ABSTRACT

SUNIL VANGARA, Code Motion techniques for STI in BBCP (Under the direction of Assistant Professor Dr. Alexander Dean)

Software thread integration (STI) helps in hardware to software migration by enabling integration of two independent software threads, which enables execution of the integrated thread in a generic processor. Automation of STI would help in obtaining efficient software versions of many hardware implemented functions. Bit banged communication protocols (BBCP) have been researched for implementation as software threads .Software integration is performed on the different threads of the implementation for obtaining efficient software versions. Though these integrated threads are efficient, these protocols include a message-level thread which is not integrated with any other threads. During integration, cocalls (coroutine calls) are introduced for context switching between integrated threads. The periodic execution requirement of cocalls results in some inefficiency introduced due to idle times introduced in the message level thread between bit level calls. We propose an optimizing algorithm to improve this and distribute the instructions in the message level thread across inter-bit calls to reduce the idle time. Our code optimization algorithm moves code from paths between inter bit calls which are larger (more execution time) to paths between inter bit calls which are smaller (lesser execution time). We adopt data flow analysis techniques similar to those used for partial redundancy elimination and dead code elimination.

CODE MOTION TECHNIQUES FOR STI IN BBCP

by

Sunil Vangara

A thesis submitted in partial fulfillment of the requirements for the degree of

Masters in Computer Science

North Carolina State University

2003

Approved by _____
Chairperson of Supervisory Committee

_____

_____

_____

Program Authorized
to Offer Degree_____

Date _____

BIOGRAPHY

Sunil Vangara was born on October 24, 1979 in Chennai, India. He graduated with a B.E. Degree in Computer Science and Engineering from the University of Madras, Chennai, India in 2001.

He the joined the masters program in Computer Science at North Carolina State University, Raleigh, NC, USA. There he was part of the Center for Embedded Systems Research and worked on his thesis under the guidance of Dr. Alexander Dean, Assistant Professor, Department of Electrical and Computer Engineering.

ACKNOWLEDGMENTS

I would like to thank every one who has played a part in the successful completion of this work. I would like to thank my parents and my brothers for providing me moral support thought out my work, which proved to be very valuable.

I would like to thank Dr. Alexander Dean, who served as my adviser and gave me the opportunity to be part of the CESR team. I would also like to thank Dr. Frank Mueller and Dr. Mihail Sichitiu, who served as my thesis committee members. Graduate studies at North Carolina State University have been a very stimulating and enlightening experience.

Finally, I would like to thank all my friends who have helped me during the course of my work.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Introduction

Bit banged communication protocols are used in some networks connecting embedded devices. These protocols are called bit-banged communication protocols, as each bit is sent one at a time using a common I/O pin. This process of sending (banging) bits one at a time could be implemented using either hardware or software. When implemented using software, these protocols impose real time requirements that have to be met by the processor on which the software is implemented. The processors efficiency is reduced due to the busy wait or interrupt processing time that is needed to meet the real time constraints. Software thread integration (STI) introduced in [DEA 1] has enabled integrating real time threads with non real time threads. This facilitates the execution of both the threads – as a single integrated thread – at the same time, thereby increasing the efficiency of the processor while meeting the real time requirements. In [KUM 1], STI concepts were extended to implement the bit banged communication protocols in software to improve efficiency. In STI for Bit Banged Communication Protocols (BBCP), the software is implemented in three different layers. Each layer in the protocol performs a different function. Hence typically the software would have three functions. They are the bit level function, the message level function and the management function (to be explained), where the bit-level is the lowest level layer. In the message level software function, cocalls to the lower bit level thread are done at regular intervals, in order to maintain proper timing. The message level function is padded with NOPs so that every inter-bit time would be equal to the maximum inter bit time. Our research is focused on improving this method. We propose a new type of code motion, different from code motions already proposed to improve this process and improve the efficiency. The code motion techniques we propose are applicable for STI in BBCP, although they can be applied to other areas with appropriate changes. We try to reduce the maximum inter bit processing time so that the amount of padding introduced is decreased. This saves significant processing time wasted by executing NOPs.

The proposed method employs the techniques of data flow analysis for the instructions at the assembly level. We have employed Control Dependence Graphs (CDGs) (chapter 4) to simplify the algorithm and make it easier for implementation. CDG is a form of graphical representation of the

nodes in a program, which identifies the dependence that exists among the different nodes of the program with respect to the control flow. Labeling (chapter 4) schemes are used to identify nodes in a CDG. Labeling is used to simplify the process of code motion and computing the insertion and deletion points during code motion easier.

The proposed method gives excellent results on many message level functon implementations of BBCP. We try to move instructions one at a time from paths with larger inter bit processing time to paths with smaller inter bit processing time. This reduces the maximum inter bit processing time which is needed for maintaining proper timing. Multiple instruction motions are not performed in this algorithm.

The thesis has been organized as follows. Chapter 2 presents the background details and the STI concepts of BBCP. Chapter 3 details relevant research conducted on the topic of code motion. Chapter 4 explains some concepts, which are used in the thesis. Chapter 5 introduces the preliminaries required for the algorithm. Chapter 6, 7 and 8 explain the algorithm in detail. Chapter 9 discusses the results obtained from the implementation in *thrint* (chapter 2).

1.2 Motivation

Due to complexity in the integration of threads, for Bit-Banged Communication protocols the management level function and the bit-level function are integrated to utilize the *coarse-grain* idle time of the bit-level function to execute management level instructions. To achieve the above integration, cocalls (coroutine calls, explained in chapter 2) are placed both in the management and the bit-level function to perform efficient context switches between them. The message level function is padded with idle time in between successive bit-level call to maintain the proper timing required for cocalls.

A sample message level function is shown below.

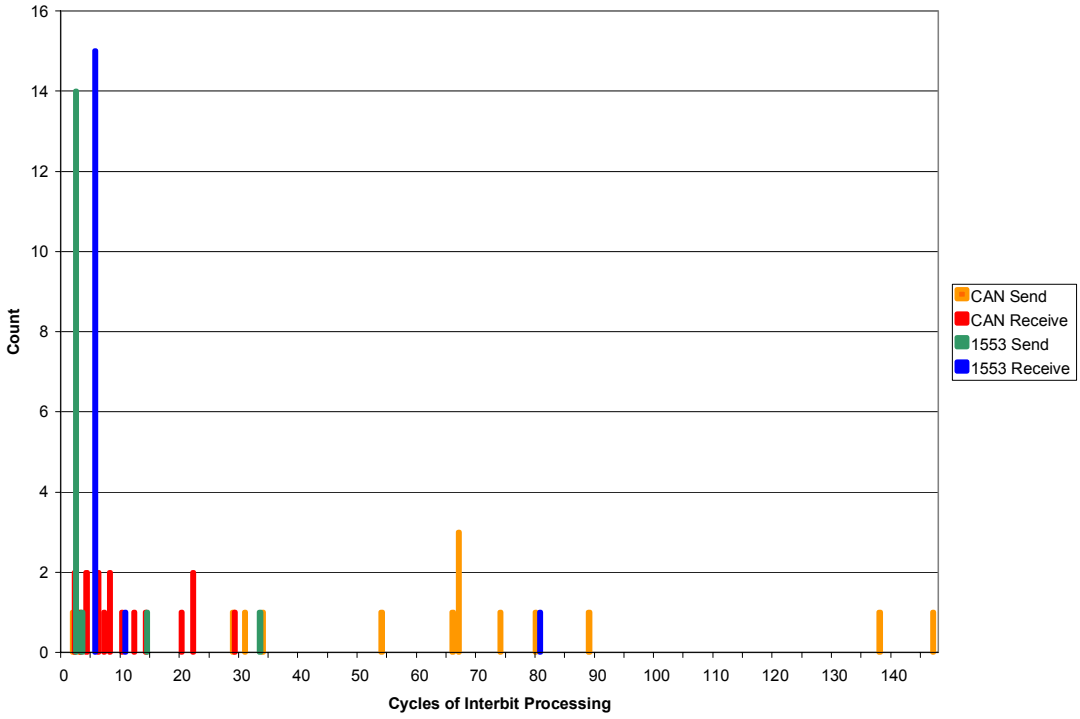
```
#include <io.h>
#include "can.h"

extern uchar oldbit_rx;
extern uchar bitcnt_rx;
extern unsigned short crc_rx;

extern uchar * CAN_bus_rx;

enum CAN_ERROR_T receive_msg(CAN_MSG_T * msg)
```

```
{
  enum CAN_ERROR_T error_code=NO_ERROR;
  short tmp;

  char i,cnt,data, n;
  ushort expected_crc;

  bitcnt_rx = 0;
  oldbit_rx = 1;
  crc_rx=0;

  /* wait until sof (start of frame) */
  while (receive_bit());
  tmp=0;

  /* Get 11 bits of Identifier */
  for (i=0; i<11; i++) {
    tmp<<=1;
    tmp += receive_bit();
  }
  msg->Identifier = tmp;
  msg->RTR = receive_bit();

  /* Get reserved bits, which according to spec can be of any value...
*/
  msg->Reserved[0] = receive_bit();
  msg->Reserved[1] = receive_bit();

  /* Get Data Length Code */
  cnt=0;
  for (i=0; i<4; i++) {
    cnt<<=1;
    cnt += receive_bit();
  }
  msg->DLC = cnt;

  /* Get up to 8 Data bytes */
  for (n=0; cnt; cnt--, n++) {
    data=0;
    for (i=0; i<8; i++) {
      data<<=1;
      data += receive_bit();
    }
    msg->Data[(int)n] = data;
  }

  /* Get 15 bits of CRC */
  expected_crc = crc_rx;
  tmp=0;
  for (i=0; i<15; i++) {
    tmp<<=1;
    tmp += receive_bit();
  }
  if (expected_crc != tmp) {
    /* CRC Error signaling is deferred until after ACK delimiter,
       according to spec. */
    error_code = BAD_CRC;
```

```
  }
  msg->CRC = tmp;

  /* Get CRC delimiter bit directly - not stuffed */
  bitcnt_rx = 0; /* disable stuffing for 5 bits */
  if (receive_bit() == 0)
    return BAD_FORM;

  /* Get ACK bit - not stuffed */
  if (receive_bit() == CAN_REC)
    return NO_ACK;

  /* Get ACK Delimiter bit - not stuffed*/
  if (receive_bit() == CAN_DOM) {
    return BAD_FORM;
  }

  /* CRC Error signaling is deferred until after ACK delimiter */
  if (error_code == BAD_CRC) {
    return error_code;
  }

  /* Get End of Frame - not stuffed*/
  for (i=0; i < 7; i++) {
    bitcnt_rx = 0;
    if (receive_bit() == 0) {
      return BAD_FORM;
    }
  }
  return error_code;
}
```

In the above code segment we can see that the `receive_bit()` function, which is responsible for sending out a bit on to the bus, is called several times. Different instructions are executed between any two subsequent `receive_bit()` function calls. This, forces us to introduce idle time between the bit-level calls, such that all the inter-bit processing times become equal. To achieve this all the inter-bit paths are padded with NOPs such that their processing time equals that of the path with the greatest processing time.

We have analyzed the message level functions of different BBC protocols and calculated the inter-bit processing times present in those threads. The plot of the various inter bit processing times are shown in figure 1.1 As can be clearly seen from the plot, a lot of processing time would be wasted, i.e., filled with NOPs between the end of processing of the instruction in that inter bit path and the call instruction (to the bit level function), so that each inter bit processing time is constant and is equal to the maximum inter bit processing time.


16
14
12
10
8
6
4
2
0
Count
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
Cycles of Interbit Processing
CAN Send
CAN Receive
1553 Send
1553 Receive

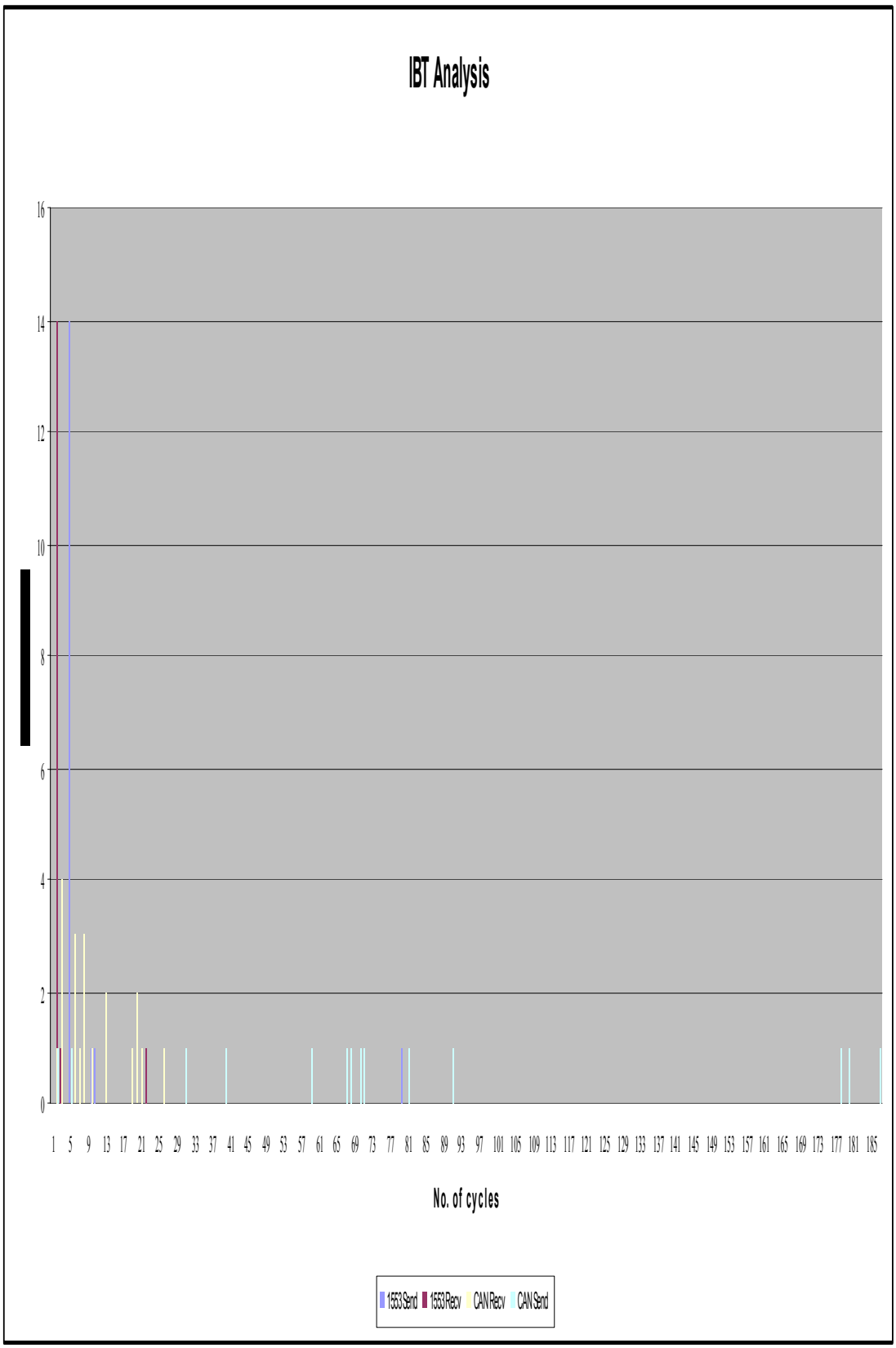

IBT Analysis
16
14
12
10
8
6
4
2
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97 101 105 109 113 117 121 125 129 133 137 141 145 149 153 157 161 165 169 173 177 181 185
No. of cycles
1553Send
1553Recv
CANRecv
CANSend

Figure 1.1 Inter-Bit processing time distribution

The question is how we can improve this scenario to utilize the time in which the NOPs are inserted. We propose to perform code motions within the message level function to utilize the NOP execution times. This thesis describes how code motion is performed from a inter bit path with the maximum inter bit processing time to inter bit paths with lower inter bit processing time. This process will reduce the maximum inter bit processing time and thereby increase the idle time available at the bit level layer to improve the efficiency of the processor. This concept is illustrated in figure 1.2.

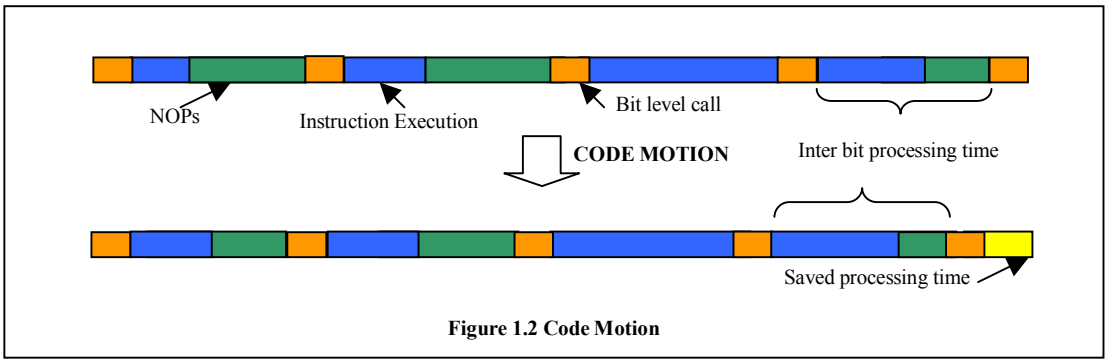


Figure 1.2 Code Motion

Much of the algorithm concepts and ideas have been derived from traditional syntactic code motion by Knuth et. al. Their papers on dead code elimination (DCE) and partial redundancy elimination (PRE), [KRS 1] [KRS 2] [KRS 3] [KRS 4], have helped us in understanding the relationships that exist among the variables of a given program and the concepts of code hoisting and code sinking. These algorithms have also helped us in understanding the intricacies of data flow analysis and helped in our own conception of a new form of data flow analysis. We have derived a new algorithm based on these concepts wherein instructions themselves are moved rather than the computations in expressions, as done by these authors. Furthermore our algorithm works with registers as opposed to variables in DCE and PRE algorithms

Chapter 2

BACKGROUND

2.1 Software Thread Integration

STI [DEA 1] is a compiler technology, which interleaves multiple assembly language threads at a fine-grain level. The resulting thread offers low-cost concurrency, but still executes on a generic processor without fast context switches. STI can be used for hardware to software migration (HSM). HSM is the process of moving functions from dedicated hardware components to real-time software. HSM helps to improve system cost, size, weight, power, function availability, time to market, and field upgrades. The main targets of HSM are embedded systems which cannot afford the luxury of a high performance microprocessor yet require fine-grain thread concurrency.

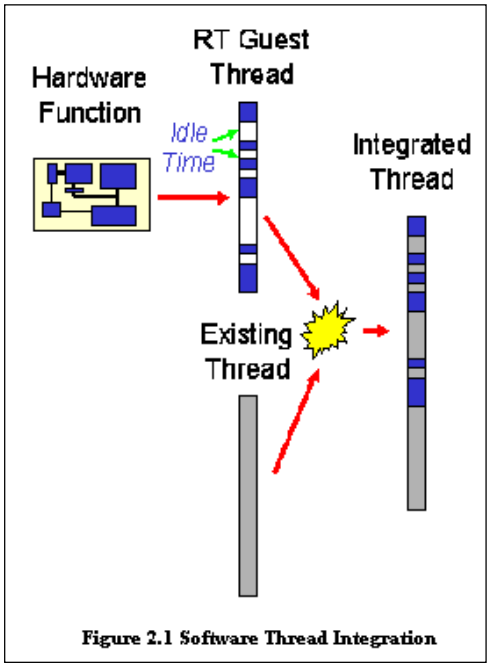


Figure 2.1 Software Thread Integration

In HSM with STI, illustrated in figure 2.1, the integration of two threads is performed. One of the threads is a real time thread with real time deadlines to meet and the other thread is a non real time thread associated with the application. The real time thread is the software implementation of the dedicated hardware function, which we want to move to software. The real time thread becomes the guest thread and the non real time thread becomes the host thread. As can be observed from figure 1.1, the real time thread does not use up the processor the entire time its running. It has a coarse grain idle time between instructions that have real time deadlines to meet. This idle time is either used up in executing NOP instruction until the next deadline to be met is reached or is used to process interrupts when the real time deadlines are met using interrupts generated through a timer. STI concepts help us to extract these fine grain idle time wasted during the execution of a real time thread and be used for executing other threads. The existing thread which does not have any real time requirements becomes the host thread and the guest is inserted into the host thread at locations which satisfy its real time constraints.

Performing this type of integration is tricky. Static timing analysis is performed on the control dependence graph generated for both the threads. This timing analysis gives us information about how early or how late an instruction would be executed when the threads are run and also how long the instructions take to complete the execution. This type of information helps in identifying insertion places in the host thread, where instructions from the guest thread can be placed so that the real time requirements of the guest thread is properly met. CDGs facilitate easier implementation of static timing analysis and provide excellent support for determining the insertion points that meet the real time constraints in the host thread and the subsequent placement of the guest code in those locations. Register reallocation techniques are also employed to enable sharing of the processor's register set by the two threads, without any conflicts.

STI is extremely valuable for recovering idle time from real-time guest threads performing low-level (e.g. MAC and data link layer) network communication and video refresh functions. Other applications with significant fine-grain idle time can benefit from STI as well. An example of an application in which STI has been implemented is given in [DEA 2] in which high temperature (185-225 C) CAN network interface is implemented in software and integrated with buffer management code for the CAN protocol. This system has been built and tested at up to 225 C.

A special compiler called *thrint* has been developed for this purpose. *Thrint* is a post-pass (back-end) compiler which reads assembly code, performs control-flow, data-flow and static timing analysis, and integrates threads.

2.2 STI for BBCP

STI concepts were employed in [KUM 1] for improving the performance of BBC protocols used in embedded networks. Examples of embedded protocols are J1850, CAN, 1553, etc. These protocols are extensively used for communication among embedded devices connected in a network. An example of such a network would be a car in which the various parts of the car are connected together in a network for proper functioning of the car.

The software implementations of these protocols are referred as bit-banged communication protocols as the bits are sent one at a time in software.

For STI in BBCP, the protocol functions are implemented in three layers. This scheme is illustrated in figure 2.2. Figure 2.3 gives an example of such a three layer stack using a send function. The three layers are explained below:

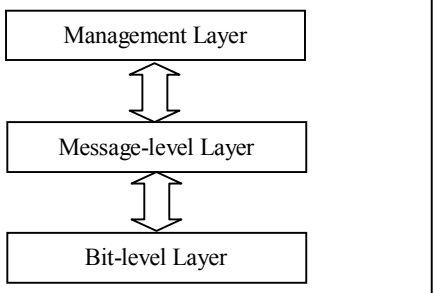


Figure 2.2 BBC Protocol Layers

The executive or manager function: This is the top-level function that runs a finite state machine which monitors the bus to either send to or receive messages from it. As shown, when a request to transmit is received, the manager layer uses a subroutine call to pass on the request to its lower layer, viz. the messaging layer.

Message level function: This is the middle layer that is called by the upper executive or manager layer. This consists of functions that handle messages, such as send_message function or receive_message function. This function performs all the encoding/decoding schemes of the protocol and passes the messages to the corresponding layers. In addition to these, depending on the protocol, the messaging layer may also be responsible for calculating the CRC (when sufficient time does not exist in bit level layer to perform these functions) and checking whether the received CRC is same as the calculated CRC.

Bit level function: This is the bottom-most layer. This layer takes care of sending or receiving bits to or from the bus. The function that runs at this layer is a real time function, as it has to meet the timing requirements of sending and receiving bits at the appropriate instant. This function is invoked by the message level layer. For example, the send message function invokes the send bit function at this layer. The bit level function may also be responsible for additional functions such as bit stuffing, CRC/parity generation/check

The three layers are implemented independently. As can be seen from figure 2.3, the bit level function has *coarse grain* idle time, which would be wasted by busy wait or by interrupt processing depending of which method is used for meeting the real time constraints. To recover these fine-grained idle times and use them to perform other tasks by the processor, the bit level function is integrated with the upper management level function. A concept called *Cocall* was introduced to switch between these two threads.

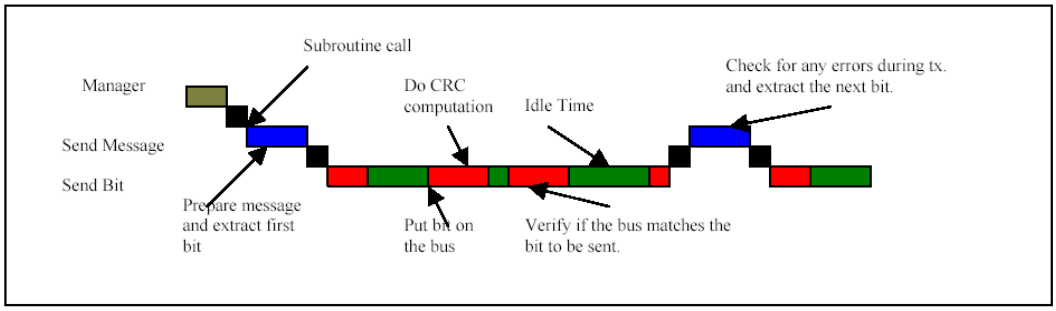


Figure 2.3 Example of a three stack implementation of send message

Cocalls are similar to subroutine calls. A co-call operation is used to transfer control between two processes. A co-call is effectively a call and return instruction combined into one operation. From the point of view of the process executing the co-call, the operation is equivalent to a procedure call and from the point of view of the process being called; the co-call operation is equivalent to a return operation.

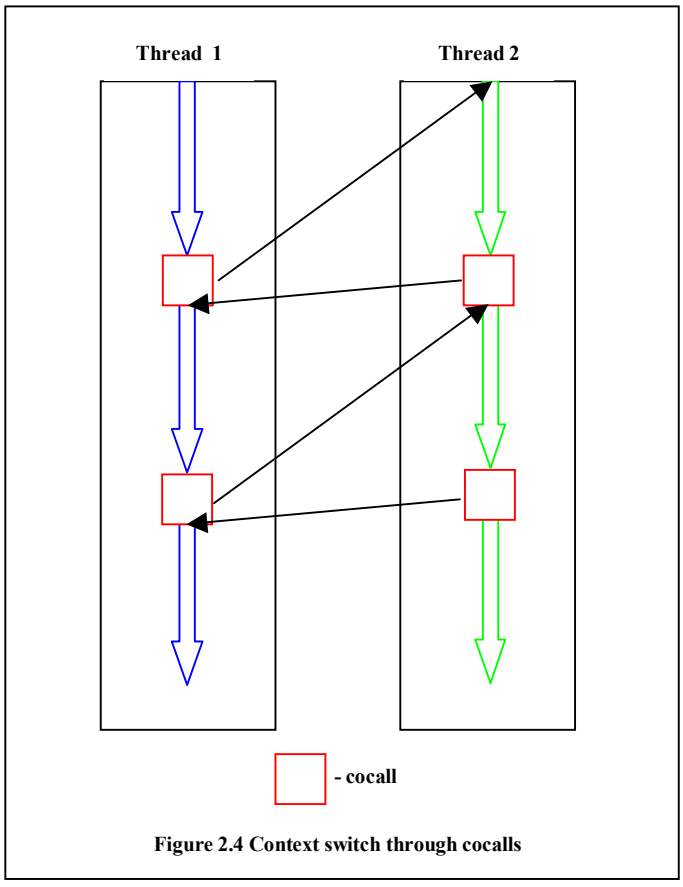


Figure 2.4 Context switch through cocalls

Thus, unlike subroutines, when the second process cocalls the first, control resumes not at the beginning of the first process, but immediately after the co-call operation. If the two processes execute a sequence of mutual co-calls, control will transfer between the two processes as shown in figure 2.4

Asynchronous Software Thread Integration (ASTI) is performed to remove the time critical code from the bit-level function to create a longer idle time for a secondary thread to run. The secondary thread is modified such that it cocalls back to the primary thread just before the end of the enlarged idle time. The code removed from the primary thread is integrated with the secondary thread so that it is executed at the correct times. Both the threads are then padded with NOPs to eliminate jitter. The bit-level functions are modified so that they run for a fixed amount of time. The message level functions are first padded to reduce jitter and then padded to make the calls to the bit level functions occur at a fixed frequency. The secondary thread which contains the management function is padded to remove jitter. Thus the resulting threads are as shown in figure 2.5. More details about the various stages of integration is given in [KSA 1]

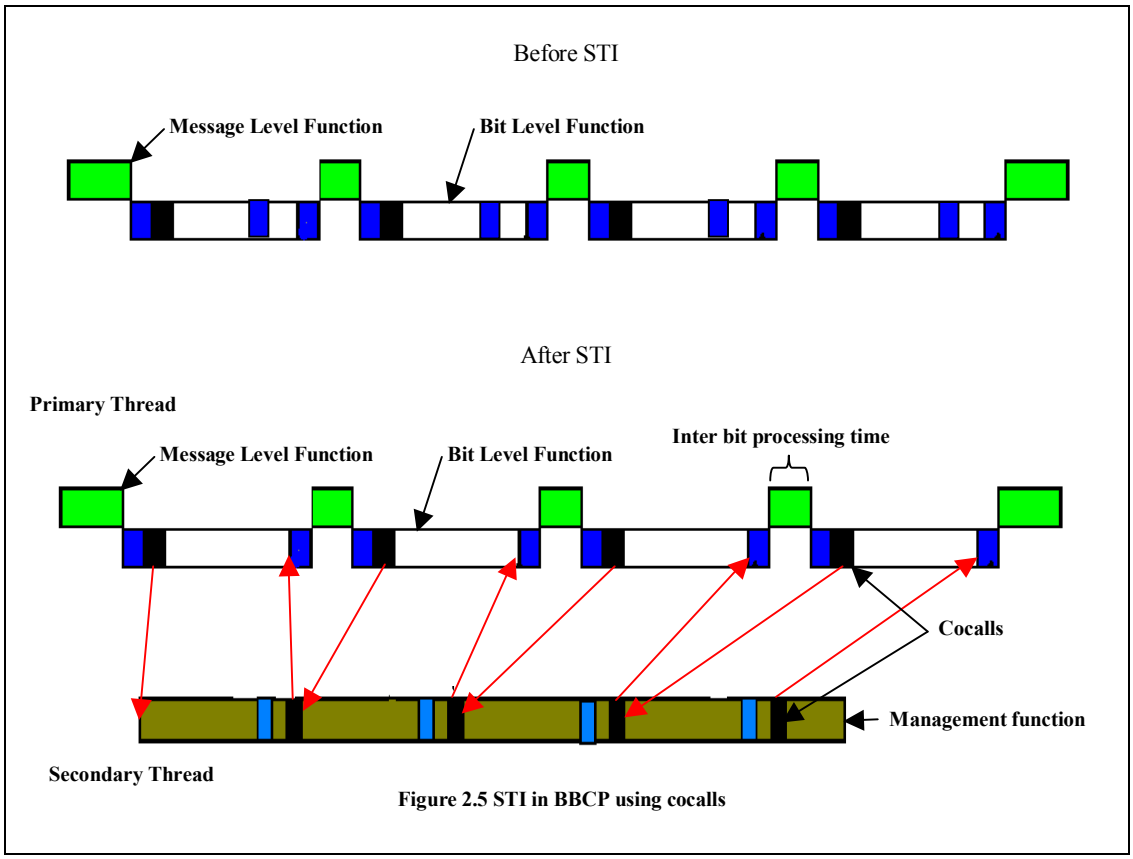


Figure 2.5 STI in BBCP using cocalls

In order for the cocalls to occur at a fixed frequency, the message level function has to be padded for each inter-bit processing time; to make sure that the calls to the lower bit-level function occurs at a fixed frequency as well. As explained in the previous chapter we try to move code in the message level function to reduce the run time of the largest inter-bit path that exists in the function. This would reduce the amount of padding required for the message level function and thereby will improve the efficiency of the process.

Chapter 3

RELEVANT RESEARCH

3.1 Code Motion

Code motion is a relatively old concept and significant research has been done in the field. This section gives a brief account of the various research results for code motion. Traditionally code motion has been aimed at improving generic programs which run in generic processors. These types of code motion can be classified as "higher-level" code motion. The intent in this type of code motion is to move instructions that are redundant (partial redundancy elimination) or dead (dead code elimination) when traversing the program along one of its flow paths. So the net effect is to move code so that the average execution time of the program can be reduces. At the other end of this code motion spectrum is the "lower-level" ("machine-level") code motion. Algorithms performing "lower-level" code motion work on the machine or assembly level of the programs. These programs try to move high-delay instructions such as loads and stores to various parts of the program to improve the run-times of a program. These types of algorithms are machine specific and can vary from architecture to architecture. Significant research has been carried out in this field as well.

3.2 Higher-Level Code Motion

Higher-level code motion as mentioned previously is performed on programs to move code that is redundant or dead along various paths of the program to improve the performance of the program. [STE 1] refers to "higher-level" code motion as any transformation that moves assignments or expression computations forwards or backwards within a flow-graph in order to remove unnecessary assignments or expressions. This results in two types of code motions *code* sinking, which moves the code forwards (along the flow of control for the program) and *code hoisting*, which moves the code upwards (against the flow of control for the program). Code motion algorithms proposed for higher-level code motion can be broadly classified as *semantic code motion* and *syntactic code motion*.

Syntactic code motion relates instructions (assignments or expressions) based on the syntax of the instruction i.e., how the instruction appears in the program. It does not consider the relationships among the values stored in the variables used in the instruction or the result of an instruction. *Semantic* code motion relates instructions based on the values of the variables involved in the instruction and the result of the instruction. In figure 3.1, the value of A on the second and the fourth line is the same (value is 3), but the syntax is different in the second and the fourth line. Thus whereas syntactic code motion might find the fourth line of code unnecessary and remove it, syntactic code motion fails to identify that the two lines of code do the same task and will not perform any code motion. Since their syntax is different syntactic code motion treats them as two separate instructions.

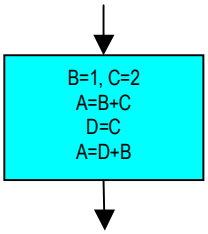


Figure 3.1 Code motion

3.3. Syntactic Code Motion

Syntactic code motion has been extensively researched by Knoop, Ruthing and Steffen. They have published various algorithms for syntactic code motion in [KRS 1], [KRS 2], [KRS 3] and [KRS 4]. This section describes the code motion and the algorithms briefly.

3.3.1 Critical Edges

A critical edge is an edge in the flow-graph from a block with multiple successors to a block with multiple predecessors. Critical edges pose a problem for many code motion algorithms, as it makes it difficult to move an assignment or expression to the middle of the edge. For example figure

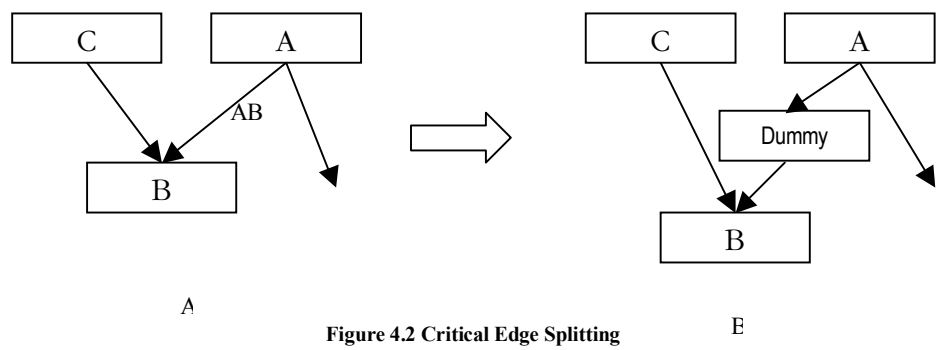


Figure 4.2 Critical Edge Splitting

3.2(A) shows a critical edge named AB. To overcome this problem the authors propose a process of *edge splitting*, which is the insertion of a block (dummy block) in the middle of the critical edge. This breaks the critical edge and enables movement of code into the dummy node.

3.3.2 Partial Dead Code Elimination

In [KRS 1], Knoop, Ruthing and Steffen propose an algorithm that eliminates partial dead code as much as possible without modifying the program structure or affecting the other parts of the program. An assignment is said to be *partially dead* if some but not all the paths from the assignment to the exit of the program contain a use of the variable that is being assigned. An example is shown in figure 3.3. In code A, the assignment X=A+B is said to be partially dead along the path through the left side of the branch, as X is redefined in X=A along that branch and there is no use of the value stored in A in between the two definitions. But it is not dead along the right side of the branch as the value of X assigned before the branch is used in the statement Z=X+Y. So after applying partial dead code elimination the new flow-graph as shown in B, eliminates the need for performing the assignment X=A+B, when traversing along the left side of the branch thereby reducing the runtime along that branch.

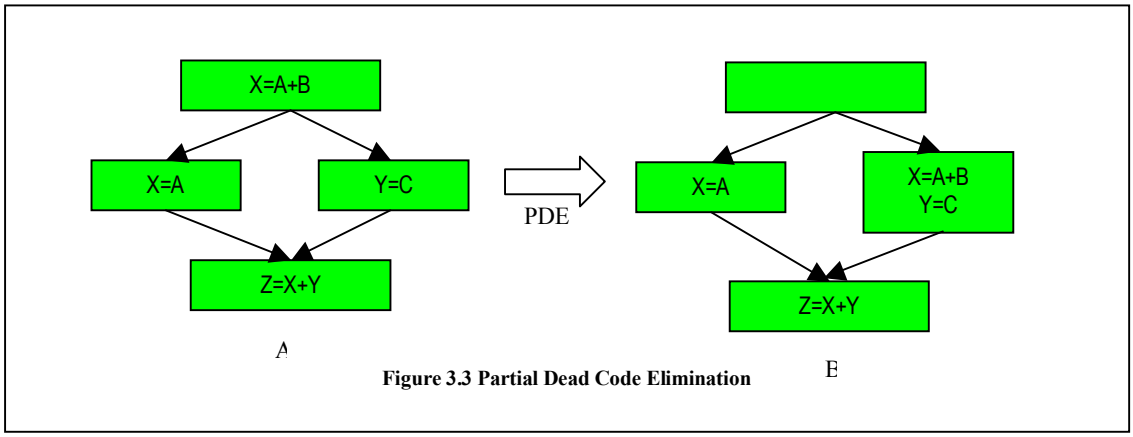


Figure 3.3 Partial Dead Code Elimination

The proposed algorithm consists of two separate steps:

Faint Assignment Elimination – In this step *faint* variables from the program are identified. A variable is said to be *faint* at a point in a flow graph if, on every path from that point to the exit of the program, any use of the variable is either preceded by a redefinition of the variable, or occurs within an assignment to a variable which is also faint. This step starts by assuming that all the

variables are faint everywhere, except arguments of a procedure call and return variables. Then, iterative backward analysis is performed as follows:

N-FAINT$_i(x)$ denotes whether the variable x is faint at the beginning of an instruction or assignment *i*. **X-FAINT**$_i(x)$ denotes whether the variable x is faint at the end of an instruction or assignment *i*.

The faint variable analysis is given as

$$\textbf{N-FAINT}_i(x) = \sim\textbf{RELV-USED}_i(x) \ \Lambda\ (\textbf{XFAINT}_i(x) \ \textbf{V} \ \textbf{MOD}_i(x)) \ \Lambda\ (\textbf{XFAINT}_i(lhs_i) \ \textbf{V} \ \sim\textbf{ASSIGN-USED}_i(x))$$

$$\textbf{X-FAINT}_i(x) = \Pi_{j \ \varepsilon \ succ(i)} \ \textbf{N-FAINT}_j(x)$$

The predicates used are
RELV-USED$_i(x)$ – x is a right-hand side variable of the relevant instruction *i*. (variables occur only on the right hand side of relevant instructions).
MOD$_i(x)$ – x is the left-hand side variable of the instruction *i*.
ASSIGN-USED$_i(x)$ – x is the right-hand side variable of the assignment statement *i*.

After computing the greatest solution for the above set of equations for the variables corresponding program transformations are performed.

Assignment Sinking – Assignment sinking is the motion of assignments within a flow-graph in the direction of program execution. Here a set of equations are computed for each sinkable statement and delay ability analysis is performed which determines how far an instruction can be sunk along the program flow.

3.3.3 Partial Redundancy Elimination

In [KRS 2], the same authors propose an effective algorithm for partial redundancy elimination. An expression is called *partially redundant* if some but not all paths from the entry of the program to the expression contain another calculation of the expression and none of the expression's operands are redefined between the two calculations. In figure 3.4(A), the expression A+B is computed twice, once when it is assigned to X and again when it is assigned to Y. In either case the value of the expression A+B remains the same, since the value of the variables is not modified in between

the two expressions. Thus the expression A+B is said to be *partially redundant*. Note that the expression is redundant only along the right side of the branch

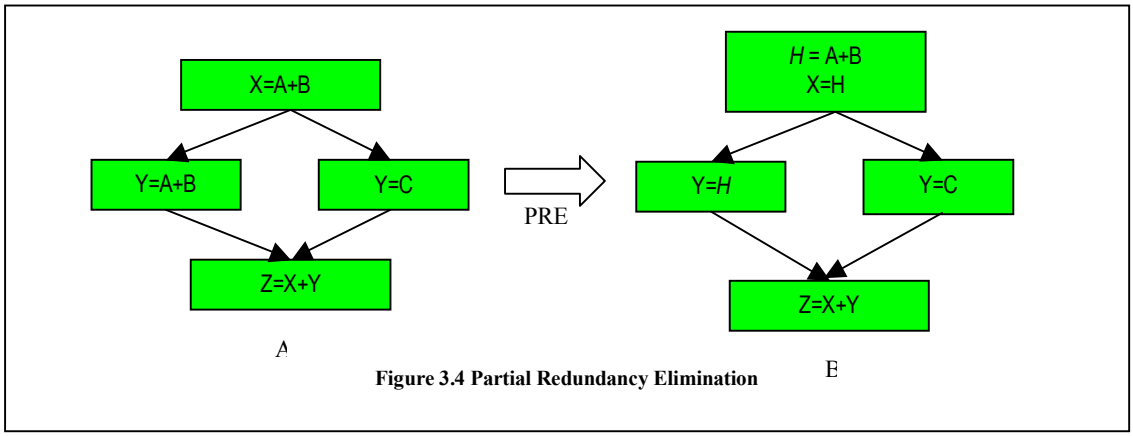


Figure 3.4 Partial Redundancy Elimination

and not along the left side of the branch. Applying PRE we can store the value of the expression A+B in a temporary variable *H* and use it in the two assignments as shown in figure 3.4(B).

The proposed algorithm determines D-SAFE points for each computation and determines how far a computation can be hoisted. Hoisting the computation as shown in the example removes partially redundant computations from the program. For a computation *t*, the D-SAFE at the various nodes (code blocks) of the program is computed using the greatest solution of the following equation system.

$$\textbf{D-SAFE}(n) = \begin{cases} false & \text{if } n = \text{EXIT NODE} \\ Used(n) \vee & \\ Transp(n) \wedge \prod_{m \in succ(n)} \textbf{D-SAFE}(m) & otherwise \end{cases}$$

D-SAFE can be defined as follows: *a node n is n-down-safe if and only if* D-SAFE(*n*) holds.

Then the earliestness of the nodes are computed using the following system of equations:

$$\textbf{EARLIEST}(n) = \begin{cases} true & \text{if } n = s \\ \Sigma_{m \in pred(n)} \; (\sim Transp(m) \vee & \\ \quad \sim \text{D-SAFE}(m) \wedge \text{EARLIEST}(m)) & otherwise \end{cases}$$

A node n is n-earliest if and only if EARLIEST(*n*) *holds.*

Using the D-SAFE and the EARLIEST values for the nodes computed, the Safe-Earliest Transformation can be performed as follows:

- Introduce a new auxiliary variable *h* for the term *t*.
- Insert at the entry of every node *n* satisfying D-SAFE and EARLIEST the assignment *h*:=*t*.
- Replace every original computation of *t* in the program by *h*.

Whenever D-SAFE(*n*) holds, there is a node *m* on every path *p* from start to *n* satisfying D-SAFE(*m*) and EARLIEST(*m*) such that no operand of *t* is modified between *m* and *n*. Thus all replacements of the above algorithms are correct. The authors also propose some special features to remove unnecessary code motion which does not improve the run-time of the program. This is called the lazy code motion algorithm.

In [KRS 4], the same authors proposed a similar algorithm with aggressive code motion. In the above lazy code motion algorithm the expressions are moved up only as far enough to perform code optimization. In the aggressive code motion algorithm, code motion is performed on every expression, resulting in the hoisting of the expressions as far up as possible. This had the disadvantage of increase in register pressure due to intermingling of expressions, which resulted in increase in the number of live variables.

A more structured approach for partial redundancy elimination is given in [CCK 1]. The authors propose an algorithm for partial redundancy elimination, which achieves optimal code motion similar to “lazy-code”. The algorithm is based on SSA for of a program and tries to eliminate the iterative data flow analysis and bit vectors in its solution. In [BGS 1], a new profile based redundancy elimination algorithm is proposed in which cost of code growth is weighed against the benefit of code motion to justify code motion. This algorithm is proposed for code motion in programs meant for embedded real-time applications.

In [STE 1], the author suggests a combined algorithm for code hoisting and code sinking. Code hoisting and code sinking work in opposite directions of the program flow and consequently when the two are applied together to a program, it would result in the two algorithms enabling each other after each pass of the algorithm. The paper hypothesizes the following points about the combined approach

1) Generate temporary variables (*h*) for each expression *e*, and split assignments as described in the initialization stage of the algorithm in [KRS 4].

2) Iterate the following until the code stabilizes:
   a) Redundant assignment elimination.
   b) Faint assignment elimination.
   c) Assignment hoisting.
3) Iterate the following until the code stabilizes
   a) Faint assignment elimination.
   b) Redundant assignment elimination.
   c) Assignment hoisting.
4) Apply a backwards copy propagation algorithm to eliminate unnecessary temporary variables (*h*).

3.4 Semantic Code Motion

Semantic code motion extends syntactic code motion algorithms to perform code motion of expression that evaluate to the same value. Semantic code motion performs better optimization compared to syntactic code motion as explained earlier. *Value numbering* has been one of the major approaches in performing semantic code motion. The primary objective of value numbering is to assign an identification number called *value* number to each value that is computed in a program in such a way that two values have the same number if the algorithm can prove that they are equal under all possible values of inputs and control flow. This section describes the research results in the field of semantic code motion.

Semantic code motion is generally based on *Static Single Assignment* (SSA) forms of programs. In the SSA, each variable name in the program occurs only once on the left hand side of the flow-graph. This makes the relationship between the definition of a variable and its use in the program more explicit. When more than one value for a variable is possible at the beginning of a block, depending on the branch along which control enters the beginning of the block, ϕ functions are used to determine which value of the variable is used in the block. A detailed description of SSA form is given in [BHS 1]. In [CFR 1], an efficient algorithm for obtaining the minimal SSA form of the flow-graph is presented.

3.4.1 Hash-Based Value Numbering

In [COC 1], the authors describe a local technique which uses hashing for identifying computations that are redundant. Each unique value is given a value number. Two values are given same value

number if they are provably equal. This algorithm though works only inside code blocks and does not expand over the entire program.

In [BCS 1], the authors extended the hash based algorithm of [COC 1], to perform hash-based value numbering over and entire program by using a single hash table for all the basic blocks. This method proceeds by traversing the control flow graph (CFG) of a given program in reverse post-order and process the ϕ functions and instructions in each block. For processing the ϕ functions in a block, the algorithm checks to see if there are any incoming back edges. If there is back-edge and the ϕ function refers a value coming through the back edge, the parameter of the ϕ function may not have a value number assigned to it. In this case, the compiler assigns a unique value number to the incoming parameter. If there are no back-edge flows into the block, the reverse post-order traversal guarantees that all the ϕ function parameters have been assigned value numbers. Thus each block of the program is analyzed completely. When processing the ϕ functions, the algorithm overwrites the operands with their value numbers, simplifies algebraic expressions and performs code motion on the resulting expression. If an expression similar to the simplified one is found in the hash table, the result is overwritten with the same value number of the expression. Otherwise, the expression is added into the hash table and the algorithm proceeds. But this algorithm cannot handle values that flow through back edges and does not work well with programs containing loops.

3.4.2. Value Partitioning

In [AHU 1], Aho, Ullman and Hopcroft propose an algorithm for DFA minimization. Alpern et. al. use the idea behind this algorithm in [AWZ 1] to partition values into congruent classes on the SSA form of the program. In this algorithm two values are said to be congruent if they are computed by the same opcode, and their corresponding operands are congruent for all expressions, two congruent values must be equal. This recursive definition would not result in a unique solution for many programs. Hence the authors propose using the maximal fixed point solution, which is the solution that contains the most congruent values. Initially all the values are assumed to be congruent and are put in a single congruent partition. The partition is then refined by identifying different congruent classes and assigning them to different partitions. This splitting goes on until the algorithm can make no more changes to the partitions or congruent classes. This type of value numbering is more complicated then the hash-based scheme and this algorithm also suffers from the problem of not handling values flowing through back edges. Both the hash based algorithm and the value partitioning algorithm are explained in detail in [MUC 1]

3.4.3 SCC-Based Value Numbering

In [COO 1], an algorithm that works in conjunction with Trajan's algorithm [TAR 1] for finding strongly connected components (SCCs). Tarjan in his algorithm uses a stack to determine which nodes are in the same SCC. Nodes which are contained in a cycle are popped out of the stack together, whereas nodes that do not belong to any cycle are popped out of the stack singly. This concept of SCC was used in [COO 1] to identify nodes in which there is a flow of data through the back-edges. When a single node is popped from the stack, value numbers would already have been assigned to the operands of the corresponding expression and hence a value number can be assigned to the expression easily. When a collection of nodes representing an SCC is popped, we would have assigned value numbers to any operands outside the SCC. The nodes of the SCC need to be handled differently, while computing their value numbers. The value numbers for the nodes in the SCC are assigned by iterating in the reverse post-order. Initially the value for each member of the SCC is assumed to be some unknown value. The unknown value indicates that the value number for that member has not been computed yet. Then optimistic assumptions are made in an iterative process and the values are stored in an optimistic table. Then when the iterative process stabilizes, entries are added to the main *valid table*. Note that for single nodes values are directly added to the *valid table*.

The reverse post order algorithm proposed by the authors in this paper is as follows

1) Initialize VN[x] = ⊤ for each SSA based name x.
2) Repeat the following:
   a) Iterate through the blocks in the reverse post-order.
      i) Iterate forward through the definitions α in the current block.
      ιι) Set VN[α] = lookup([α]).
   b) Remove all entries from the hash table.
3) Until no changes to NV[] occurs.

3.4.4 Value-Driven Code Motion

In [COO 2], Cooper and Simpson propose a value-driven code motion algorithm which performs code motion to eliminate computations of redundant values. This is in contrast to the syntactic code motion which performs code motion to eliminate redundant assignments. Value driven redundancy analysis is performed over a domain of values instead of assignment. The predicates used in his algorithm are

- EXECUTED-LOCAL(*b*) – this is a set of values computed by assignments in the block *b*.
- REDUNDANT-IN(*b*) – this set contains values which have already been computed along every execution path from start to the beginning of the block *b*.
- REDUNDANT-OUT(*b*) – this set contains values which have already been computed along every execution path from start to the end of the block *b*.

For local analysis is performed to determine the EXECUTED-LOCAL set for each block of the program. Then REDUNDANT-IN and REDUNDANT-OUT which are defined recursively as follows are computed for each block:

$$\textbf{REDUNDANT-IN}(b) = \begin{cases} \{\} & \text{,for } b = \text{start} \\ \prod_{p \ \varepsilon \ pred(b)} \textbf{REDUNDANT-OUT}(m) & \text{,otherwise} \end{cases}$$

$$\textbf{REDUNDANT-OUT}(b) = \textbf{EXECUTED-LOCAL}(b) \ \cup \ \textbf{REDUNDANT-IN}(b)$$

The above analysis is applied to the program from the start node to the end node in the forward direction, until the values of the predicates REDUNDANT-IN(*b*) and REDUNDANT-OUT(*b*) stabilize. This method is called iterative forward analysis. The actual transformation is performed as follows:

- Scan through each block of the program in the forward direction.
- Update the contents of REDUNDANT-IN(*b*) after each instruction.
- Delete any assignments encountered that computes values which are already contained in REDUNDANT-IN(*b*).

C h a p t e r 4

CONCEPTS

4.1 Control Flow Graphs (CFGs)

Control flow graphs are graphical representations of the flow/sequence of execution of a program. We use control flow graphs as a first step towards obtaining the control dependence graphs.

Definition 4.1: A control flow graph is a directed graph G augmented with a unique entry node START and a unique exit node STOP such that each node in the graph has at most two successors.

We assume that nodes with two successors have attributes "T" (true) and "F" (false) associated with the outgoing edges in the usual way. We further assume that for any node N in G there exists a path from START to N and a path from N to STOP.

4.2 Control Dependence Graphs (CDGs)

Control dependence graphs are also graphical representation of programs. But CDGs represent the dependence relationship that exists between instructions in the program. For example, an instruction *i*, present inside an *if* instruction is dependent on the *if* instruction. Ferrante et. al. define CDGs as follows in [FER 1]

Definition 4.2: Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X iff

1. *There exists a directed path P from X to Y with any ∈ in P (excluding X and Y) post-dominated by Y and*
2. *X is not post-dominated by Y.*

We use the same definition of CDGs. But the CDGs constructed in our process is different from that of the CDGs obtained using Ferrantes method. The CDGs can be generated only for structured code.

Definition 4.3: A sequence of basic blocks is said to be structured iff the CDG generated has an hierarchical structure. i.e., if a → b (a depends on b) and a → c, then either b → c or c → b is true (a, b and c are different basic blocks).

This type of CDG is chosen for STI because of the following reasons

- The timing analysis can be performed correctly without ambiguity using the CDG.
- Accurate timing intervals can be identified in the code
- Insertion of guest code into the host CDG can be done accurately to meet the timing constraints of the guest code

A simple example for a CFG and its corresponding CDG is shown in figure 4.1

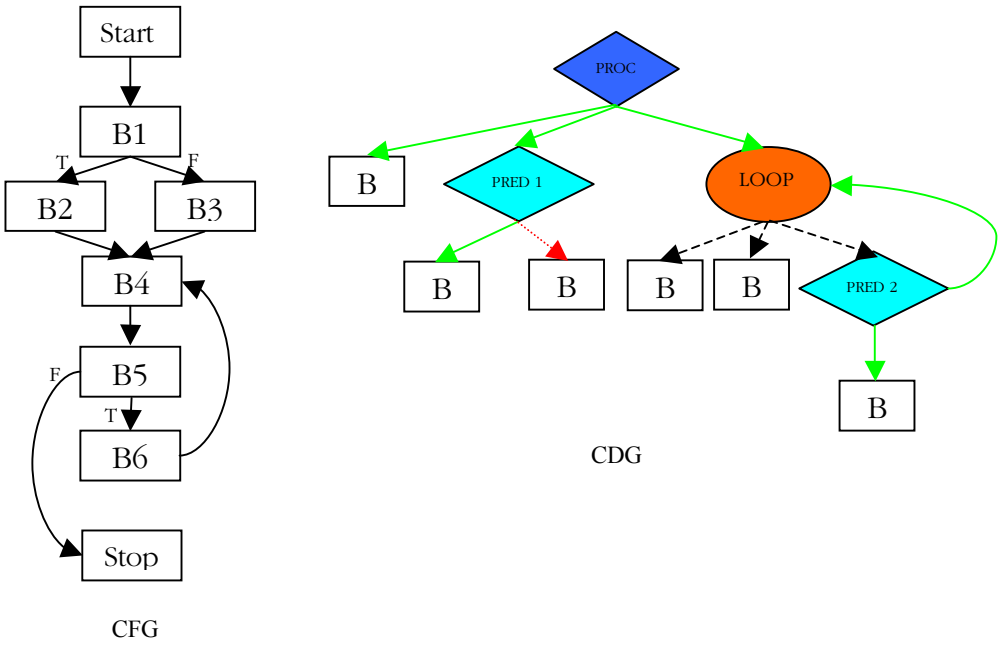


Figure 4.1 CFGs and CDGs

Every code block in the CFG is represented as CODE node (square shaped nodes) in the CDG. The node B2 is represented as a CODE node in the CFG. When a basic block has a conditional branch at its end, the conditional instruction is placed in a PRED node (diamond shaped nodes) in the CDG. The code block B1 has a conditional branch at its end. This instruction is represented as the PRED node PRED 1. Loops are represented using a LOOP node (ellipse shaped node) as shown in figure. The loop involving the nodes B4, B5 and B6 is represented as the node LOOP 1.

The arrows in the CDG point to the dependents of each node. The green or line arrow indicates a dependency based on TRUE result of the conditional instruction (Example – arrow from PRED 1 to B2). The red of dotted arrow indicates a dependency based on FALSE result of the conditional instruction (Example – arrow from PRED 1 to B3). The black or dashed arrows indicate a straightforward unconditional relation (Example – arrow from LOOP to B4). This type of arrow is used only for unconditional children of a LOOP node. In other words this arrow links nodes which execute at least once in a loop to the LOOP node. A detailed explanation fo the CDG format is given in [DEA 1].

4.3 Labels

Labels were introduced in [NEW 1]. The node labeling scheme proposed can be used for assigning labels to each node in a CDG. This labeling scheme helps to determine the relative positions of the nodes by just comparing their labels. This also facilitates computation of reachability, dominance and control dependence relationships, as well as node-to-node traversals. Without the labeling scheme, the evaluation of the above relations would require traversing the entire tree from node-to-node.

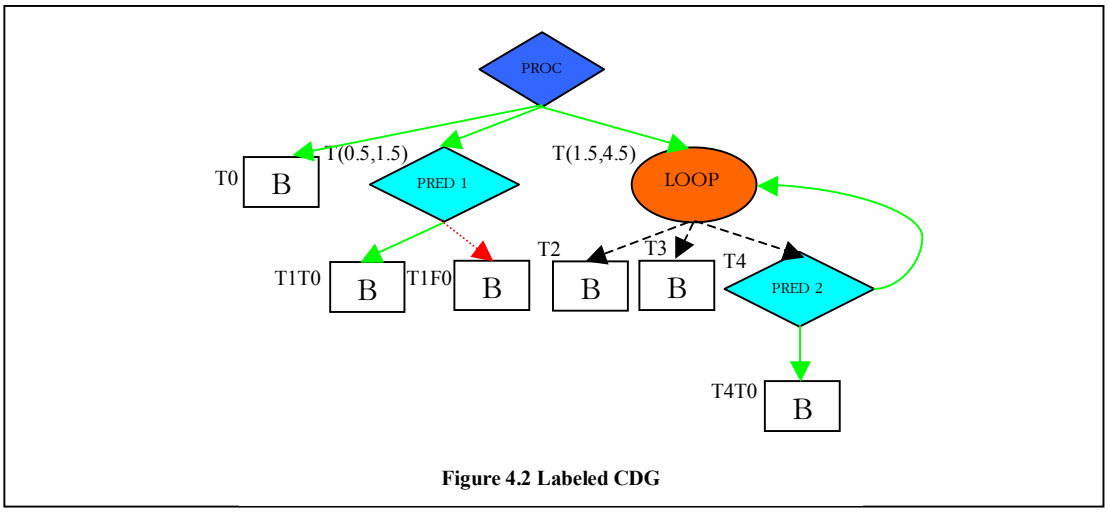


Figure 4.2 Labeled CDG

We use labels in our algorithms for the same reasons of improving the efficiency in obtaining the relationships between the various nodes in the CDG. Each node is assigned a label in this scheme. Each label consists of an array of cells. The number of cells a label contains depends on the depth

of the node in the CDG. So if a node is at a depth of n from the root of the CDG it will contain n cells. For the CDG constructed in the previous section, the labeled CDG is shown in figure 4.2.

The i^{th} cell of a label corresponds to the position of the node in the CDG with respect to the conditional node at level *i*. Each cell in the label has three fields. The first field specifies which *side* of the conditional the node is present. There can be two values either T for true or F for false. The second field contains the *section number*. The section number is the relative distance of the node from the first node in the same side of the conditional. So the values of the section number start from 0. When new nodes are inserted in between existing nodes in the CDG section numbers with non-integer values can be used to avoid renaming all the node labels. The third field contains a pointer to the conditional node at the level indicated by the cell i.e., the i^{th} conditional in the hierarchy for the i^{th} cell.

For example, consider the node B3 in figure 4.3. Its label (T1 F0) can be separated into 2 cells and its position information can be extracted as follows.

Table 4.1 Labels

Cell 1	T	1
	Indicates that the node is in the True side of the first conditional (in this case PROC).	Indicates that the node is the second section on that side
Cell 2	F	0
	Indicates that the node is in the False side of the second conditional (in this case PRED 1).	Indicates that the node is the first section on that side

Note that for conditional nodes (PRED nodes) and group nodes (LOOP nodes) there is no fixed section number for the last cell in the label. Instead these nodes have a range of values for the section numbers. This is provided to accommodate the section numbers of their child nodes. In the case of PRED the child nodes form a new level of hierarchy. Hence the PRED node reserves a value of the section number for each of its child nodes. In our implementation we have made the section numbers of the PRED node to range from (s-0.5) to (s+0.5), where s is the section number reserved by the PRED node for its immediate child nodes. In the case of LOOP nodes, their child nodes do not introduce a new hierarchy. The child nodes belong to the same level of hierarchy as the LOOP node, but they are grouped under the LOOP node. Hence the LOOP node reserves as many section numbers as the number of child nodes it has. In our implementation we have made the section numbers of the PRED node to range from (s+0.5) to (s+n+0.5), where s is the section number of the node immediately preceding the LOOP node and n is the number of immediate

children to the LOOP node. For example, the PRED node PRED1 has a range of T0.5 to T1.5. Its child nodes have a section number of T1 for the corresponding level of hierarchy. Similarly LOOP node LOOP1 has a range of T1.5 to T4.5. It has three immediate children and they each have the section number T2, T3 and T4 from left to right as shown. The main PROC node does not have any label and is at the top level of the hierarchy.

In [NEW 1], the following list of properties is provided for the labels.

- *Use of labels* – represents the location within the program structure locally, which facilitates evaluation of relations, traversal and code motion, especially along multiple paths
- *Every label is unique* – no ambiguity exists among labeled nodes, guarantees that nodes can always be inserted without changing any other labels.
- *Each cell of a label has a pointer to a CDG node* – facilitates the assignment of control dependences and arcs from *group* nodes during CDG construction.
- *Each nodes has a label for each aggregate conditional path that it is on* – facilitates the computation of reach ability and domination, makes certain path properties more easily provable.
- *Beginning and ending labels* – limits the scope of CDG traversal in order to assign labels to newly-inserted region nodes; aids in upwards and downward CDG traversal.

In the case of unstructured code, there is a possibility that a node may have two labels. This is due to the fact that a single node may be the child of two different nodes in the CDG caused by the unstructured nature of the code which allows a node to be dependent on two different nodes. For our case this will not occur as we deal only with structured code. In structured code, each node is dependent on only one other node and hence each node has only one parent. [NEW 1] also describes some algorithms can be used effectively for determining the location details and relative spacing of nodes in a CDG. We will present the details of these methods and how they are used in the algorithm later.

4.4 Movable and Non-movable statements

Code motion is performed at the assembly level. The assembly code generated for the programs would be more or less equivalent to the machine code that is obtained from the assembler. The assembly instructions include many features which depend on the architecture of the processor

under consideration. So while designing the algorithm for optimizing the inter-bit processing time, the architecture of the processor is also considered to some extent.

A major difference in low level code motion and high level code motion is the fact that some instructions in the low level code motion cannot be moved while preserving the semantic correctness of the program. We classify these types of instructions as non movable instructions. Our algorithm des not perform any action on these instructions and no data flow analyses are performed either. When a non movable instruction occurs it is just skipped and the analysis carries on for the next instruction (either in the up direction or the down direction).

We have analyzed the different types of assembly instructions and have arrived at the following list of non-movable instructions.

1. Skip instructions & Branch instructions – (sbrc, jmp, jnez) These instructions decide the flow of control along the program. Moving these instructions would alter the control flow.

2. Conditional instructions – (cmp, tst) Any instruction which defines the flag, which is used by a conditional branch instruction is the conditional instruction. Control flow of the program is decided by these instructions. Moving this type of instructions would affect the control structure of the program.

3. Pointer instructions – (ld, st) These are indirect memory addressing instructions. These instructions cannot be moved as they define or use values which are stored in memory locations, the address of which is stored in registers. Further these instructions also affect the way in which other memory based instructions are moved. As the memory locations addressed by these instructions are not predictable, other memory based instructions could not be moved past these instructions.

4. Flag affected instructions – These are pairs of instructions in which the first instruction defines a flag and the following instruction uses the flag defined by the first instruction. Since the algorithm is based on moving one instruction at a time, this type of instructions are not considered.

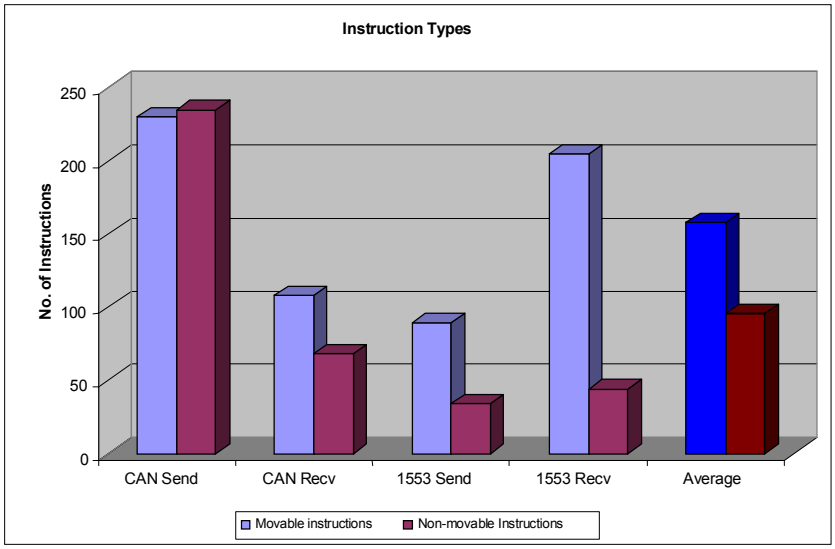


Figure 4.3 Movable Vs Non-movable instructions

The distribution of the non-movable instructions is shown in figure 4.4

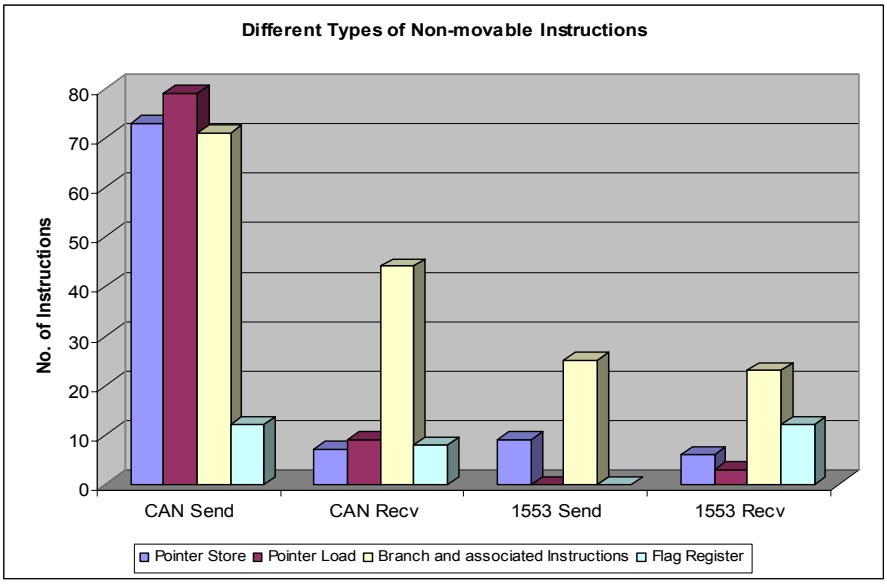


Figure 4.4 Types of Non-movable instructions

We analyzed several different programs and obtained the graph shown in figure 4.3. From the figure we can see that on average; two thirds of the instructions are movable in a program. As the percentage of movable instructions increases, we can obtain a better optimization.

From the figures we can observe the following.

- The number of branch and conditional instructions is in general proportional to the number of instructions present in the program.
- The number of flag affected instructions is very less. In most cases it is a very small negligible percentage of the total number of instructions.
- The number of pointer instructions depends on the structure of the original source code. Here the large number of pointer instructions in the CAN_Send routine is due to the use of "structs" in the original C source code. (refer Addendum 1 for C source code)

Chapter 5

PRELIMINARIES

5.1 Why Code Motion is done at assembly level?

Since the thread integration is performed at a lower assembly level, the timing analysis takes place at the assembly level as well. Another reason for performing timing analysis at the assembly level is the accuracy of the timing analysis as the assembly instruction timings can be predicted accurately. The same would not be possible at a higher language level.

Assembly level is the most appropriate level to perform code motion. As mentioned in chapter 4 most of the relevant research has been either in the higher level or lower level. In the higher level code motion, code movement is performed for expressions and assignments which are redundant or dead at places. The lower level code motion involves the movement of the machine level code to exploit the microprocessor design underneath for extracting some efficiency. Our code motion is different from these two approaches. Our code motion tries to move assignment and other register manipulation instructions at the assembly level to improve the inter-bit processing times that exist when code integration is performed for Bit-Banged communication protocols. This is not a generalized code optimization technique. But rather this is a software thread integration specific code optimization algorithm. This thesis concentrates on explaining the algorithm as we have implemented it for the AVR-RISC processors. But in general this algorithm can be expanded to any RISC processor with some changes in the timing analysis and classification of the instructions that can be moved.

5.2 Why CDGs are used?

Program graphs have been traditionally constructed in two formats, Control Flow Graphs (CFG) and Control Dependence Graphs (CDG). These concepts are explained in chapter 3. As seen in chapter 4, much of the traditional code motion is done using CFGs as are they are straightforward and simple to understand. But traversing through a CFG is not an efficient process and significant additional processing is required compared to a CDG traversal of the same program. Further using

the concept of labels (explained in chapter 3), node identification and search becomes very easy and efficient while using a CDG. Thus we have designed our algorithm to work on the CDG representation of the program. Though for clarity purposes and easy understanding we explain the algorithm in detail using CFGs and explain its implementation on the CDG representation.

5.3 Inter-Bit Timing Analysis

In order to perform code optimization and code motion, we need to determine from which paths code needs to be moved. Code has to be moved from paths that take a longer time to run, to paths which take shorter time to run. In order to identify this we need to perform timing analysis of all the inter-bit paths that exist in the program.

In this section we present a simple yet efficient timing analysis algorithm which we use to identify the run times of the different inter-bit paths that exist in the program. We present a brief discussion of the algorithm here.

Definition 5.1: An inter-bit path is defined as the set of nodes (CODE/LOOP/PRED) between subsequent CALL nodes that can occur in the CDG of a program. By subsequent we mean the subsequence during the run-time of the program.

There may be many inter-bit paths for a given CDG of a program. Some inter-bit paths are traversed many times when the program is executed and some may not be traversed at all. But all the inter-bit paths that are possible have to be taken into account when timing analysis is performed.

Timing analysis gets a little complicated for inter-bit paths as there can be numerous possible ways in which a program flow can take place. For example for the simple CDG shown in the figure 5.1, the number of inter-bit paths possible is 6.

The various paths are

C1 → B1 → P1 → B2→ B4→ B5 → C2
C1 → B1 → P1 → B2→ B4→ B5 → C3
C1 → B1 → P1 → B3→ B4→ B5 → C2

C1 → B1 → P1 → B3→ B4→ B5 → C3

C2 → B6 → B4 → B5 → C2

C2 → B6 → B4 → B5 → C3

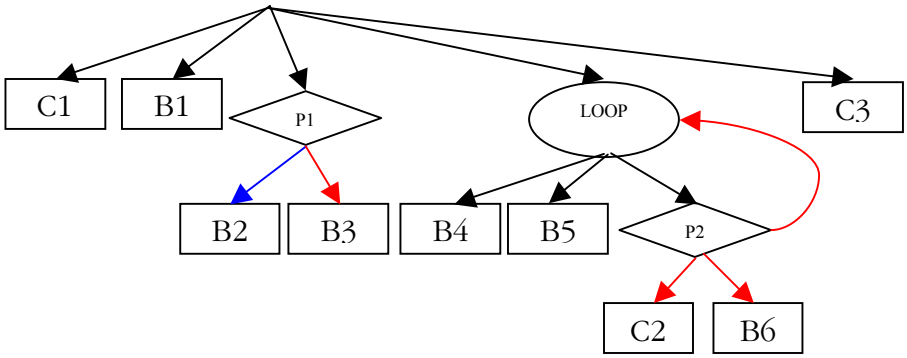


Figure 5.1 Inter-bit Timing Analysis

To simplify the task we do not explore the various paths possible for PRED and LOOP nodes which do not have any CALL nodes below them. In our example we have the PRED node P1 which has two children B2 and B3, but does not have any CALL node. But instead of considering the two possible paths that are available at the PRED node (along either branch), we take just the PRED node as a single node with a finite run-time. This run-time would be the greatest run-time of the branches of the PRED node

In the case of LOOP nodes which do not have any CALL nodes below them, we determine the run-time of the entire LOOP node by using data-flow analysis or directions from the user about the number of times each loop executes. The user can provide this information through the ID file in our implementation (thrint). Based on the number of times a particular loop runs we can calculate its run-time by using simple calculations. Thus after this simplification each PRED or LOOP node that does not have any CALL node below it, would be simplified as a single node with finite run-time.

For our example the simplification reduces the number of paths to 4 and the paths are

C1 → B1 → P1 → B4 → B5 → C3

C1 → B1 → P1 → B4 → B5 → C2

C2 → B6 → B4 → B5 → C2

C2 → B6 → B4 → B5 → C3

5.3.1 Algorithm details

LivePaths – Set that contains the list of paths for which the run-time is being calculated.

AppendNew(pathlist,i) – Appends a new path to pathlist. The new path is a copy of the path number i in pathlist.

AddDuration(pathlist,dur) – Adds dur cycles to the duration of all the paths in pathlist.

RemovePath(pathlist,i) – Removes the path i from pathlist.

Type(node) – Gives the type of node.

NextChild(node,tv) – Gives the next child of node with truth value tv.

LoopReturn(node) – Is true, if node is the LOOP node and is already visited or is being visited.

HasCALL(node) – Is true, if node has a CALL node as one of its descendants

Duration(node) – Duration of node in cycles

SDuration(node) – Duration of the entire subgraph with node as the root

IsLCP(node) – Is true, if node is a loop closing PRED

node – current node (the algorithm is started with node=start)

createnew – indicates whether new paths have to be created (initially false)

5.3.2 Algorithm

The algorithm for computing the various inter-bit paths is as follows

```
Inter-Bit Analysis(node,createnew)
if Type(node) = PROC do
        next= nextchild(node,TV_T)
        while next do
                Inter-BitAnalysis(next)
                next=nextchild(node,TV_T)
else if createnew
        for every path p in LivePaths
                AppendNew(LivePaths,p)
if  LoopReturn(node)
        exit
if Type(node)=CODE
```

```
        AddDuration(LivePaths,Duration(node))
If Type(node)=LOOP
        If not HasCALL(node)
                AddDuration(LivePaths,SDuration(node))
        else
                next=nextchild(node,TV_T)
        while next do
                Inter-BitAnalysis(next)
                next=nextchild(node,TV_T)
        while next do
                Inter-BitAnalysis(next)
                next=nextchild(node,TV_T)
if Type(node)=PRED and IsLCP(node)
        if node called first time
                If not HasCALL(node)
                        AddDuration(LivePaths,SDuration(node))
                Else
                        AddDuration(LivePaths,Duration(node))
                        tv=truth value of first child of node
                        next=nextchild(node,tv)
                        Inter-BitAnalysis(next,true)
                        next=nextchild(node,tv)
                        while next do
                                Inter-BitAnalysis(next)
                                next=nextchild(node,tv)
        else
                AddDuration(LivePaths,Duration(node))
if Type(node)=PRED
        if not HasCALL(node)
                AddDuration(LivePaths,SDuration(node))
        Else
                AddDuration(LivePaths,Duration(node))
                tv=truth value of first child of node
                next=nextchild(node,TV_T)
                Inter-BitAnalysis(next,true)
                next=nextchild(node,tv)
                while next do
                Inter-BitAnalysis(next)
                        next=nextchild(node,TV_T)
```

```
            while next do
                    Inter-BitAnalysis(next)
                    next=nextchild(node,TV_F)

if Type(node)=CALL
        for every path p in LivePaths ending in node
                Remove(LivePaths,p)
```

5.4 Code Motion Algorithm Specifics

5.4.1 Predicates Used

Two predicates IS-DEFINED(*i,r*) and IS-USED(*i,r*) are used in the algorithm

IS-DEFINED(*i,r*) – indicates whether the instruction *i* defines the value in the register *r.* In case *r* denotes a register set, this indicates whether instruction *i*, defines at least one of the registers in the register set *r*.
IS-USED(*i,r*) – indicates whether the instruction *i* uses the value in the register *r.* In case *r* denotes a register set, this indicates whether instruction *i*, uses at least one of the registers in the register set *r*.

Two sets USED(i) and DEFINED (i) are used

USED(i) – holds the registers whose values are used by the instruction i.
DEFINED (i) – holds the registers whose values are defined by the instruction i.

The D-SAFE and U-SAFE predicates (to be described later) can hold any of the four values R_FALSE, R_TRUE, C_FALSE and C_TRUE. These four values are defined as follows:

1. NONE: Indicates that the predicate has not yet been computed.
2. C_FALSE: Indicates that code could be *possibly* not moved to the node.
3. C_TRUE: Indicates that code can be *possibly* moved to the node.
4. R_FALSE: Indicates that code cannot be moved into the node.
5. R_TRUE: Indicates that code can be moved into the node.

The “C_” prefixed values are used for nodes, for which it is exactly not known at the time, whether code motion could or could not be performed. These values are also used for nodes into which we would not be moving our code in general for the current data flow analysis. For example when we are trying to perform a code hoist from a node A and another node B always follows A, we would not be considering B to move the code from A. these values are also used when processing loop nodes as some complexity is involved when performing data-flow analysis for them. We will explain more about this in later chapters. The “R_” prefixed values are used for nodes which can be potentially candidates for code motion, as we want to know exactly whether code can be moved or cannot be moved to these nodes.

The operations for these predicates are similar to the Boolean operations performed in other code motion algorithms, though slightly different. We define here the operations that can be performed using these predicate values and the results of the operations. The operations are the same as the Boolean operation when

- The operand values have only C prefixes or no prefixes
- The operand values have only R prefixes or no prefixes

The result in the above two cases can be obtained by removing the prefixes from the operand values and performing normal Boolean operations. The result is then prefixed with the prefix of the operands.

AND operation: The AND operation is similar to the Boolean AND operation. If the operands have a mix of both C prefixed and R prefixed values, the AND operation is defined as follows:

Table 5.1 AND Operation

A	**B**	**A AND B**
R_TRUE	C_TRUE	R_TRUE
R_TRUE	C_FALSE	R_FALSE
R_FALSE	C_TRUE	R_FALSE
R_FALSE	C_FALSE	R_FALSE

OR operation: The OR operation is similar to the Boolean OR operation. If the operands have a mix of both C prefixed and R prefixed values, the OR operation is defined as follows:

Table 5.3 OR Operation

A	B	A OR B
R_TRUE	C_TRUE	R_TRUE
R_TRUE	C_FALSE	R_TRUE
R_FALSE	C_TRUE	R_TRUE
R_FALSE	C_FALSE	R_FALSE

For CFGs, *start* and *end* denote the start node and end node respectively unless specified otherwise. The start node does not have any predecessors and the end node does not have any successors. The programs as already mentioned, are assumed to be structured code and there should not exist any multiple dependencies.

Instr(*n*) denotes the set of instructions in the node *n*. **PrevInstr**(*i,n*) denotes the set of all the instructions that occur before the instruction *i*, in the node *n*.

5.4.2 Labels

Labels assumed to have been assigned to each node in the CDG. The functions that are provided by the labels are:

Before(*node1,node2*) : returns true if *node2* precedes *node1* in the control flow sequence.
After(*node1,node2*) : returns true if *node2* follows *node1* in the control flow sequence.
Locate(*labels*): returns the node *n* with label *labels* if present in the CDG
FirstDifference(*node1,node2*): returns the highest level at which the labels of the two nodes difference, i.e., the level below their common ancestor.

5.4.3 Dummy nodes

Dummy nodes are inserted into the CDG at various places. These are CODE nodes with no instructions. Dummy nodes help in simplifying the data flow analysis of the CDG and also act as place holders for code placement during code motion.

Dummy nodes are inserted at the following places in the CDG:

For PROC node: For PROC nodes, a single dummy node is inserted as the last child of the PROC node. This node acts as a place holder while moving instructions down. This node is names as proc_EN, where proc is the name of the PROC node and EN stands for END.

For CALL node: For CALL nodes, dummy nodes are inserted before and after each CALL node. These nodes mainly serve are place holders during code placement to hold instruction that are moved. These nodes are names name_CP and name_CS, where name is the name of the CALL node and CP stands for CALL predecessor and CS stands for CALL successor.

For PRED node: For PRED nodes, dummy nodes are inserted one each on either branch of the PRED. This is to make sure that there exists at least one node into which code can be moved while code placement is done in one of the branches. These dummy nodes are named name_P1 and name_P2 for either branch and are included as the last children of the branches (name is the name of the PRED node).

For LOOP node: For LOOP nodes, two dummy nodes are inserted. One is inserted before the LOOP node. This is named as name_PR. Another dummy node is inserted as the last child of the loop closing PRED for the LOOP node. This node is names name_PD

Chapter 6

UP-SAFETY AND DOWN-SAFETY

6.1 Introduction

In this chapter we explain two main concepts behind our code motion algorithm. The concepts are "safety of nodes" and "set of maximal nodes". The "safety of nodes" determines whether a node is safe for the motion of code to that node from another node. The "set of maximal nodes" is the set of all *safe* nodes which are at the greatest distance from the node from which the code is moved. These two concepts are used to decide whether a node is safe for code motion. When those set of safe nodes are determined, it is also possible for us to identify how far the code can be moved either along the program flow (code sinking) or against the program flow (code hoisting). Semantic correctness which is used to verify whether a program transformation changes the behavior of a program is defined in the next section. The subsequent sections describe the two concepts in terms of code hoisting and code sinking.

6.2 Semantic Correctness

As code hoisting or code sinking is performed in a program we have to consider the semantic correctness of the program due to code motion. The program's behavior should not change by the movement of a piece of code from a node *A* to another node *B*. To verify the correct operation of the program is preserved during code motion, we define semantic correctness of a program.

Definition 6.1: A program is said to be semantically correct after the movement of an instruction i from node A to node B if node B is DOWN-SAFE for code hoisting and UP-SAFE for code sinking.

The above definition holds true for individual instruction code motions and not for a block of instructions. The above definition is also restricted to only movable statements as described in chapter 4.

6.3 Code Hoisting

Code hoisting is the motion of code in the upward direction, i.e., against the control flow of the program. As we move code in the *upward* direction, we have to consider the semantic correctness of the program when we move code. We define the predicate D-SAFE and *down-safety* for code hoisting

Down-safety is defined for code hoisting as follows:

Definition 6.2: A code hoist of instruction i, from node A to node B is said to be down-safe iff the execution of i at B does not introduce a new value along any path between B to the exit node.

For an instruction *i*, we define two predicates for each node *n* in the program. They are DSAFE-IN(*i,n*) which is the *D-SAFEness* for the instruction *i* at the beginning of the node *n* and DSAFE-OUT(*i,n*) which is the *D-SAFEness* for the instruction *i* at the end of the node *n*. These two predicates cannot be defined easily in words. Hence we present a recursive definition of these predicates as follows.

Equation 6.1

$$\textbf{DSAFE-OUT}(i,n) = \begin{cases} \text{C\_TRUE} & \text{if } n \text{ is the } \textit{EXIT} \text{ node} \\ \prod_{m \,\varepsilon\, succ(n)} \textbf{D-SAFE-IN}(i,n) & \text{otherwise} \end{cases}$$

Equation 6.2

$$\textbf{DSAFE-IN}(i,n) = \begin{cases} \text{R\_TRUE} & \text{if } i \text{ is present in node } n \\ \textbf{TRANSP\_R}(i,n) \;\Lambda\; \textbf{DSAFE-OUT}(i,n) & \text{if } \textbf{DSAFE-OUT}(i,n) = \text{R\_TRUE or R\_FALSE} \\ \textbf{TRANSP\_C}(i,n) \;\textbf{V}\; \textbf{MASK}(i,n)) \;\Lambda\; \textbf{DSAFE-OUT}(i,n) & \\ & \text{, otherwise} \end{cases}$$

Equation 6.3

$$\textbf{TRANSP\_R}(i,n) = \prod_{j \,\varepsilon\, \textbf{Instr}(n)} (\sim (\textbf{IS-DEFINED}(j, \textbf{USED}(i)) \;\Lambda\; \textbf{IS-USED}(j, \textbf{DEFINED}(i)) \;\Lambda\; \textbf{IS-DEFINED}(j, \textbf{DEFINED}(i)))$$

Equation 6.4

$$\textbf{TRANSP\_C}(i,n) = \prod_{j \,\varepsilon\, \textbf{Instr}(n)} \sim \textbf{IS-USED}(j, \textbf{DEFINED}(i))$$

Equation 6.5

$$\mathbf{MASK}(i,n) = \begin{cases} \text{TRUE} & \text{if } \exists j \in \mathbf{Instr}(n) \mid \mathbf{IS\text{-}DEFINED}(j, \mathbf{USED}(i)) \wedge \\ & \quad \forall_{k \in \mathbf{PrevInstr}(j,n)} (\mathbf{IS\text{-}USED}(k, \mathbf{DEFINED}(i)) = \text{FALSE}) \\ \text{FALSE} & \text{,otherwise} \end{cases}$$

TRANSP_R and TRANSP_C are transparency conditions for the nodes. Note that the transparencies are different for nodes which have "C_" prefixed values and which have "R_" prefixed values. Equation 6.5 denotes the case in which a new definition of the variable defined in the instruction *i,* is present in the node *n*. The mask condition facilitates the movement of a definition along a path, when another path diverging from the path has a definition of the variable, before its use. This is illustrated in figure 6.1. Use of the mask predicate enables us to move the instruction 1 to node A even though the register defined by 1 is defined along the other path.

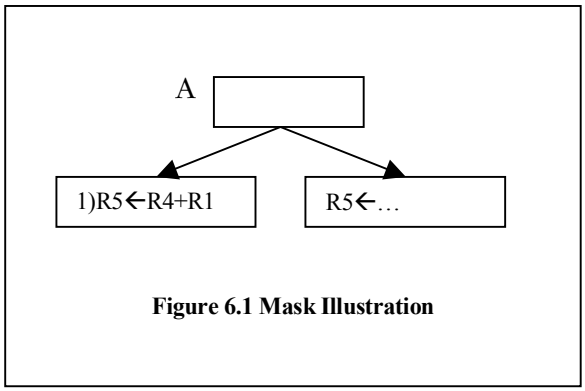


Figure 6.1 Mask Illustration

These definitions hold true only for programs without any loops, i.e., programs which do not have any back edges. Note that the operations are as defined in the previous chapter and not regular Boolean operations.

We have defined the predicates IS-DEFINED, IS-USED and the sets USED, DEFINED, Instr and PrevInstr in chapter 5. The above equations are explained in the context of CFGs. But as explained in chapter 5 the code motion algorithm is developed to work on Control Dependent Graphs and not Control Flow Graphs. For the straightforward non-loop equations 6.1 to 6.3, the corresponding CDG equivalents can be obtained by simply replacing the *succ(n)* by the *next(i), next(i)* being the next node in the CDG when the CDG is traversed in a post-order fashion. The post-order traversal can be performed using the following algorithm. Note that since there are no loops in the program the CDG will have only START, CODE and PRED.

Post-order algorithm:

1) Begin with the start (top level) node.

2) For every node do
   a) Visit the first child if present
   b) Visit the next sibling if present
   c) Process the node

For handling loops, the equations 6.1 to 6.3 can be used for a CFG of the program. However, the straight forward algorithm explained earlier has to be changed to handle the loops. Loops in programs are characterized by back-edges in the CFG of the program. So when the recursive equations 6.2 through 6.3 are applied, we would have predicates on the right side which are not yet computed. For example consider the code segment shown in figure 6.2. The blocks of code are processed from the *end* node to the *start* node. The code blocks in the figure would then be processed D first, then B and finally C or A depending on the algorithm. Note that when we reach B, in order to find its D-SAFE predicate values, we need the D-SAFE predicates of its successors to be computed. But block C which is one of the successors is not yet processed.

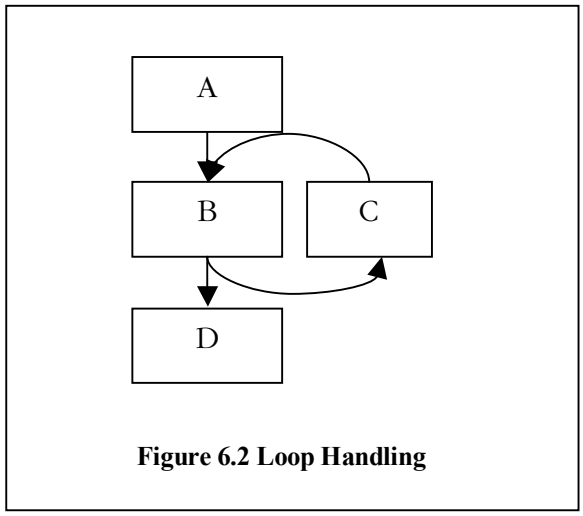


Figure 6.2 Loop Handling

In order to overcome this type of situation, in [KRS 1], the authors assume that the predicate values of the node that has not been visited as *true* in their algorithm. This is referred as "the maximum solution of the set of equations". This concept works fine for moving just computations. But for our case where the whole instructions are moved, this solution would give wrong values for the predicates, resulting in code motion that will generate programs which are not semantically correct.

Our solution to overcome this problem is to assume a structure as shown in the figure 6.3. The block C is removed and is replaced by two new blocks which contain the same code as block C. the new blocks are named C(1) and C(2). Now when B has to be processed and its predicates determined, its successors D and C(2) would have been processed already. Note C(2) does not have any successor. Similarly when the node C(1) is processed B would have been computed. Thus we have eliminated the back edges from the CFG.

When using a CDG, the above assumption can be embedded in the algorithm. This is done by handling the LOOP nodes differently when encountered. When a LOOP node is encountered, we should start processing from the loop closing PRED node. Since the loop closing PRED node has children which are not processed yet, they would be called for processing (post order algorithm). This step is similar to having C(2) be processed before B. Then all the children of the LOOP node get processed from right to left excluding the LCP. Then the LOOP node is processed. After the LOOP node is processed, we process all the children of the loop closing PRED node from right to

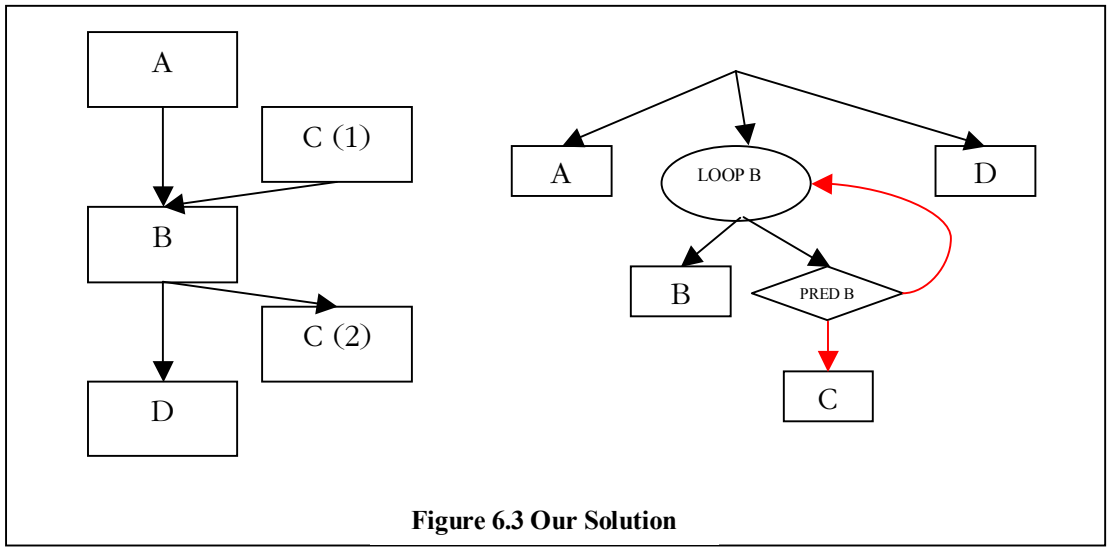


Figure 6.3 Our Solution

left. This step is similar to having C(1) be processed after B.

6.4 Code Sinking

Code sinking is the motion of code in the downward direction, i.e., along the control flow of the program. Similar to *upward* code motion, as we move code in the *downward* direction, we have to consider the semantic correctness of the program. We define the predicate U-SAFE and *up-safety* for code sinking

Up-safety is defined for code hoisting as follows:

Definition 6.2: A code sink of instruction i, from node A to node B is said to be down-safe iff the execution of i at B does not introduce a new value along any path between the start node and B.

Two predicates are defined for each node *n* in the program. They are USAFE-IN(*i,n*) which is the *U-SAFEness* for the instruction *i* at the beginning of the node *n* and USAFE-OUT(*i,n*) which is the *U-SAFEness* for the instruction *i* at the end of the node *n*. The recursive definition of these predicates is as follows.

Equation 6.6

$$\textbf{USAFE-IN}(i,n) = \begin{cases} \text{FALSE} & \text{if } n \text{ is the } \textit{START} \text{ node} \\ \prod_{m \in pred(b)} \textbf{USAFE-OUT}(i,n) & \text{otherwise} \end{cases}$$

Equation 6.7

$$\textbf{USAFE-OUT}(i,n) = \begin{cases} \text{TRUE} & \text{if } i \text{ is present in node } n \\ \textbf{TRANSP\_R}(i,n) \ \Lambda \ \textbf{USAFE-IN}(i,n) & \\ & \text{if USAFE-IN(i,}n\text{)=R\_TRUE} \\ \textbf{TRANSP\_C}(i,n) \ \Lambda \ \textbf{USAFE-IN}(i,n) & \\ & \text{if USAFE-IN(i,}n\text{)=C\_TRUE} \\ \textbf{USAFE-IN}(i,n) & \\ & \text{,otherwise} \end{cases}$$

Equation 6.8

$$\textbf{TRANSP\_R}(i,n) = \prod_{j \in instr(n)} (\sim (\textbf{IS-DEFINED}(j, \textbf{USED}(i)) \ \Lambda \ \textbf{IS-USED}(j, \textbf{DEFINED}(i)) \ \Lambda \ \textbf{IS-USED}(j, \textbf{DEFINED}(i)))$$

Equation 6.9

$$\textbf{TRANSP\_C}(i,n) = \prod_{j \in instr(n)} (\sim (\textbf{IS-DEFINED}(j, \textbf{USED}(i)) \ \Lambda \ \textbf{IS-DEFINED}(j, \textbf{DEFINED}(i)))$$

These definitions hold true only for programs without any loops, i.e., programs which do not have any back edges. Note that the operations are as defined in the previous chapter and not regular Boolean operations.

Note the definitions of TRANSP are different from that of the DSAFE definitions. This is due to the change in the code motion direction. In the case of code sinking the data flow analysis for the

CFG is performed in the reverse direction as the recursive definitions 6.6 and 6.7 suggest. This along the flow analysis in the CDG representation can be easily achieved by using the pre-order traversal.

The pre-order traversal algorithm is as follows:

1) Begin with the start (top level) node.
2) For every node do
    a) Process the node
    b) Visit the first child if present
    c) Visit the next sibling if present.

Loops are handled in a way similar to that explained for computing DSAFE. The code structure is assumed to be as shown in the figure 6.3. When a LOOP node is encountered, the processing takes place as follows

- The child nodes of the LCP are processed.
- The child nodes of the LOOP are processed next.
- The child nodes of the LCP are processed again.

6.5 Other Details

Now we have completed computing the DSAFE and USAFE predicates for each node for a given instruction *i*. All the nodes that have their predicate values as R_TRUE are candidates for code motion, i.e., it is possible for us to move code into those code points without any additional data flow analysis. Note at this point though that these nodes are only candidates for code motion and are not the final code motion points. We still have to remove nodes which would lead to semantically incorrect nodes.

Since we have used a very aggressive data flow analysis technique for computing our code motion points, some of the points would still be unsafe for code motion. These unsafe points are introduced by the way we handled the LOOP nodes. The problem is caused by self dependent nodes and the presence of LOOP structures. This is illustrated in figure 6.3.

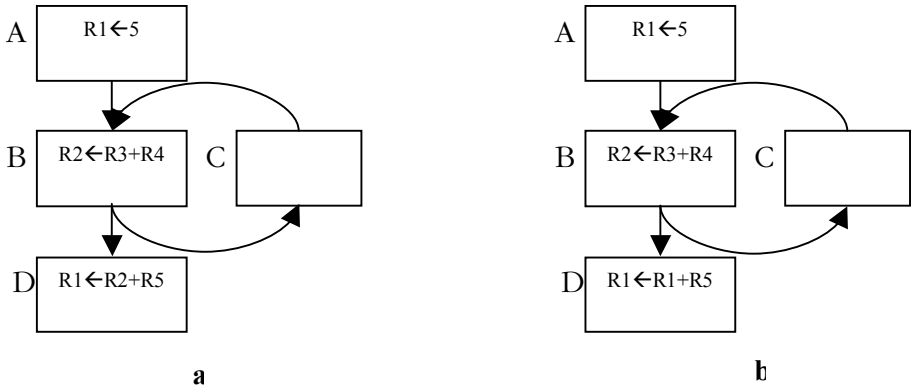


Figure 6.4 Loop Problem due to Self Dependent Instructions

Consider the code structure as shown in figure 6.4(a). For the instruction in block D, our code analysis gives a safe value for block B as there is no use of register R1 in block C or block B. This would be perfectly fine and we could move the instruction from block B to anywhere in block B after the instruction R2←R3+R4, which defines R2. This code motion is semantically right and we can go ahead with the motion if it improves the performance. On the other hand, consider the code structure shown in figure 6.4(b). For the instruction in block D, our code analysis again gives a safe value for the node B as there is no use of the register R1 in block C and block B. But if we move the instruction to block B, the value of R1 is incremented each time the loop executes as it is self dependent. In our code analysis we have considered only the situation in which other instructions either define or use the registers defined or used by the instruction which we were moving. We have not considered the effect of an instruction on itself during code motion. Syntactic code motion, explained in chapter 4, does not need to consider this situation as it moves only computations and the computation results are stored in temporary variables. Thus code placement in syntactic code motion always results in movement of code which is not self dependent. Further syntactic code motion tries to avoid moving code into loops as this would in fact deter performance rather than improving it. We present a simple definition of self dependent instructions at this point.

Definition 6.4: An instruction i is said to be self dependent if it contains a register r which is both used and defined by the instruction i, i.e., $\boldsymbol{IS\_DEFINED}(i,r) = TRUE$ *and* $r \in \boldsymbol{USED}(i)$.

To solve the above problem, we introduce a slight modification to the code analysis that we have explained in the previous sections. Since this problem is caused by LOOP nodes and self-dependent instructions, we can solve this problem by handling the LOOP nodes slightly differently, when data flow analysis is performed for self dependent instructions. When data-flow analysis is done for an instruction *i*, which is self dependent, the safety values of the LOOP nodes are either inherited from its child node or the previous node depending on the direction of the dataflow.

Our Solution: In the case of DSAFEness, the DSAFE values are obtained from the DSAFE_IN value of the first child CODE node of the LOOP. The inclusion of dummy nodes guarantees that there exists such a CODE node. In the case of USAFEness, the USAFE values of the LOOP nodes are obtained from their respective immediate sibling CODE nodes. Again the insertion of the dummy nodes guarantees that such a CODE node exists.

Thus we have eliminated the possibility of the LOOP node giving a safe value for code motion, if code motion is possible only to the inside of the LOOP structure. We will explain this concept more when we compute the "set of maximal nodes" in the next section.

6.6 UpNodes and DownNodes

In this section we define the sets of UpNodes and DownNodes for instructions which have been referred so far as the "sets of maximal nodes". These sets help to determine how far instructions can be moved in the program. Determining how far instructions can be moved helps in deciding the optimal placement of the instructions based on the various inter-bit processing times associated with each node. The process of computing inter-bit processing times was explained in chapter 5. These sets are computed for every instruction considered for motion after the data flow analysis step.

6.6.1 UpNodes

UpNodes(*i*) is the set of nodes which have their DSAFE_IN values as R_TRUE for an instruction *i* and are at maximal distance from the node *n* containing the instruction. These nodes are in the direction that is against the flow of the program. All nodes along the paths from UpNodes(*i*) to the node *n*, including the nodes in UpNodes(*i*) are candidates for code placement during code hoisting. We give a formal definition of the set UpNodes(*i*) as follows:

Definition 6.5: UpNodes(i) is a set of nodes such that for any node $k \in$ *UpNodes(i), DSAFE_IN(k)=R_TRUE and* $\exists\ j \in$ *prev(k) such that DSAFE_OUT(j) = R_FALSE.*

We use the above definition in our algorithm to compute the set of UpNodes.

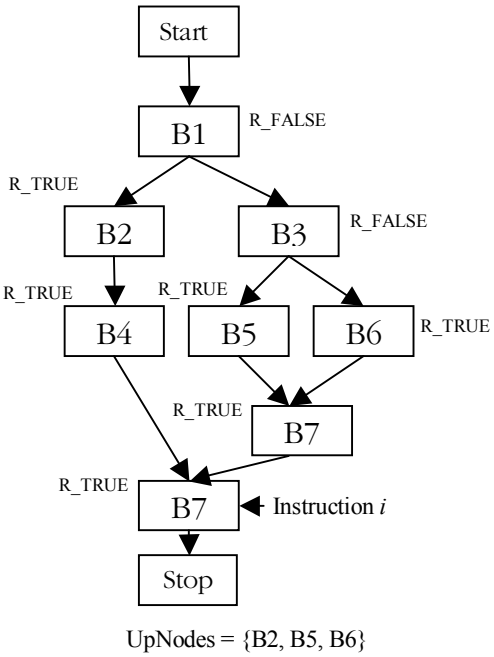


Figure 6.5 UpNodes

The concept is further illustrated by figure 6.5. The instruction *I* considered for code motion is present in the node B7. The values of the DSAFE_IN(*i*) predicate for each node is shown beside the node. For computing the UpNodes in the CDG, we can simply do a traversal of the CDG to determine the UpNodes. Some special processing is required while processing nodes which follow PRED nodes where the number of previous nodes to be considered is more than 1. These nodes can be easily identified, as they would be the *dummy* nodes that we inserted following each PRED node.

6.6.2 DownNodes

DownNodes(*i*) is the set of nodes which have their USAFE_OUT values as R_TRUE for an instruction *i* and are at maximal distance from the node *n* containing the instruction. These nodes are in the direction that is along the flow of the program. All nodes along the paths from DownNodes(*i*) to the node *n*, including the nodes in DownNodes(*i*) are candidates for code placement during code sinking. We give a formal definition of the set DownNodes(*i*) as follows:

Definition 6.5: DownNodes(i) is a set of nodes such that for any node $k \in$ *DownNodes(i), USAFE_OUT(k)=R_TRUE and* $\exists\ j \in$ *succ(k) such that USAFE_IN(j) = R_FALSE.*

Similar to UpNodes, we use the above definition in our algorithm to compute the set of DownNodes. The concept is further illustrated by figure 6.6. The instruction *i* considered for code motion is present in the node B1. The values of the USAFE_OUT(*i*) predicate for each node is shown beside the node. For computing the DownNodes in the CDG, we can simply do a traversal of the CDG to determine the UpNodes. Some special processing is required while processing PRED nodes where the number of successor nodes to be considered is more than 1.

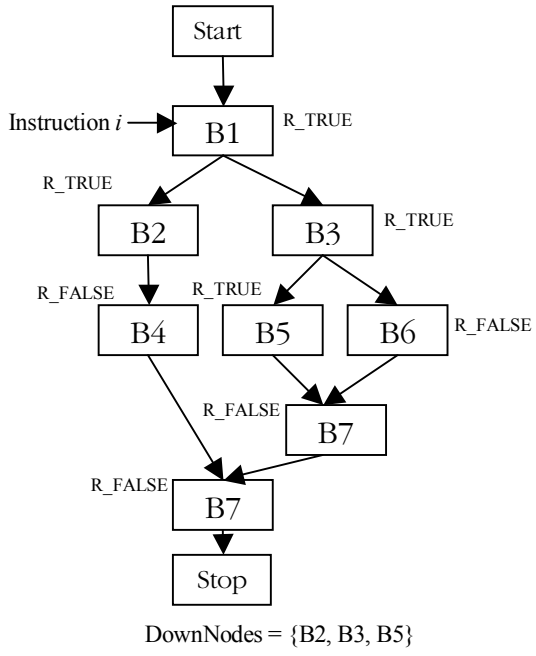


Figure 6.6 DownNodes

Chapter 7

CODE MOTION POINTS

7.1 Introduction

This chapter explains the process of obtaining insertion and deletion points while moving an instruction. We assume at this point that we have decided to delete the instruction *i* at the node n and move it to the node m. The actual process of determining the instruction *i* is deferred until the explanation of the detailed algorithm in the next chapter. Similarly, the process of how we determined node m as the destination of the code motion is also deferred until the next chapter. Insertion points is the set of all nodes into which a copy of the instruction *i* has to be placed for code motion. Deletion points is the set of all nodes from which the instruction *i*, if present has to be deleted. We also discuss the concept of parallel paths in conjunction with insertion points. For obtaining the insertion points using CDGs we use parents of UpNodes and DownNodes which are also explained in this chapter.

7.2 Insertion Points

As explained earlier insertion points is the set of all the nodes into which a copy of the instruction *i* has to be placed for code motion. Since we have already decided the point to which we are going to move the instruction, i.e., the destination node m, we can easily determine the insertion points associated with the node m. Now the question arises as to why we need to compute other insertion points once we have determined that node m is going to be the destination node. The answer is illustrated in figure 7.1. In this case the instruction 1 is moved from the block D to the block B. This code motion would affect the semantic correctness of the program. Assume the case in which the control flows through A→C→D. The value of R1 which has to be computed before the instruction 2 executed is not calculated is not done as the program did not execute the code present in node B. so in order to keep the semantic correctness of the program, when we insert an instruction *i*, we have to consider all the paths along which control can flow through.

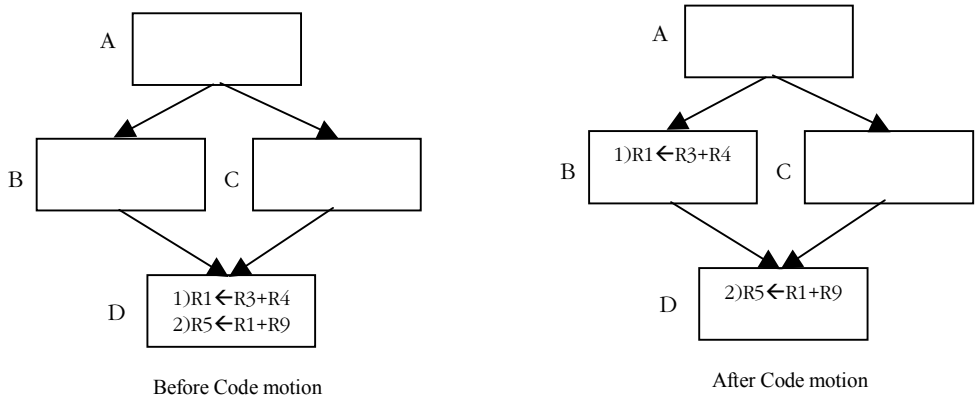


Figure 7.1 Insertion points

This can be formally defined as

Definition 7.1: The set of insertion points associated with a node m when code hoisting moves an instruction from another node n is given in IP{m}, such that for any path p from Start node to the node n, ∃ k ∈ IP{m} which is a node in the path p. Note: m is also in IP{m}.

According to the above definition, for our example IP{B} ={B, C}. Hence a copy of instruction *i* is placed in the code blocks B and C. Now the code is semantically correct and which ever way the control of the program flows, the instruction 1 would be executed before instruction 2. The same holds for code sinking except that the definition is a little different as follows

Definition 7.2: The set of insertion points associated with a node m when code sinking moves an instruction from another node n is given in IP{m}, such that for any path p from node n to the end node ∃ k ∈ IP{m} which is a node in the path p. Note: m is also in IP{m}.

7.3 Deletion Points

The set of deletion points is corollary to insertion points. It is the set of nodes from which the instruction *i*, if present is removed. This requirement arises as an effect of moving code into more than one node as explained in the previous section. When an instruction which was already subjected to a code motion is further moved, we need to delete the instruction, if necessary from all

the nodes into which we placed it in the earlier code motion. So the set of deletion points can be defined as follows:

Definition 7.3: The set of deletion points associated with the node n when code sinking moves an instruction i to another node m is given by DP{n}, such that for every node $k \in DP\{n\}$ *is a node along a path p from the start node the node n and k has the instruction i.*

A similar definition for the deletion points can be given for code hoisting as follows

Definition 7.4: The set of deletion points associated with the node n when code hoisting moves an instruction i to another node m is given by DP{n}, such that for every node $k \in DP\{n\}$ *is a node along a path p from the node n to the end node and k has the instruction i.*

Once the insertion and deletion points have been computed using the above definitions for the code motion of the instruction *i*, all that remains to be done is deleting the instances of the instruction *i* from the deletion points and inserting a copy of the instruction *i* in the insertion nodes. For inserting however, the insertion takes place at the beginning of the insertion nodes when code hoisting is performed and at the end of the end of the insertion nodes for code sinking.

7.4 Parallel Paths Nodes

Loops result in additional insertion and deletion points during code motion. The additional insertion points introduced by the loops are explained in figure 7.2. In figure 7.2(a), a code hoist is performed with the instruction 1 in the code block G. The instructions which are marked by letters are copies of the instruction 1 after code motion was performed. This code is perfectly fine and is semantically correct. But consider the same code motion performed in the code structure shown in figure 7.2(b). Our data flow analysis gives R_TRUE for the nodes B,C and D in the CDG. Now, if we move the instructions based on the definition we discussed above, we would end up with the same code placement as 7.2(a). This placement however is wrong, because in the case the code flows through code D, the value of one of the operands of the instruction is changed and this would not be reflected in the value defined by the instruction. This type of situation occurs only for code hoisting and not for code sinking as we adopted different conditions of transparencies for the nodes as explained in the previous chapter.

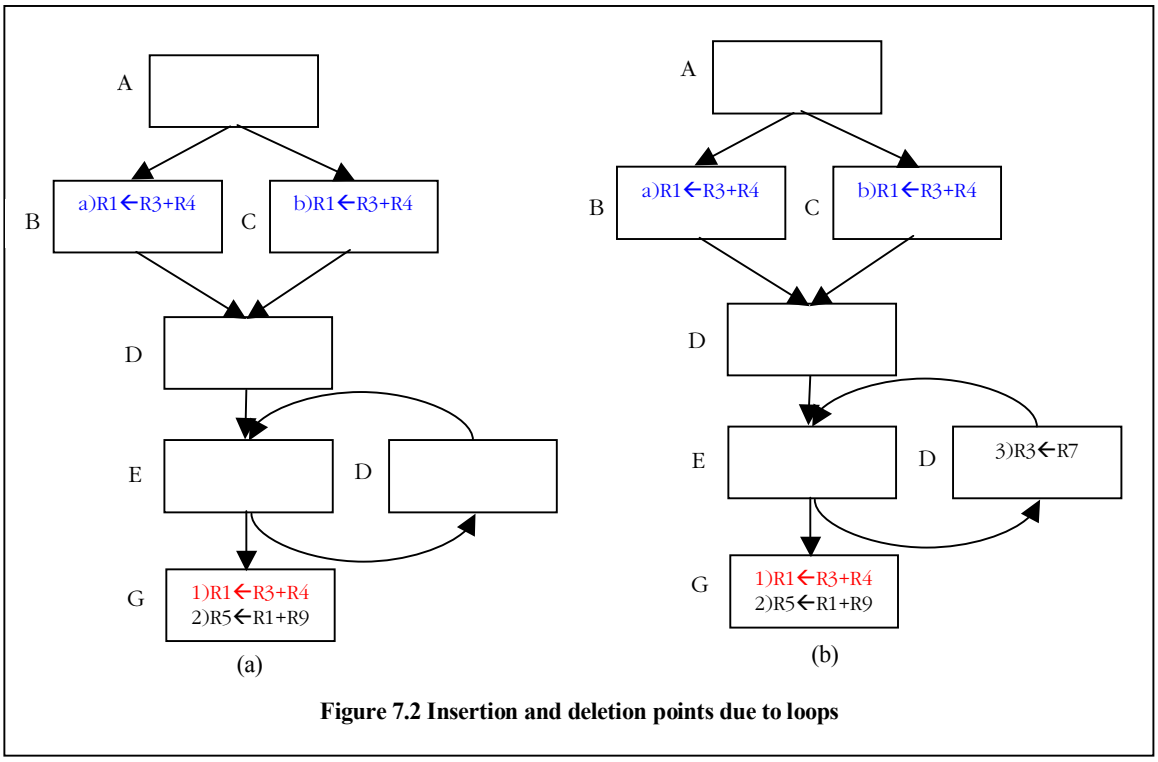


Figure 7.2 Insertion and deletion points due to loops

To overcome this problem, we define *parallel paths nodes* as follows:

Definition 7.5: Parallel path nodes for a code hoist of i from a node n to a node m, are nodes which are in the set of UpNodes and which occur after the insertion point under consideration. Mathematically a parallel path node n can be given as $n \in UpNodes(i)$ *and* $After(n,m)$ *is true.*

In the same way, if an insertion point is the node D, we need to make sure that the value is available when the control for the program does not flow though the node D. If we did not perform insertion of the instruction *i* anywhere else, the instruction 1 would not be executed at all. This situation can be easily identified in the CDG structure of the program, as the node which we are considering for insertion would be the descendant of a loop closing PRED node, which has to be handles specially. Thus when insertion is performed in a loop closing PRED node, the CODE node immediately preceding the corresponding LOOP node will become the *parallel path node*. This condition could occur both in code hoist and code sink. The insertion of *dummy* node before the LOOP node guarantees that there is a code into which can be used as the insertion point. For code sink however, the *parallel path node* would be the code node immediately following the LOOP node.

7.5 Redundant Nodes

Redundant nodes are introduced when repeated code motion is performed on a single instruction. When code motion performs code hoist and code sink on a single instruction *i*, we would have nodes in the *code motion region*, which have a copy of the instruction being moved. We define *code motion region* as follows:

Definition 7.6: A code motion region for a particular code motion is given as a set of nodes CMR, such that $\forall\ j \in CMR$ *,*

- *j is a node in some path between IP{n} and DP{m}for code hoist*
- *j is a node in some path between DP{m} and IP{n} for code sink*

The set excludes nodes in DP{m}.

It is semantically incorrect to have nodes within the *code motion region* that have a copy of the instruction being moved. These nodes are called redundant nodes. Redundant nodes if present have to be detected and the copies of the instruction present in them have to be deleted. Redundant nodes can be easily computed using the CDG representation and labels. All nodes within the *code motion region* have labels which lie between the insertion and deletion points as suggested by definition 7.6. Thus we can investigate all the nodes in the insertion region by incrementing the labels and checking if the node with that label has a copy of the instruction *i*.

Note, if data flow analysis is performed without taking redundant nodes into consideration, we would get results which are affected by the copies of the same instruction due to earlier code motion. In order to prevent this we number all the instructions in the program initially. When data flow analysis is performed for an instruction with number *r* and we encounter another copy of the instruction with the same number, we skip the instruction and move on to the next one.

7.6 Implementing code motion points in CDG

The algorithms for obtaining the code motion points for the CDG representation of the program are explained in this section. The CDG representation and the functionality provided by the introduction of labels make it easier for computing the code motion points. To take advantage of

the CDG representation while computing the code motion points, we determine the parents of the UpNode or DownNode which we are considering for code motion. We have already mentioned that UpNodes occur for code hoisting and DownNodes occur for code sinking. Let *m* be the node from which code is moved to the node *n*. The parents of these two nodes are computed relative to each other and depend on the levels at which these nodes appear in the CDG. Note that relative parents are defined for pairs of nodes and not for a single node.

For a code hoist or a code sink, the relative parents of the nodes *m* and node *n* can be defined as follows and is illustrates in figure 7.3.:

Definition 7.6: For the code motion of an instruction i into a node n moved from a node m, the relative parents of m and n are their respective ancestors, which are same or are siblings at the lowest level in the CDG.

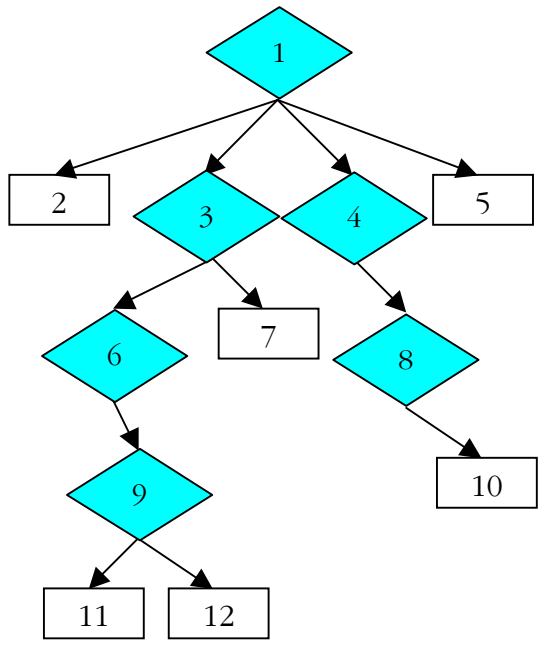


Figure 7.3 Relative Parents

For example the relative parents of nodes 12 and 10 are 3 and 4 respectively as they are the parents at the lowest level which are siblings. The relative parent for the nodes 11 and 7 is the same node, node 3.

Determining the parent nodes enables us to easily compute the insertion points. The insertion points for a code hoist include the current node *n*, into which we are considering to move our code and the last nodes of all the other branches beneath the parent node. From our discussions so far it should be obvious that only PRED nodes in the CDG accommodate the different braches of the program. Using the above two conditions, we can easily compute the insertion points by the following algorithm:

Obtain Insertion Points for code hoist:

set parent = *n*
set *p* = relative parent of node *n* (for node *m*)
while *parent* is not *p* do
  parent = parent(*parent*)
  if *parent* is of type PRED and not *LCP*
    include the last child in the other branch in *IP{n}*

For a code sink the above algorithm would find the first node in the other branch of the PRED node, instead of the last node. The insertion of the *dummy* nodes on either branches of a PRED node guarantees that we would have a CODE node for insertion.

The above algorithm can also be extended for computing the deletion points in the same way for node *m*. Though in the case of deletion points instead of directly including the last node in the other branch for the PRED node, we search the entire other branch for a node which contains an instance of the instruction *i*, we are considering for motion. If we find such a node, it should be included in *DP{m}*. This applies for both code hoist and code sink.

To sum it all up, the various points into which code has to be placed and deleted for the various types of code motion is given in the following table:

Table 7.1 Insertion and Deletion points

Type of code motion	Points where code is placed	Points where code is deleted
Code hoist into a node which is a descendant of an LCP	IP{n} and Parallel Path Nodes	DP{m} and Redundant Nodes
Code hoist into other nodes	IP{n} and Parallel Path Nodes	DP{m} and Redundant Nodes
Code sink into a node which is a descendant of an LCP	IP{n} and Parallel Path Nodes	DP{m} and Redundant Nodes
Code sink into other nodes	IP{n}	DP{m} and Redundant Nodes

Chapter 8

COMPLETE ALGORITHM

8.1 Introduction

In this chapter, we discuss the complete code motion algorithm to achieve optimum inter-bit processing time. So far we have discussed all the details that go into the various aspects of the algorithm. This chapter consolidates all those details and presents the underlying algorithm.

8.2 Instruction and Node Selection

The basic idea of our algorithm is to distribute the instructions present in the program in such a manner that the maximum inter-bit processing time is reduced. In order to achieve this, we move the instructions up and down the program to get optimal placement. When trying to move code out of a path that has the maximum inter-bit processing time, we need to consider the following two things.

1. Which instruction should be moved from the path (nodes of the path)?
2. Which node should we move the instruction to?

In this section we explain how this is handled in our algorithm. Assume that we have a path *p* from which we need to move code. In order to decide which instruction to move, we need to know which instruction can move out of the path into other nodes. This can be found out for each instruction by performing dataflow analyses for each instruction in the path. The instructions which do not have any nodes from the path *p* in their corresponding UpNodes or DoenNodes are all candidates for code motion as they can move out of the path *p*. Thus we have isolated a set of instructions *CanMoveOut* that we can consider for code motion.

From the *CanMoveOut* set, we have to pick one instruction which we will subject to code motion. We could pick some instruction *i*, which we could then subject to code motion. But this random selection could result in problems later. Assume that we had two instructions *a* and *b*. Assume data

flow analysis showed us that *a* could be moved up to the node *x* and *b* could be moved up to the node *y*. For simplicity assume that *a* and *b* are node dependent on each other. This situation is illustrated in figure 8.1. Now if we decide to move the instruction b to the node *y*, afterwards, we may need to move some instruction away from *y*. In such a case, we cannot move the instruction *b* from the node *y* as it cannot move past node *y*. Consider the case if we had moved the instruction *a* to the node *y*. Now if we need to move an instruction away from the node *y,* we can move the instruction *a* as it can move up to node *x*. Thus, by moving the instruction which "travels the maximum distance" we can cover the possibility of additional code motions at the destination node.

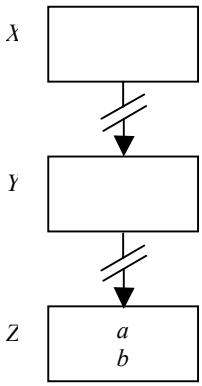


Figure 8.1 Code Motion Candidates

To simplify the process of selecting the instruction as explained above, we number the nodes while constructing the CDG. The numbering mechanism assigns smaller numbers to nodes which are closer to the *start* node and larger numbers as we move away from the *start* node. Then the node with the largest number in the UpNodes denotes the *distance* the corresponding instruction can move. Since we would like to move the instruction that can go as far up as possible, we have to select the instruction with the smallest *distance*. The situation is reversed in the case of DownNodes.

In order to prevent lots of code motions back and forth, we perform a conservative type of code motion. We mean conservative in the sense that even though we are moving the instruction *i* that can travel the largest distance, we do not move that instruction to the highest or lowest point while code motion. We perform code motion just to make sure that the instruction *i* is moved out of the path *p* in which it present. Our solution is to move the instruction to the node immediately preceding or succeeding the path. Thus for code hoist the instruction selected would be moved to the node immediately before the CALL node at the beginning of the path *p* and for code sink we move the instruction to the node immediately after the CALL node at the end of the path *p*. In the next section we will consider which paths have to be subjected for code motion and what type of code motion is to be performed, whether code hoist or code sink needs to be performed.

8.3 Loop Paths and Cascading Effect

In this section we explain two unique concepts that affect our algorithm. The first one is called *loop paths*. *Loop paths* are critical to our algorithm and provide a serious block for improving the inter-bit processing times. The second is cascading effect which deals with the effect of the movement of one instruction on other instructions. Cascading effect helps our algorithm in optimizing large paths

8.3.1 Loop Paths

Loop paths are inter-bit paths that occur in the program. But *loop-paths* occur entirely within loops. We define *loop paths* in a CDG context as follows:

Definition 8.1: Loop paths are inter-bit paths in which all the nodes of the path are descendants of some LOOP node.

Note that the nodes need not be the descendants of a single node. But can be distributed across more than one adjacent LOOP node, i.e., it is possible for a path to begin at the CALL node which is a descendant of some LOOP node and end at the CALL node, which is a descendant of another LOOP node which can be the immediate sibling of the first LOOP node.

In our algorithm we cannot move an instruction out of its LOOP if it is a descendant of one. We assume that a redundancy elimination algorithm as mentioned in chapter 4 is performed before our algorithm. This redundancy elimination algorithm if performed removes all the loop invariant code from inside the loops and gives an optimal program. Since we are not performing any redundancy elimination code motion, our algorithm does not move any code out of the loop as it might affect the semantic correctness of a program. As a result of this, *loop paths* present a blocking point to our algorithm. If a *loop path* exists that begins and ends at the same CALL node inside a loop, we will not be able to improve on the run-time of that path, as we cannot move instructions out of the loop and only the instructions within the loop are in the *loop path*. Thus the algorithm would not help improve a program's inter-bit times, if we have a large loop with a single CALL node in it. But this type of code structure occurs rarely in the message level functions and could be prevented by avoiding large loop structures while coding the message-level functions. We will discuss more about this in the results section, where we subject our algorithm to a message-level thread with the above mentioned code structure.

8.3.2 Cascading Effect

Cascading effect is the effect that one code motion has on the other movable instructions of the program. A simple example of this is shown in figure 8.2. Dataflow analysis would not let us move the instruction 2 out of the block C, since the instruction 1 defines R2 which is used by instruction 2. Now consider the case in which we were able to move instruction 1 to the code block B. Data flow analysis of instruction 2 will now allow us move instruction 2 out of block C. Thus each code motion affects every other movable instruction. We call this the *cascading effect.*

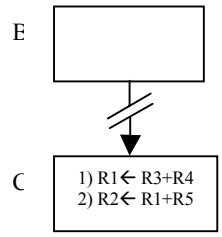


Figure 8.2 Cascade Effect of code motion

Cascading effect has its own advantages and disadvantages. As the example shows, it allows us to move instructions which we were not able to move initially. This helps when we are trying to move instructions out of a very large path into other smaller paths. But the disadvantage of this is that we have to perform the data flow analysis for all the instructions again to get the new changed UpNodes and DownNodes. This affects the runtime of the algorithm. Since the optimization algorithm would be run only once during the development process of the message-level thread and on the worst case a few times for getting the best solution with a few different threads, this disadvantage overweighs the advantage that we would be able to optimize as much as we want if we select the right instruction to move as mentioned in the previous section.

8.4 Feasibility Analysis and Direction of Motion

In this section we explain the feasibility analysis which determines whether a code motion is indeed useful. Further we also discuss when to perform code hoist of instruction and when to perform code sink. Then we move to the algorithm itself in the next section.

8.4.1. Feasibility Analysis

Feasibility analysis tells us whether the movement of an instruction would help us to achieve optimal code placement or not. The feasibility analysis adopted is very simple. Once we have decided that we would be moving the instruction *i*, we can obtain the insertion points of the instruction as explained in chapter 7. Since we know the inter-bit times associated with each node

of the program, we can obtain the largest destination path from the set of insertion points. The run-time of this largest path would be increased by the number of cycles the instruction *i* takes to execute. The following formula helps in deciding whether code motion would result in subsequent performance gain:

Feasibility analysis: Let r be the inter-bit processing time of the current path from which instructions are moved. If s denotes the largest inter-bit path associated with the destination insertion points of an instruction i which takes q cycles to run, code motion is performed only if r+q<=s.

If the destination path would become larger than the current path being optimized, then the code motion would bring down the effective performance rather than improving it. But on the other hand if we move code into a destination path that would become equal to the current path, it would not improve the performance immediately. But the destination path may be subjected to code motion and instructions could be moved out of that path resulting in subsequent performance gain. Since we cannot predict the effect of code motion on other instructions, we use this simple feasibility test to verify that our code motion only results in optimization of the code.

8.4.2 Direction of Code Motion

So far we have discussed everything we need to consider before and during code motion. We have explained the concepts for both code sinking and code hoisting. But which code motion to perform once we decide that we need to move code out of some path *p*? There is no way to determine the answer for this question based on the code structure. So, in our algorithm we try to perform code hoist first. If code hoist fails we try to sink the code. If both code hoist and code sink fails then we cannot move code out of the path.

The above solution though intuitive suffers from the cascading effect problem in which the code in some path *p* may not be move out of the path *p*, unless we perform code hoist or code sink to some adjoining path. Note that this situation can occur only if the adjacent paths do not satisfy the feasibility condition before code is removed from them. Otherwise we could have indeed moved the code out of the path *p* to its adjoining nodes. This type of situation thereby occurs when the path *p* and the adjoining paths have the same maximum inter-bit processing time. In order to take cascading effect into consideration, we use the following solution.

Let *MaxPaths* denote all the paths which have the maximum inter-bit processing time. *MaxPaths* contains all the paths in the order in which their paths begin in the program structure. The order of the paths in *MaxPaths* can be easily obtained as we have numbered the nodes from *start* to *end*. We use the following algorithm for code motion directions:

```
for every path p in MaxPaths from the first path to the last path
     try code hoist for the instructions in path p
     try code sink for the instructions in path p
for every path p in MaxPaths from the last path to the first path
     try code sink for the instructions in path p
     try code hoist for the instructions in path p
```

The above sequence of code motion will eliminate the problem mentioned earlier.

8.5 Code Motion Algorithm

So far we have presented all the details of the code motion algorithm for optimizing the inter-bit processing times in the message level threads. In this section we give a complete algorithm consolidating all the details explained so far.

8.5.1 The Algorithm

```
IBPOptimization

Inter-bit_Analysis(proc,false)
Next:
  Number the CDG nodes and instructions
  Compute the set MaxPaths
  Set Max-value = inter-bit time(MaxPaths)
  for every path p in MaxPaths from the first path to the last path
    do code-hoist(p) until no more code hoist possible
    do code-sink(p) until no more code hoist possible
  for every path p in MaxPaths from the last path to the first path
    do code-hoist(p) until no more code hoist possible

    do code-sink(p) until no more code sink possible
```

```
  if Max-value is unchanged exit else go to Next
end



Code_Hoist(p)

for each movable instruction i in p do
  compute DSAFE for all nodes for instruction i
  compute UpNodes(i)
set movable-instructions = Travels_Max(p)
for each instruction i in movable-instructions do
  set insertion-points = Get_Insertion_Points(p,i)
  set Max-Insertion-IBT = Maximum_InterBitPath(insertion-points)
  perform feasibility analysis
  if feasibility analysis succeeds
    delete copies of instruction i in the code motion region
    set deletion-points = Get_Deletion_Points(i,p)
    delete copies of instruction i from deletion-points
    insert copies of instruction i into insertion-points
    exit
end



Code_Sink(p)

for each movable instruction i in p do
  compute USAFE for all nodes for instruction i
  compute DownNodes(i)
set movable-instructions = Travels_Max(p)
for each instruction i in movable-instructions do
  set insertion-points = Get_Insertion_Points(p,i)
  set Max-Insertion-IBT = Maximum_InterBitPath(insertion-points)
  perform feasibility analysis
  if feasibility analysis succeeds
    delete copies of instruction i in the code motion region
    set deletion-points = Get_Deletion_Points(i,p)
    delete copies of instruction i from deletion-points
```

```
    insert copies of instruction i into insertion-points
    exit
end
```

TravelsMax(*p*) – returns the set of instructions that can travel the farthest

Get_Insertion_Points(*i,p*) – returns the set of insertion points for the instruction *i* in path *p*

Maximum_InterBitPath(*insertion-points*) – returns the maximum inter-bit path that is associated with the insertion points.

Get_Deletion_Points(*i*,*p*) – returns the set of deletion points for the instruction *i* in path *p*

Chapter 9

RESULTS

9.1 Introduction

In this chapter we discuss the results obtained for our algorithm. We applied our algorithm to 5 different message level functions and the results obtained are described in the next sections. The first section describes the optimizations achieved and the second describes the increase in code size due to code motion.

The five message level functions on which we ran our algorithms are

- A sample message function which receives a bit arriving at the bus
- 1553 communications send packet function
- 1553 communication receive packet function
- CAN send packet function
- CAN receive packet function

9.2 Code Optimization

Our optimization results for the five threads specified in section 9.1 are shown in figure 9.1 and figure 9.2. As can be seen in figure 9.1 considerable improvement is achieved for the sample function, CAN receive function, 1553 send function and 1553 receive function. But the improvement for CAN send function is very less, of the order of a few cycles. The reason for this is due to *loop paths* explained in earlier chapters. For the CAN send function, a loop path spanning two big loops proved to be the blocking path. This path started near the beginning of the first loop and ended near the last node of the second loop. Thus only a few instructions could be moved of this loop into adjoining paths.

The above mentioned improvements are reflected in the percentage reduction achieved in the maximum inter-bit processing time as shown in figure 9.2.

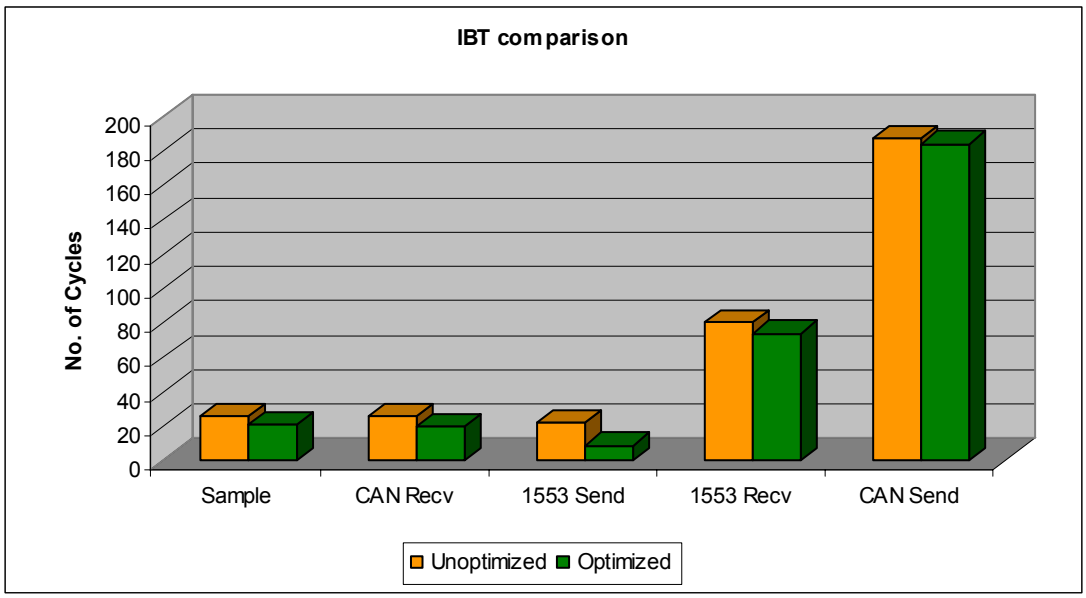


Figure 9.1 IBT before and after code motion

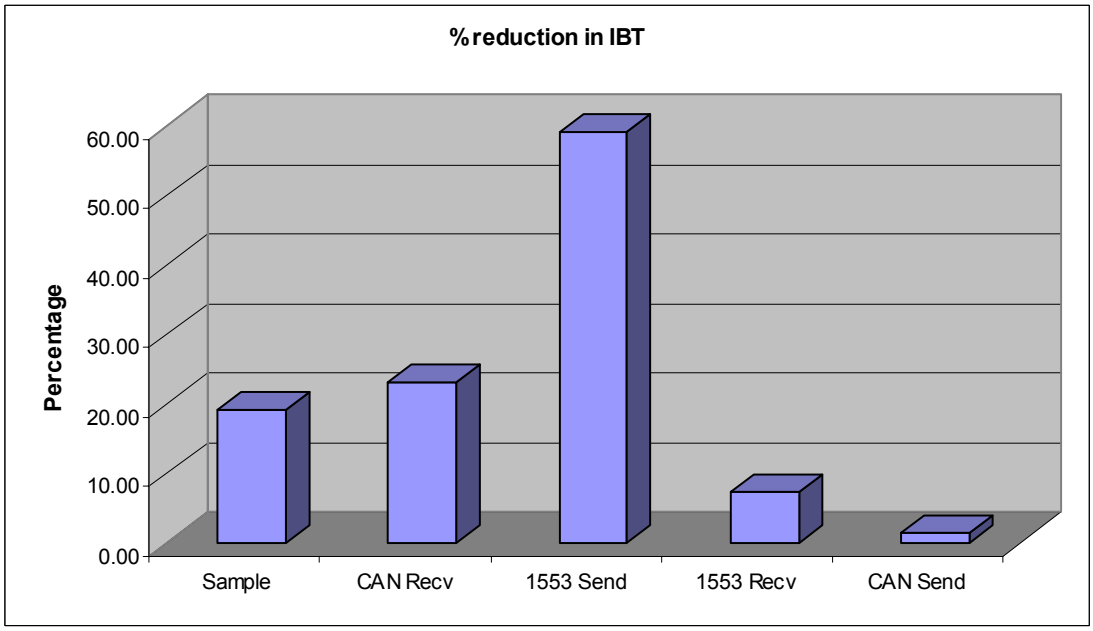


Figure 9.2 Percentage reductions in cycles of the maximum IBT

In order to prevent the situation encountered for CAN send packet function; we suggest a few guidelines to follow during coding of message level functions for better optimization of the resulting function in section 9.4 where more analysis is done.

9.3 Code Size Increase

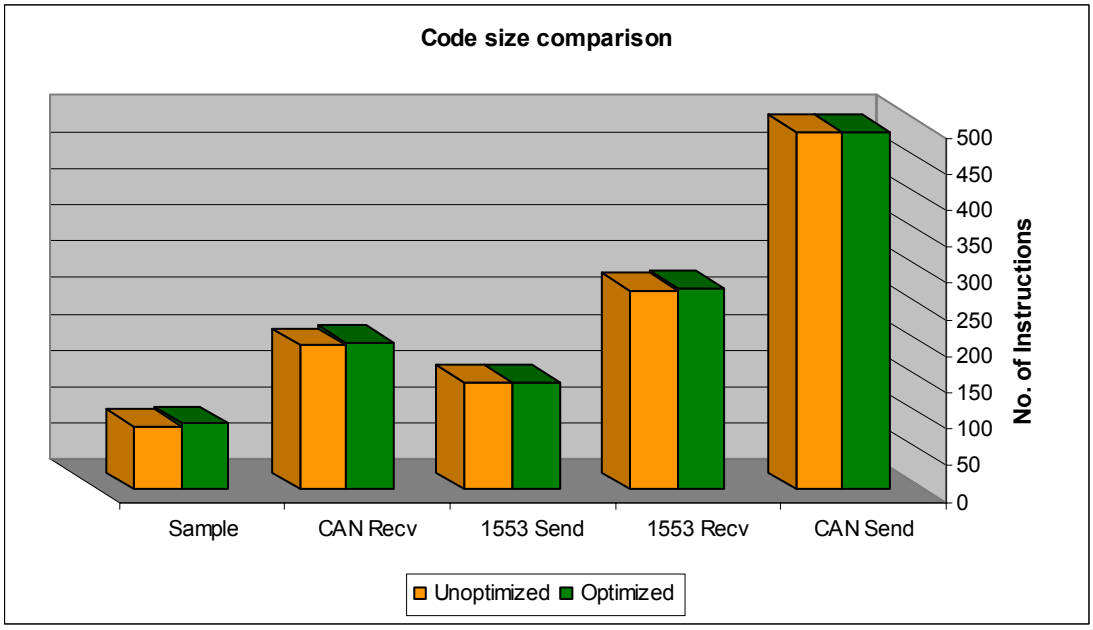


Figure 9.3 Code Size before and after code motion

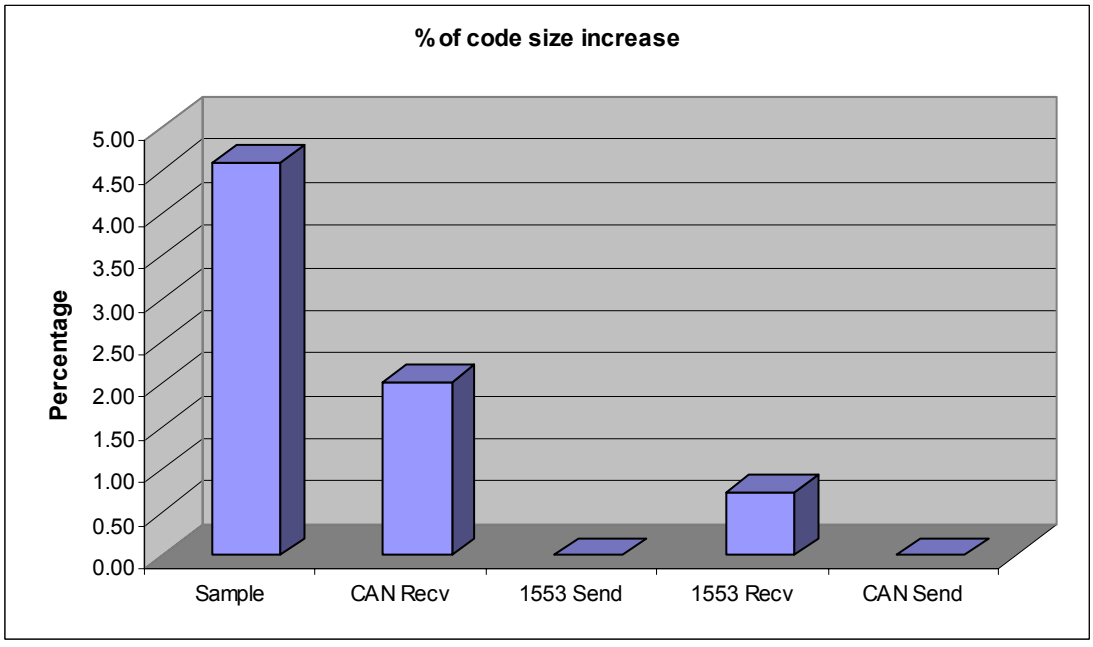


Figure 9.4 Percentage increase in the code size

The code size increase due to code motion is shown in figure 9.3 and figure 9.4. As can be seen from figure 9.3, the amount of code increase is not considerable compared to the gain in the idle time obtained between inter-bit calls. Figure 9.4 gives the percentage increase in code size, which can be compared with figure 9.2. It can be observed that as the optimization percentage increases so does the code size increase percentage. The reason for this is that for achieving large optimizations in the run time of the algorithm, quite a few number of instructions have to be moved around. As a result a few instructions might need to be copied at more than one node to satisfy the insertion points as explained in earlier chapters. This also explains the no increase in code size of functions which have been optimized for only a few cycles.

9.4 Analysis and Guidelines

The source code (in c) of the various functions is given in the Addendum 1.

The optimization percentage is affected by the number of movable instructions that are present in the function. Movable instructions were explained chapter 4. Comparing the above results with the movable instructions graph in figure 9.5, we observe that the percentage of movable instructions affects the optimization levels achieved. Clearly, the more the percentage of movable instructions in a given function, the more optimization can be obtained for the function.

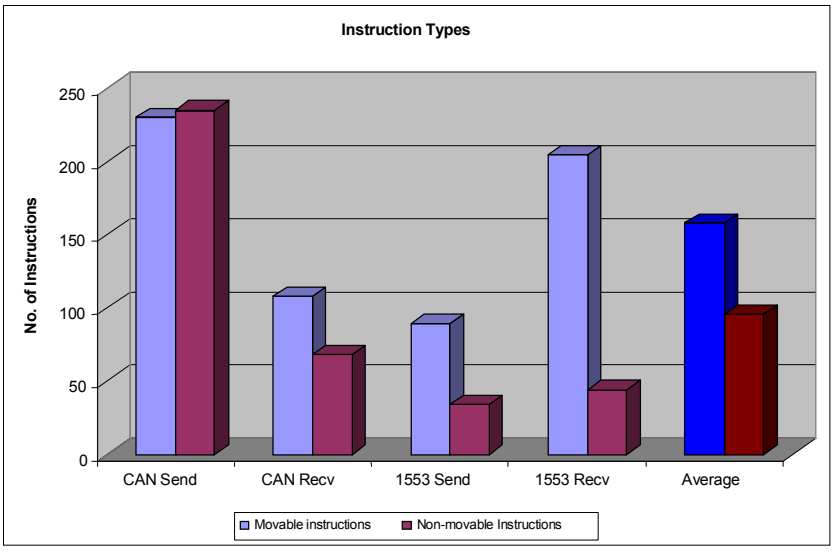


Figure 9.5 Movable Vs Non-movable

Use of pointers or structures in the message-level function will result in operands which are memory locations. This type of instructions is not subjected to code motion by our algorithms. Figure 9.6 shows the distribution of the non-movable instructions

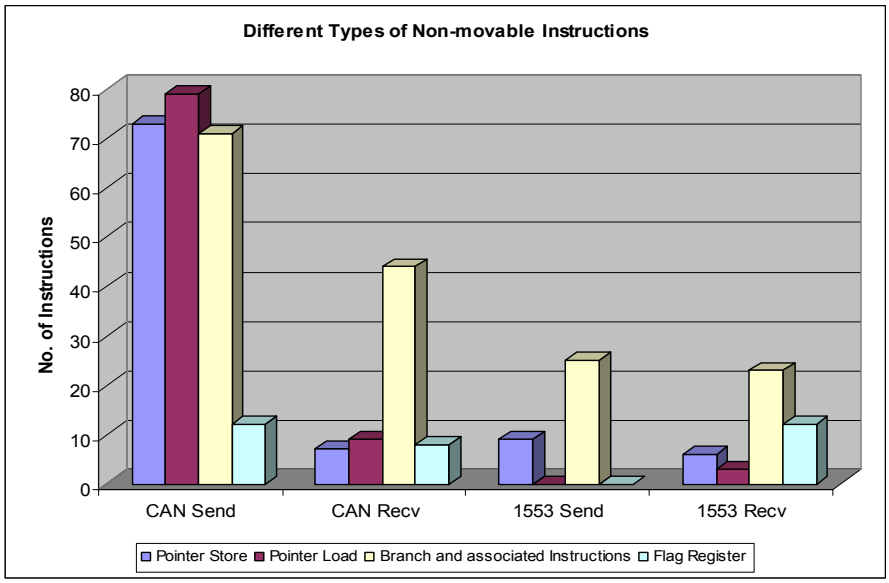


Figure 9.6 Types of Non-movable instructions

Another factor that affects the percentage of optimization obtained is the distribution of the inter-bit times. This does not occur for all programs, but it is worth analyzing. The inter-bit times must have as much deviation as possible from the average inter-bit time. This would enable easy movement of code from paths which are large into paths which are small. If the deviation is very less, then all the paths would have their inter-bit times near the average. We would not be able to improve much in this situation.

Another factor related to the above mentioned one is the location of the larger paths in the function. If the larger paths are localized to one part of the function, it may not be possible to move the code from an inner path to the outer paths of the function by our algorithm. Optimization would not be possible under such localized situations.

Having analyzed the results of our algorithm, we suggest the programmer of the message-level function to follow a few guidelines to obtain optimal code. These guidelines may be difficult to follow under some circumstances, but nevertheless they will help in obtaining the most optimized message-level function.

1. Do not include large loops in the message-level function if it could be prevented. In case this is unavoidable try to roll out the loop into its individual instructions. This might allow the algorithm to move instructions to achieve some optimization
2. Avoid using pointer instructions whenever possible. Try to localize these instructions to one part of the program, so that other paths can be optimized.
3. Try to distribute the larger paths across the function if possible
4. Move intensive operations in-between bit-level calls to the management function if possible.

Chapter 10

SUMMARY AND FUTURE WORK

10.1 Summary

Software thread integration helps to move functionality from hardware to software. STI applied for communication protocols, facilitates the hardware to software migration of these protocols. *Coarse grain* idle time which otherwise goes wasted is utilized by integration of the management-level function and the bit-level function.

In achieving utilization of the fragmented idle times present in the bit-level thread, some efficiency is lost due to the padding of the message level thread for achieving fixed intervals in the threads for insertion of coroutine calls (cocalls). Reducing this idle time will increase the efficiency achieved by software thread integration.

This thesis attempts to reduce the idle time introduced in the message-level thread. Code motion is used to reduce the maximum inter-bit processing time present in the message-level thread as much as possible. Our code motion works by moving one instruction at a time to preserve the semantic correctness of the program. Data flow analysis techniques similar to that of higher level code motion techniques are employed at the assembly level for identifying the instructions that can be moved. The source and the destination paths for code motion, known as the code motion points are computed. These code motion points are subjected to feasibility analysis to identify the profitability of the code motion.

We employ CDGs, instead of CFGs, to enable easier analysis and easier code motion. Problems caused by loops are identified at each stage and the algorithm has been designed to handle these situations.

Significant gains were achieved in reducing the idle time introduced in the message-level thread. On average, 15-20% reduction of idle-time was achieved. Loop paths affected the gain obtained, as it is difficult to move instructions to or from a loop.

10.2 Future Work

The gain obtained due to code motion can still be improved in a few ways. In this section we explain some of them.

Currently code motion is performed for only one instruction at a time. This could be improved upon to move more than one instruction at a time by performing additional analysis. By moving more than one instruction at a time, we can include *flag dependent instructions* in our code motion, which reduces the number of non-movable instructions.

Code motion can also be performed on 'if' code structures. Part of the 'if' structure can be hoisted or sunk until the condition value of the structure is not modified. This would require additional data analysis for the conditional instruction and the branch instruction of the 'if' structure. Either the 'if' structure as a whole or a part of it can be moved.

Pointer instructions are not moved by our code motion. This can be used by value based code motion analysis of the registers to determine whether the value of the memory locations held by registers is affected by code motion. Our data flow analysis technique is more of a syntactic analysis technique, than a semantic technique. This direction of analysis would be different from that done by our algorithm and may affect the analysis. Careful analysis on the effects on either type of code motion when combined together has to be done to achieve optimization while preserving correctness.

REFERENCES

[DEA 1] Dean, A. *Software thread integration for Hardware to Software Migration*. Phd Dissertaion, Carnegie Mellon University, May 2000.

[DEA 2] Dean, A., Grzybowski, R.R. *A High-Temperature Embedded Network Interface Using Software Thread Integration.* Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99) October 1-3, 1999, Washington, D.C.

[KUM 1] Nagendra, J., Kumar. *STI Concepts for Bit-Bang Communication Protocols*. Masters Thesis, North Carolina State University, May 2003.

[KSA 1] Nagendra J. Kumar, Siddhartha Shivshankar and Alexander G. Dean. *Asynchronous Software Threading for Efficient Software Implementations of Embedded Communication Protocol Controllers*. Center for Embedded Systems Research, NCSU, 2003.

[NEW 1] Chris Newburn. *Node labeling*. CMuART, Carnegie Mellon University, 1997.

[STE 1] David Stephenson. *Code Motion.* Technical Report, December 1997.

[KRS 1] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. *Partial dead code elimination.* SIGPLAN Notices, June 1994.

[KRS 2] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. *Lazy code motion*. SIGPLAN Notices, July 1992.

[KRS 3] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. *Optimal code motion: Theory and practice.* ACM Transactions on Programming Languages and Systems, July 1994.

[KRS 4] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. *The power of assignment motion*. SIGPLAN Notices, June 1995.

[BHS 1] Preston Briggs, Tim Harvey, and Taylor Simpson. *Static Single Assignment Construction.* Massively Scalar Compiler Project, Rice University, July 1992.

[CFR 1] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman and Kenneth Zadeck. *Efficiently computing static single assignment form and the control dependence graph*. ACM trancsactions on programming languages and systems, October 1991.

[FER 1] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. *The program dependence graph and its use in optimization*. ACM Transactions on Programming
Languages and Systems, 9(3):319--349, 1987.

[BGS 1] Rastislav Bodik, Rajiv Gupta and Mary Lou Soffa. *Complete Removal of Redundant Expression*. SIGPLAN Conference on Programming Language Design and Implementation, 1998.

[CCK 1] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo and Peng Tu. *A New Algorithm for Partial Redundancy Elimination based on SSA Form*. SIGPLAN Conference on Programming Language Design and Implementation, 1997

[COC 1] John Cocke and Jocob T.Schwartz. *Programming languages and their compilers: Preliminary notes*. Technical report, Courant Institute of Mathematical Sciences, NY University, 1970.

[BCS 1] Preston Briggs, Keith D.Cooper, and L.Taylor Simpson. *Value Numbering: Technical Report*. Center for Research on Parallel Computation, Rice University, 1997. Submitted to Software-Practice and Experience.

[MUC 1] Steven Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kauffman Publishers, California, 1997.

[AHU 1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and Analysis of Computer Algorithms.* Addison-Wesley, Massachusetts, 1974.

[AWZ 1] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. *Detecting equality of variables in programs*. Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, 1988.

[COO 1] Keith D.Cooper and L.Taylor Simpson. *SCC-based value numbering*. Extended abstract submitted to SIGPLAN PLDI '96.

[COO 2] Keith D.Cooper and L.Taylor Simpson. *Value-Driven Code Motion*. Technical report, Center for Research on Parallel Computation, Rice University, TX, 1995.

[TAR 1] Robert E.Tarjan. *Depth first search and linear graph algorithms*. SIAM Journal on computing, June 1972.

[CAN 1] *CAN Specification 2.0.* http://www.can-cia.de/downloads/specifications

C source code of message level functions

CAN Send Message:

```
#include <io.h>
#include "can.h"

extern uchar * CAN_bus_tx;

extern uchar oldbit_tx;
extern uchar bitcnt_tx;
extern unsigned short crc_tx;

enum CAN_ERROR_T send_msg(CAN_MSG_T * msg)
{
  /* This function transmits msg and places the bits in the array
pointed
     to by CAN_bus_tx.  This array must be filled with recessive bits,
as
     send_msg waits until the bus is idle before transmitting.  The
return
     value describes the status of the send attempt.
  */
/*  int i; */
  char i;
  uint mask, CurData;
  uchar CurBit;
  enum CAN_ERROR_T err=NO_ERROR;

  /* wait for bus to become idle for long enough */
  /*
  i = 0;
  while (i < ACK_DELIM_LEN + EOF_LEN + INTERMISSION_LEN) {
    i = (*CAN_bus_tx)? i+1 : 0;
    TX_DELAY_FOR_BIT;
  }
  */

  crc_tx = 0;
  oldbit_tx = 0;
  bitcnt_tx = 0;

  /* send start bit */

  sbi(PORTA,0x2);
  if (!send_bit(CAN_DOM, TRUE))
    return BIT_ERROR;

  /* Send identifier and arbitrate for bus access */
  CurData = msg->Identifier;
```

```
  for (mask = 1 << (IDENTIFIER_LEN-1); mask && !err; mask >>= 1) {
    CurBit = (uchar) ((CurData & mask)? CAN_REC : CAN_DOM);
    if (!send_bit(CurBit, TRUE))
      err = CurBit ? LOST_ARB : BIT_ERROR;
  }
  if (err)
       return err;

  /* Send RTR bit */
  CurBit = (uchar) (msg->RTR ? BUS_MASK : 0);
  if (!send_bit(CurBit, TRUE))
               return (CurBit)? LOST_ARB : BIT_ERROR;

  /* Send Control (inc. Data Length) field.  DLC must be <=8 */
  CurData = msg->DLC;
  for (mask=1 << (CONTROL_FIELD_LEN-1); mask && !err; mask >>= 1) {
    CurBit = (uchar) ((CurData & mask) ? CAN_REC : CAN_DOM);
    if (!send_bit(CurBit, TRUE)) {
      err = BIT_ERROR;
    }
  }
  if (err)
    return err;

  /* Send Data Field */
  for (i=0; (i < msg->DLC) && !err; i++) {
    /* select Data byte */
    CurData = msg->Data[(int)i];
    for (mask=1 << 7; mask && !err; mask >>= 1) {
      /* select Data bit */
      CurBit = (uchar) ((CurData & mask) ? CAN_REC : CAN_DOM);
      if (!send_bit(CurBit, TRUE))
        err = BIT_ERROR;
    }
  }
  if (err)
    return err;

  /* Send CRC field */
  CurData = crc_tx;
  for (mask=1 << (CRC_FIELD_LEN-2); mask && !err; mask >>= 1) {
    CurBit = (uchar) ((CurData & mask) ? CAN_REC : CAN_DOM);

    if (!send_bit(CurBit, TRUE))
        err = BIT_ERROR;
  }
  msg->CRC = crc_tx;
  if (err)
       return err;

  /* Send CRC delimiter*/
  if (!send_bit(CAN_REC, FALSE))
               return BAD_FORM;

  /* Send ACK Field (recessive) */
#ifdef SIM_RX
  /* pretend there's a receiver out there which acks the msg */
```

```
  send_bit(CAN_DOM, FALSE);
#else
  /* we EXPECT the recessive bit to be overridden by a dominant ACK
flag,
     so the send bit should NOT match the received bit */
  if (send_bit(CAN_REC, FALSE))
    return NO_ACK;
#endif

  /* Send ACK Delimiter */
  if (!send_bit(CAN_REC, FALSE))
    return BAD_FORM;

  /* Send End of Frame Field */
  for (i=0; (i < EOF_LEN) && !err; i++) {
    if (!send_bit(CAN_REC, FALSE))
      err = BAD_FORM;
  }
  if (err)
    return err;

  cbi(PORTA,0x2);

  return NO_ERROR;
}
```

CAN Receive Message:

```
#include <io.h>
#include "can.h"

extern uchar oldbit_rx;
extern uchar bitcnt_rx;
extern unsigned short crc_rx;

extern uchar * CAN_bus_rx;

enum CAN_ERROR_T receive_msg(CAN_MSG_T * msg)
{
  enum CAN_ERROR_T error_code=NO_ERROR;
  short tmp;

  char i,cnt,data, n;
  ushort expected_crc;

  bitcnt_rx = 0;
  oldbit_rx = 1;
  crc_rx=0;

  /* wait until sof (start of frame) */
  while (receive_bit());
  tmp=0;

  /* Get 11 bits of Identifier */
```

```
  for (i=0; i<11; i++) {
    tmp<<=1;
    tmp += receive_bit();
  }
  msg->Identifier = tmp;
  msg->RTR = receive_bit();

  /* Get reserved bits, which according to spec can be of any value...
*/
  msg->Reserved[0] = receive_bit();
  msg->Reserved[1] = receive_bit();

  /* Get Data Length Code */
  cnt=0;
  for (i=0; i<4; i++) {
    cnt<<=1;
    cnt += receive_bit();
  }
  msg->DLC = cnt;

  /* Get up to 8 Data bytes */
  for (n=0; cnt; cnt--, n++) {
    data=0;
    for (i=0; i<8; i++) {
      data<<=1;
      data += receive_bit();
    }
    msg->Data[(int)n] = data;
  }

  /* Get 15 bits of CRC */
  expected_crc = crc_rx;
  tmp=0;
  for (i=0; i<15; i++) {
    tmp<<=1;
    tmp += receive_bit();
  }
  if (expected_crc != tmp) {
    /* CRC Error signaling is deferred until after ACK delimiter,
       according to spec. */
    error_code = BAD_CRC;
  }
  msg->CRC = tmp;

  /* Get CRC delimiter bit directly - not stuffed */
  bitcnt_rx = 0; /* disable stuffing for 5 bits */
  if (receive_bit() == 0)
    return BAD_FORM;

  /* Get ACK bit - not stuffed */
  if (receive_bit() == CAN_REC)
    return NO_ACK;

  /* Get ACK Delimiter bit - not stuffed*/
  if (receive_bit() == CAN_DOM) {
    return BAD_FORM;
  }
```

```
  /* CRC Error signaling is deferred until after ACK delimiter */
  if (error_code == BAD_CRC) {
    return error_code;
  }

  /* Get End of Frame - not stuffed*/
  for (i=0; i < 7; i++) {
    bitcnt_rx = 0;
    if (receive_bit() == 0) {
      return BAD_FORM;
    }
  }
  return error_code;
}
```

1553 Send Message

```
#include "1553sim.h"

BYTE_t _1553_SendPacketToBus(BYTE_t len, BYTE_t* p_data)
{
  BYTE_t itr;
  UINT16_t value;

  ASSERT(0 == (len & 0x01));

  if (0 == g_config.TX_on)
  {
    return ERR_TX_OFF;
  }

  for (itr=0; itr<len; itr+=2)
  {
    /* We need to send the bytes from the queue, with the sync
sequences between the words */

    /* Send a control sync before the first word of the packet (the
control or status word) */
    _1553_SendControlToBus((0 == itr));

    /* Read the value */
    /* We will send the high byte and then the low byte */
    value  = *p_data++;
    g_parity_bit = 0;
    _1553_SendBitToBus((BYTE_t) value, 0x80);
    _1553_SendBitToBus((BYTE_t) value, 0x40);
    _1553_SendBitToBus((BYTE_t) value, 0x20);
    _1553_SendBitToBus((BYTE_t) value, 0x10);
    _1553_SendBitToBus((BYTE_t) value, 0x08);
    _1553_SendBitToBus((BYTE_t) value, 0x04);
    _1553_SendBitToBus((BYTE_t) value, 0x02);
    _1553_SendBitToBus((BYTE_t) value, 0x01);

    value = *p_data++;
```

```
    _1553_SendBitToBus((BYTE_t) value, 0x80);
    _1553_SendBitToBus((BYTE_t) value, 0x40);
    _1553_SendBitToBus((BYTE_t) value, 0x20);
    _1553_SendBitToBus((BYTE_t) value, 0x10);
    _1553_SendBitToBus((BYTE_t) value, 0x08);
    _1553_SendBitToBus((BYTE_t) value, 0x04);
    _1553_SendBitToBus((BYTE_t) value, 0x02);
    _1553_SendBitToBus((BYTE_t) value, 0x01);

    _1553_SendBitToBus((BYTE_t) g_parity_bit, 0x01);
  }

  return ERR_NONE;
}
```

1553 Receive Packet

```
#include "1553sim.h"

/* _1553_RecvPacketFromBus should never write more than */
/* MAX_PKT_SIZE_IN_BYTES bytes to the p_data buffer.    */
/* if *p_len is 0, then the length will be determined   */
/* and returned in *p_len, otherwise, *p_len will be    */
/* ASSUMED to be the correct length.  This can be used  */
/* when reading a status word (no word count field).    */
BYTE_t _1553_RecvPacketFromBus(BYTE_t* p_len, BYTE_t* p_header, BYTE_t*
p_payload)
{
  BYTE_t itr;
  UINT16_t temp_16, value;
  BYTE_t temp_8;
  BYTE_t len_in_bytes = 2;
  BYTE_t retval=ERR_NONE

  if (0 == g_config.RX_on)
  {
      retval=ERR_RX_OFF;
  } else {
    for (itr=0; (itr<len_in_bytes)&&(retval==ERR_NONE)
    {
      ASSERT(0 == (len_in_bytes & 0x01));

      /* We need to send the bytes from the queue, with the sync
sequences between the words */

      /* Recv a control sync before the first word of the packet (the
control or status word) */
      temp_8 = _1553_RecvControlFromBus((0 == itr));
      if (ERR_NONE != temp_8)
      { /* Did not find the sync */
        retval=temp_8;
      } else {
        g_parity_bit = 0;
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x80);
        ASSERT(ERR_NONE == temp_16);
```

```
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x40);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x20);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x10);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x08);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x04);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x02);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x01);
        ASSERT(ERR_NONE == temp_16);

        value = ((UINT16_t) temp_8) << 8;

        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x80);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x40);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x20);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x10);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x08);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x04);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x02);
        ASSERT(ERR_NONE == temp_16);
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x01);
        ASSERT(ERR_NONE == temp_16);

        value |= temp_8;

        /* Verify the parity bit, after we read the parity bit, it
should be 0 */
        temp_16 = _1553_RecvBitFromBus((BYTE_t*) &temp_8, 0x01);
        ASSERT(ERR_NONE == temp_16);
        if (g_parity_bit)
        {
          retval=ERR_WRONG_PARITY
        } else {
          /* Determine the length */
          if (0 == itr)
          {
            if (*p_len)
            { /* The length is already known */
              /* This is used when a status word is being read (i.e. we
are the BC) */
              len_in_bytes = *p_len;

              ASSERT(g_config.act_as_BC);
            }
            else
```

```
            { /* The length should be determined from the word just
read */
              ASSERT(0 == g_config.act_as_BC);

              if (value & _1553_CMD_WORD_IS_A_TX_MASK)
              { /* TX from the RT, this packet is only a one word
command from the BC */
                *p_len = len_in_bytes = 2;
              }
              else
              { /* RX by the RT, this packet contains the data */
                /* The number of bytes = number of words times 2 so
left  */
                /* shift to multiply by two.
*/
                len_in_bytes = (value & _1553_CMD_WORD_WORDCOUNT_MASK)
>> (_1553_CMD_WORD_WORDCOUNT_SHAMT);
                if (len_in_bytes)
                {
                  *p_len = len_in_bytes = (len_in_bytes << 1) + 2;
                }
                else
                {
                  *p_len = len_in_bytes = 64 + 2;
                }
              }
            }

            ASSERT(0 == (len_in_bytes & 0x01));
            ASSERT(len_in_bytes <= MAX_PKT_SIZE_IN_BYTES);
            ASSERT(len_in_bytes >= MIN_PKT_SIZE_IN_BYTES);

            *p_header++ = (value & 0xff00) >> 8;
            *p_header++ = (value & 0x00ff);
          } /* end if itr == 0 */
          else
          { /* itr != 0 */
            /* Write the value */
            *p_payload++ = (value & 0xff00) >> 8;
            *p_payload++ = (value & 0x00ff);
          }
        }
      }
    }
  }

  return retval;
}
```