

VISTA: A System for Interactive Code Improvement

Wankang Zhao, David Whalley,
Robert van Engelen, Xin Yuan, Kyle Gallivan
Florida State University

Baosheng Cai Mark Bailey
Oracle Corporation Hamilton College

Jason Hiser, Jack Davidson
University of Virginia

Douglas Jones
University of Illinois at Urbana-Champaign

Embedded Systems: a Difficult Target

- unusual architectural features
 - low overhead looping hardware
 - specialized address and arithmetic functions
 - highly irregular instruction sets
- stringent application constraints
 - real-time deadlines
 - absolute memory limitations
- efficient code requires specific user knowledge
 - value ranges
 - memory disambiguation
 - determining loop bounds

Choices for Coding Embedded Systems Applications

- high-level language
 - difficult to exploit special-purpose hardware
 - less control over performance
- assembly language
 - difficult to maintain and retarget
 - coding is slow
 - error prone
- hybrid at the module level
 - too coarse grain

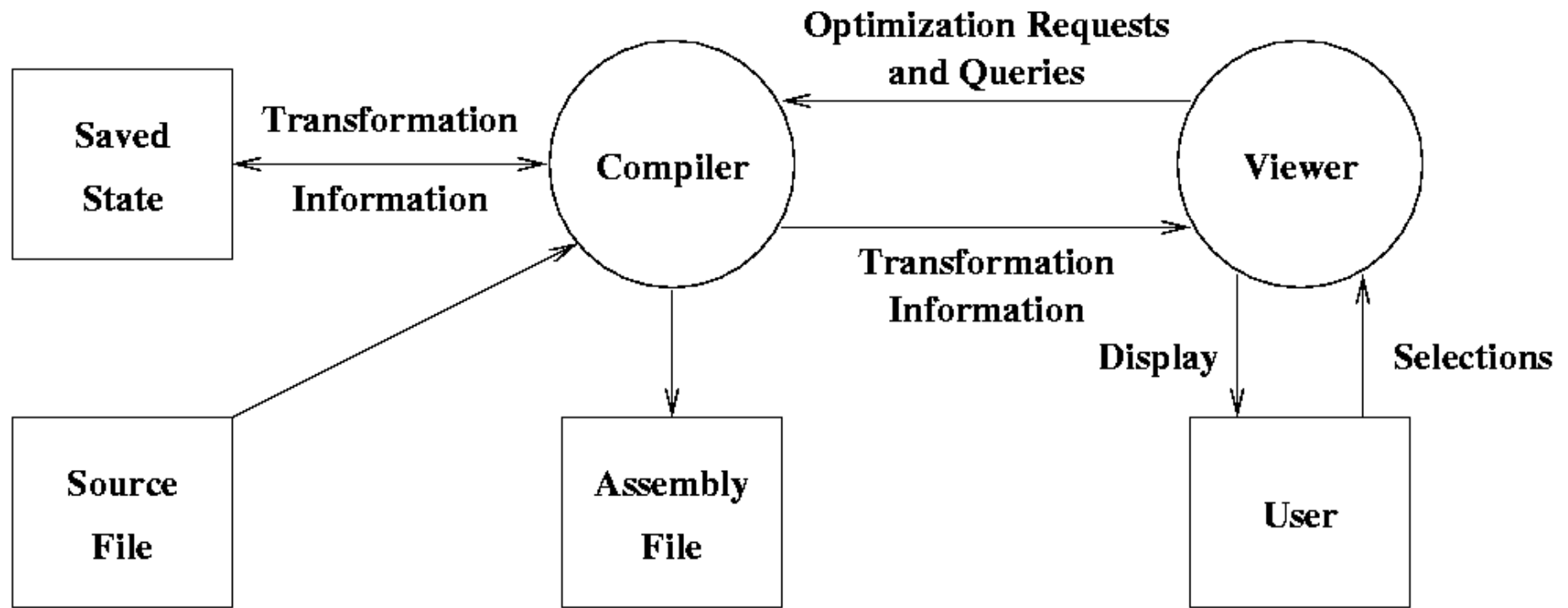
Interactive Code Improvement

- Application development in a high-level language
- Low-level code improvement assisted by developers
 - Selecting the order and scope of traditional optimization phases
 - User-specified code improvements
- User guided code improvement assisted by the compiler

Related Work

- Compiler debugging
 - XVPODB: Boyd, Whalley
- High-level parallelization of programs
 - Pat toolkit: Appelbe, Smith, McDowell
 - Paraphrase-2: Polychronopoulos, Girkar, et al.
 - Pittsburgh system: Dow, Chang, Soffa
 - SUIF Explorer: Liao, Diwan, Bosch, et al.

VISTA: Vpo Interactive System for Tuning Applications



Features of the Environment

- View the representation of a function at any optimization point.
- Specify the order and scope of optimization phases.
- Specify code-improving transformations manually.
- Visualize performance of the application.
- Reverse previously applied transformations.
- Obtain information from the compiler.
- Specify improvements over multiple sessions.

Viewing the Low-Level Representation

- Natural level for embedded systems performance tuning.
- Supports a variety of display options.
 - RTLs
 - assembly
 - control flow
- Eases debugging of compiler errors.
- Provides a better understanding of the code improvement process to a user.

History of Compilation Phases

UserInterface

Function	Trans Number	Total
State	Before	418

transformations

Jump Minimization	1	
Instruction Selection	5	
Register Allocation	12	
Dead Variable Elimination	7	
Register Allocation	11	
Instruction Selection	(15)	64
Register Allocation	1	
Dead Variable Elimination	6	
Common Subexpr Elimination	16	
Instruction Selection	15	
Dead Variable Elimination	2	
Instruction Selection	28	
Common Subexpr Elimination	16	
Instruction Selection	14	
Dead Variable Elimination	1	
Instruction Selection	15	
Fix Entry Exit	4	
Fill Delay Slots	7	

Setup Trans Sequence Specify Trans by Hand **RTLs**

< << < > >> >| Option Exit

Message: RTL type is going to be changed

```
1 |
r[10]=r[12];
r[9]=r[11];
r[8]=r[9];
r[8]=r[8]-1;
r[13]=r[8];
r[8]=r[9];
IC=r[8]?0;
PC=IC:0,L002;
```

```
2 | L92
r[8]=r[10];
r[9]=r[8];
r[8]=r[9];
r[8]=r[8]+1;
r[10]=r[8];
r[8]=r[9];
r[8]=(B[r[8]}{24})24;
IC=r[8]?0;
PC=IC!0,L93;
```

```
3 |
r[8]=32;
```

Control Flow: A Bird's Eye View

The screenshot displays a software interface titled "UserInterface". On the left, there is a table with the following data:

Function	number	Trans Number	308
State	After	Total	308

Below the table, a section titled "transformations" lists various operations and their counts:

Register Allocation	5
Instruction Selection	30
Register Allocation	4
Dead Variable Elimination	3
Common Subexpr Elimination	12
Instruction Selection	10
Dead Variable Elimination	2
Code Motion	1
Dead Variable Elimination	1
Instruction Selection	14
Common Subexpr Elimination	12
Instruction Selection	9
Dead Variable Elimination	2
Instruction Selection	10
Fix Entry Exit	1
Instruction Selection	22

At the bottom of the left panel, there are buttons for "Setup Trans Sequence", "Specify Trans by Hand", and a dropdown menu set to "Ctl Flow". Below these are navigation buttons: "<", "<<", "<|", ">|", ">>", and ">". There are also "Option" and "Exit" buttons. A message box at the bottom left shows "Message: No Message".

On the right side of the interface, a control flow graph is shown. It consists of seven rectangular nodes, each with a blue border and a light blue fill. The nodes are labeled as follows:

- Node 1: -1 |
- Node 2: -2 | L005
- Node 3: -3 | L37
- Node 4: -4 | L43
- Node 5: -5 | L41
- Node 6: -6 | L42
- Node 7: -7 | L004

Vertical arrows point downwards from each node to the next. A black arrow points from the right side of Node 1 to the right side of Node 3. A black arrow points from the right side of Node 3 to the right side of Node 6. A black arrow points from the right side of Node 6 to the right side of Node 7. A green arrow points from the right side of Node 3 to the right side of Node 5. A red arrow points from the right side of Node 4 to the right side of Node 6.

Specifying Compilation Phases

- Gives the user control over the code improvement process.
- Helps to address the phase ordering problem.
- Phases can be specified to be performed repeatedly until no more changes are made.
- Can limit the scope of the program representation where a phase is applied.
- Certain restrictions still have to be enforced.

Phase Order Control

The screenshot displays a compiler's UserInterface window. On the left, there are two panels: 'Transformation Selection' and 'Transformation Sequence'. The 'Transformation Selection' panel contains a grid of options, with 'if changes goto' selected. The 'Transformation Sequence' panel shows a list of transformations, with '3. Register Allocation' highlighted. At the bottom, there are buttons for 'Loops', 'Undo Last Change', 'Done', and 'Cancel', and a 'Message' field showing 'No Message'.

The main area of the window shows a control flow graph with three nodes, each containing assembly code:

```
1 |
add  %r30, .1_argc, %r32
ld   [%r32], %r33
mov  2, %r34
cmp  %r33, %r34
bge  .L29

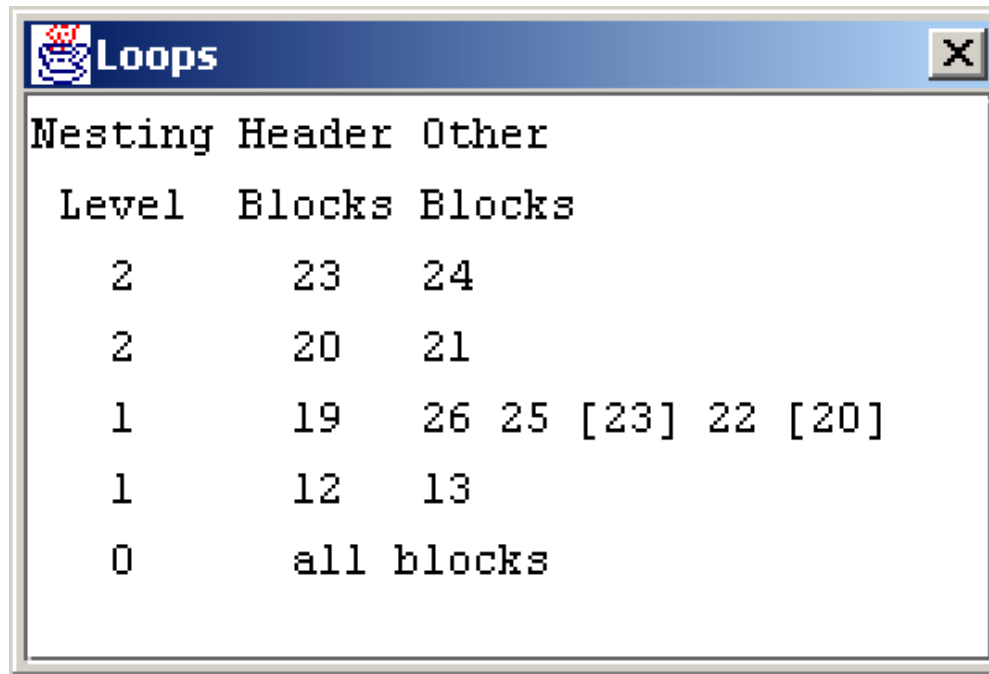
2 |
sethi %hi(.L31), %r32
add  %r32, %lo(.L31), %r32
sethi %hi(printf), %r33
add  %r33, %lo(printf), %r33
mov  %r32, %r8
call %r33
mov  %g0, %r32
sethi %hi(exit), %r33
add  %r33, %lo(exit), %r33
mov  %r32, %r8
call %r33

3 | L29
add  %r30, .1_argc, %r32
ld   [%r32], %r33
```

Arrows indicate the flow from node 1 to node 2, and from node 2 to node 3. A return arrow also points from node 2 back to node 1.

Restricting the Scope of Phases

- set of basic blocks by clicking on each block
- set of loops by clicking on loops in the loop report



The screenshot shows a window titled "Loops" with a close button in the top right corner. The window contains a table with the following data:

Nesting Level	Header Blocks	Other Blocks
2	23	24
2	20	21
1	19	26 25 [23] 22 [20]
1	12	13
0	all blocks	

User Specified Improvements

- Often difficult to exploit embedded features.
- User can tune compiler generated code.
- User can make queries to the compiler.
 - What registers are live at a given point?
 - Which blocks dominate a specified block?
 - What loops exist in the function?
 - ...
- Useful for prototyping code improvements.

Manually Specifying a Transformation

The screenshot displays a software interface titled "UserInterface" with the following components:

- Function:** main
- Trans Number:** 24
- State:** (empty)
- Total:** 24

Changes in transformation:

1. Move RTL 36 in block 2 to block 2
2. Modified RTL 65 in block 2

The main editor shows three code blocks:

- Block 2:** Contains RTL instructions: `r[8]=HI[L31];`, `r[9]=HI[printf];`, `r[8]=r[8]+LO[L31];`, `r[9]=r[9]+LO[printf];`, `r[8]=r[8];`, and `ST=r[9];`. The instruction `ST=r[9];` is highlighted in red, and a context menu is open over it with options: "Move a RTL", "Insert RTL after specified RTL.", "Delete specified RTL.", "Modify specified RTL.", and "List registers live before RTL".
- Block 3 | L29:** Contains RTL instructions: `r[8]=r[30]+.1_argc;`, `r[8]=R[r[8]];`, `r[9]=2;`, `IC=r[8]?r[9];`, and `PC=IC!0,L33;`.
- Block 4:** Partially visible at the bottom.

Navigation arrows indicate flow between blocks. A green arrow points from the context menu to Block 3.

Buttons: Query, Undo Last Change, Done, Cancel & Back

Message: Please select instruction operations

Visualizing Performance

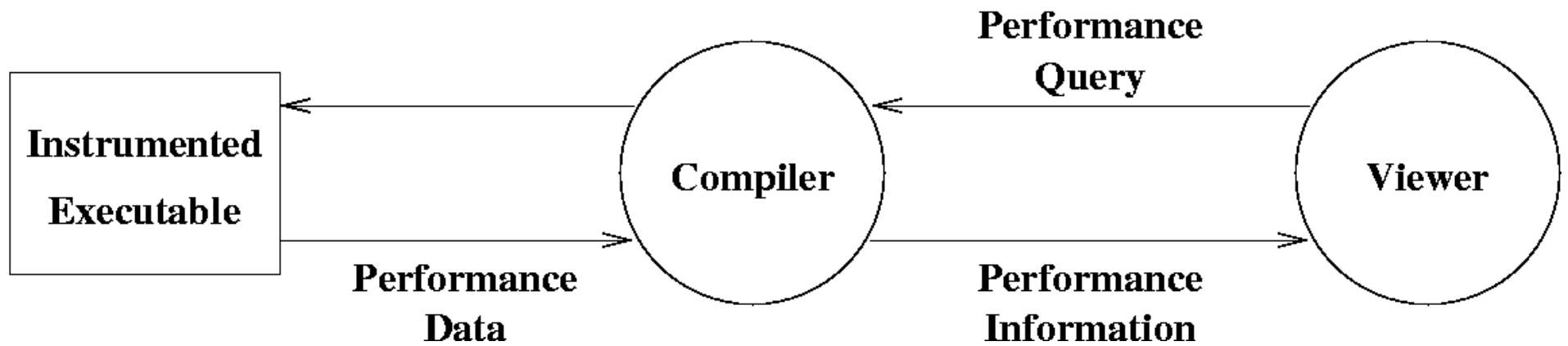
- Can obtain performance measurements and can view them on blocks or loops.

The screenshot displays the 'UserInterface' window, which is used for visualizing performance. The window is divided into several sections:

- Function and State:** The 'Function' field is set to 'main', and the 'State' field is empty.
- Trans Number and Total:** Both are set to 1890.
- transformations:** A list of transformations and their counts is shown:

Transformation	Count
Branch Chaining	4
Eliminate Empty Blocks	1
Useless Jump Elimination	3
Dead Code Elimination	4
Reverse Branches	8
Basic Block Reordering	4
Merge Basic Blocks	5
Other Transform	1
Instruction Selection	610
Eval Order Determination	3
Instruction Selection	97
Register Assignment	23
Instruction Selection	97
Jump Minimization	4
Merge Basic Blocks	4
Instruction Selection	8
Register Allocation	46
Dead Variable Elimination	17
- Assembly Code Blocks:** The main area shows three assembly blocks connected by arrows, indicating control flow:
 - Block 1 (Top):** Contains instructions: `bl printf`, `ldr r0,[r4,#36]`, `ldr r9,[r4,#12]`, `bl printf`, and `ldr r11,[r4,#40]`.
 - Block 2 (Middle):** Labeled '-18 | L22 | freq: 0.053%', containing: `mov r12,#0`, `mov r0,#0`, and `mov r1,r10`.
 - Block 3 (Bottom):** Labeled '-19 | L24 | freq: 98.656%', containing: `strb r0,[r1,r12]`, `add r12,r12,#1`, `cmp r12, #432`, and `blt .L24`. A loop arrow indicates a branch from the bottom back to the top of this block.
 - Block 4 (Bottom):** Labeled '-20 | L002 | freq: 0.013%', containing: `ldr r1,[r9,r5, lsl #2]`, `mov r0,r11`, `mov r6,#0`, `mov r8,r10`, and `bl printf`.
- Control Flow:** Arrows show the flow from Block 1 to Block 2, Block 2 to Block 3, and Block 3 to Block 4. A loop arrow also connects Block 3 back to itself.
- Message:** A message box at the bottom states: 'Message: The program representation is Assembly'.

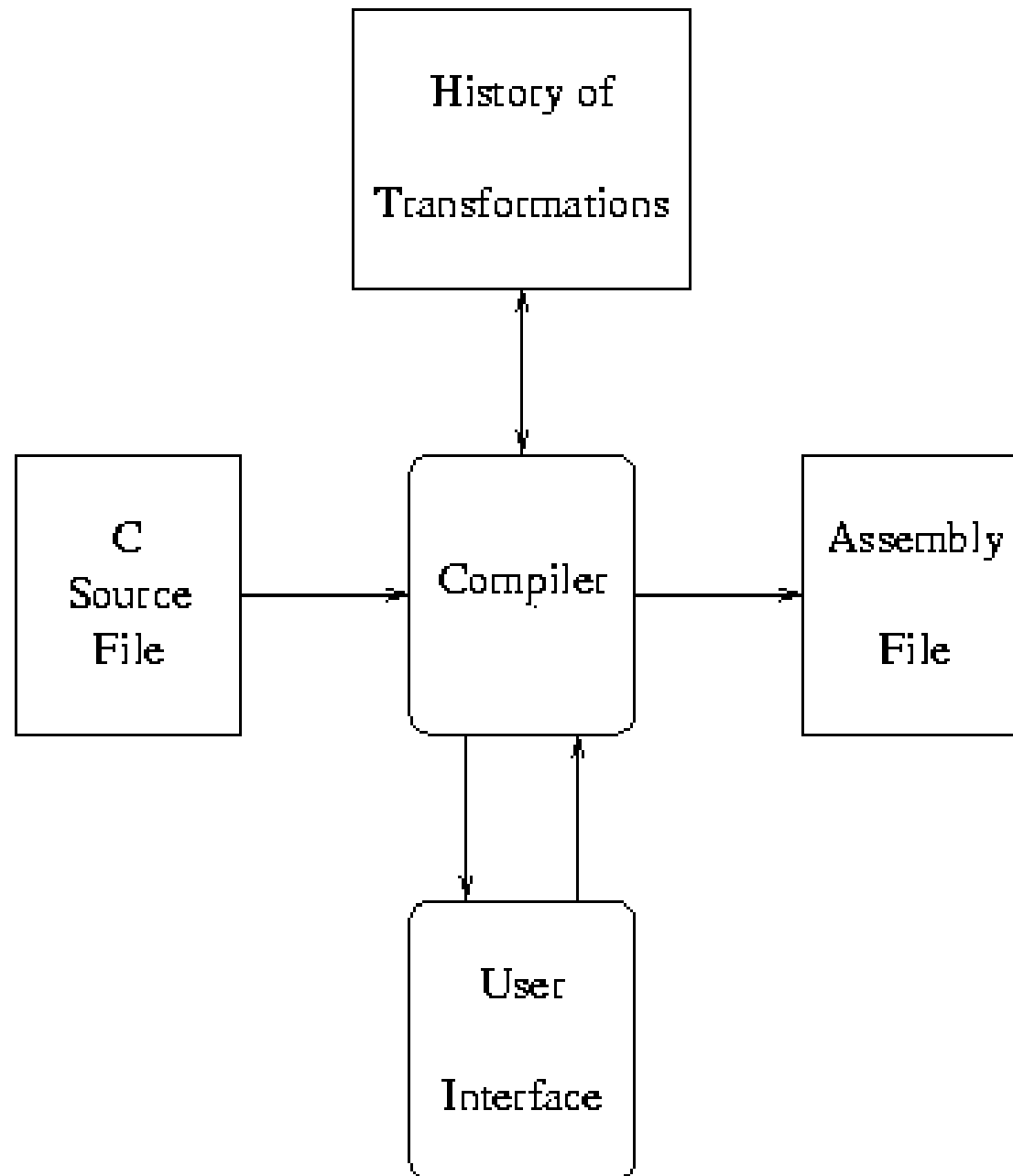
Performance Information Collection



Traversing Applied Transformations

- Can apply or undo transformations.
- Allows a user to experiment with different compilation phase orderings.
- All changes are stored.
- Changes, both compiler and user specified, are saved to a file.

Transformation History Is Saved



Implementation Issues

- Used Java for the user interface to enhance its portability.
- Communication between the compiler and user interface was accomplished using UNIX sockets.
- Analysis needed for or invalidated by each optimization phase had to be identified.
- Translators were required to convert a human specified RTL or assembly instruction into an encoded RTL.

Future Work

- Patterns for detecting code improvement opportunities.
- Show performance improvement.
- Support iterative compilation to meet specified constraints on speed, size, and power.
- Include a mapping between source and assembly.

Conclusions

- Useful for effective embedded systems development.
 - Benefits of coding in a high-level language.
 - Flexibility of coding in assembly.
 - Compiler can exploit user knowledge.
 - User can use compiler supplied information.
- Useful for debugging compiler errors.
- Useful for prototyping.