5

CODE OPTIMIZATION

5.1. Motivation and Approach ................................................. 123
5.2. Detection of Redundant Bank Switching Codes ........ 126
   - Relation Matrix Formulation
   - Realization
   - Tool Evaluation
5.3. Optimization Technique .................................................. 141
   - Variable Partitioning
   - Optimum Memory Bank Allocation
5.4. Redundant I/O Port Configuration ................................. 147
5.5. Redundant ADC Channel Selection ....................... 148
5.6. Software Realization .............................................. 149
5.7. Summary ......................................................... 150

Optimization is a procedure that mainly seeks to maximize performance and minimize code size. As processor architectures have exponentially increased in complexity, the compiler optimization techniques [38, 39, 178] are continually advancing. However, no single optimization technique will work for every application. It is best to apply one optimization at a time, verify the results and measure any performance improvements before moving on. Optimization is an important task when developing resource intensive applications like embedded systems. Embedded systems are usually designed for a single or a specified set of tasks. Being specific the system design as well as its hardware/software development can be highly optimized.
Optimization by elimination of redundant codes is one among the optimization technique adopted by many software development tools. An architecture oriented approach towards the optimization, by the static analysis of embedded system machine codes resulting in the elimination of redundant code is described in this chapter. Many of the embedded system controllers use partitioned memory architecture. Memory banking and memory paging are common techniques used for microcontrollers, which increases the size of program and data memory without extending the address buses of the CPU. In these memory banks that cannot be accessed simultaneously, switching between them requires at least one bank selection instruction, which induce extra overhead in code size and execution time. The code size is a major factor rather than speed for the programs running in many embedded systems, since smaller code size often means less consumption of ROM as well as energy, and hence minimizing the number of bank selection instructions is an important research topic. Current compilers provide limited support to optimum generation of bank switching codes.

This work presents a code optimization technique to minimize the data and program memory bank switching instructions that assist a programmer in developing efficient embedded software. A relation matrix formed for the memory bank state transition corresponding to each bank selection instruction is used for the detection of redundant codes. It also proposes the optimum memory bank allocation to the variables in a program by the compiler that results in minimum number of bank switching codes. An algorithm is developed and utilized to detect the redundant memory bank switching instructions in the resulting machine codes from a compiler for different data allocation schemes of the application program. Then it selects the program with minimum bank switching instructions as the optimum solution. The basics of the algorithm developed, enhancements made to
the algorithm in order to suppress the false warnings and the results of the case study using Microchip’s PIC16F877 microcontroller are presented.

The technique presented here achieves optimization of bank switching instructions without much computational burden by statically analyzing the machine code with a comparatively simple algorithm. Our algorithm can detect redundant data memory bank selection instructions that remain even after the application of optimization techniques by the compilers. For a program developed in assembly language also, the redundant bank switching instructions, especially for lengthy programs can be eliminated with this technique. With a well defined Control Flow Graph (CFG) constructed from the machine codes this algorithm fits well into large problem sizes as well. Redundant data and program memory bank selection instructions in the intraprocedural sequence, loops and interprocedural routines in the application program can be eliminated.

5.1. Motivation and Approach

For any memory space, larger the memory is, the larger the address bus needs to be. Previous efforts on partitioned memory are to enable memory access in parallel thereby increasing memory bandwidth and thus improving program performance. Such partitioned memory banks are found in processors like Motorola DSP 56000, Intel 8086, i80186 etc onwards. One way of avoiding large address buses is to divide the memory into a number of smaller blocks - called banks/ pages - each identical in size in most of the cases so that a smaller address bus can be used [35]. Smaller address buses result in smaller chip die size, higher clock frequencies and less power consumption. It can access all banks in an identical way, with just one of the banks being identified at any one time called the active memory bank (AMB) [17] as the target of the address specified. The contents of memory temporarily bank-switched out of the processors address space
are inaccessible to the processor. Many MCUs have banked memories that cannot be addressed simultaneously. For example, Freescale 68HC11 8-bit microcontrollers [179] allow multiple 64KB memory banks to be accessed by their 16-bit address registers with only one bank being active at a time. Bank switched SRAMs are employed with ultra-low-power sensors to achieve high code density [180] and allow the gating of individual memory banks [181]. Other examples include Intel 8051 processor family and MOS technology 6502 series microcontrollers. Certain modern microcontrollers use bank switching to manage read-write memory, non-volatile memory, input-output devices and system management registers. Most of the PIC microcontrollers [165] adopt a banked structure for their data as well as program memory of which a case study on PIC16F87X series of microcontrollers has been made in this work.

Unlike other memory management techniques, bank switching is nearly always initiated by the application program explicitly, although some real time operating systems take detailed control of the bank switching operation out of the application programmer's hands. A bank-sensitive program statement requires that the appropriate bank is to be made active prior to its execution. Otherwise, the program semantics are violated. This introduces an additional burden on the programmer; there is always a possibility for redundant bank switching instructions. Thus, if data in one bank must be copied to another bank, bank selection instructions are always necessary. Obviously, placing all the variables accessed by a function in the same memory bank will reduce the number of bank selection instructions and the total required cycles for the application. However, conventional compilers have no way of knowing which functions call which variables and are therefore unable to optimize their memory assignment. Nor do these compilers have any way of knowing whether or not a particular memory bank will be selected at any point in the code. As a result, these compilers automatically
generate bank selection instructions for every memory access, whether or not that bank is already selected, unnecessarily bloating the code - often increasing the code space requirement. Compiler vendors have addressed this issue by providing bank qualifiers - extensions to the C-code. This allows the compiler to see the exact bank an object resides in and reduces the number of bank selection instructions for more compact code. However, trying to track all the memory addresses across multiple code modules and ensuring all pointers to have the appropriate qualifiers is a time consuming and tedious process. This requires substantial expertise as well as run the risk of introducing programming errors [8]. All the related works [17, 31, 32, 33, 34] are analyzing the source programs for minimal placement of bank switching instructions.

Analysis of a high level program cannot easily determine the current bank state. But with a static analysis of the machine code, the state transitions at each bank switching instruction can be easily determined. Static analysis examines the code of programs to determine properties of the dynamic execution of these programs without running them. This technique has been used extensively in the past by compiler developers to carry out various analysis and transformations aiming at optimizing the code [40, 96]. Today’s compilers cannot efficiently exploit the architectural features of advanced embedded processors. This results in the introduction of redundant codes in their output. This work presents an algorithm developed, to detect the redundant bank switching codes in the machine code generated by the compiler. So the compilers can insert bank selection instructions for every memory access in the conventional way and the output file in the Intel Hex file format is tested with the algorithm developed to detect all the redundant bank switching code. So the compiler is deprived of any complicated analysis needed during compilation to minimize the bank switching code as done by some advanced compilers like HI-TECH OCG (Omniscient Code
Generation)[8]. Now appropriate allocation of data variables to the available memory banks can again increase the redundant bank switching codes detected by the algorithm developed, resulting in minimum number of such codes in a given application program. To the best of the authors’ knowledge only [34] presents a data partition technique aimed at minimal placement of bank selection instruction resulting in code as well as runtime saving. Our non profile-guided compiler method is static and is independent of the compiler but implementation of algorithm depends on certain architectural features of the target processor. The optimization of the code is done again following certain algorithms. Some of the instructions inside a loop will be executed unnecessarily which can be eliminated by code motion.

5.2. Detection of Redundant Bank Switching Codes

The goal of our optimization is to eliminate the redundant bank selection instructions in a program while ensuring that the banked memory is accessed correctly. The detection of redundant bank switching code is done with the help of a relation matrix derived from the architectural features of the target processor like number of memory banks and instruction set (memory bank switching codes). Though the implementation depends on the target processor the formation of the relation matrix can be generalized as explained below. The feasibility of the approach has been verified on systems based on PIC16F87X series of microcontrollers.

A detailed study on the various PIC families of microcontrollers has been made in this regard. PIC 16F84A have just two banks [35] and the address of either bank is the 7-bit RAM address. The active bank is selected by bit 5 in the Status register. The programmer must ensure that the bank bit in the Status register is correctly set before making any access to memory. The data memory in
PIC16F87X devices is partitioned into four banks of 128 Bytes each, which contain the general purpose registers and the Special Function Registers (SFR). For selecting a particular bank, bits \( RP1;bit\ six\ of\ status\ register (status<6>) \) and \( RP0 \) \( (status<5>) \) are to be configured appropriately. All PIC16F87X devices are capable of addressing a continuous 8K word block of program memory. The \textit{call} and \textit{goto} instructions provide only 11 bits of address to allow branching within any 2K program memory page. While doing a program branching with \textit{call} or \textit{goto} instruction the upper two bits of the address are provided by \( pclath<4:3> \). When doing such branching, the user must ensure that the upper page select bits are programmed so that the desired program memory page is addressed.

The data memory space in PIC18F series devices is divided into as many as 16 banks that contain 256 bytes each [182]. The Bank Select Register, \( BSR<BSR3:BSR0> \) holds the four bit bank; the instruction itself includes the 8 Least Significant Bits, which can be thought of as an offset from the bank’s lower boundary. The BSR can be loaded directly by using the movlb instruction.

In general, the address space is partitioned into memory banks and the CPU can access one bank at a time, which is called the active memory bank (AMB), using bank selection bits or bank selection instruction. For implementing this code optimization through static analysis of machine code, the memory bank that was active just before the execution of a bank switching instruction is named as Previously Activated Memory Bank (PAMB). A \textit{bank switching/ selection instruction is said to be redundant when the execution of such an instruction switches the memory bank to an Active Memory Bank (AMB) that does not alter the PAMB}.

Based on the study on the various PIC families of microcontrollers following generalizations are made for the partitioned data memory architecture. If
Chapter 5

$P$ is the number of memory banks, so that $2^r = P$, then the number of bits that decide the bank selection in the bank selection register will be $r$. The number of machine codes controlling the bank selection will be $P$ if the bank register is loaded with a `mov` instruction. For each PAMB state there will be one bank selection instruction, which is redundant. If `bitset` and `bitclear` instructions on the $BSR$ are used for bank switching there will be $2r$ number of machine codes for this operation and for each PAMB state there will be $r$ number of bank switching instructions, which are redundant.

5.2.1 Relation Matrix Formulation

The family of Microchip PICmicro MCUs constitutes a RISC-based Harvard architecture with instruction size of 14 bits and data width of 8 bits [165]. The data memory banks in these embedded controllers contain the General purpose Registers and Special Function Registers. For proper functioning of the device, proper configuring of these registers is essential. Since these registers are spread across different banks they are to be accessed through the bank switching instructions, which limits the data partitioning optimization for hardware dependent code. In case of program memory paging when a branching instruction that crosses the page boundary is made, the programmer should ensure the required page switching before these instructions. The disadvantage of bank-switched architectures is the code size and runtime overhead caused by bank selection instructions.

Instead of having a single bank selection instruction, the PIC16F87X architecture provides only bit access to the bank selection register, which is the status register. The assembly instructions that clear or set the bits $RP0$ and $RP1$ of the status register are `bcf status, RP0`; `bcf status, RP1`; `bsf status, RP0`; and `bsf status, RP1` and are represented by the symbols $a$, $b$, $c$ and $d$ respectively. The hex
codes corresponding to these instructions are 1283h, 1303h, 1683h and 1703h respectively. The four data memory banks are named B0, B1, B2 and B3. On a power on reset, the default bank that is active is bank’0’ represented as B0. Depending on the PAMB state, the AMB state occurs with each bank switching instruction. The assembly instructions that set or clear the bits RP0 and RP1 of the status register, corresponding machine code sequence, and their symbols used in the state transition diagram are shown in Table 5.1.

Table 5.1 Bank switching instructions and their symbols

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Machine Code in Hex</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcf status,RP0</td>
<td>1283</td>
<td>a</td>
</tr>
<tr>
<td>bcf status,RP1</td>
<td>1303</td>
<td>b</td>
</tr>
<tr>
<td>bsf status,RP0</td>
<td>1683</td>
<td>c</td>
</tr>
<tr>
<td>bsf status,RP1</td>
<td>1703</td>
<td>d</td>
</tr>
</tbody>
</table>

The Active Memory Bank is a discrete function \[169\] of Previously Activated Memory Bank (PAMB) and bank switching instruction. Let the finite sets

\[ B = \{B0, B1, B2, B3\} \]

represent the symbols of PAMB states and

\[ I = \{a, b, c, d\} \]

represent the symbols of bank switching instructions respectively.

\[ \partial : B \times I \rightarrow B \]

is a mapping of \( B \times I \rightarrow B \) which denotes the next-state function.

Then \( \Delta \), a \((2^r \times 2^r)\) relation matrix, can be obtained by first constructing a table whose columns are preceded by a column consisting of successive elements of \( B \) and whose rows are headed by a row consisting of the successive elements of \( I \) as shown in Table 5.2. The relation matrix \( \Delta \) is obtained as...
\[
\Delta = \begin{bmatrix}
B0 & B0 & B1 & B2 \\
B0 & B1 & B1 & B3 \\
B2 & B0 & B3 & B2 \\
B2 & B1 & B3 & B3 \\
\end{bmatrix}
\]

Previously Activated Memory Bank | Active Memory Bank with Bank Switching Instructions
--- | ---
B0 | B0 B0 B1 B2
B1 | B0 B1 B1 B3
B2 | B2 B0 B3 B2
B3 | B2 B1 B3 B3

Fig. 5.1 State transition diagram showing the bank switching scheme.

Elements of \( \Delta \) represent the AMB for each mapping of \( B \times I \rightarrow B \). A state transition diagram representing the data memory bank switching with the execution of each bank switching instruction to the corresponding AMB is shown in Fig 5.1.

The nodes represent the PAMB states. The occurrence of a loop on each state in the state transition diagram corresponds to an unnecessary bank switching or a redundant bank switching instruction, which can be identified and eliminated by...
incorporating the necessary algorithm. Eliminating such instructions from a machine code sequence results in a code optimized for space and speed metric.

For the target processor considered, most of the time the compiler/macros/user places two instructions to select the required data memory bank. They are

\[(\text{bcf status, RP0} \lor \text{bsf status, RP0}) \land (\text{bcf status, RP1} \lor \text{bsf status, RP1})\].

i.e. \((a \lor c) \land (b \lor d)\)

To select bank B3 (i.e. \(\text{status}^{<\text{RP1:RP0}>} = b'11'\)), the two probable instructions are \(\text{bsf status, RP0} (c)\) and \(\text{bcf status, RP1} (d)\). With a PAMB state B2 (i.e. \(\text{status}^{<\text{RP1:RP0}>} = b'10'\)), only instruction \(c\) is needed and instruction \(d\) is redundant since; \(\partial (B2, c) = B3\) which is evident from the matrix \(\Delta\) as well as the state transition diagram. This redundancy corresponds to a loop in the state transition diagram which the algorithm identifies and that instruction is eliminated. Even though the order of the instructions is reversed, the algorithm identifies the first instruction as redundant since the state transition is to B2 itself or \(\partial (B2, d) = B2\). The other situation is selecting a bank which is already the active bank. For selecting the bank say B1, the two probable instructions are \(\text{bsf status, RP0} (c)\) and \(\text{bcf status RP1} (b)\). With a PAMB state B1;

\(\partial (B1, c) = B1\) and \(\partial (B1, b) = B1\)

The algorithm identifies both the bank selecting instructions which are redundant as evident from the matrix \(\Delta\) as well as the state transition diagram and can be removed. The relation matrix is independent of the application program, but it depends on the architectural features of the target processor. If \(P\) is the number of memory banks, so that \(2^r = P\), then the number of rows of the relation matrix will be \(P\). If the bank switching is done with a data transfer instruction then the number

\text{Cochin University of Science and Technology}
of columns of the relation matrix also will be $P$ and in case the bank switching is done with individual bit set/reset instructions the number of columns will be $2r$. With an identical approach the redundant program memory page switching instructions (which are $bcf\ pclath, 3$; $bcf\ pclath, 4$; $bsf\ pclath, 3$; $bsf\ pclath, 4$) to switch the four pages in the program memory also can be eliminated. Hence almost all possible redundancy introduced in the program with respect to the bank selection instructions are identified and can be eliminated.

### 5.2.2 Realization

A novel algorithm to assist software developers for eliminating redundant data and program memory bank selection instructions has been developed; this also helps developing efficient embedded software utilizing static analysis of machine codes.

The relation matrix $\Delta$, formed for the AMB state transition, corresponding to each bank switching instruction in the machine code sequence of an application program, is used for eliminating the redundancy. For the implementation of the code optimization the machine code is read from the Intel hex file and stored in an array. Checking of redundant bank switching instructions should follow the sequence of instructions executed by the processor which correspond to a path in the program graph. In order to get the correct sequencing of instructions, the program (machine code) is partitioned into blocks of instructions by disconnecting from every merge node (a node in the program graph with more than one incoming arc) all of its incoming arcs [167]. Hence the program graph is partitioned into a collection of disconnected subgraphs where each subgraph corresponds to a set of instructions or subprogram. Since each subgraph is a tree, they have only one entry point (root node) and there is a unique path, and hence a unique sequence of instructions, from entry point to each of the exit points. Now the CFG can be
constructed where each subgraph of the program graph is represented as a single node and the arcs represent valid control flow between subgraphs \([38, 150]\). From the CFG the set of elementary paths in a subprogram are identified in the same way as described in section 4.1.3 of chapter 4.

The flow chart given in Fig. 5.2 explains the algorithm to detect the redundant bank switching codes. N represents the total number of CFG nodes. M represents the number of paths in the \(n^{th}\) subgraph. L represents the number of machine codes (nodes)

![Flowchart](image)

*Fig. 5.2 Flowchart explains the identification and pruning of redundant MBSWC in the machine code sequence of a program.*
in the \( j^{th} \) path of the \( n^{th} \) subgraph. \( C(n,j,k) \) represents the \( k^{th} \) machine code in the \( j^{th} \) path of the \( n^{th} \) subprogram. Since the bank \( B0 \) is the default active bank on \textit{reset}, \( B0 \) is assigned at start to the PAMB of each path of the \( 1^{st} \) CFG node. For each memory bank Switching Code (MBSWC) in a valid path the AMB state is obtained from the matrix \( \Delta \). A redundant MBSWC is located when AMB = PAMB. The AMB associated with the \( v_{if} \), which is given an Exit ID is assigned to the Exit Active Memory Bank (EAMB) which becomes the starting PAMB of each path of the next CFG node. For the analysis of a subprogram a linear scan is sufficient. Analysis of a subprogram takes care of the redundancy of the memory bank switching instructions associated with the intraprocedural routines in an application program.

Analysis of the last CFG node is followed by the processing of the merge nodes. The initial node of a subprogram is a merge node where there is more than one incoming edge. So the first MBSWC in each path of a subprogram cannot be eliminated just by observing it to be redundant from the EAMB state of its previous subprogram. Hence the AMB associated with each of the incoming arc at the merge node are to be considered. Each of these incoming edges corresponds to a source node which is nothing but a leaf node, and hence an AMB is associated with it. Hence the AMB at the entry node of a subprogram need not be unique. A typical case is that of a function call from different call sites. A call site corresponds to a node which contains an instruction implementing a function call. All the call sites need not have the same AMB state. A loop in a program is another case. Therefore, for all the CFG nodes, even though the first (pair of) bank switching instruction in any path is found to be redundant, they are not reported till the interprocedural analysis is over and the redundancy is confirmed. This is the first step done towards the suppression of false warnings. Hence the AMB associated with the first
instruction in a subprogram is taken as the union of AMBs of its incoming arcs. During processing of the merge nodes, if it is found that all the incoming edges to a merge node are having the same AMB associated with them, then the redundancy marked for the first (pair of) bank selection instruction in that node or in any path of that subprogram is considered to be redundant and can be eliminated. When it is not so a decision is made by considering the AMB combinations of the incoming edges as follows.

If B0 and B2 only then the instruction $bcf\ status,RP0$ is redundant
If B0 and B1 only then the instruction $bcf\ status,RP1$ is redundant
If B1 and B3 only then the instruction $bsf\ status,RP0$ is redundant
If B2 and B3 only then the instruction $bsf\ status,RP1$ is redundant

As a second step towards suppression of the false warnings, the algorithm considers all the transparent nodes which do not contain any bank switching instructions. If the active memory bank associated with the incoming edges of a transparent node are not equal then the leaf nodes of this subprogram are assigned with the combination of incoming edges’ AMBs. Again within a CFG node if any of the paths is without a bank switching instruction its leaf node is treated similarly. When the initially detected redundant codes are pruned the AMBs associated with all the incoming edges to the entry node are taken care of. Hence the algorithm takes care of the redundant data memory bank selection instructions associated with all the loops and interprocedural routines of the application program.
Chapter 5

5.2.3 Tool Evaluation

The code analyzer developed for the detection of redundant bank switching instructions in an application program is realized in software using Visual Basic. The tool is evaluated using programs typically run on microcontrollers. For programs developed in assembler the necessary pair of MBSW instructions were inserted prior to all bank sensitive instructions and tested. Fig. 5.3 shows the CFG of a sample program used for the analysis which has got six nodes ‘n1’ through ‘n6’. Each bank sensitive instruction in the program is preceded by an appropriate pair of MBSWC. Each node in a program graph is assigned with an address and its associated machine code. The hex values of the addresses corresponding to the pair of MBSWC are shown encircled and the resulting active memory banks such as B0, B1 etc. are also shown. B0 results with the instructions $a \& b$, B1 results with the instructions $c \& b$, B2 results with the instructions $a \& d$ and B3 results with the instructions $c \& d$. The AMB associated with the incoming edges of ‘n1’ through ‘n6’ are also shown. With the MC_CODE ANALYZER v1.02 only the inraprocedural analysis has been done. Here the analysis of each CFG node considers the EAMB associated with the exit node of its predecessor only. Results of the analysis for the sample program above with MC_CODE ANALYZER v1.02 are given in Fig. 5.4 which show all the redundant bank switching codes associated with all the inraprocedural routines along with their address locations. The source node address, the machine code at this address location and the destination node address of the program graph are also displayed in the screenshot. The addresses of these redundant codes are single starred or double starred in the Fig. 5.3, the latter being the first (pair of) MBSWC in the subprogram.
Fig. 5.3 CFG of the sample program for the analysis.

With the MC_CODE ANALYZER v3.00 the inraprocedural, interprocedural and transparent node analysis has been conducted. The first (pair of) redundant bank switching code/codes in any of the subprogram (the nodes which are marked **), already identified with the MC_CODE ANALYZER v1.02 are pruned with this analysis to avoid any false warnings. Here the first/first pair of bank switching code/codes of each subprogram which were found redundant by the previous analysis are reported to the programmer only if they are found redundant with the interprocedural analysis too.
Fig. 5.4 Screen shot of the developed MC_CODE ANALYZER v1.02 for the sample program.

Screenshot explaining the results of this analysis for the same sample program with the MC_CODE ANALYZER v3.00 are given in Fig. 5.5. The machine codes at addresses 8h, 23h, 2Dh, 2Eh, 29h and 12h are pruned as follows. The redundancy reported in the first analysis for the code at location 23h is eliminated in the second analysis since ‘n2’ is a transparent node and therefore the leaf nodes of this subprogram are assigned with the combination of incoming edges’ AMBs. Then the

Figure 5.5 Screen shot of the developed MC_CODE ANALYZER v3.00 for the sample program.
incoming edges of node ‘n3’ can have active memory banks either B1 or B2. With a PAMB of B1, the instruction ‘c’ is redundant since ∂ (B1, c) = B1, but with a PAMB of B2, the instruction ‘c’ is not redundant as evident from the state diagram; hence the code at location 23h is eliminated from the result. Similarly for the node ‘n4’, codes at 2Dh and 2Eh are reported redundant in the first analysis since EAMB of the exit node of ‘n3’ is B3. But with the second analysis only code at location 2Eh is reported and 2Dh is eliminated since the incoming edges AMB combination is B3 and B2 only. With a PAMB of B3 or B2 the instruction ‘c’ is not redundant, but the instruction ‘d’ is redundant since ∂ (B3, d) = B3 and ∂ (B2, d) = B2. For the node ‘n5’, since the incoming edges are having the same AMB B3, code at location 29h is reported in both the analysis which is clear from the relation matrix. For the machine codes at addresses 8h and 12h no change since node ‘n1’ is having only one incoming edge and for ‘n6’ the incoming edges are having the same AMB which is B1. The codes which are found redundant in the first analysis but eliminated later lead to the suppression of false warnings.

Results of the analysis done on machine codes generated with different compilers as well as assembler are given in Table 5.3. HI-TECH Software is a world-class provider of development tools for embedded systems and is the number one third party vendor of compilers for Microchip Technology Inc. For a program module ‘delay_time_rout’ downloaded from [183] and compiled using HI-TECH C PRO, the algorithm detected six redundant codes. Sample programs available with HI-TECH C PRO compiler are tested and the results are given as sl. no. 2 to 6. These programs are compiled with the optimization enabled; hence the results prove that the tool developed is superior to the compiler. Serial numbers 7 to 12 gives the results of the analysis on programs available with PROTEUS VSM design tool. The results of the analysis for an ADC program compiled using HI-TECH C PRO, mikroC and also the same program developed in assembler are also included.
(sl. No. 13 to 15) to test the independence of the tool developed on the compiler. Serial number 12 is a program compiled with PICBASIC.

For a traffic signaling program developed in assembler with each bank sensitive instruction preceded by a pair of necessary bank switching instructions, the algorithm detected all the redundant bank switching codes and this is presented as sl. no. 16 of the table.

**Table 5.3 Results of the analysis**

<table>
<thead>
<tr>
<th>Sl.No.</th>
<th>Program</th>
<th>Code size</th>
<th>MBSWC present</th>
<th>Redundant MBSWC detected.</th>
<th>% Saving in Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>delay_time_rout</td>
<td>223</td>
<td>6</td>
<td>6</td>
<td>2.7</td>
</tr>
<tr>
<td>2</td>
<td>Lcd_demo</td>
<td>176</td>
<td>12</td>
<td>10</td>
<td>5.7</td>
</tr>
<tr>
<td>3</td>
<td>Timer_demo</td>
<td>49</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Intr_demo</td>
<td>44</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Pic_demo</td>
<td>700</td>
<td>16</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Bootloader</td>
<td>225</td>
<td>19</td>
<td>1</td>
<td>0.44</td>
</tr>
<tr>
<td>7</td>
<td>ADC</td>
<td>63</td>
<td>7</td>
<td>1</td>
<td>1.6</td>
</tr>
<tr>
<td>8</td>
<td>Doorbell</td>
<td>643</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>PICCLOCK</td>
<td>292</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>RS232LCD</td>
<td>102</td>
<td>5</td>
<td>1</td>
<td>0.98</td>
</tr>
<tr>
<td>11</td>
<td>GEPE456</td>
<td>1403</td>
<td>10</td>
<td>2</td>
<td>0.14</td>
</tr>
<tr>
<td>12</td>
<td>GLCD_T~1</td>
<td>1044</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>HiTecC_ADC</td>
<td>84</td>
<td>18</td>
<td>8</td>
<td>9.5</td>
</tr>
<tr>
<td>14</td>
<td>mikroC_ADC</td>
<td>56</td>
<td>10</td>
<td>2</td>
<td>3.6</td>
</tr>
<tr>
<td>15</td>
<td>ASM_ADC</td>
<td>81</td>
<td>9</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td>16</td>
<td>Traffic_signalling</td>
<td>48</td>
<td>16</td>
<td>7</td>
<td>14.6</td>
</tr>
</tbody>
</table>
The tool developed counts the total number of bank switching codes originally present in the program as well as the number of redundant bank switching codes. Using the simulation log in PROTEUS VSM the number of program words in each program is also found. Hence the percentage saving in code size is computed and presented in the table. A corresponding saving in run time can also be computed. Including the profile data can give the execution frequency of each node so that the better approximation of the runtime saving can be computed which will be conducted as a future work. The same application program can be compiled using different compilers and the machine codes from each of these compilers can be tested with the tool developed. Now by counting the number of redundant codes reported in each of these cases, an evaluation of the different compiler’s optimization capability in this regard is possible.

5.3 Optimization Technique

This work considers a compiler strategy of allocating \( z \) number of data variables in an application program to \( P \) number of data memory banks in the target processor, with the objective to deliver the machine code with minimum number of bank switching codes. Since the number of bank switching codes cannot be expressed as a linear function of the data variable, an ILP solver is not applied in our approach.

5.3.1 Variable Partitioning

For a banked memory with \( P \) banks each of equal size, \( z \) number of data variables can be assigned to the available banks in \( P^z \) possible ways provided \( z \leq \) bank size. If the banks are of unequal size the case reduces to the same, provided \( z \leq \) smallest size of the banks. When \( z > \) bank size the data mapping can be considered as the problem of finding all possible \( z \times P \) integer matrices [184, 185, 186] \( A \) with
\( a_{ij} \in \{0,1\} \), that satisfies the given constraints on its rows and columns. The cardinality of the set of such data mapping matrices depends on these constraints. The first constraint is that, every data variable is considered as a single unit and is allocated to only one memory bank:

\[
(\forall i): 1 \leq i \leq z: \quad \sum_{j=1}^{p} a_{ij} = 1
\]

Second constraint is that the sum of the sizes of all variables allocated to a particular memory bank \( B_j \) must not exceed the size of that memory bank \( m(B_j) \):

\[
(\forall j): 1 \leq j \leq p: \quad \sum_{i=1}^{z} a_{ij} \leq m(B_j)
\]

Third constraint is that \( z \) must not exceed the sum of sizes of all banks:

\[
z \leq \sum_{j=1}^{p} m(B_j)
\]

The polynomial-time solvability of this case has been proved [187]. Indeed, more constraints may decrease the runtime by decreasing the space of feasible solutions. For example six variables can be allocated to two memory banks in \( 2^6 \) (64) ways provided each bank size \( \geq 6 \). But with the constraint of bank size=3, the feasible number of data mapping matrices (cardinality of the set of matrices) reduce to 20.

The set of data mapping matrices can be obtained with a depth first search algorithm. Adding one more row and column to an \( z \times p \) matrix subject to the following constraints gives the matrices.

\[
(\forall j) = 1 \text{ to } p
\]

\[
a_{(z+1,j)} = m(B_j)
\]

\[
(\forall i): 1 \leq i \leq z
\]

\[
a_{i,(p+1)} = 1
\]
So without any HLL directives the compiler can try all possible combination of data variable allocation. Prior to all bank sensitive instructions the compiler can insert as many bank switching instructions as needed. The resulting machine codes are tested with the algorithm developed to detect the redundant bank switching codes. The program that results in the maximum number of redundant bank switching code corresponds to the minimum number of bank switching codes in the program and can be selected as the optimum data allocation scheme for a given application.

5.3.2 Optimum Memory Bank Allocation

The compiler designers and MCU manufacturers suggest certain tips for speed optimization. In processors using banked memory architecture, the bank switching instructions can be reduced by properly selecting the order in which the variables are initialized at the start up of a program. They also suggest using variables in same bank in arithmetic expressions, to avoid bank switching. Another suggestion is that the variables accessed most often in the program can be allocated to the memory spaces that are cheapest to access. A careful assignment of program variables to registers is the most important optimization of a compiler for RISC.

For a given application program, the data variables can be allocated to the available memory banks by considering all possible permutation of memory banks and combination of data as represented by the set of data mapping matrices explained in section 5.3.1. In each of these programs corresponding to the various data allocation schemes, the compiler puts the necessary MBSWC prior to all bank sensitive instructions without applying any algorithm for the minimal placement of bank switching codes. This results in a unique Intel hex file output corresponding to each of these programs. These files become the input to the machine code analyzer developed which detects the number of redundant bank switching
instructions present. The more the reported number of redundant codes, optimum is the memory bank assignment. So the number of eliminated code is compared each time and the most efficient code is selected.

We now discuss an example to illustrate how the approach described above works in practice. For the target processor under study there are four memory banks. So \( z \) number of data variables can be assigned to the 4 memory banks in \( 4^z \) ways when \( z \leq \text{bank size} \). For testing this tool for optimum data allocation a traffic signaling program having three data variables is considered. The three data variables are named \( S, T \) and \( U \) and are assigned to the four banks in \( 4^3 \) (64) ways resulting in 64 programs each with a unique data allocation scheme. In these programs the three data variables \( S, T \) and \( U \) can be placed in the four memory banks available, first by placing the entire three in one bank, second by placing the three data in any of the two banks and third in any of the three banks out of the four available. Considering the permutation of memory banks and the combination of data in each of the above cases, programs one to four are with all the three data allocated to any one of the banks so that there are \( 4P_1 = 4 \) ways of data allocation; programs five to forty are selecting any of the two banks at a time, so that for the three variables there are \( 4P_2 \times 3C_2 = 36 \) ways of allocating the data and programs forty one to sixty four are selecting any of the three banks for the three variables in \( 4P_3 \times 3C_3 = 24 \) ways.

For the target processor since the special function registers are implemented in data memory bank, accessing these registers must ensure the proper bank switching. The \( SFRs \) used in the program considered are \( triSB \) and \( portB \). Each bank sensitive instruction in the program is made preceded by a pair of necessary bank switching instructions. There are eight number of bank sensitive instructions so that the number of bank switching instructions altogether in the program is
sixteen. Fig. 5.6 shows the CFG of this program in which the data variables $S$ and $U$ are allocated to $B3$ and $T$ is allocated to $B0$. This results in the worst case allocation where the number of redundant codes identified is two which are starred in the figure.

Fig. 5.6 CFG of the sample program with the worst case data allocation scheme.
Fig. 5.7 shows the bar graph for the number of redundant bank switching instructions reported in the 64 data allocation schemes of the program considered. The first four cases are with all the three variables S, T and U in one bank. Programs five to forty are with the data variables S, T and U assigned to any of the two memory banks. Similarly programs forty one to sixty four are with data assigned to any of the three banks out of the available four. The worst case reported is when S, in B3, T, in B0 and U also in B3 (sl.no.19 in bar graph) where out of the sixteen bank switching instructions only two are redundant. The optimum data assignment is with S, T and U assigned to B0 (sl.no.1 in bar graph) where fourteen out of the sixteen are redundant. The total number of bank switching instruction depends also on the use of special function registers in a program which are implemented in these memory banks. Data allocation schemes 5, 6, 34 and 36 in the bar graph give the indication that there is a tendency for optimum data assignment even though all the data are not in B0. Distributing the data allocation to two banks in these cases is more efficient than allocating all the data to B2 or B3.

From the results the conclusion obtained is that a compiler can insert the required bank switching instructions prior to any bank sensitive instruction without any complicated analysis on the source code. The compiler can attempt all possible data allocation schemes for a given application program. Using this tool it can determine all the bank switching code to be eliminated along with the optimum data allocation to the available banks. When the reported redundant codes are eliminated, the program runs successfully.
5.4 Redundant I/O Port Configuration

The I/O ports available in PIC16F87X series of microcontrollers have tris registers associated with them. These register configurations control the direction of the I/O pins even when they are being used as analog inputs. The user must ensure the bits in these registers are maintained set/reset appropriate to the situation. When a lengthy program is developed configuring these pins unnecessarily may result in code redundancy. Compilers also can introduce such errors. Even though the use of macros [18] simplifies the program development, when they are used without care they can also introduce such errors.

![Bar chart showing the number of redundant bank switching instructions reported in the 64 data allocation schemes of the program.](chart.png)
These redundant codes can be identified and reported through the static analysis of machine code developed. If a processor has got ‘M’ number of I/O port pins then ‘M’ number of port_pin_flags are used in the analysis. A flag is set if the corresponding I/O port bit is configured as input in the program code sequence and reset otherwise. Initialization of each of these flags is according to the power on reset condition of that processor. Using the relation matrix $\Delta$ described in section 5.2.1 of this chapter and the bank switching codes in the given program the use of all registers as well as ports in the executables can be identified. A linear scan through the successive nodes in the CFG of an application program can reveal the existence of a code, the execution of which results in the same pin configuration for a port as that of the previous one. The resulting codes can be pruned with the interprocedural analysis where the statuses of the port_pin_flags of all the incoming edges of a merge node are taken into account.

5.5 Redundant ADC Channel Selection

Programs spend most of their time going around loops. Loops therefore are the most promising sources to attempt speedups in a program, and it behoves an optimizing compiler to generate particularly efficient code for loops. In embedded systems a real time program will be executed in an infinite loop. Though the control flow trace may indicate a number of repeated paths, the possible execution paths are fixed. Analysis is to be conducted through the possible execution paths only. The paths taken by the program can be obtained from the possible combinations of control flow edges.

Some of the instructions inside a loop will be executed unnecessarily. A typical case is that of selecting the $ADC$ channel which should be done only once unless a new channel is to be selected. So channel selection instruction is to be outside the loop. Evaluation of rule (4) of Table 4.4(c) included in section 4.4.1 of
chapter 4 achieves this result. Similar instances can be identified and the programmer can be instructed to revise the program for optimization.

### 5.6 Software Realization

Algorithms for the detection of redundant codes in an application program are realized in software using Visual Basic. The block diagram given in Fig. 5.8 explains the various steps in the static analysis of the machine code developed. The selected Intel Hex file of the application program is read and an array of machine codes are stored with their address values. For generating the program graph a
control flow array is formed which identifies the edges in the program graph with
the source and destination addresses and the corresponding codes. The CFG is
constructed by eliminating the incoming edges of the merge nodes, thereby
identifying the entry nodes, exit nodes and leaf nodes. Then the number of paths in
each CFG node and the codes in each path are identified. Now the algorithm is
applied to identify the redundant codes which are reported to the developer.

5.7 Summary.

This chapter describes the optimization techniques and their realization, for
the static machine code analyzer. This tool helps to eliminate the redundant codes
in bank switching instructions in partitioned memory architectures, in port
configuration as well as in ADC channel selection. Formation of a relation matrix
with PAMB and bank switching instructions is illustrated. A state transition
diagram representing the data memory bank switching with the execution of each
bank switching instruction to the corresponding AMB state is presented. A data
allocation technique to minimize the bank switching is described. Various steps in
the software realization of the machine code analyzer are discussed. Elimination of
redundant codes in the intraprocedural sequence, loops and interprocedural routines
in the application program are illustrated. A prototype based on PIC16F87X
microcontrollers is described and the experimental results obtained with sample
programs are also presented.