A Compiler Integrated Assistance for Optimum Data Allocation in Banked Memory Embedded Processors

Mariamma Chacko[1] and K. Poulose Jacob[2]

[1]Department of Ship Technology, [2]Department of Computer Science,
Cochin University of Science and Technology, Cochin, India
mariamma.cusat@gmail.com, kpj@cusat.ac.in

Abstract

Bank switching in embedded processors having partitioned memory architecture results in code size as well as run time overhead. An algorithm and its application to assist the compiler in eliminating the redundant bank switching codes introduced and deciding the optimum data allocation to banked memory is presented in this work. A relation matrix formed for the memory bank state transition corresponding to each bank selection instruction is used for the detection of redundant codes. Data allocation to memory is done by considering all possible permutation of memory banks and combination of data. The compiler output corresponding to each data mapping scheme is subjected to a static machine code analysis which identifies the one with minimum number of bank switching codes. Even though the method is compiler independent, the algorithm utilizes certain architectural features of the target processor. A prototype based on PIC 16F87X microcontrollers is described. This method scales well into larger number of memory blocks and other architectures so that high performance compilers can integrate this technique for efficient code generation. The technique is illustrated with an example.

Keywords: *Banked Memory; Optimization; Data Allocation; Compilers; Embedded Systems*

1. Introduction

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. It is common to select a microprocessor/microcontroller based on its performance and to rely on the compiler to deliver this performance. This is particularly true of high-performance RISC devices. Optimization is an important task when developing resource intensive applications. Embedded software must meet conflicting requirements such as high reliability, operation on resource-constrained platform and rapid development. Due to strict timing constraints owing to real time concerns, the code optimization problem is more complex than for general purpose systems. An efficient compiler can provide compact code, without having to learn the intricacies of the device architecture. This makes these devices more accessible to engineers with limited programming experience who are increasingly using MCUs in their product designs [1], [2].

The integration of processor cores and memory in the same chip effects a reduction in the chip count, leading to cost effective solutions. Typical examples of optional memory modules integrated with the processor on the same chip are: Instruction Cache,

Data Cache, and on-chip SRAM. Efficient utilization of on-chip memory space is extremely important in modern embedded system applications based on microprocessor cores. Memory banking and memory paging are common techniques, which increase the size of data and code memory without extending the address bus. Many MCUs have banked memories that cannot be addressed simultaneously. For example, Freescale 68HC11 8-bit microcontrollers [3] allow multiple 64KB memory banks to be accessed by their 16-bit address registers with only one bank being active at a time. Other examples include Intel 8051 processor family, MOS technology 6502 series microcontrollers and most of the PIC microcontrollers [4]. Switching between the memory banks 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 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.

The related literature for minimal placement of bank switching instructions is motivated by objectives, such as less runtime, low power, small code size, or a combination of these parameters. Scholz et.al. in [5] assume the variables have already been assigned to memory banks and presents a novel optimization technique that minimizes the overhead of bank switching through cost effective placement of bank selection instruction. They formulate the placement of bank selection instructions as a discrete optimization problem that is mapped to a partitioned Boolean quadratic programming (PBQP) problem. Allocating variables to shared memory is useful to eliminate bank selection instructions. Mengting et al. in [6] presents a dynamic programming algorithm to generate the optimal assignment of variables in the shared memory to minimize bank selection instructions. Li *et al*. [7] prove that it is NP-Hard to insert the minimum number of bank selection instructions if all the variables are pre-assigned to memory banks. So they introduce a 2-approximation algorithm using a rounding method. They consider the case when there are some nodes that do not access any memory bank and design a dynamic programming method to compute the optimal insertion strategy when the Control Flow Graph (CFG) is a tree. An algorithm is presented in [8] devoted to reduce the number of page selection instructions with careful allocation of functions into pages.

The work presented in [9] aims to utilize variable partitioning techniques to minimize the size and time overhead introduced by bank switching. Current practice typically leaves it to the programmer to partition the data among different memory banks. Whether programming is done in assembly language or in a high-level language, the programmer has to provide data manually by using assembler directives or compiler pragmatics. Compiler methods are preferable to programmer directives as they do not require programmer effort; are portable across different systems; and are likely to make better decisions, especially for large, complex programs [10]. Most of the current variable partitioning techniques aim at achieving the maximum instruction level parallelism for processors equipped with dual data memory banks accessible in parallel [11],[12],13]. But these techniques will not benefit the bank switching optimization because no parallel bank accessing is allowed in this architecture. The problem of partitioning data into *scratch pad* SRAM and cache with the objective of maximizing performance has been addressed in [14]. A compiler method for automatically allocating program data among the heterogeneous memory units in embedded processors without caches resulting in reduced runtime is presented in [10]. Allocation techniques to statically allocate data to the scratchpad for energy saving were introduced in [15] and [16] whereas [17] presented a dynamic approach. These techniques are all based on the frequency of data

accesses and make no attempt to reduce the code size. All the previous works mentioned above are analyzing the source programs for optimum data partitioning.

It is characteristic for embedded system programmers to inter-leave fragments of assembly code in high level language, to enable direct access to the device's hardware. Performing static analysis on the high level representation of the source code requires transforming the embedded assembly code to the high level representation. Static analysis on machine code rather than source code eliminates the requirement of knowledge of the semantics of high level language/assembly language and it is independent of the compiler so that, developers are free to change compilers or compiler versions [18]. Current compilers provide limited support to generate bank switching code optimally.

The work presented here is targeted towards the optimum allocation of data variables to on-chip memory banks that cannot be accessed simultaneously. A compiler strategy that can automatically determine the optimum data partition among the memory banks is presented. 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. A static analysis of machine codes which is in Intel hex file format is used to give the feedback to the compiler to decide the optimum allocation. To the best of the authors' knowledge only [9] 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 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 [19] microcontroller are presented.

2. 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's 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 [20]. Smaller address buses result in smaller chip die sizes, 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) [5] 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 popular manufactures of microcontrollers adopt this technique. 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 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.

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 to such an extent that it will not fit in device memory. 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 [2].

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. 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) [2]. 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.

3. 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.

PIC 16F84A have just two banks [20] 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. The data memory space in PIC18F series devices is divided into as many as 16 banks that contain 256 bytes each [21]. 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 Previously Activated Memory Bank (PAMB). ***A 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 **P** is the number of memory banks, so that $2^{\mathbf{r}} = \mathbf{P}$, then the number of bits that decides 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 2**r** number of machine codes for this operation and for each PAMB state there will be **r** number of bank switching instructions, which are redundant.

3.1 Relation Matrix Formulation

The family of Microchip PICmicro MCUs constitutes a RISC-based Harvard architecture with instruction size of 14 bits and an 8-bit wide data bus [4]. 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.

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 Active Memory Bank is a discrete function [22] of Previously Activated Memory Bank (PAMB) and bank switching instruction. Let the finite sets

B = {B0, B1, B2, B3} represents the symbols of PAMB states and

I = {*a*, *b*, *c*, *d*} represents the symbols of bank switching instructions respectively.

∂ is a mapping of B×I →B which denotes the next-state function.

Then Δ, a (2^{r}×2r) 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 1. The relation matrix Δ is obtained as

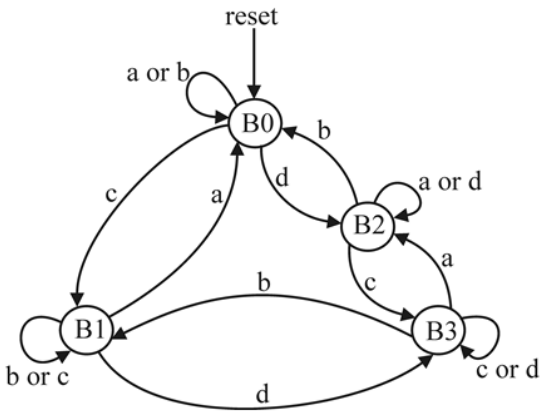
$$\Delta = \begin{bmatrix} B0 & B0 & B1 & B2 \\ B0 & B1 & B1 & B3 \\ B2 & B0 & B3 & B2 \\ B2 & B1 & B3 & B3 \end{bmatrix}$$

Figure 1. State Transition Diagram Showing the Bank Switching Scheme

Elements of Δ represent the AMB for each mapping of B×I→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 Figure 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.

Table 1. Relation Matrix Formation with PAMB and Bank Switching Instructions

Previously Activated Memory Bank	Active Memory Bank with Bank Switching Instructions			
	a	*b*	*c*	*d*
B0	B0	B0	B1	B2
B1	B0	B1	B1	B3
B2	B2	B0	B3	B2
B3	B2	B1	B3	B3



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

(bcf status, RP0 ∨ bsf status, RP0) ∧ (bcf status, RP1 ∨ bsf status, RP1).

i.e. $(a \vee c) \wedge (b \vee d)$

To select bank B3 (i.e. *status*<RP1:RP0> = b‘11’), the two probable instructions are *bsf status, RP0* (*c*) and *bsf status, RP1* (*d*). With a PAMB state B2 (i.e. *status*<RP1:RP0> = b‘10’), only instruction *c* is needed and instruction *d* is redundant since;

∂ (B2, c) = B3 which is evident from the matrix Δ 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. The other situation is selecting a bank which is already the active bank. For selecting the bank say B1, the two probable instructions are *bsf status, RP0 (c)* and *bcf status RP1 (b).* With a PAMB state B1;

∂ (B1, *c*) = B1 and ∂ (B1, *b*) = B1

The algorithm identifies both the bank selecting instructions which are redundant as evident from the matrix Δ 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 $2^{\mathbf{r}} = \mathbf{P}$, 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 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 2**r**.

3.2 Realization

For the implementation of the code optimization the machine code is read from the *Intel hex* file and stored in an array. Intel hex file format is widely used in microprocessors and microcontrollers as de-facto standard for representation of code for programming into microelectronic devices. 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 [23]. 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 [24], [25].

From the CFG the set of elementary paths in a subprogram are identified as follows.
Let N represent the total number of CFG nodes.
Each subprogram (subgraph) can be represented as an acyclic digraph $G = \langle V_i, E_i, e_i, V_{if}, P_i \rangle$, where,

$(\forall i) = 1$ to N

$V_i = \{v_{i1}, v_{i2}, \ldots..,v_{ik}\}$ represents the nodes in the i^{th} subgraph.

$E_i = \{x \mid x \in (V_i \times V_i)\}$ represents control flow edges between the nodes in the subprogram.

e_i, is the initial node (merge node) of the i^{th} subgraph .

$V_{if} = \{x \mid x \in V_i \wedge x \text{ is a leaf node}\}$.

$| V_{if} | = M$, represents the number of paths in the i^{th} subgraph. One of the leaf nodes v_{if} is given an Exit ID such that there exists an arc (v_{if}, e_{i+1}) in the program graph. If L represents the number of machine codes (nodes) in the j^{th} path of the i^{th} subgraph, then $(\forall j) = 1$ to M

$P_i = \{x \mid x = \{x_{j1}, x_{j2}, \ldots x_{j(L-1)}, x_{jL}\} \wedge x_{j1} = e_i \wedge x_{jL} \in V_{if} \wedge (\forall q) = 1$ to L $(x_{jq}, x_{j(q+1)}) \in (V_i \times V_i)\}$, represents the set of elementary paths in the i^{th} subprogram. The CFG can be

represented as a digraph G = <V,E> where V = {v_1,v_2,……, v_N} represents the set of subprograms and E = { x | x ∈ (V×V)}.

The flow chart given in Figure 2 explains the algorithm to detect the redundant bank switching codes. Since the bank B0 is the default active bank on *reset*, B0 is assigned at start to the PAMB of each path of the 1st CFG node. For each memory bank Switching Code (MBSWC) in a valid path the AMB state is obtained from the matrix Δ. 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.

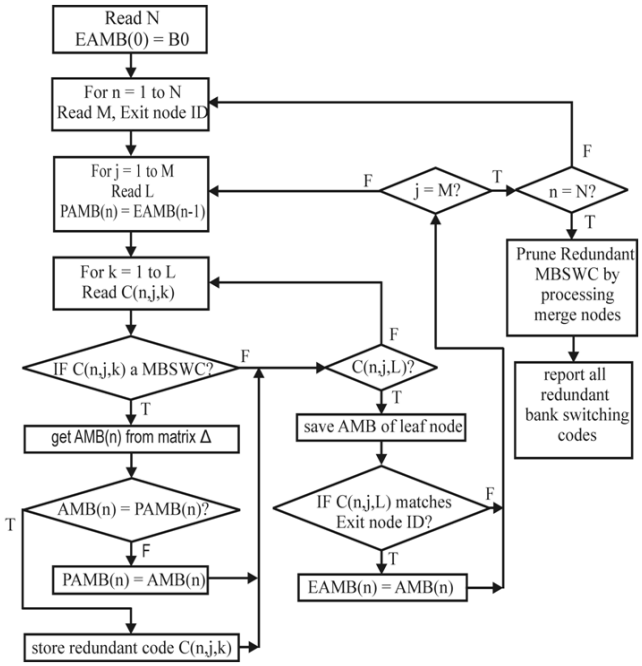


Figure 2. Flowchart Explains the Identification and Pruning of Redundant MBSWC in the Machine Code Sequence of a Program

Completion of the analysis of the last CFG node enables the processing of the merge nodes. The initial node of a subprogram is a merge node where there is more than one incoming arc. 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 arcs corresponds to a source node which is nothing but a leaf node, and hence an active memory bank is associated with it. Hence the active memory bank 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 arcs of a CFG node (arcs to a merge node) are having the same active memory bank 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

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.

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 arcs of a transparent node are not equal then the leaf nodes of this CFG node 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.

3.3 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. Figure 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 $a \wedge b$, B1 results with the instructions $c \wedge b$, B2 results with the instructions $a \wedge d$ and B3 results with the instructions $c \wedge d$. The AMB associated with the incoming arcs 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 Figure 4 which shows the redundant bank switching codes along with their address locations identified by the tool with the intraprocedural analysis. 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 Figure 3, the later being the first (pair of) MBSWC in the CFG node.

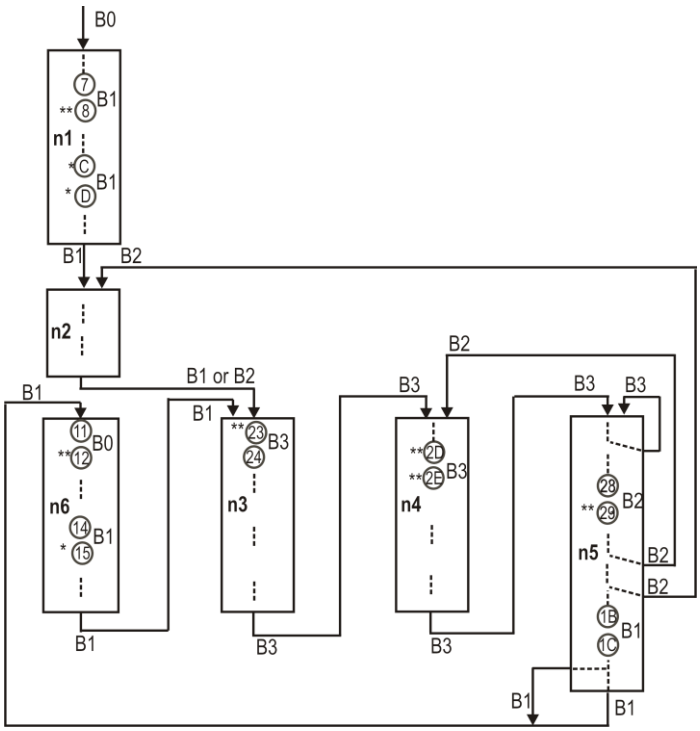


Figure 3. CFG of the Sample Program for the Analysis

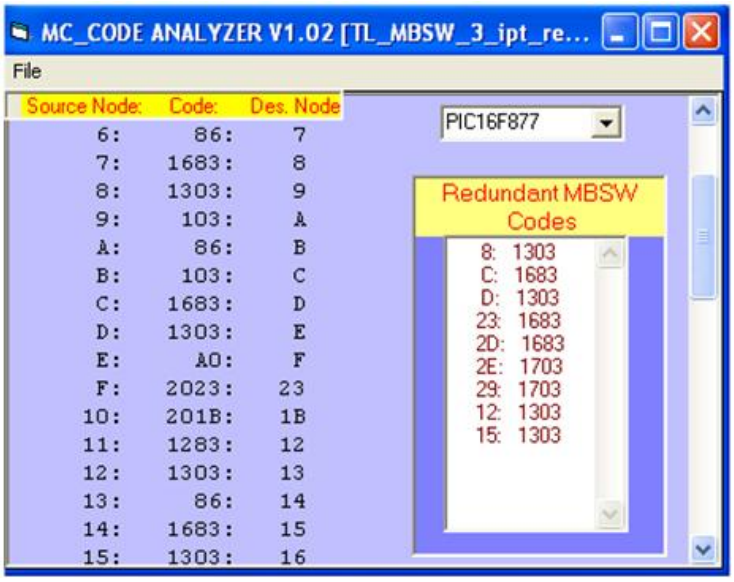


Figure 4. Screen Shot of the Developed MC_CODE ANALYZER v1.02 for the Sample Program

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 CFG node which were found

redundant by the previous analysis are reported to the programmer only if they are found redundant with the interprocedural analysis too.

Screenshot explaining the results of this analysis for the same sample program with the MC_CODE ANALYZER v3.00 are given in Figure 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 subgraph are assigned with the combination of incoming edges' AMBs. Then the 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 there is no change since node '*n1*' is having only one incoming edge and for '*n6*' the AMB associated with the incoming edges are the same. The codes which are found redundant in the first analysis but eliminated later lead to the suppression of false warnings.

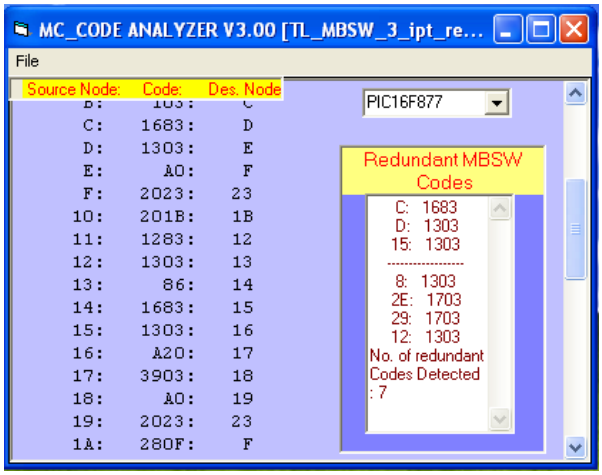


Figure 5.Screen Shot of the Developed MC_CODE ANALYZER v3.00 for the Sample Program

Results of the analysis done on machine codes generated with different compilers as well as assembler are given in Table 2. 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 [26] 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. 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.

Table 2. Results of the Analysis

Sl. No.	Program	Code size	MBSWC present	Redudnt. MBSWC detected.	Saving in code size %
1	delay_time_rout	223	6	6	2.7
2	Lcd_demo	176	12	10	5.7
3	Timer_demo	49	3	0	0
4	Intr_demo	44	2	0	0
5	Pic_demo	700	16	14	2
6	Bootloader	225	19	1	0.44
7	ADC	63	7	1	1.6
8	Doorbell	643	2	0	0
9	PICCLOCK	292	2	0	0
10	RS232LCD	102	5	1	0.98
11	GEPE456	1403	10	2	0.14
12	GLCD_T~1	1044	16	0	0
13	HiTecC_ADC	84	18	8	9.5
14	mikroC_ADC	56	10	2	3.6
15	ASM_ADC	81	9	1	1.2
16	Traffic_signalling	48	16	7	14.6

4. 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.

4.1 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 $\mathbf{P}^{\mathbf{z}}$ possible ways provided **z** ≤ bank size. If the banks are of unequal size the case reduces to the same, provided **z** ≤ smallest size of the banks. When **z** >

bank size the data mapping can be considered as the problem of finding all possible **z** × **P** integer matrices [27] ***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:

$(\forall i)$: $1 \leq i \leq \mathbf{z}$: $\sum_{i=1}^{P} a_{ii} = 1$

Second constraint is that the sum of the sizes of all variables allocated to a particular memory bank $\mathbf{B}_j$ must not exceed the size of that memory bank $m(\mathbf{B}_j)$:

$(\forall j)$: $1 \leq j \leq \mathbf{P}$: $\sum_{i=1}^{z} a_{ii} \leq m(\mathbf{B}_i)$

Third constraint is that **z** must not exceed the sum of sizes of all banks:

$\mathbf{z} \leq \sum_{i=1}^{P} m(\mathbf{B}_i)$

The polynomial-time solvability of this case has been proved [28]. 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 ≥ 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 × **P** matrix subject to the following constraints gives the matrices.

$(\forall j) = 1$ to P

$a_{(\mathbf{z}+1),j} = m(B_j)$

$(\forall i)$: $1 \leq i \leq \mathbf{z}$

$a_{i,(\mathbf{P}+1)} = 1$

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.

4.2 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. 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 4.1. 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 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 $4^{\mathbf{z}}$ ways when $\mathbf{z} \leq$ 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 $3C_2 \times 4P_2 = 36$ ways of allocating the data and programs forty one to sixty four are selecting any of the three banks for the three variables in $3C_3 \times 4P_3 = 24$ 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. *TRISB* and *PORTB* are the *SFRs* used in the program considered. 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.

Figure 6 shows 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 the same bank. Distributing the data allocation to two banks in these cases is more efficient than allocating all the data to B2 or B3.

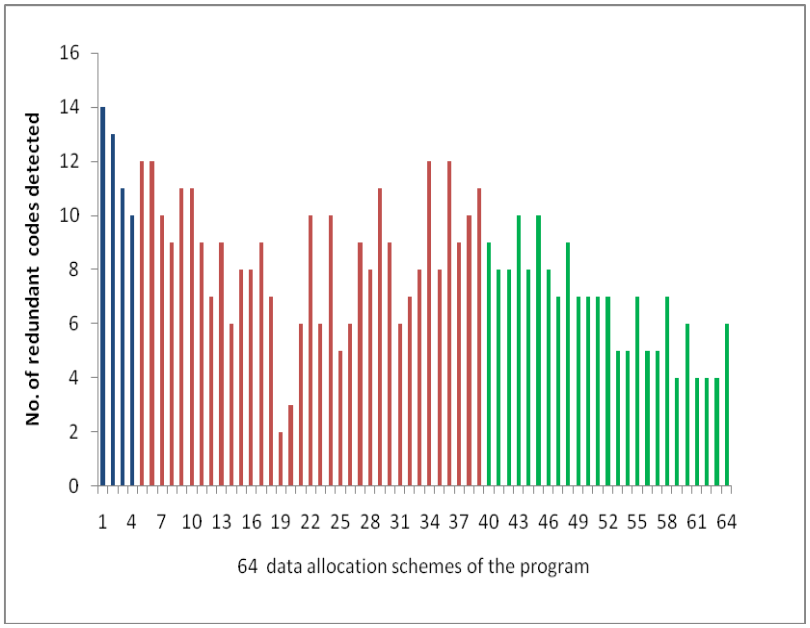


Figure 6. The number of redundant bank switching instructions reported in the 64 data allocation schemes of the program

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. Conclusion

As processor architectures advance, new instructions and enhancements appear. To take advantage of these features, compiler technology must necessarily advance. This paper describes an algorithm to detect the redundant bank switching codes in a program introduced by the compiler/user for partitioned memory architectures with the help of a static machine code analysis. 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. The algorithm detects the redundant memory bank switching instructions inserted by the compiler for each data allocation scheme of the program and helps to identify the program with minimum bank switching codes. With this knowledge the compiler can eliminate all the redundant codes in the optimum program resulting in reduced code size as well as increased execution speed. The compiler introduced redundancy can be identified since the proposed approach is realized through the static analysis of machine code. Since the input file is Intel hex file, the method is independent of the compiler but realization of the technique depends on certain architectural parameters of the target processor.

The technique presented in this work achieves optimization of bank switching instructions without much computational burden by analyzing the machine code with a comparatively simple algorithm. A static analysis of machine code can provide information which can hardly be discovered by traditional simulation or test techniques. In contrast to dynamic

techniques, static techniques can explore abstractions of all possible program behaviors, and thus are not limited by the quality of test cases in order to be effective. With a well defined CFG constructed from the machine codes this algorithm fits well into large problem sizes as well. Redundant data memory bank selection instructions in the intraprocedural sequence, loops and interprocedural routines in the application program can be eliminated. The relation matrix assists the code analyzer in identifying the active memory bank associated with each code in the instruction stream. The suppression of false warning is done by considering the transparent nodes which is a node without any bank switching instructions and also by considering the union of the active memory bank associated with the incoming edges of a CFG node for interprocedural analysis. The example illustrated in this paper proves the feasibility of the approach.

This technique can be used for a processor core based system to select the number of data memory banks and the size of each bank resulting in the optimized design instead of using a single scratchpad RAM.

The execution frequency of an instruction is not considered since it is not a dynamic analysis, nor the run time optimizing attempted. It is very hard to furnish a general solution that handles all the problems associated with the control flow analysis of machine code, but with more information and some architecture specific heuristics the problems become manageable. Instruction reordering without affecting the program within the basic blocks can further improve the bank selection optimization.

References

[1] J. Labrosse et al., Embedded Software: Know It All, Elsevier Inc. **(2008)**

[2] P. Riachi, "HI-TECH OCG Compiler Support for Microchip's PIC10/12/16", Embedded Systems Conference, San Jose, California **(2008)** April 15.

[3] Freescale. *http://www.freescale.com*.

[4] PICmicro mid-range MCU family reference manual, Microchip Technology Inc. **(1997)**

[5] B. Scholz, B. Burgstaller, and J. Xue. "Minimal placement of bank selection instructions for partitioned memory architectures", ACM Transactions on Embedded Computing Systems (TECS), **(2008)** February 7(2), pp. 1-32.

[6] Y. Mengting, W. Guoqing, and Y. Chao, "Optimizing Bank Selection Instructions by Using Shared Memory", the International Conference on Embedded Software Systems (ICESS), pp. 447-45 **(2008)**

[7] M. Li, C. J. Xue, T. Liu and Y. Zhao, "Analysis and Approximation for Bank Selection Instruction Minimization on Partitioned Memory Architecture", ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES'10), Stockholm, Sweden, **(2010)** April 13-15 pp. 1-8.

[8] Q. Li, Y. He, Y. Chen, W. Wu and W. Xu, "A Heuristic Algorithm for Optimizing Page Selection Instructions", proceedings of the IEEE 2nd International Conference on Software Technology and Engineering(ICSTE), **(2010)** pp. v2-143 to v2-148.

[9] L. Tiantian, M. Li, and C. J. Xue, "Joint Variable Partitioning and Bank Selection Instruction Optimization on Embedded Systems with Multiple Memory Banks", Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC 2010), **(2010)** pp. 113-118.

[10] O. Avissar and R. Barua. "An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems", ACM Transactions on Embedded Computing Systems, **(2002)** November Vol.1(1), pp. 6–26.

[11] J. Cho, Y. Paek, and D. Whalley, "Fast Memory Bank Assignment for Fixed-Point Digital Signal Processors", ACM Transactions on Design Automation of Electronic Systems **(2004)** Vol.9(1), pp.52-74.

[12] M. A. R. Saghir, P. Chow, and C. G. Lee, "Exploiting DualData-Memory Banks in Digital Signal Processors", In Proceedings of the SIGPLAN'96 International Conference on Architectural Support for Programming Languages and Operating Systems **(1996)** pp. 234-243.

[13] Q. Zhuge, B. Xiao, and E. H. M. Sha "Exploring Variable Partitioning for Dual Data-memory Bank Processors", In Proceedings of the 34th International Symposium on Microarchitecture **(2001)** pp. 42–55.

[14] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of Scratch-pad memory in embedded processor applications", In European Design and Test Conference, **(1997)** March.

[15] S. Steinke, L.Wehmeyer, B.-S. Lee, and P. Marwedel, "Assigning Program and Data Objects to Scratchpad for Energy Reduction", Design, Automation and Test in Europe (DATE), **(2002)** pp. 409–417.

[16] L. Wehmeyer, U. Helmig and P. Marwedel, "Compiler-optimized usage of partitioned memories", In Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture, Munich, Germany, **(2004)** June 20, pp. 114-120, DOI=10.1145/1054943.1054959.

[17] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic Management of Scratch-Pad Memory Space", In Proceedings of the 2001 ACM Design Automation Conference(DAC), **(2001)** June.

[18] O. Goloubeva, M.S. Rebaudengo, M.S. Reorda and , M. Violante, "Improved software-based processor control-flow errors detection technique", in proceedings of the Reliability and Maintainability Symposium, **(2005)** January, pp. 583–589.

[19] *Data sheet, PIC16F87X*, Microchip Technology Inc.., **(1999)**. Available: http://www.microchip.com.

[20] T. Wilmshurst, Designing Embedded Systems with PIC Microcontrollers-Principles and applications, Newnes, Elsevier, London, UK **(2007)**

[21] *Data Sheet, PIC18F2455/2550/4455/4550*, Microchip Technology Inc. **(2004)**

[22] J. P. Tremblay and R. Manohar, Discrete Mathematical Structures with Applications to Computer Science, McGraw-Hill, Singapore **(1987)**

[23] T. Sridhar and S.M. Thatte. "Concurrent checking of program flow in VLSI processors", In Proceedings of the 12th Int. Test Conf. **(1982)** November, pp. 191-199.

[24] A.V. Aho and J.D. Ullman. Principles of Compiler Design, Addison-Wesley/Narosa, New Delhi, India **(1985)**

[25] M. A. Schuette and J. P. Shen. "Processor control flow monitoring using signature instruction streams", IEEE Trans.on Computers **(1987)** March, vol. C-36, No 3, pp. 264-275.

[26] http://www.microchip.com.

[27] Ryser, H. J. "Combinatorial Properties of Matrices of Zeros and Ones." Canad. J. Math., **(1957)** Vol.9, pp. 371-377.

[28] S. E. Wright, "Integer matrices with constraints on leading partial row and column sums", Elsevier journal of Discrete Applied Mathematics **(2010)** Vol. 158, pp. 1838-1847, doi:10.1016/j.dam.2010.06.010.

Authors

Ms. Mariamma Chacko was born in 1961 at Changanacherry, India. She received her Bachelor's Degree in Electrical Engineering from University of Kerala in 1985, Master's Degree in Electronics from Cochin University of Science and Technology in 1987.

She has been working as Reader in the Department of Ship Technology at Cochin University of Science and Technology since 1990. From 1987 to 1990 she was associated with the Department of Electronics, Cochin University of Science and Technology, as a research Associate. Her research interests include validation and optimization of embedded software

Dr. K. Poulose Jacob was born in 1955 at Cochin, India. He received his Batchelor's Degree in Electrical Engineering from University of Kerala in 1976, Masters Degree in Electronics from University of Cochin in 1981 and Ph D in Computer Engineering from Cochin University of Science and Technology in 1991.

Dr. K. Poulose Jacob, Professor of Computer Science at Cochin University of Science and Technology since 1994, is currently Director of the School of Computer Science Studies.

A National Merit Scholar all through, Dr. K.Poulose Jacob joined the Cochin University as a member of the faculty in 1983. He has presented research papers in several International Conferences in Europe, USA, UK and other countries. Dr. K.Poulose Jacob is a member of the ACM (Association for Computing Machinery) and a Life Member of the Computer Society of India.

He has more than 60 research publications to his credit. His research interests are in Information Systems Engineering, Intelligent Architectures and Networks.