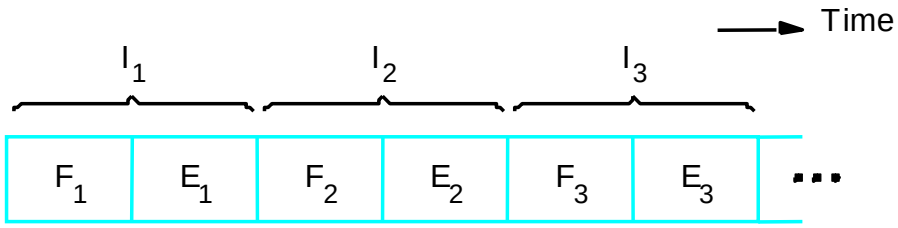
- ***Pipelining*** improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

- ***Performance*** :
  - The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
  - However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
  - Unfortunately, this is not true.

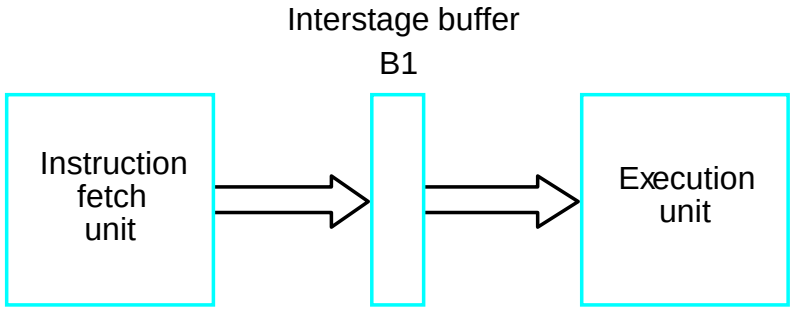
Resume

Use the Idea of Pipelining in a Computer

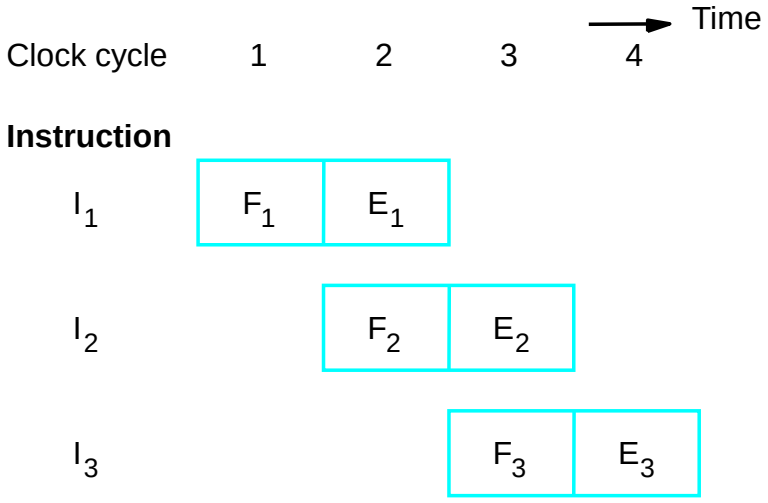
Fetch + Execution



(a) Sequential execution



(b) Hardware organization



(c) Pipelined execution

Figure1. Basic idea of instruction pipelining.

Contd..

Fetch + Decode
+ Execution + Write

Pipeline Performance

Resume

Role of Cache Memory

- Each pipeline stage is expected to complete in one clock cycle.
- The clock period should be long enough to let the slowest pipeline stage to complete.
- Faster stages can only wait for the slowest one to complete.
- Since main memory is very slow compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.
- Fortunately, we have cache.

- Resume

Optimization Techniques

- **Common themes**
  - Optimize the common case
  - Avoid redundancy
  - Less code
  - Fewer jumps by using *straight line code*, also called *branch-free code*
  - Locality
  - Exploit the memory hierarchy
  - Parallelize
  - More precise information is better
  - Runtime metrics can help
  - Strength reduction

Specific techniques

- **Loop optimizations**
  - Induction variable analysis
  - Loop fission or loop distribution
  - Loop Fusion or Loop Combining
  - Loop inversion
  - Loop interchange
  - Loop- Invariant Code Motion
  - Loop nest Optimization
  - Loop Reversal
  - Loop unrolling
  - Loop Splitting
  - Loop unswitching
  - Software Pipelining
  - Automatic Parallelization
- Data Flow Optimizations
  - Common sub-expression elimination
  - Constant Folding and Propagation
  - Induction Variable recognition and elimination
  - Alias Classification and pointer analysis
  - Dead store elimination
- SSA Based Optimizations
  - Global Value Numbering
  - Sparse conditional constant propagation
- Code Generator Optimizations
  - Register allocation
  - Instruction selection
  - Instruction scheduling
  - Rematerialization
  - Code Factoring
  - Trampolines
  - Reordering Computations
- Functional Language Optimizations
  - Removing Recursion
  - Deforestation (data structure fusion)
- Others
  - Bounds checking elimination
  - Branch offset optimization
  - Code block reordering
  - Dead code elimination
  - Factoring out invariants
  - Inline/Macro Expansion
  - Jump threading

Inline expansion or macro expansion

- When some code invokes a procedure, it is possible to directly insert the body of the procedure inside the calling code rather than transferring control to it.
- This saves the overhead related to procedure calls, as well as providing great opportunity for many different parameter-specific optimizations, but comes at the cost of space;
- The procedure body is duplicated each time the procedure is called inline.
- Generally, inlining is useful in performance-critical code that makes a large number of calls to small procedures.
  - A "fewer jumps" optimization.

Macro Compressions

- A space optimization that recognizes common sequences of code, creates subprograms ("code macros") that contain the common code, and replaces the occurrences of the common code sequences with calls to the corresponding subprograms.

- This is most effectively done as a machine code optimization, when all the code is present.

- The technique was first used to conserve space in an interpretive byte stream used in an implementation of Macro Spitbol on

Interprocedural optimizations

- Interprocedural optimization works on the entire program, across procedure and file boundaries.
- It works tightly with intraprocedural counterparts, carried out with the cooperation of a local part and global part.
- Typical interprocedural optimizations are:
  - procedure inlining
  - interprocedural dead code elimination
  - interprocedural constant propagation
  - procedure reordering.
- As usual, the compiler needs to perform interprocedural analysis before its actual optimizations.
- Interprocedural analyses include alias analysis, array access analysis, and the construction of a call graph.
- Due to the extra time and space required by interprocedural analysis, most compilers do not perform it by default. Users must use compiler options explicitly to tell the compiler to enable interprocedural analysis and other expensive optimizations.