ABSTRACT

KANG, SANG YEOL. Providing Static Timing Analysis Support for an ARM7 Processor Platform. (Under the direction of Professor Alexander G. Dean).

Scratchpad memory provides faster speed but smaller capacity than other memories do in embedded systems. It provides a visibly heterogeneous memory hierarchy rather than abstracting it as cache memory does. Unlike cache memory, program code and data can be allocated into the scratchpad memory as desired. This enables optimizing the performance in real-time embedded systems. Static timing analysis helps the optimization processes by providing microscopic information of the application program’s timing information. Based on the WCET and BCET estimated by static timing analysis, the techniques using scratchpad memory may be enhanced.

This study aims to provide a method of static timing analysis for an ARM processor platform (ARM7TDMI). Basic analysis is performed relying on well-known program analysis graphs such as control flow graphs, call graphs, depth-first search trees, and post-dominance trees. During this basic analysis, loops and unstructured code are also identified, which make static timing analysis more difficult. A control dependence analysis is a convenient way to analyze the WCET and BCET, since it represents the hierarchical control structure of a program. By traversal of the control dependence graph, the WCET and BCET are estimated.

To confirm the feasibility of this study, a real target system and its development environment tool chains are developed and an existing application is ported. In addition, the static timing analysis framework of this study is implemented by the tool named ARMSAT. Experiments are performed in these all environments. The experimental results show that the actual execution times are bounded by the calculated analytical WCET and BCET bounds, although there are a few factors which interfere with computing the analytical execution times.
Providing Static Timing Analysis Support for an ARM7 Processor Platform

by

Sang Yeol Kang

A thesis submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Master of Science

Computer Engineering

Raleigh, North Carolina

2008

APPROVED BY:

Dr. D. S. Reeves

Dr. J. M. Tuck

Dr. A. G. Dean
Chair of Advisory Committee
BIOGRAPHY

Sangyeol Kang was born in the Republic of Korea. He graduated from Seoul National University, Seoul, Republic of Korea. He worked as a hardware and software engineer for mobile cellular phones manufacturers over two years and began the study of his master degree at North Carolina State University, Raleigh, North Carolina in the fall of 2006.
ACKNOWLEDGMENTS

I would like to appreciate many people’s help with this study.
My father, mother and brother always encouraged me to go through this work, and my girlfriend Eun-Ha also does.

Dr. Alexander G. Dean served as my academic advisor. He gave me the opportunity to commit to this study and precious advice whenever I met obstacles. He expressed his enthusiasm and deep insight for research.

My friend Sang-Chang inspired me to study abroad.
Karthic and Gregory helped me revise this thesis draft.
# TABLE OF CONTENTS

**LIST OF TABLES** .................................................  v

**LIST OF FIGURES** .............................................. vi

1 Introduction .....................................................  1

2 Basic Analysis ..................................................  5
   2.1 Control Flow Graph and Call Graph ..........................  5
   2.2 Depth-First Search Tree and Identification of Loops ...........  8
   2.3 Dominance and Post-Dominance Tree .........................  11
   2.4 Unstructured Code ...........................................  14
      2.4.1 Fundamental Types of Unstructuredness ................  15
      2.4.2 Node Labeling ..........................................  16

3 WCET and BCET by Control Dependence Analysis ..................  20
   3.1 Control Dependence Graph ...................................  20
   3.2 Computing Worst and Best Case Execution Time ...............  22

4 Physical Target System .........................................  25
   4.1 Physical Target System .....................................  25
      4.1.1 Heterogeneous Memories ................................  25
      4.1.2 Instruction Execution Cycle ............................  26
   4.2 Development-Environment Tool Chains ......................  29

5 Experiments .....................................................  32
   5.1 Test Benchmark .............................................  32
   5.2 Experimental Results ......................................  33

6 Conclusion ................................................................  38

Bibliography .......................................................  40


LIST OF TABLES

Table 4.1 Memories of OKI ML674000 ...................................................... 26
Table 4.2 Memory Remap Register Values of OKI ML674000 ......................... 26
Table 4.3 ARM Instruction Speed Summary .............................................. 27
Table 4.4 Execution Cycles of Major Instructions ..................................... 29
Table 5.1 Analyzed Functions and Results ............................................. 35
LIST OF FIGURES

Figure 2.1 Process Flow ................................................................. 6
Figure 2.2 Algorithm for Depth-First Search ................................. 10
Figure 2.3 Algorithm for Identifying Loops .................................. 11
Figure 2.4 Example Graphs ............................................................ 12
Figure 2.5 Algorithm for Seeking Immediate Dominator .................. 14
Figure 2.6 Basic Unstructured Types .............................................. 16
Figure 2.7 Node Labeling Example ............................................... 18
Figure 3.1 Algorithm for Determining Control Dependence ............... 22
Figure 3.2 Example Control Dependence Graph .............................. 23
Figure 4.1 Development-Environment Tool Chain ........................... 30
Figure 4.2 Software Modules of open On-Chip Debugger .................. 31
Figure 5.1 Call Graph of Analyzed Functions ................................. 33
Figure 5.2 Error Plot of WCET&BCET Ratio ................................. 36
Chapter 1

Introduction

As processor technology has evolved dramatically, embedded systems are also equipped with remarkable resources compared to those of the past. Thanks to this rising tide, real-time embedded systems support multi-task scheduling in a real time manner for various applications such as multimedia, communication, automobile and so on. Tasks of a real-time system must complete their job before a promised deadline. Missing the deadline is catastrophic since it can lead to the failure of the functionality of the whole system. To avoid this situation, programmers put their efforts into meeting the deadline. This requires tasks’ timing information, and static timing analysis is able to provide it.

In another sense, the static timing analysis may have more particular applicability. In the real-time systems domain, it is helpful to meet the real-time deadlines more easily for better performances. Several factors are involved with meeting real-time deadlines efficiently: the task scheduling algorithm, decomposition of a huge task into appropriate size tasks, optimized memory allocation and so on. Memory allocation is an important factor since it directly affects the execution time of each instruction, as the processor’s pipeline does. Embedded systems compose their memory systems with heterogeneous memories. Each memory system component shows different access latency. Therefore, in embedded systems, memory allocation of program code and data has to be considered when the timing performance becomes an issue.

Cache memory accelerates execution of program by keeping data with high space and time locality in faster memory. Cache memory is located at the top level of memory hierarchy. This makes the estimation of a task’s timing more difficult. There is a huge penalty of execution time when cache miss occurs. If a task has cache misses while executing, the
actual execution time becomes longer. At this time, to prevent the timing deadline missing, the programmer must lengthen the margin of the deadline. Eventually the performance is lost or else the system must be over-provisioned, which is expensive. Of course, if it is possible to predict cache hit or miss, it can help solve this problem. Unfortunately it is not easy to estimate the cache memory contents at a certain point in time, although this is an area of research such as [WHW+97]. The ambiguity of predicting execution time in cache memory limits the utility of cache memory for real time embedded systems.

For this reason, traditional real-time embedded systems do not use cache memory. Instead, on-chip SRAM, which is called scratchpad memory, offers performance improvement for the systems. The architectural difference of scratchpad memory and cache memory rests on the fact that the scratchpad memory is architecturally visible in the system’s address space, unlike cache memory. As a result, while at the top level cache memory abstracts the memory lower in the hierarchy to the processor, the scratchpad memory becomes a part of the heterogeneous memory systems.

To use scratchpad memory wisely, much research has been performed from various directions[AMF+04, EKJ+06, ABS01, GPD07]. These investigations can be categorized by which program component is allocated into scratchpad memory. [AMF+04] and [EKJ+06] load program code into scratchpad memory. These research papers are about partitioning and selecting candidates from program code for allocation, or about positioning and loading the candidates into scratchpad memory. [AMF+04] allocates code into scratchpad memory and loads it statically, but [EKJ+06] does it dynamically. Both of them derive a performance metric function to select appropriate code fragments by estimating the execution time.

[ABS01] and [GPD07] allocate data variables into scratchpad memory. [GPD07] especially uses preemption threshold technique in order to make the allocation process virtuous. They also have an objective function to estimate the worst case performance. The objective function can be represented by 0-1 knapsack problem and solved by integer linear programming. The final solution tells an optimized allocation. When data variables are allocated into a new space of scratchpad memory leaving the original location, data variables are assigned new addresses. This process derives another new longest execution path as well as its corresponding WCET. Even after applying the most recently optimized allocation, the worst case execution path still exists. In spite of the repetition of generating new worst case execution path, the WCET converges to a fixed value. This framework requires repeated WCET analysis.
This study is about static timing analysis. The timing analysis of a task mainly deals with \textit{worst case execution time} (WCET) since the longest execution time of a task affects the schedulability of other tasks. [PB00] reviews previous research about the WCET analysis. The WCETs of tasks can be estimated by either dynamic analysis or static analysis. Unlike dynamic timing analysis, static timing analysis guarantees the upper bound of the WCET by testing all execution paths possible[MW98]. Overestimation of the WCET is crucial for task scheduling not to fail.

Conversely, the BCET analysis estimates the lower bound of execution time. The actual execution time can be represented by the linear sum of the BCET and additional execution time terms. The additional execution time components can be represented by a vector expression which indicates the memory access and their execution latency. These parametric information may be referred as the lower bound of execution time when an optimization for allocation is processed. The BCET is also required for EDF (Earliest-Deadline-First) schedulability analysis with mutually exclusive tasks.

This study computes the WCET and BCET by traversal of the control dependence graph (CDG). [GPD07] also developed its own static timing analysis tool, but the tool works by relying on the control flow graph (CFG) of a program; its computing algorithm of the WCET is not scalable to the number of basic blocks. [CP01] has suggested a tree-based static timing analysis framework. They used the CFG and the syntax tree to represent high level language during analysis. It also takes three simulation results into account such as instruction cache behavior, branch prediction and pipelined execution. This study is similar to the tree-based approach in a sense but differs by relying on the CDG and assembly language level representation. The approach to estimate the WCET depending on the program dependence graph (which is the union of the CDG and the flow dependence graph) was introduced by [SEGL06]. It suggested a program slicing technique relying on the program dependence graph to reduce the complexity of flow analysis when the WCET is computed. This study relies on only the CDG as [Dea00] does.

Although we can know the worst and the best case execution path from program analysis, estimation of execution times along with those paths is hard because of many factors affecting the execution times. [KP05] lists those factors with classification of several WCET analysis methods. It divides the whole spectrum of interfering factors into three groups: representation level, flow facts and execution time modeling. In this study, instruction overlapping in the processor’s pipeline, operand type and the memory allocation of
instructions and data, and conditional execution introduce variation to the exact estimation. Bounded loop iteration counts and error handling code at the entry of a function also affect timing. Due to the hierarchy of function calls, errors in analysis lower in the hierarchy may accumulate when rising in the hierarchy. Some of these factors are abstracted by measurement, some are supported directly and some are ignored.

This study aims to provide an automatic method to compute the WCET and BCET (best case execution time) by statically analyzing a program relying on several well-known graphs, such as the control flow graph (CFG), post-dominance tree, control dependence graph (CDG) and their relevant graphs. At the first stage, the control flow graph and depth-first search tree (DFST) are constructed. From them, loops are identified and unstructured code fragments are identified by the double-painting method and a simplified node labeling technique from [CFS90] and [Dea00]. Before creating the control dependence graph, a post-dominance tree is constructed. Finally the WCET and BCET are estimated from the control dependence graph by referring to program flow information, such as function calls and loop iteration counts, and the memory timing models and instruction timing models.

This work contributes an efficient way to estimate the WCET and BCET for the ARM processor. To implement the methodology of this study, an actual static timing analysis tool for the ARM7 processor named ARMSAT is implemented. To verify the feasibility and applicability of this study, a real target system (the core of which is ARM7TDMI) and its development tool chains are developed. An existing application, Helix MP3 decoder is also ported to the target system by the development tool chains. Experiments with 19 functions are performed and the results show the methodology of this study is applicable and reasonable.

This thesis will introduce the methodology and algorithms to analyze programs statically before the control dependency analysis in chapter 2. Chapter 3 will suggest the method to construct the control dependence graph and to compute the WCET and BCET from it. Chapter 4 will describe the real target system and the development environment tool chain, and chapter 5 will show the corresponding experimental results. The overall conclusions will be described in chapter 6.
Chapter 2

Basic Analysis

The static timing analysis of this study can be conquered by two steps: basic analysis and control dependence analysis for computing the WCET and BCET. The basic analysis step constructs the well-known program analysis graphs required for the control dependence analysis step.

First, the annotated assembly code is scanned and parsed into instruction, label, basic block and procedure data structures. To generate the annotated assembly code, the GNU ARM assembler is modified. The modified assembler generates the instruction information such as opcode, conditional suffix, operands information, label, etc. into an external file while assembling. Based on the parsed input data from the annotated assembly file, the call graph and control flow graph are constructed. Starting from the control flow graph, loops are identified and unstructured code is identified. This process requires a control flow graph spanned by a depth-first traversal and node labeling respectively. The post-dominance tree is constructed from the control flow graph then, the control dependence graph is constructed as the next step. The auxiliary graphs constructed are depth-first search tree, breadth-first search tree and dominance tree. The overall analysis follows the sequence shown by figure 2.1.

2.1 Control Flow Graph and Call Graph

Static analysis for a certain program starts with building the control flow graph (CFG). Before constructing the CFG, basic blocks have been identified since the CFG uses the basic block as its node. The following is the definition of a CFG[FOW87].
Definition 1 A control flow graph is a directed graph $G$ augmented with a unique ENTRY node and a unique EXIT node such that each node in the graph has at most two successors.

The CFG is built through three phases in this study. For the first phase, the basic blocks are identified by detecting a chunk of instructions starting with an internal label or just after termination of preceding basic block, and finishing with particular termination instructions. The common concept of basic blocks is used, i.e. if an execution flow enters into the entry of a basic block, it must emanate from the exit of the basic block. There cannot be another exit, thus only unique control flow exists inside a basic block. Since only branch (B) or conditional branch (e.g. BEQ) can transfer the control flow in the ARM instruction architecture targeted by this study, only those instructions can terminate a basic block[ARM00]. This policy also does not allow more than two successors of a basic block because those instructions can have only two destination. As a result there are only two successors: taken or not-taken branch. Although the CFG of this phase is generated from the annotated instructions directly, it is temporary since it should be verified to meet the
above definition of the CFG. This will be discussed at the third phase.

During the second phase, linear consecutive basic blocks are merged into a single basic block. The predecessors and successors of a certain basic block already are known at the first phase. Referring to the temporary CFG, the second phase merges a basic block B into a basic block A if the basic block B has only the basic block A as a unique predecessor, and the predecessor A has also only the basic block B as a unique successor. This phase reduces the number of basic blocks. As a consequence, future analysis of graphs derived from the CFG can be done more easily and faster. In some sense, this follows the definition of a basic block more strictly.

At the last phase, the CFG definition that the CFG has unique ENTRY and EXIT nodes are verified. The node which does not have a predecessor in the temporary CFG by the end of the second phase is regarded as the ENTRY node. Identifying the EXIT node is a little more complicated. If a basic block terminates with an instruction updating the PC (program counter) register, that basic block is looked upon as the EXIT. This criterion identifies ENTRY and EXIT nodes.

If multiple ENTRY nodes exist, a dummy basic block with no instructions is augmented at the head of the CFG. The corresponding control flows of multiple ENTRY nodes are also updated so that the inserted dummy basic block is followed by the multiple ENTRY basic blocks. This seems reasonable in the sense that the dummy basic block does not affect the computational result or execution times but it keeps the original execution path. On the other hand, the case that there is no ENTRY does not occur since the first basic block in the program order of a certain function always can be considered as the ENTRY.

In the case of absence of an explicit EXIT from a procedure, the dummy basic block is augmented at the tail of basic blocks sequence as a unique EXIT node similarly to the case of the ENTRY node. This situation may occur when a procedure ends with an infinite loop. In this case the instruction which loads an address into the PC register disappears. The potential problem of the control structure without a unique EXIT node is exposed when computing the post-dominance tree; the algorithm starts from the EXIT node and works toward the ENTRY node. If there is no EXIT node, the algorithm cannot construct the post-dominance tree from the CFG. This situation can be avoided by augmenting a dummy basic block as the successor of the infinite loop’s tail. If there are multiple EXIT nodes, the same policy is applied as in the case of multiple ENTRY nodes, which augments
a dummy basic block as a common EXIT node. Of course, the control structure of multiple EXITs is transformed to have the augmented block as their successor. This is essential for computing the post-dominance tree for the same reason.

While constructing the CFG, the call graph is also obtained. The call graph shows the relationship between caller and callee procedures. Detecting branch with link (BL) and conditional branch with link (e.g. BLEQ) instructions identifies that a procedure calls another procedure. The called procedure is identified through the label field of those instructions’ operand[ARM00].

2.2 Depth-First Search Tree and Identification of Loops

Intuitively, a loop is a group of code whose execution repeats. The actual execution cycle count of a loop is calculated by multiplying the execution cycle count of a loop code with the number of loop iterations. Loops occupy such a considerable portion of total execution cycle counts of a program that identifying loops must be done before computing the WCET and BCET in order to reflect the actual execution cycles. This section introduces the method to identify loops by a depth-first search tree.

Identifying loops is a two fold problem. The first problem is to identify basic blocks which are parts of the loop, including loop head and loop tail. Since the loop body can be distinguished after the head and tail are found, identifying the head and tail is the key to the first problem. The second problem is to get the information about the loop iteration count. However, it is a somewhat complicated problem. If the iteration count is assigned statically in the program, the loop head or loop tail may contain the iteration counts explicitly. In the case that the iteration count is determined dynamically i.e. during the runtime, the loop head or loop tail does not contain the iteration counts explicitly until the actual execution of the loop is monitored. Even in the case when statically determined, the actual iteration count cannot always be attained from the head or tail; consider an exit from loop body by forced break like the break statement in C language. In all cases, recognition of the loop iteration count requires additional work. For this study, the loop iteration counts are imported from an external input file which is created manually from instrumentation on coder knowledge. The process flow diagram of this study, figure 2.1, shows the external input file for providing the loop iteration counts. This can be regarded as an imported component from the dynamic analysis method.
Loops can be identified by simply examining the control flow; following the control flow directly and remembering the visited nodes in the past. If a remembered node is revisited while traversing the CFG, a loop is detected, and the examined control flow edge at that time is the back arc of identified loop. But this approach is not scalable with the number of nodes (basic blocks), particularly the number of predicate (conditional) basic blocks of a program. When following the control flows, the history of nodes visited has to be kept in each node to check whether the node is revisited or not. If the node is revisited, that node is regarded as the loop head and the node which is visited immediately ahead of the node as the corresponding loop tail. This algorithm has a very high time and space complexity, although it is not quantified in this study. Control flow at a conditional branch is split into two directions: branch taken or not taken. Before splitting, the history stored must be replicated for propagating the history to the next two nodes. This process takes a long time and requires a huge memory space. Therefore, more predicate basic blocks lead to more time to copy the visited history, and the processing time increases exponentially as the number of basic blocks increases.

Rather than this primitive method, a more systematic and simple method has been suggested by [Hav97]. Relying on a depth-first search (DFS) tree, the algorithm removes the time for replicating the visited history. A DFS tree is a tree representation of the CFG spanned by the depth-first search order. The node index of each node is assigned sequentially in DFS order. [Hav97] augments an additional information to a node, which is the index of the last child node below a node. This information can tell the ancestor node of a certain node in the control flow. The last child information eliminates the need to store the visited history. Actually, despite the total history of visiting being saved, only a fraction of the whole history is referred to while finding the back edge. Figure 2.2 describes the algorithm for identifying loops based on the DFS tree of this study from [Hav97]. The DFS tree can vary depending on which successor is visited first when there are two successors of a node. Here fall-through (i.e. not-taken branch) is visited first.

The step of identifying loops is composed mainly of traversing the DFS tree and examining the successor node in the CFG. In the DFS tree, a control flow edge which is headed from a left-side branch to a right-side branch cannot exist since every node of the control flow edge’s head will be visited next time. Assume that the current source node of a control flow edge’s tail is named as A and the edge’s head is the target node B, i.e. A→B. The edge’s head node B has greater index than node A in the DFS tree because it
static current := 1
number[*] := UNVISITED
DFS(ENTRY)

**procedure DFS** (a)
node[current] := a
number[a] := current++
for each node b such that $\exists$ edge(a, b) do
  if (number[b] == UNVISITED) then DFS(b)
lst[number[a]] := current-1

Figure 2.2: Algorithm for Depth-First Search

appears just below the current node A. Thus if a control flow from left-side to right-side branch exists in the DFS tree, the target nodes must be located below the branch node and the flow edge should point downward. Finally, if there exists a control flow A $\rightarrow$ B in the downward direction then, the index of B is greater than the index of A.

However, the control flow edge from right-side branch to left-side branch can exist. One sufficient condition for this case is where the index of B is greater than the index of A although A $\rightarrow$ B. Another sufficient condition is where there is a back arc of loop. It is apparent that in the DFS tree the nodes of the right-side always have a greater index than the nodes of the left-side have. The case of back arc is also definite because that is the definition of back arc. If two cases can be distinguishable, loops can be identified. If the DFS index of node A is greater than the DFS index of the target node, that edge is either a normal control transition from the right-side branch to the left-side branch or a back arc of loop. When the edge is a normal control transition, the index of node B is greater than the index of node A's last child node. Otherwise, the edge is a back arc of loop. This can be explained well with an example. See figure 2.4(b).

In figure 2.4(b), the DFS indexes are shown in parenthesis. The dotted lines show omitted control flows only in the DFS tree. Definitely there is no control transition from the left-side branch to the right-side branch. While traversing the DFS tree, one successor of node D is a back arc toward node C. First the algorithm detects that the DFS index of node C is smaller than the DFS index of node D. Then while examining the last child node of node C i.e. E(5), the arc must be a back edge because the index of node D is smaller than E(5). The node C becomes the loop head and the tail node D of the back edge becomes the corresponding loop tail automatically. Figure 2.3 describes the algorithm to identify a
**Figure 2.3: Algorithm for Identifying Loops**

loop head when $G$ is the control flow graph.

The property of the DFS tree helps to identify the loop body. Starting from the loop tail, the nodes on the path toward the loop head are painted with the first color. Since the loop head is already known, the painting may stop at the loop head during traversal of the CFG. Then starting from the loop head, the nodes are painted again with a different color until the loop tail is reached. If the visited node is not painted with the first color, the node and the successors of that node will not be visited any more, hence that control flow is eliminated from the traversal path automatically. All the double painted nodes are in the body of the loop. This *double-painting* method also works for the nested loops.

### 2.3 Dominance and Post-Dominance Tree

Before computing the WCET and BCET from the control dependence graph, generating the post-dominance tree is a crucial step since the control dependence is determined by considering both control flow and post-dominance together. Although the immediate goal is to get the post-dominance tree, the dominance tree is discussed ahead of the post-dominance tree since the algorithms to build the dominance tree and the post-dominance tree are almost identical. [CHK01] summarized the definition of *dominance* and *immediate dominator* as below.

**Definition 2** A certain node $A$ in a control flow graph $G$ dominates $B$ if $A$ lies on every path from the ENTRY node of the control flow graph to $B$, and is denoted $A \in \text{DOM}(B)$.

**Definition 3** For a node $B$, the set $\text{IDOM}(B)$ contains exactly one node, the *immediate
Figure 2.4: Example Graphs
**dominator** of such that if \( N \) is \( B \)'s immediate dominator, then every node in \( \text{DOM}(B) - B \) is also \( \text{DOM}(N) \).

By definition, \( \text{DOM}(B) \) contains every node \( N \) that dominates \( B \) as well as node \( B \) itself. \( \text{DOM}(B) \) contains whole dominators of \( B \), although the closest dominator of node \( B \) is useful practically. Another point to pay attention to is that the definition of \( \text{DOM}(B) \) specifies the existence of a unique ENTRY node. The verification stage of the CFG ensures a unique ENTRY node exists as mentioned. Therefore, there is no need to be concerned with the existence of a unique ENTRY node at this time. Intuitively, \( IDOM(B) \) gives us the information which node must be visited in order to reach node \( B \).

[CHK01] suggests an engineered algorithm evolved from [LT79]. Figure 2.5 describes the pseudo code from [CHK01]. The algorithm finds the common ancestor node in the CFG while tracing all the predecessors of a certain node. At the very first cycle, all dominators are assigned as the first predecessor of the node. While the loop is iterated, the common ancestor node in the CFG is found and it replaces the old dominator with itself. Before entering the algorithm the CFG needs to be spanned in post order, and the nodes are visited in post order.

Post-dominance can be defined as below. Intuitively, post-dominance means the dominance in the reverse control flow graph obtained by reversing the direction of all control flow edges and interchanging the roles of the ENTRY and the EXIT node with each other. \( \text{PDOM}(A) \) tells which node must be visited after executing node \( A \) in advance.

**Definition 4** A certain node \( A \) is **post-dominated** by a node \( B \) in control flow graph \( G \) if every directed path from \( A \) to EXIT (not including \( A \)) contains \( B \), and is denoted \( B \in \text{PDOM}(A) \).

The property of \( \text{PDOM} \) gives a clue for revealing the control dependence among basic blocks. As mentioned above, post-dominance tells which node will be executed after executing a certain basic block in the future. That means two basic blocks share the same control dependence since if the previous node is executed, then the immediate post-dominant basic block will be executed unconditionally. The relationship between post-dominance and control dependence will be discussed in the chapter 3 in detail.

Just like the dominance tree, a unique EXIT instead of ENTRY node is required, and the control flow graph has been verified already when the post-dominance tree is constructed. In addition, the same algorithm for creating the immediate dominance tree is
procedure Dominator (START, G)
    for each node b ∈ G do
        doms[b]:=UNDEFINED
        doms[START]:=START
        changed:=true
    while (changed) do
        changed:=false
        for each node b in reverse post-order of G except START do
            newidom:=first predecessor
            for all other predecessor p of b do
                if doms[p]≠UNDEFINED do
                    newidom:=Intersect(p, newidom)
            if doms[b] ≠ newidom do
                doms[b]:=newidom
                changed:=true
procedure Intersect (B1, B2)
    finger1 = B1
    finger2 = B2
    while (finger1 ≠ finger2) do
        while (finger1 < finger2) do
            finger1:=doms[finger1]
        while (finger2 < finger1) do
            finger2:=doms[finger2]
    return finger1

Figure 2.5: Algorithm for Seeking Immediate Dominator

utilized. This is simply achieved by replacing predecessors with successors in the algorithm for the dominance tree. Before going through the algorithm, the reversed control flow graph must be spanned by the post order traversal.

2.4 Unstructured Code

After the basic analysis, unstructured code needs to be identified to facilitate the control dependence analysis. Although an original source code usually has a structured control form before compiling, assembly code optimized by the compiler may have unstructured codes such as multiple exits from a procedure and exits from a loop’s body, jumping into a decision region and so on. Structuring unstructured code is a prerequisite step to get a simple control dependence graph. If unstructured codes are not removed properly,
nodes of the corresponding control dependence graph will have multiple predecessors. That means a basic block is control dependent on multiple basic blocks. The multiple control dependence causes multiple instances of traversing the control dependence graph when the WCET and BCET are calculated. Then, the algorithm is not scalable with the number of basic blocks. However, structuring unstructured code requires complicated work with the CFG. In this study, in order to pay more attention to the computation of the WCET and BCET, only structured code will be assumed. However, to avoid processing unstructured code when the execution times are computed, identifying unstructuredness is necessary.

2.4.1 Fundamental Types of Unstructuredness

At the C or C++ language level, structured programs are composed of simple, hierarchical program flow structures, which do not contain \textit{GOTO} instructions (which changes program’s execution flow arbitrarily). However, at the assembly language level, there are many branch and conditional branch instructions. Moreover, after optimization during compiling, unstructuredness grows more serious due to the optimization techniques.

[Oul82] has suggested a method to manipulate unstructured code as well as the specification of six basic types and four forms of unstructured codes. They also prove that unstructured code can be converted into structured code by applying schema algebra in the particular order to convert the unstructuredness of the basic types into a structured code. This is feasible since generally the unstructured forms can be obtained by super-positioning of the basic types. Structuring the four basic type is enough to eliminate unstructuredness. The four basic types from [Oul82] are depicted on figure 2.6.

In figure 2.6, \textit{ID} means “jump into decision”, \textit{OD} means “jump out of decision”. Similar to the ID, \textit{IL} means “jump into loop” and \textit{OL} means “jump out of loop”. In the case of the ID type, there is an entry of control flow from outside of the decision region (which consists of node A, B and C) into the decision region. A jumping out of the decision region results in the OD type. A jumping into the loop region A-B-C gives the IL type and a jumping out of the region causes the OL type.

The ID and OD types of unstructuredness make a certain node control dependent on multiple nodes. For example on figure 2.6(a), node B is control dependent on the branch node A and an unknown external node out of the decision region. In the case of the OD type, a control flow emanates from the decision region and is headed for another node. It
also makes figuring out the control dependency more complicated. When the IL or the OL occurs, computing the actual execution cycle count of loops is also hard due to the ambiguity of the ancestor of the corresponding control dependency. Since a loop is a group of instructions which repeats multiple times without affecting the instructions outside, the IL and the OL structure are not favorable when the WCET and BCET are computed. In figure 2.4(a), the OD type unstructured form is shown between node I and node J.

### 2.4.2 Node Labeling

Identifying the type of unstructuredness is not easy to do since generally decision and loop regions are nested within another region and control structures have a more complicated combination of basic unstructuredness in the real world. The IL and the OL type unstructuredness are identified by the double-painting method as introduced earlier. While examining successors or predecessors of the loop’s body nodes, if unpainted or differently painted nodes are detected at the loop bodies’ predecessors and successors, unstructuredness is found. However, for the case of the ID and OD, the regional information is not known yet. In this study, a simplified node labeling technique is used similar to the prior works [Dea00, NNS94]. With this technique, the branch and rejoin node of a decision region
are identified. Then, the double-painting method will be applied again to identify the type of unstructuredness.

A predicate node has two successors: not-taken branch (fall-through, false) node and the taken branch (true) node. If a control form is structured, there always exists a complete branch-rejoin pair without any entry from or any exit to other nodes outside since there must be only one control flow streaming down sequentially except for in a loop. This initiates the idea for the node labeling algorithm. A node label consists of the preceding predicate basic blocks' index and the corresponding branch condition i.e. taken (true) or not-taken (false). The rejoin node is identified by combining the ancestors’ branch history of nodes.

The node label is inherited by the successor nodes. A predicate basic block generates a new node label and appends it to the existing label i.e. inherited labels. If duplicated node labels occur, they are merged into one depending on the branch condition. The merging rules are

(1) When true and false branches are found, the merged branch condition is always
(2) When true and true branches are found, or false and false branches are found, the merged branch condition inherits the original one without any change.
(3) When always and other branches are met, the merged branch condition is always.

Starting from the ENTRY node, if a predicate node is met, a new node label which contains the predicate node index is generated. This label propagates into the two successors while augmenting branch conditions until it reaches the rejoin node. If the split label reaches its rejoin node, the branch condition becomes always (neither true nor false).

In addition, if always branch condition is met with either true or false branch, it represents unstructuredness because always branch condition is not propagated any more after it is generated. This may be another way to identify unstructuredness, but this study uses the double-painting method.

Figure 2.7 depicts a node labeling example of the previous control flow graph example. The node label appears to the left or right to the node. The brace symbol, “{” or “}” means merging of the node labels.

Labeling starts with node A. Since node A is a predicate node, it generates two node labels with corresponding branch resolution: A(F) and A(T). Node B has the label “A(F)” which means node A is the predicate node and node B can be reached by following the false branch condition from node A. The node F has the label “A(T)”. Node C and
D are not predicate nodes so they do not generate a new label and the label “A(F)” is propagated until it reaches node E. Even though the label “A(F)” reaches the end node of the procedure, node E is not yet the rejoin node of node A, because it has other predecessors: node H and J. Node G is a predicate node, hence it generates two new labels “G(T)” and “G(F)” . Its successors, node H and I inherit the previous node label “A(T)” from node G commonly as well as inherit the new label “G(F)” and “G(T)” respectively. Node I also generates a new label. Node H gets all node labels from its predecessors. Node H combines all inherited nodes together. Since the two condition branches from node G reach at node H together, the node H becomes the rejoin node for the predicate node G.

Although the decision region starting from node I is nested in the decision region starting from node G, the rejoin node of node I is located out of the decision region of node G. At this point, we may know there are unstructured codes around node H, specifically the predicate node I. But the identification of the predicate and its corresponding rejoin node
is enough outcome for this step. Identifying unstructuredness rests on the double-painting method. It provides an easier way to detect a jump into a region or jump out of a region. At node E, every branch rejoins again, thus node A as well as node I has node E as its rejoin node.
Chapter 3

WCET and BCET by Control Dependence Analysis

A control dependence graph (CDG) plays an essential role in the analysis of a program and the computation of both WCET and BCET. It presents the hierarchical information of a program. A control dependence graph is a component of a program dependence graph, which covers the data dependence as well as the control dependence. While the control flow graph shows the sequential relationship of the control flow, the control dependence graph shows the hierarchical relationship of the control flow. A control dependence graph can be established based on the control flow graph and the post-dominance tree. By traversing the control dependence graph, the WCET and BCET can be computed.

3.1 Control Dependence Graph

Nodes are also basic blocks in the CDG. However, the edge from a node to another node represents the control dependence rather than control flow between them. The CDG shows control-equivalent nodes of a certain node. If a node is executed conditionally depending on another node’s computational result, the former node is control dependent on the latter node. The control dependence between two nodes X and Y is defined as follows [FOW87]. It also suggests the concept and the way to build the control dependence graph, and [CFS90] reduces the time and space complexity. This study uses the method of [CFS90]. The detail of building the CDG will be explained later.
Definition 5  When a graph G is a control flow graph and X and Y are nodes in G, Y is control dependent on X if and only if
(1) there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and
(2) X is not post-dominated by Y.

What part (1) of the definition 5 says can be rephrased as, if a node Y is control dependent on node X, then X must have two successors in the CFG. If one of these edges is followed, Y should not be executed and when another edge is followed, Y should be executed. Part (2) of the definition is satisfied when node X is identical to node Y.

Combining the post-dominator tree with the control flow graph gives a basis to build the CDG. The post-dominator tree and the CFG have been already constructed in the prior steps. Before combining the CFG with the post-dominance tree, one thing must be mentioned. Unlike building the CFG, there is no explicit ENTRY node while building the CDG since there is no explicit node that can be followed by the entry parts of the CDG although they must have a certain control dependence node. Therefore, a fake ENTRY node is inserted conceptually at the top level of the CDG. Therefore, the ENTRY node means a certain execution that then invokes the execution of the current procedure.

After assuming the conceptual ENTRY node, the CFG is traversed. Consider a control flow edge (W, S) that emanates from node W to node S. If node W is post-dominated by node S, node S is not control dependent on node W because executing node W means that node S is executed inevitably. They have same control dependence on an ancestor node of node W. Otherwise, node W and node S will be examined in the post-dominance tree together. Assume another node L that is the least common ancestor of node W and node S in the post-dominator tree. Then, node L is either node S or the parent of node S in the post-dominator tree. If node L is the parent of node S, all nodes in the post-dominator tree on the path from node L to node W, including node S but not node L, are control dependent on node W. If node L is identical to node S, then all nodes between node S and node W in the post-dominator tree are control dependent on node W including node W and node S. This case can capture loop dependence. Figure 3.1 describes the pseudo code of forming a CDG.

The algorithm can be explained well with the previous example shown in figure 2.4(a). If we choose the control flow edge (A, B), the least common ancestor of both nodes
for each w of CFG do
  for each s ∈ successors of w do
    if w=s then skip /* Skip the single block loop */
    if ipdom(w) ≠ s do
      l:=searchLeastIPDOM(w, s)
      if l=w then a:=w, b:=s
        else a:=next parent node in post-dominator tree after l toward s, b:=s
        for each node in post-dominator tree from a to b do
          register control dependence of each node on w

procedure searchLeastIPDOM(a, b)
  node t:=b
  while t exists such that t≠ipdom(a) do
    t:=ipdom(t)
  return node t

Figure 3.1: Algorithm for Determining Control Dependence

is E in the post-dominator tree. Nodes on the path from node E to node B in the post-
dominator tree are control dependent on node A except node E. Therefore, node D, C and
B are control dependent on node A. In the case of the edge (A, F), node G and F are
control dependent on node A. For the edge (B, C), since node C is post-dominator of node
B, node C is not control dependent on node B. The example of figure 2.4(a) has the OD
type unstructured code between node I and node J. This makes node H is dependent on
both node G and node I simultaneously. Therefore, as mentioned above, unstructuredness
is an obstacle to the traversal of the CDG. Figure 3.2 shows the CDG example of the CFG
in figure 2.4(a).

3.2 Computing Worst and Best Case Execution Time

In a CDG, a node represents a basic block. In order to facilitate computing the
execution times, this study sorts nodes into three types: code, loop and predicate. Code
type nodes become leaf nodes in the CDG. The code type nodes are all nodes that share
common control dependence and do not have a child node. Of course, code type nodes do
not generate control dependence. Control dependence arcs emanate from predicate nodes.
Predicate nodes bring new control dependence out. The basic block terminating with a
conditional branch can be a predicate node in the CDG, so that a predicate type node has
two successors in the CFG. In this study, multi-way jumps are not allowed since branch instructions can have only one or two successors. Multi-way jumps are divided into multiple basic blocks corresponding to the definition of basic blocks like definition 5. Nodes of two branches from the predicate node have control dependence on the predicate node. Loop nodes represent parts of a loop, which means a loop head, a loop body, and a loop tail. Since special attention is required for the parts of a loop when the execution cycle counts are computed, nodes in a loop become a loop type. An exit from a loop also results in control dependence, since the exit node must have two successors in the CFG. In this sense, loop nodes are a special type of predicate node.

The control dependence graph suggests an algorithm to compute the WCET and BCET more elegantly than tracing every possible execution path in the CFG. A related technique is tree-based static timing analysis [CP01]. When we use the CFG, every possible execution path needs to be traced while storing its accumulated execution cycle counts, and then compared to each other path reaching the exit node of the CFG. This causes exponentially growing time and space complexity. On the other hand, calculating and merging the execution cycles from the bottom level upward to the top-level gives the WCET and BCET in the CDG. Since the merging process chooses the paths that we want and discards other paths, less computing is required for calculating the execution times. The
complexity becomes linear to the number of basic blocks.

Traversal of the CDG starts from the node at the lowest hierarchical level. After summing the execution cycles of the same branch condition, the cycle counts for the two conditions are compared. For the WCET, the longer value is chosen and for the BCET the shorter is chosen. The chosen execution cycles are propagated into the upper level and the same process repeats again until the top level is reached. Since the leaf nodes of the CDG are code nodes, the execution cycle counts of a code node are simply summed up and propagated to the upper level. In the case of loop nodes, the execution cycles must be computed by multiplying the iteration counts with the execution cycles of the loop parts.

For instance, node C and D are loop type nodes on the false branch condition from node A. Execution times with node C and D are computed considering loop iteration counts, and summed with the time of node B. This gives the execution time for the false branch. After computing the time for the true branch with node F and node G when we assume that the execution times for node G are already computed in figure 3.2, we choose one of them as the WCET and another one as the BCET for node A. Node A and E are always executed, so we add their execution times together so that we finally have the WCET and BCET for this procedure.

The execution along with the worst-case or the best-case paths may occur or not when the program is actually executed. This is because control paths are dependent on data variables such as conditional variables or loop count variables. However, we have no idea about the data flow simply with control flow analysis and control dependence analysis at this time. To uncover this ambiguity of the execution at the stage of static analysis, data flow analysis is required. Since this study examines the control flow and dependence analysis, the computed execution times merely bound the actual execution times. The analytical WCET will likely be greater than the actual WCET, and the analytical BCET will likely be smaller than the actual BCET.

Beyond this, other factors affect the tightness of the execution time bounds. The first one is caused by the analysis method. As mentioned, we do not have any information about the data flow so that the actual control of the branches or loops is limited at this time. Another factor rests on the characteristics of the target hardware. This is caused by the pipelined processor architecture, different latencies of memory accesses and the ARM instruction architecture. These will be discussed latter.
Chapter 4

Physical Target System

To confirm the feasibility of this study, experiments are performed with a test benchmark program on a real target system. Additional work includes developing the integrated development environment tool chains from several already existing frameworks. This chapter will describe the real physical target system, and the integrated development-environment tool chains.

4.1 Physical Target System

The target system, OKI ML674000 evaluation board is equipped with an ARM7TDMI core which works with a maximum 33MHz clock frequency. The system has 4 heterogeneous memories: internal SRAM, external SRAM, flash ROM and external SDRAM. The processor core has a 3-stage pipeline and each instruction has its own bus cycle count. The internal data bus width is 32 bits and the external memory bus width is 16 bits. These features make setting up an instruction’s execution timing model difficult.

4.1.1 Heterogeneous Memories

The target system has four memories. See table 4.1[OKI03, SST03, OKI99, GSI00]. The on-chip SRAM (scratchpad memory) has the fastest access speed but the smallest capacity among the memories. It can read and write a data chunk in one clock cycle, and the size of the data chunk that we can access at one time is 8, 16, or 32 bits. The core of the target has an internal 32 bit data bus. Thus 32 bits of data can be accessed in a single clock cycle. The external SRAM and the flash ROM show intermediate performance and
Table 4.1: Memories of OKI ML674000

<table>
<thead>
<tr>
<th>Spec.</th>
<th>Int. SRAM</th>
<th>Ext. SRAM</th>
<th>Flash ROM</th>
<th>Ext. SDRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>8KB</td>
<td>1MB</td>
<td>2MB</td>
<td>2MB</td>
</tr>
<tr>
<td>Data Bus</td>
<td>8/16/32 bits</td>
<td>16 bits</td>
<td>16 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>Read Speed</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>2 cycles</td>
<td>2~5 cycles</td>
</tr>
<tr>
<td>Write Speed</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>460~461 cycles</td>
<td>2~5 cycles</td>
</tr>
</tbody>
</table>

the external SDRAM shows the worst access latency. The memory bus width for external memories is 16 bits. The mismatch of data bus width between the internal bus and the external bus splits bus cycles for memory access into two parts.

Generally the flash ROM is used for the purpose of storing program code, and the external SRAM is used as RAM during program execution. However, by using the memory remapping register every memory can be used for any purpose. In the ARM architecture, the execution of the program code starts from the address 0 of the address space. The memory remapping register of the microcontroller selects which memory is mapped to the address 0. By setting the appropriate value in the memory remapping register, the memory mapped to bank 0 can be switched as a programmer wishes. Table 4.2 lists the values for remapping. In this study, the program code is stored in the flash ROM and the stack and heap region are allocated into the external SRAM in order to have a simplified instruction execution-timing model.

Table 4.2: Memory Remap Register Values of OKI ML674000

<table>
<thead>
<tr>
<th>Register Value</th>
<th>Bank 0 Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Flash ROM</td>
</tr>
<tr>
<td>0x08</td>
<td>External SRAM</td>
</tr>
<tr>
<td>0x09</td>
<td>External SDRAM</td>
</tr>
<tr>
<td>0x0C</td>
<td>Internal SRAM</td>
</tr>
</tbody>
</table>

4.1.2 Instruction Execution Cycle

ARM7TDMI has a three-stage pipeline: fetch, decode and execute. The core has four types of bus cycle: internal cycle (I), non-sequential cycle (N), sequential cycle (S), and coprocessor register transfer cycle (C). During the internal cycle I, the processor core does not request a transfer from or to memories because an internal function is executing and
no useful prefetching can be done at the same time. The non-sequential cycle N requests a transfer from or to an address that is unrelated to the address used in the preceding cycles. At the sequential cycle S, the processor core requests a transfer from or to an address which is either one word or one and half-word greater than the address used in the preceding cycle. During a coprocessor register transfer cycle, the processor uses the data bus to communicate with a coprocessor but does not require any action by the memory system[ARM01].

Table 4.3 summarizes the timing parameters of the bus cycles needed for each instruction. In the table, b is the number of cycles spent in the coprocessor busy-wait loop, and n is the number of words to be transferred. m may have from 1 to 3; 1 if bits [32:8] of the multiplier operand are all zero or one, 2 if bits [32:16] of the multiplier operand are all zero or one, and 3 if bits [31:24] of the multiplier operand are all zero or all one.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles Needed</th>
<th>Additional Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Processing</td>
<td>S</td>
<td>+1 for SHIFT(Rs)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+S+N if R15 written</td>
</tr>
<tr>
<td>MSR, MRS</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>S+N+I</td>
<td>+S+N if R15 is loaded</td>
</tr>
<tr>
<td>STR</td>
<td>2N</td>
<td></td>
</tr>
<tr>
<td>LDM</td>
<td>nS+N+I</td>
<td>+S+N if R15 is loaded</td>
</tr>
<tr>
<td>STM</td>
<td>(n-1)S+2N</td>
<td></td>
</tr>
<tr>
<td>SWP</td>
<td>S+2N+I</td>
<td></td>
</tr>
<tr>
<td>B, BL</td>
<td>2S+N</td>
<td></td>
</tr>
<tr>
<td>SWI, trap</td>
<td>2S+N</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>S+mI</td>
<td></td>
</tr>
<tr>
<td>MLA</td>
<td>S+(m+1)I</td>
<td></td>
</tr>
<tr>
<td>MULL</td>
<td>S+(m+1)I</td>
<td></td>
</tr>
<tr>
<td>MLAL</td>
<td>S+(m+2)I</td>
<td></td>
</tr>
<tr>
<td>CDP</td>
<td>S+bI</td>
<td></td>
</tr>
<tr>
<td>LDC, STC</td>
<td>(n-1)S+2N+bI</td>
<td></td>
</tr>
<tr>
<td>MCR</td>
<td>N+bI+C</td>
<td></td>
</tr>
<tr>
<td>MRC</td>
<td>S+(b+1)I+C</td>
<td></td>
</tr>
</tbody>
</table>

These execution cycle properties could help us estimate exactly the execution cycle counts of instructions. However, we are not able to estimate precisely. The actual execution cycle counts vary since the preceding instruction’s bus cycle can be merged with the next instruction’s bus cycle or vice versa. This is a common overlap in the processor’s pipeline. The particular cases when the pipeline overlap occurs vary. For example, the behavior of
the branch predictor or cache memory can either stall or expedite the pipeline operation occasionally. The target system of this study does not have a branch predictor or a cache memory. But external memory access still must be considered. Non-uniform data bus width of processor core and external memories also makes accurate calculation hard. Since the external memories have 16 bit bus width and the processor core use 32 bit bus width, the external memory controller of the microcontroller arbitrates memory accesses.

Beyond these obstacles from the hardware such as the processor, other factors also need to be taken into account when computing the execution cycle counts. One interfering factor is the location of the program code and the data in the memory systems. If the code is allocated into a slow memory (for example, the external SDRAM), the corresponding execution time is longer than when in the faster memory since the instruction fetch can be affected by the latency of the code memory. This situation also occurs when the accessed data is allocated into the slower memory. If the data variables and the program code are scattered on various memories, then for each case, the bus cycle parameters $N$, $S$ and $I$ must have different values. For example, $N$ values of LDR and STR instruction can be different depending on the location of operands. Therefore, we can apply a particular bus cycle parameter for each instruction.

The ARM instruction set architecture also affects the computation of the execution time. As in table 4.3, an operand’s register affects the execution cycle count. When we shift a value by the value stored in a register, instead of an immediate value, it requires additional cycles. In the case of conditional execution, if the condition is not met, it takes only a single clock cycle. As a result, due to the limit of the analysis method and the hardware characteristics the minimum possible execution cycle count is used as the best case and the maximum possible execution cycle count is used as the worst case for an instruction.

As preliminary experiments before estimating the execution times, the execution cycle count of each instruction is measured by the timer of the target system. From this information and the timing specification of the target processor, bus cycle parameters are extracted and bounded with a small margin such as 1 clock cycle[ARM00, ARM01, OKI03]. While parsing the annotated input assembly files, this information is used to set the execution cycle count for each instruction. The execution cycles of the most common instructions listed in table 4.4.
Table 4.4: Execution Cycles of Major Instructions

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Parameters Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data operations (ADD, SUB, etc)</td>
<td>N=4, S=4, I=1</td>
</tr>
<tr>
<td>Branch (B, BL)</td>
<td>N=4, S=4</td>
</tr>
<tr>
<td>Load Register (LDR)</td>
<td>N=4, S=4 I=2</td>
</tr>
<tr>
<td>Load Registers (LDM)</td>
<td>N=4, S=4 I=2</td>
</tr>
<tr>
<td>Store Register (STR)</td>
<td>N=4 (Worst N=6)</td>
</tr>
<tr>
<td>Store Registers (STM)</td>
<td>N=4, S=4</td>
</tr>
<tr>
<td>Swap (SWP)</td>
<td>N=4, S=4 I=4</td>
</tr>
<tr>
<td>Multiply (MUL, MULL, MLA, etc)</td>
<td>S=4, I=1</td>
</tr>
</tbody>
</table>

4.2 Development-Environment Tool Chains

To develop the application on the target system and perform experiments, a development-environment with tool chains is developed. The tool chains mainly consist of an on-chip debugger for run-time debugging via the JTAG interface, a cross compiler, a linker, an assembler, and a software debugger. As the on-chip debugger, Open On-Chip Debugger (OpenOCD) is ported. GNU ARM tool chains offer the cross-compiler, the linker, the assembler and the software debugger for this study. For the graphical user interface, all these components are integrated into eclipse so that users can work with the unified interface. Olimex JTAG-USB Tiny provides the interface between the target and the host systems through JTAG over USB and UART. The overall tool chains are depicted in figure 4.1.

To provide debugging functionality, openOCD is ported and its relevant configurations and scripts are developed. As the debugging front-end, GNU GDB is used. Hence, relevant debugging scripts are also developed. Debugging software for embedded systems is different from debugging general-purposed systems. Since embedded systems have limited hardware resources (such as memory or I/O devices and so on), a software debugger rarely runs together with the debugged software module on the target system. Instead of traditional debugging, embedded systems can be debugged by remote debugging so that the debugger runs outside of the target, such as on a host desktop computer, and controls the target through hardware or a small size software agent running on the target.

The most primitive debugging relies on tracing hardware signals such as data and address buses by logic analyzer or other type of hardware tracer. This method generates a
A tremendous amount of debug data, which is not manageable, and it cannot trace instruction execution directly. Another method is debug monitors running on the target system that transmit monitored debugging information to the host computer. But they require extra software that initializes necessary hardware for debugging (memory, communication channels, etc) and share the system’s resources with the original application software. Therefore, the monitor is not portable to multiple target systems and limits the performance of the original debug software. An in-circuit emulator offers debugging functionality by replacing the target hardware with a special debug variant. However, this requires an additional and specially prepared hardware module.

An alternative is an on-chip debugger, which integrates debug circuitry into the controller, and offers the benefit of in-circuit emulators with much higher flexibility. Since every processor contains the debug functionality, additional and special hardware are not required. The integrated debug circuit communicates with its host system through a special
communication channel and reports debug information without interfering with the original system’s execution. In the case of ARM, the JTAG interface is adopted to communicate between the internal debugging circuit and the host systems. OpenOCD provides the functionality of an on-chip debugger via the JTAG interface for ARM. It consists of several functional blocks. See figure 4.2[Rat05].

![Diagram of software modules of open On-Chip Debugger](image)

The *daemon* module manages the program configuration by evaluating the command line arguments and the CLI module parses the commands. Only the JTAG module accesses the hardware; it offers an interface for other modules to communicate with the target through the JTAG interface. The GDB module uses the GDB remote serial protocol and the flash module works with memory accesses. Among the modules, the *flash* module is modified to support the target OKI ML674000. The target system has two flash ROM modules from SST with 512KB memory capacity each [OKI03]. Although OpenOCD aims to support most flash memories, the flash memory of the target has a slightly different interface from the common interface [SST03]. Therefore, several parts for initialization of the flash memories are modified. When openOCD is launched, several configurations for openOCD itself can be performed. These include JTAG settings and memory settings, such as flash memory space and size, openOCD working space, communication speed, etc.
Chapter 5

Experiments

Together with the target development tool chains, a static analysis tool for ARM named ARMSAT is developed by implementing the prior static analysis methodology. In addition, an existing application, Helix MP3 decoder[HEL] is ported and tested. The actual execution cycle count of each procedure is also measured by a programmable overflow timer of the target system and compared with estimated execution times by the static analysis tool. This chapter will show the actual WCET and BCET are bounded with the analytic WCET and BCET.

5.1 Test Benchmark

As a test benchmark, the Helix MP3 decoder is ported. The size of the executable binary is about 87KB and the stack depth is about 3KB. An MP3 file playable for 5 seconds is located in the flash ROM and fed to the decoder. From the static analysis, 77 functions are identified, of which 55 functions are structured functions and 22 functions are unstructured. Unstructured functions are excluded from these experiments because this study focuses only on structured functions. 27 of the structured functions are in the decoder itself and the others provide auxiliary functionalities such as timer management and interrupt service and so on. Finally, 19 functions directly related with the decoder are selected for the experiments. Figure 5.1 shows the caller and callee relationship of the analyzed 19 functions.

Before analyzing a certain function, its callee functions (located at the lower level in the call graph) must be analyzed since the execution time of the callee function is included
into the caller function’s execution time. For instance, to estimate the execution cycle counts of \textit{AntiAlias}, the execution cycle counts of \textit{MULSHIFT32} must be computed before starting analysis of \textit{AntiAlias}. In the call graph, \textit{CLZ}, \textit{FASTABS}, and \textit{MULSHIFT32} are located at the bottom level of the graph. This implies that in order to analyze other functions at the higher level, these three functions must be analyzed first. This is done by the tool automatically. The test benchmark functions are selected to have both a hierarchy of function calls and stand-alone leaf-node functions in the CG, which do not call other functions.

5.2 Experimental Results

Execution cycle counts are measured by the processor’s 16 bit timer. A certain amount of overhead is included in the results by the measurement code itself. The function
initiating the measurement is called immediately after the entry of the target function, and the function recording the measurement results is called immediately before the return statement. This function call increases the measured execution times by the execution time required for manipulating timer and call & return process of functions. To compensate this error, the overhead is also measured and considered in the final result. Another overhead is caused when the timer overflow is serviced. Since the target system has only 16 bit wide timer counters, timer overflows must be counted when measuring cycle counts longer than the full bits of the timer counter. This results in extra overhead from the timer interrupt service routine. However, when the timer overflow occurs, the actual execution time of the target function dominates the overflow overhead so much that it is ignorable.

Another preparation for analysis is measurement of the iteration counts of loops. If the loop count is hard-coded, it is easily recognizable. However, if the iteration count is determined dynamically, instrumentation is required. The instrumentation gives the range of the actual iteration counts, because during execution of a program, a loop can be executed many times. This information (in an external parameter input file) is used when computing the execution cycle counts of loops in the CDG.

Table 5.1 lists the experimental results. The table shows each actual execution time is correctly bounded by the analytical execution time limits. The error bounds are varied function by function. For certain functions, the error bounds are relatively larger than others. This is because the analysis of this study takes a conservative estimation, but it does not mean the analysis fails to get reasonable results. The conservativeness comes from the three groups as mentioned: representation level, flow facts and execution time modeling. In this study, it is induced mainly from flow facts and execution time modeling rather than representation level.

In the sense of flow facts, absence of data flow analysis affects the result. Data flow analysis can help estimate the execution time more accurately by giving hints when predicate basic blocks are examined to determine the proper execution path. When a predicate basic block is examined, the two branches are considered equally possible execution paths without data flow analysis, even though one of them is hardly taken during actual execution.

For the difference between two BCETs (actual BCET and analytical BCET), an error handling routine such as a condition check at the entry of a function is a good example. Figure 5.2 plots the ratio of two WCET and two BCET together in error bar format from the table 5.1. Here HybridTransform checks assertive conditions before doing its main work and
Table 5.1: Analyzed Functions and Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Best</td>
<td>Worst</td>
<td>Best</td>
</tr>
<tr>
<td>1</td>
<td>CLZ</td>
<td>791</td>
<td>1,883</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>FASTABS</td>
<td>143</td>
<td>143</td>
<td>123</td>
</tr>
<tr>
<td>3</td>
<td>MULSHIFT32</td>
<td>157</td>
<td>160</td>
<td>127</td>
</tr>
<tr>
<td>4</td>
<td>AntiAlias</td>
<td>144</td>
<td>266,077</td>
<td>73</td>
</tr>
<tr>
<td>5</td>
<td>ClearBuffer</td>
<td>157</td>
<td>707,908</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>DequantBlock</td>
<td>947</td>
<td>26,462</td>
<td>740</td>
</tr>
<tr>
<td>7</td>
<td>FreqInvertRescale</td>
<td>147</td>
<td>1,271</td>
<td>129</td>
</tr>
<tr>
<td>8</td>
<td>HybridTransform</td>
<td>836,698</td>
<td>1,103,098</td>
<td>258</td>
</tr>
<tr>
<td>9</td>
<td>idct9</td>
<td>4,431</td>
<td>4,454</td>
<td>3,664</td>
</tr>
<tr>
<td>10</td>
<td>imdct12</td>
<td>2,615</td>
<td>2,625</td>
<td>2,177</td>
</tr>
<tr>
<td>11</td>
<td>IMDCT12x3</td>
<td>26,345</td>
<td>29,385</td>
<td>7,603</td>
</tr>
<tr>
<td>12</td>
<td>IMDCT36</td>
<td>28,538</td>
<td>36,696</td>
<td>20,544</td>
</tr>
<tr>
<td>13</td>
<td>RefillBitstreamCache</td>
<td>551</td>
<td>589</td>
<td>289</td>
</tr>
<tr>
<td>14</td>
<td>WinPrevious</td>
<td>5,106</td>
<td>6,978</td>
<td>371</td>
</tr>
<tr>
<td>15</td>
<td>xmp3_CalcBitsUsed</td>
<td>231</td>
<td>231</td>
<td>207</td>
</tr>
<tr>
<td>16</td>
<td>xmp3_FDCT32</td>
<td>34,577</td>
<td>34,747</td>
<td>14,793</td>
</tr>
<tr>
<td>17</td>
<td>xmp3_GetBits</td>
<td>1,193</td>
<td>1,193</td>
<td>289</td>
</tr>
<tr>
<td>18</td>
<td>xmp3_MidSideProc</td>
<td>4,785</td>
<td>391,726</td>
<td>373</td>
</tr>
<tr>
<td>19</td>
<td>xmp3_SetBitstreamPointer</td>
<td>211</td>
<td>211</td>
<td>165</td>
</tr>
</tbody>
</table>

so does CLZ. Thus, if the assertion condition is met, the later part of the function cannot be executed. Although this error handling case hardly occurs in the real world, without any information about this situation, the tool decides that case is the best execution case.

The BCET error of the function WinPrevious is caused by a bit different reason. This function splits its execution into two parts in the middle of execution. One part executes during a relatively shorter time than the other part does. The tool takes that path as the best case execution path, but this path is not executed, too. Another reason of conservatively estimated BCET is accumulation of errors from the functions at the lower hierarchy. IMDCT12x3 and xmp3_MidSideProc are examples. They call another function during their execution frequently, i.e. by loops. Thus, the called function’s errors are accumulated when the function is called.

The difference between the two WCETs also results from a reason similar to that of the BCET as mentioned above. In figure 5.2, DequantBlock and FreqInvertRescale estimate
unexpected cases as the actual worst case execution. The two functions also have two big parts inside, which are executed exclusively of each other. In the case of \textit{HybridTransform} and \textit{IMDCT12x3}, the errors of the functions at the lower call hierarchy are accumulated.

Conditionally executed instruction may cause some error, but it is covered by the tool. While parsing the annotated assembly files, instructions with execution condition are modeled, so their variation is supported by the tool without difficulty.

In the sense of the execution timing model, the processor’s pipeline overlap, instruction’s operands type and the location of instructions and operands in the memory can be regarded as the error sources. The variation by instruction’s operand types is also supported by the tool. In this study, the program code is allocated into the flash ROM and the stack and heap region are allocated into external SRAM. Given this simple memory usage, the location of program code and data variables are also simply modeled without
difficulty. However, the overlap in the pipeline is a hard problem to deal with. In this study, as discussed in section 4.1.2, the execution time of each instruction is abstracted by the measurement and bounding. This results in conservative estimation.
Chapter 6

Conclusion

The WCET and BCET analysis are essential and useful information when scheduling tasks efficiently and allocating program code and data into the scratchpad memory in real time embedded systems. This study provides a method to estimate the WCET and BCET of a program using well-known program analysis graphs.

The control flow graph is a starting point for the whole analysis steps and the DFS tree from the control flow graph enables identifying loops without increasing time and space complexity so that it is scalable to the number of basic blocks of the test program. Unstructured code is identified by node labeling and the double-painting technique. Since unstructured code makes the overall process much more complicated, this study considers only structured code. The post-dominance tree helps determine the control dependence between the basic blocks. Considering the control flow graph and the post-dominance tree simultaneously, the control dependence graph is constructed, which is the most important immediate step of this study. The ultimate goals, the WCET and BCET, are computed simply by traversal of the control dependence graph.

To check the feasibility of this study, the real target system and its development tool chains are developed. For experiments, an application is also ported into the target system. While developing the development tool chain, on-chip debugger (openOCD) is modified to support the real target system, and integrated into eclipse together with GNU ARM tool chains for user facility. With the real target system and test benchmark, the worst and best execution times are measured as well as loop iteration counts. These actual execution times are compared with the analytical execution times.

The experimental results show that the estimated WCET and BCET bound the
actual WCET and BCET. However, since this study estimates the execution times conservatively, there are some differences between the actual and analytical execution times. Although the differences may increase for certain cases due to lack of data flow analysis and accumulated errors via call hierarchy, the WCET and BCET are estimated properly only relying on the control dependence graph.

This study also contributes by setting up a development tool chain for OKI ML674000 test board. Development of the tool chain starting from the GNU ARM assembler to static analysis tool for the ARM (ARMSAT) processor is also a contribution.

The WCET and BCET are not sufficient information for optimizing the usage of scratchpad memory, but just necessary information. Providing parametric information of data variables along with the worst and the best case execution path is also required in the future. In this study, unstructured code is avoided. However, structuring unstructured code could give better applicability for this work. Data flow analysis could abate the conservativeness and tighten estimation boundaries.
Bibliography


