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 belongs to the heterogeneous memory hierarchy of the systems rather than abstracting the memory at a higher hierarchical position as cache memory does. Unlike cache memory, we can allocate part of program code and data into the scratchpad memory. This enables optimizing the real-time performance in real-time embedded systems. Static timing analysis helps the optimizing processes by providing microscopic information of the application program. The worst case and the best case execution times are estimated by static analysis. Based on these timing information, the techniques using scratchpad memory have been 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, post-dominator trees. During the basic analysis, loops and unstructured code are also identified, which are obstacles to static timing analysis. A control dependence analysis is a convenient way to analyze the execution times, since it shows the hierarchical information of the control structure. By traversal of the control dependence graph, the worst and the best case execution time are computed.

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. With the developed test environment experiments are performed. The experiments show that actual execution times are bounded by the encountered best case and the worst cases. The analytical worst and best case execution times bound the actual execution time boundary again although there are several 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

Electrical and Computer Engineering

Raleigh, North Carolina

2008

Approved By:

Dr. D. S. Reeves
Dr. J. M. Tuck

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

Sangyeol Kang was born in the Republic of Korea in 1976. He graduated from Seoul National University, Seoul, Republic of Korea in 2003. 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, NC in the fall of 2006.
ACKNOWLEDGMENTS

I would like to appreciate people who helped me with this study.

My father, mother and brother always gave me strength to go through this work. My girl friend Eun-Ha also encouraged me.

Dr. Alexander G. Dean served as my academic advisor. He gave me the opportunity to commit for 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 at NC State University. My alumni Byoung-Don and friend Eui-Kyoung were also helpful.
# TABLE OF CONTENTS

**LIST OF TABLES** .............................................................. vi

**LIST OF FIGURES** .......................................................... vii

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

2 Basic Analysis ................................................................. 5
   2.1 Control Flow Graph and Call Graph ................................. 5
   2.2 Depth-First Search Tree and Identifying 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 Real Target System ........................................................... 25
   4.1 Real 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 and Future Work ............................................. 37

Bibliography ............................................................................ 39
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 The 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 ........................................ 9
Figure 2.3 Algorithm for Identifying Loops ......................................... 10
Figure 2.4 Example Graphs ............................................................. 12
Figure 2.5 Algorithm for Seeking Immediate Dominator ....................... 13
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 The Software Module of open On-Chip Debugger .................. 31
Figure 5.1 Call Graph of Analyzed Functions ...................................... 33
Chapter 1

Introduction

As processor technology has evolved dramatically, embedded systems are also equipped with remarkable resources compared to that of the past. This tide enables real-time embedded systems which support multi-task scheduling in real time manner for applications such as multimedia and communication and so on. A task in real-time systems must complete its job within the promised time due. Missing the deadline is catastrophic since it can mean the failure of the system itself. This rule lets programmers put their efforts to secure the timing margin to the deadline and to meet the deadline. However this requires concrete timing information of tasks.

The timing analysis of a task mainly deals with the worst case execution time (WCET) since that is the longest execution time of the task, which affects the schedulability of other tasks. The timing knowledge of tasks can be attained by either dynamic analysis or static analysis. Unlike dynamic timing analysis, static timing analysis guarantees the upper bound of the worst case execution time by testing all execution paths possible [WM01]. Although over-estimated execution time keeps the timing margin too much enough to meet the deadline, it does not let the task scheduling fail. On the contrary, less-estimated execution time imposes the possibility to ruin the whole task scheduling, and eventually systems can fail to operate. [PB00] reviews remarkable researches about the worst case execution time analysis.

On the other sense, static timing analysis might have more particular applicability. In the real-time systems domain, it is helpful to meet real-time deadlines more easily when we try to get better performance. Meeting real-time deadlines effectively depends on several factors such as the task scheduling algorithm, decomposition of a huge task into
appropriately size tasks and so on. Memory is also an important factor for guaranteeing realtime performance since it directly affects the execution cycle counts of instructions.

Cache memory has a virtue of accelerating the execution by keeping data having high space and time locality in the fastest memory. But cache memory abstracts the lower hierarchical memory systems to the processor so that it is not easy to estimate time performance. There is a huge penalty of execution cycles between cache hit and cache miss. If a task has cache misses while executing, the actual execution time becomes longer than the other case i.e. cache hit. To avoid missing the deadline in this situation, the programmer has to lengthen the deadline and eventually real-time performance is lost or else the system must be over-provisioned, which is expensive. If it is possible to predict cache hit or miss, it can help solve this problem. Unfortunately it is not easy to estimate the memory situation at a certain time point, although this is an area of research [WHW+97]. The ambiguity of predicting execution time limits the utility of cache memory for real time embedded systems.

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

To use scratchpad memory effectively and wisely, much research has been performed from diverse directions [ABS01, EKJ+06, AMF+04, GPD07]. These investigations can be categorized by which objects they try to allocate into scratchpad memory. [EKJ+06] and [AMF+04] load program code into scratchpad memory. These research are about partitioning and selecting candidate code from programs for scratchpad memory, or about positioning and loading the candidate code into scratchpad memory. [AMF+04] allocates codes into scratchpad memory and load statically while [EKJ+06] does it dynamically. Both of them import an objective function to select appropriate code fragments by estimating execution time. The result value of the objective function can be optimized by 0-1 knapsack problem solver using integer linear programming. Then the solution of the problem determines the allocation of the code. [ABS01] and [GPD07] try to allocate data into scratchpad
memory. Especially in [GPD07] preemption threshold technique is used in order to make the allocation process into a virtuous cycle. Both of them rely on the WCET to select data variables for the scratchpad memory.

When program code and data are allocated into scratchpad memory leaving from the original location, the newly allocated address of data variables and code is changed. This process generates another worst case execution time and path. Even after applying optimized allocation of the program code and data, the worst case execution path is vulnerable but converges toward a certain point. Static analysis of the program is necessary in order to create a parametric model which provides prompt the WCET and BCET information about this code repeatedly.

Estimation of the execution time is necessary regardless of whether allocating of code or data. This study provides an automatic method to compute WCET and BCET (best case execution time) by statically analyzing a program relying on several well-known graphs such as control flow graph, post-dominance tree, control dependence graph and their relevant graphs. Firstly, the control flow graph (CFG) and depth-first search tree (DFST) are computed and loops are identified. While constructing the CFG, unstructured code fragments are found. Unstructured code can be identified by a simplified node labeling technique from [CFS90] and [Dea00]. To create the control dependence graph (CDG), a post-dominance tree is constructed. The CDG is analyzed to compute WCET as well as BCET. [CP01] has suggested a similar tree-based static analysis framework based on the CFG and the syntax tree to represent high level language during analysis. However, [CP01] adopts three simulation results such as instruction cache, branch prediction and pipelined execution, and works relying on the CFG. However the static analysis framework of this study works relying on the CDG and aims at the technology using scratchpad memory to optimize the real-time performance.

This work contributes an efficient way to get WCET and BCET for ARM processor code. To verify the feasibility of this study and its applicability, a real target system (the core of which is ARM7TDMI) and its development environment tool chains are developed. A real embedded application is also ported to the target with help of the development tool chains. To confirm the actual execution time of the test benchmark is bounded by the analytical execution time, a static analysis tool for ARM named ARMSAT is developed by applying the methodology of this study. The experimental results show that the methodology of this study is applicable.
This thesis introduces a method and algorithms to analyze programs statically before control dependency analysis in chapter 2. Chapter 3 suggests a method to construct control dependence graph and to compute WCET and BCET from it. Chapter 4 describes the real target and the development environment and chapter 5 shows the corresponding experimental results. The overall conclusions are made in chapter 6.
Chapter 2

Basic Analysis

The static analysis of this study can be conquered by two steps; basic analysis, and control dependence analysis to compute WCET and BCET. Basic analysis generates the well-known program analysis graphs required for the step of control dependence analysis.

First, 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 search and node labeling respectively. The post-dominance tree is constructed from the control flow graph then, the control dependence graph is constructed at the next step. As auxiliary graphs, the depth-first search tree, breadth-first search tree and dominance tree are also obtained. 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 consists of basic block as its node. The following is the definition of a CFG from [FOW87].
Definition 1 A control flow graph is a directed graph $G$ augmented with an unique ENTRY node and an 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, basic blocks are identified by detecting a chunk of instructions starting with an internal code label or just after terminating another basic block, and finishing with particular termination instructions. A common concept of basic blocks is used, i.e. if control flows into the entry of a basic block, it should emanate from the exit of the basic block. There can not be another exit, thus only one and unique control flow exists inside a basic block. There is no instruction which transfers the control flow to another direction in a basic block. Since only branch(B) or conditional branch (ex. BEQ) can transfer the control flow in the ARM instruction architecture which is targeted by this study, only those instructions can terminate a basic block[ARM00]. This also does not allow more than two successors because those instructions can have only one destination operand. In the ARM instruction set architecture, normal conditional instructions besides conditional branch are executed for
resolving the corresponding condition so that they do not terminate a basic block. Although the CFG of this phase is generated from the instruction information directly, it is temporary since it must be verified to meet the above definition. This will be discussed in 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 basic block B has only A as a unique predecessor, and the predecessor A has also only B as a unique successor. This phase reduces the number of basic blocks. As a consequence following analysis of relevant graphs in future can be done more easily and faster. In some sense, this follows the definition of a basic block more strictly.

At the last phase, unique ENTRY and EXIT nodes are verified. This is for following the definition of a CFG. A CFG must have a unique ENTRY and EXIT node. The node which does not have predecessor in the temporary CFG by the second phase is regarded as the ENTRY node. To identify the EXIT node is a little more complicated. If a basic block terminates with an instruction touching the PC(program counter) register or loading an address into PC register, the basic block is looked upon as the EXIT. This criterion identifies ENTRY and EXIT nodes. Then, the uniqueness of ENTRY and EXIT node is going to test immediately.

If there exist multiple ENTRY nodes, a dummy basic block with no instructions is augmented at the head of CFG. The corresponding control flow is also updated so that the added dummy basic block is followed by the multiple ENTRY basic blocks. This is reasonable in the sense that the dummy basic block does not affect the computational result or execution times but it conserves the original execution path. On the other hand, the case that there is no ENTRY does not occur since we may always consider the first basic block in program order as the ENTRY.

In the case of no existence of an explicit EXIT from a procedure, the same dummy basic block is augmented at the tail of basic blocks list as EXIT node. This occurs when a procedure ends with infinite loop. In this case the instruction which loads an address into the PC register is missing. The potential problem of this control structure becomes serious when computing post-dominance tree: while computing the post-dominance tree, the computing algorithm starts from EXIT node and works toward ENTRY node. This situation can be avoided by augmenting a dummy basic block as the successor of the infinite loop’s tail. If
there are multiple EXITs, same policy is applied too, which augments a dummy basic block as a common EXIT node. Of course the control structure of multiple EXITs are transformed to have the augmented block as their successor. This is also 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 of procedure calls. By detecting branch with link (BL) conditional branch with link (ex. BLEQ) instructions, we can know a procedure calls another procedure. The label operands of those instructions identify called procedures [ARM00]. Hence, while examining all instructions to construct the CFG, call graph can be obtained.

2.2 Depth-First Search Tree and Identifying Loops

Intuitively, a loop is a group of code whose execution repeats. The actual execution cycle count of a loop is calculated by multiplying execution cycle count of a loop code with the number of loop iterations. Loops occupy such a considerable portion of total execution cycles 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 way to identify loops by a depth-first search tree.

Identifying loops is two fold problem. The first problem is to identify basic blocks which are part of the loop, including loop head and tail. The loop body can be distinguished after loop head and tail are found. Therefore, identifying the loop head and tail is the key to the first problem. The second problem is to know 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 contains the information explicitly. In the case of dynamic assignment i.e. assigned during runtime, the loop head or loop tail does not provide the exact count until we monitor actual execution. This is more complicated than the former case, although the former case requires additional data flow analysis. In any case, since recognition of loop count requires additional work, loop counts are imported from external inputs which is gathered by a manual method (like instrumentation) for this study. The process flow diagram of this study, figure 2.1 shows an input file with this information.

Identifying loops can be achieved by simply following the control flow directly and remembering the visited nodes in CFG. If a remembered node is visited again while traversal of the CFG, a loop is detected. But this approach is not scalable with the number of nodes
(basic blocks), particularly the number of predicate (conditional) basic blocks of program. When following control flows, the visit history of a node should be kept in the node to check whether that is revisited or not. If the node is revisited, that node is regarded as loop head and the node which is starting node of the control edge as loop tail. This incurs high time and space complexity when a predicate basic block is visited. Control flow of branch and conditional branch is split into two directions - branch taken or not taken. Before splitting, the stored history must be replicated for propagating the history to next node. This process is a burden in time and requires huge memory space. More predicate basic blocks lead to more time to copy 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 conceals the time of replication. 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 the DFS order. [Hav97] augments additional information to a node, which is the index of last child node below a node. This information can tell the control flow ancestor node of a certain node. The last child information eliminates the need to store visit history. Actually, although we store the total history of visiting, finding the back edge refers only a fraction of the whole history. 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.

```plaintext
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)
  last[number[a]]:=current-1
```

Figure 2.2: Algorithm for Depth-First Search
The step of identifying loops is composed mainly of traversing the DFS tree and examining a CFG node’s successors. In the DFS tree, a control edge which is headed from a left-side branch to a right-side branch can not appear 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 A and the edge’s head is 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 appears just below the current node A. Thus if a control flow from left-side to right-side branch exists, the branch should point downward. Finally, if there exists a control flow A→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 is where the index of B is greater than the index of A although A→B. Another sufficient condition for this case is back arc of loop. It is apparent that in DFS tree the right-side always has a greater index than the left-side. The case of back arc is also definite because that is the definition of back arc itself. If two cases can be distinguishable, loops can be identified. If the DFS index of A is greater than the DFS index of the target node, that edge is either a simple control transition from right-side branch to left-side branch or a back arc of loop. When the edge is for a simple control transition, the index of node B is greater than the index of the node A’s last child node. Otherwise, the edge is back arc of loop. This can be explained well with an example. See figure 2.4(b).

```
procedure IdentifyLoop (G)
    number nodes of G by DFS order
    for w:=head of G to tail of G do
        for each node b such that ∃ edge (w, b) do
            if (IsAncestor(b, w)) then register loop

procedure IsAncestor (x, y)
    if (DFS index of y < DFS index of y’s last child) then
        return true
    else return false
```

Figure 2.3: Algorithm for Identifying Loops

In figure 2.4(b), 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 left-side branch to right-side branch. While traversing DFS tree, one successor of D is a back arc toward the node C. First the algorithm detects the DFS index of C is smaller than the DFS index of D. Then while examining the last child node of C i.e. E(5), since the index of node D is smaller than E(5), we conclude the arc must be back edge. The node C becomes loop head and the tail node D of the back edge becomes corresponding loop tail automatically. Figure 2.3 describes the algorithm to identify 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 has been already known, 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 successors of that node will not be visited, hence that control flow is eliminated from the traversal path. Afterwards all double painted nodes are in the body of the loop. This double-painted method also works for 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 an important step since the control dependence is determined by considering both control flow and post-dominance simultaneously. Although our immediate goal is to get post-dominance tree, the dominance tree is discussed ahead of post-dominance tree since the algorithms to build dominance tree and 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 IDOM(B) contains exactly one node, the immediate dominator of such that if N is B's immediate dominator, then every node in DOM(B)-B is also DOM(N).
Figure 2.4: Example Graphs
By definition, DOM(B) contains every node N that dominates B as well as the node B itself. Practically the closest dominator of node B is useful, since DOM(B) contains whole dominators of B. Another point to pay attention to is that the definition of DOM(B) specifies the existence of a unique ENTRY node. The verifying stage of the CFG ensures a unique ENTRY node exists. Hence we do not need to be concerned with the ENTRY node here. IDOM(B) gives us the information which node should be visited in order to reach the node B.

**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

[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 is 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 ENTRY and EXIT node with each other. PDOM(A) let us know which node must be visited after executing the 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 $B \in PDOM(A)$.

The property of PDOM comes with a clue to reveal control dependence among basic blocks. As mentioned above, we can know which node will be executed after executing a certain basic block in future. That means two basic blocks share same control dependence since if the previous node is executed, then the immediate post-dominant basic block will be executed unconditionally. The relationship between the post-dominance and the control dependence will be discussed in the chapter 3 in detail.

As with the dominance tree, when the post-dominance tree is constructed, a unique EXIT instead of ENTRY node is required, and the control flow graph has been verified already. In addition, the same algorithm for creating the immediate dominance tree is utilized again. This is simply achieved by replacing predecessors with successors in the dominance tree algorithm. Thus, before going through the algorithm, the reversed control flow graph should be spanned by the post order traversal.

### 2.4 Unstructured Code

After the basic analysis, unstructured code is need to be identified to facilitate control dependence analysis. Although an original source code usually has a structured control form before compiling, optimized assembly code generated by the compiler can have unstructured code such as multiple exits from a procedure and exits from a loop’s body,
jumping into a decision and so on. Structuring unstructured codes 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, which
means a basic block is control dependent on multiple basic blocks. Multiple control depen-
dence causes multiple instances of traversing the control dependence graph when the WCET
and BCET are calculated, so the algorithm is not scalable with the number of basic blocks.
However, structuring unstructured code requires complicated work with the CFG. Here, to
pay more attention to the computation of the WCET and BCET, structured code will be
assumed. However, to avoid processing unstructured code while computing the execution
times, identifying it is required.

2.4.1 Fundamental Types of Unstructuredness

At the C or C++ language level, structured programs are composed of simple, hi-
erarchical program flow structures, which do not contain GOTO instructions (which changes
control flow arbitrarily). However, at the assembly language level, there are many branch
and conditional branch instructions. Moreover, after compiler optimization, unstructured-
ness 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. It also proves that
unstructured code can be converted into structured code by applying schema algebra in the
particular order to solve the basic types. This is available since the forms can be obtained
by super-positioning 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, ID means ”jump into decision”, OD means ”jump out of decision”.
Similarly IL means ”jump into loop” and OL means ”jump out of loop”. In the case of
the ID type, there is an entry from outside of the decision region (which consists of node A,
B, C) into the decision region. Jumping out of the decision region results in the OD type.
Jumping into loop region A-B-C gives the IL type and 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), the node B is control dependent on the
split node A and 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 the procedure to figure out control dependency more complicated. When the IL or the OL occurs, computing the actual execution cycle count of loops is disturbed. 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. In figure 2.4(a), the OD type unstructured form is shown between the node I and the 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 real world. In the case of the IL and OL type unstructuredness, the unstructuredness is identified by the double-painted method as mentioned already. While examining successors or predecessors of the loop body nodes, if unpainted or differently painted nodes are identified from loop bodies’ predecessors and successors, they point out unstructuredness. However, for the case of the ID and OD, we do not have any information yet. In this study, a simplified node labeling technique is used similar to prior work [Dea00] and [NNS94]. With this technique, the split and rejoin nodes
of a decision region are identified. Then, we apply the double-painted method again to identify the type of unstructuredness.

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

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

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 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 condition until the rejoin nodes are met. If the split label reaches its rejoin node, the branch condition becomes always not true nor false. In addition, if always branch condition is met with either true or false branch, it represents unstructuredness because always branch resolution is not propagated. This may be another way to identify unstructuredness, but here the double-painted method is used.

Figure 2.7 depicts an node labeling example of the previous control flow graph. 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 the node A. Since A is a predicate node, it generates two node labels with corresponding branch resolution; A(F) and A(T). The node B has the label ”A(F)” which means the node A is the predicate node and this node B can be visited
by following false path from the node A. The node F has the label "A(T)". The node C and D are not predicate nodes so they do not generate new labels and the label "A(F)" reaches the node E. Even though the label reaches the end point, we can not conclude the node E is the corresponding rejoin node since it has two other predecessors; H and J. We must await the receipt of the label through them. G is a predicate node, hence it generates the new label "G(T)" and "G(F)". Its successors, H and I inherit the previous node label "A(T)" from G commonly as well as inherit the new label "G(F)" and G(T)" respectively. The node I also generates a new label, then the node H gets all node labels from its predecessors. H combines all inherited nodes. Since the two control streams from the node G reach at H, H becomes the rejoin node for the split node G. However, that is not for the node I, although the decision region starting from I is nested in that starting from G. As mentioned, at this point, we may conclude there are unstructured codes around node H, specifically the split node I. But we will be satisfied with the identification of the split and rejoin node.
Identifying unstructuredness rests on the double-paint method, since it gives an easier way to compare the node label. At the node E, every branch stream rejoins again, hence other predicate nodes rejoin normally.
Chapter 3

WCET and BCET by Control Dependence Analysis

A control dependence graph plays a useful role to analyze a program. It gives the hierarchical information of a program. A control dependence graph is a part of a program dependence graph, which covers the data dependence as well as the control dependence. While control flow graphs show the sequential relationship of the control flow, control dependence graphs show the hierarchical relationship of the control flow. A control dependence graph can be established based on control flow graph and post-dominance tree. By traversal of the control dependence graphs, the WCET and BCET are computed without difficulty.

3.1 Control Dependence Graph

Nodes in a control dependence graph (CDG) are also basic blocks. Some forms of CDG introduce region nodes to group common nodes together explicitly, representing procedures, loops, and areas of predicated (conditional) execution. 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 computing 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 a control dependence graph, and [CFS90] reduces the time and space complexity. Here the way of [CFS90] is used. The detail of building a 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 $Y$.

Part (1) of the definition 5 tells 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 those edges is followed, $Y$ should not be executed and when another edge is followed, $Y$ should be executed. (2) of the definition is satisfied when node $X$ is identical to node $Y$.

Combining a post-dominator tree with a control flow graph gives a basis to build a 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 which can be followed by 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, ENTRY node means a certain execution which invokes the execution of the current procedure.

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

The algorithm can be explained well with our previous example shown in figure 2.4(a). If we choose the control flow edge $(A, B)$, on the post-dominator tree, the least common ancestor of both nodes is $E$. Then, the nodes on the path from $E$ to $B$ on the
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

post-dominator tree are control dependent on node A except E. Therefore, the node D, C and B are control dependent on A. In the case of the edge (A, F), the node G and F are control dependent on A. For the edge (B, C), since C is post-dominator of B, C is not control dependent on B. Our example has an OD type unstructured code between the node I and J. This makes the node H is dependent on both G and 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 time, 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 which share a common control dependence and do not have child node. Of course, code type nodes do not generate control dependence. Control dependence arcs emanate from predicate nodes. Predicate nodes bring a 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, multiway jumps are not allowed since branch
instructions can have only one or two successors. Multiway jumps are divided into multiple basic blocks corresponding to the definition of basic blocks like the definition 5. The nodes on two branches from the predicate node have control dependence on the predicate node. Loop nodes represent parts of loop, which means loop head, loop body and loop tail. Since special attention is required for the part of loop when the execution cycle counts are computed, nodes in a loop become a 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 nodes.

The control dependence graph suggests an algorithm to compute the WCET and the 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 huge a time and space complexity exponential. On the other hand, in the CDG, calculating and merging the execution cycles from the bottom level upward to the top level gives the WCET and the BCET. Since the merging process chooses the paths which we want and discards other paths, fewer computing instances are required for getting the execution time. The complexity becomes linear.
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, the node C and D are loop type nodes and they stand on the false branch condition from A. Execution times with C and D are computed considering loop iteration counts, and summed with the time of B. This gives the execution time for the false branch. After computing the time for the true branch with F and G when we assume that the execution times for G are already computed in figure 3.2, we choose one of them as the WCET and another one as the BCET for the node A. The node A and E are executed always, we add the execution times of them together so that we have finally 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 changing control paths relies on the data of 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 is with the control flow and dependence analysis, the computed execution times bounds the actual execution times. The analytical worst case execution time will likely be greater than the actual worst case execution time, and the analytical best case execution time will likely be smaller than the actual best case execution time.

Beyond this, other factors affect the tightness of our 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 characteristic of the target hardware. This is caused by the pipelined processor architecture, different latencies of memory accesses and ARM instruction architecture. These will be discussed latter.
Chapter 4

Real Target System

To confirm the feasibility of this study, experiments are done with a real target system and a test benchmark is ported. This is done together with developing the integrated development environment tool chain from several already existing frameworks. This chapter will describe the real target system, and the integrated development-environment tool chain.

4.1 Real Target System

The target system, OKI ML67400 evaluation board is equipped with 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. 3 stage pipeline processor core and instruction’s bus cycle make estimation of instruction’s execution time difficult as well as heterogeneous latency of memories.

4.1.1 Heterogeneous Memories

Our target system has four memories. See table 4.1.1[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 which we can access at one time is 8, 16, or 32 bits. The core of our target has an internal 32 bit data bus. Thus 32 bits data can be accessed by a single clock cycle. The external SRAM and the flash ROM show immediate performance and the external SDRAM is the slowest memory. The memory bus width for external memories except scratchpad memory is 16 bits. This splits bus cycles into two parts since the processor
has 32 bit wide bus.

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>

Generally the flash ROM is used for storing program code, and the external SRAM is used as RAM for the program execution. However, every memory can be used for any purpose by using the memory remapping register. In the ARM architecture, the code is executed starting from the address 0 of the address space. The memory remapping register selects which memory is mapped to the address 0. By setting appropriate values in the memory remapping register, the memory mapped to the bank 0 of the address space can be switched to what we want. Figure 4.1.1 shows the values for remapping. Here, the program code is stored in the flash ROM and the the stack and heap region are allocated into the external SRAM in order to have a simplified instruction execution time set.

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)[ARM01]. During internal cycle I, the core does not request a transfer because an internal function is performing and no useful prefetching can be done at the same time. A non-sequential cycle N requests a transfer from or to an
address which is unrelated to the address used in the preceding cycles. At 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 coprocessor register transfer cycle, the processors use the data bus to communicate with a coprocessor but does not require any action by the memory system.

Table 4.1.2 summarizes 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 4.3: ARM Instruction Speed Summary

<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>+I 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 the execution cycle counts of instructions. However, we are not able to estimate precisely the actual execution cycle counts of an instruction from these information. The actual counts vary since the preceding instruction’s bus cycle can be merged with the next instruction’s bus cycle. Non-uniform
data bus width of processor core and external memories also disturbs our accurate calculation. Since the external memories use 16 bit bus and the processor core use 32 bit bus, the external memory controller of the microcontroller arbitrates memory accesses. Beyond the obstacles from the processor, there are other factors to take into account when computing the execution cycle counts.

Another 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 cycle count is larger than when in the faster memory since the instruction fetch can be affected by the latency of the code memory. This situation also appears 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 parameter line N, S and I must have different values. For example, N values of LDR and STR instruction can have difference values depending on the location of operands. Therefore, we can apply a bus cycle parameter for every instruction.

The ARM instruction set architecture also affects the computation of the execution time. As in table 4.1.2, operand’s register affects the execution cycle count. In the case of conditional execution, if the condition is not met, it takes only 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 fundamental experimental information for computing the execution timing, the execution cycle count of each instruction is measured. From this information and the timing specification of the target processor [ARM00, ARM01, OKI03], bus cycle parameters are extracted. While parsing the annotated input assembly files, this information is used to set the execution cycle count for each instruction. Here the program code is allocated into the flash ROM and the stack and the heap region are allocated into the external SRAM. The execution cycles of the most common instructions listed in table 4.1.2.
Table 4.4: The 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</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 are developed. The tool chains mainly consist of an on-chip debugger for run-time debugging via 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 offers 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 a 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 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 on outside of the target such as 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
Figure 4.1: Development-Environment Tool Chain

tremendous amount of debug data which is not manageable, and it can not trace instruction execution directly. Another method is debug monitors run on the target system and transmit monitored debugging information to its host computer. But they require extra software which initializes necessary hardware for debugging (memory, communication channels, etc) and share the system’s resources with original application software. Therefore the monitor is not portable to multiple target systems and limits the performance of 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 with the internal debugging circuit with the host systems. OpenOCD provides the function of an on-chip debugger via the JTAG interface for ARM. It consists of several functional blocks. See figure 4.2[Rat05].

![Figure 4.2: The Software Module 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 JTAG interface. The GDB module uses the GDB remote serial protocol and the flash module works with memory accesses. Among the modules, *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 slightly different interface from the standard [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

Along with the target development tool chain, a static analysis tool for ARM named ARMSAT is developed relying on the prior static analysis methodology. An existing application is ported and tested. The actual execution cycle count of each procedure is also measured by a programmable overflow timer of the real target and compared with estimated execution times by our static analysis tool. This chapter will show the actual execution cycle counts bounded with the analytic counts.

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 external SRAM and fed to the decoder while doing these experiments. After the static analysis, 77 functions are identified, of which 55 functions are structured functions and 22 functions are unstructured. Unstructured functions are excluded from the experiments because this study focuses on structured functions. 27 of the structured functions are in the decoder itself and the others provide an auxiliary functionality such as managing timer and interrupts. Finally, 19 functions directly related with the decoder are selected. Figure 5.1 shows the caller and callee relationship of the analyzed 19 functions.

Before analyzing a function, its callee functions (located at the lower level in the call graph) must be analyzed. For instance, to estimate the execution cycle counts of AntiAlias, the execution cycle counts of MULSHIFT32 must be computed before starting AntiAlias.
In the call graph (see figure 5.1.), CLZ, FASTABS, and MULSHIFT32 are located at the bottom level of the graph. This implies that in order to analyze other functions which are placed at the higher level, these three functions must be analyzed first. The test benchmark functions are selected to have various types: both a hierarchy of function calls and stand-alone leaf-node functions which do not call other functions.

### 5.2 Experimental Results

Execution cycle counts are measured by the processor’s timer. Some overhead is caused by the measurement code itself. The function starting the measurement is called immediately after to the entry of the measured function, and the function finishing the measurement is inserted immediately before the return statement. These function calls increase the actual execution cycle counts but do not count the entry part and the returning
part of the measured function. To compensate for these errors, the overhead is also measured and included in the result. Another overhead is servicing the overflow of the timer. Since the target system has only 16 bit wide timer counters, overflows are also counted to measure longer cycle counts than supported by the full bit size of the counter. This results in measurement overhead from both the timer interrupt routine and counting and resetting the overflow. When the timer overflow occurs however, the actual execution time of the function dominates the timer overhead so much that it is ignorable.

Another preparation for analyzing is measuring the iteration counts of each loop. If the count is hard-coded, then the iteration value is recognized easily. However, if the iteration count is determined dynamically instrumentation is used. The instrumentation gives the range of the iteration counts, too. This information (in an external parameter input file) is used when computing the execution cycle counts of loops in the CDG.

Table 5.2 lists up the experimental results of the analysis. Each actual execution time is correctly bounded by the analytical execution time limits. This means that the analysis finds a conservative estimation of the execution time, but it does not mean our analysis fails to get reasonable results.

The conservativeness mainly comes from absence of data flow analysis. Data flow analysis can contribute to estimating the execution time more accurately by giving hints for predicate basic blocks or dynamically determined loop iteration counts. Since the loop iteration counts are derived from instrumentation, the predicate basic block results in over-estimated execution times. When there is a predicate basic block, without data flow analysis the two branches are equally considered possible execution path even though one of them is not be performed while actually executing. A condition check at the entry part of a function is a good example. For example, the entry part of HybridTransform checks assertive conditions before doing its main work. Hence if the assertion condition is met, the execution of later body can not occur. But while testing, assertion does not occur.

Loop iteration counts can also affect the analysis. If a function has multiple loops (whether nested or not), the worst case execution time assumes the largest count for each loop. Sometimes this is unrealistic. For example, assume the loop $L1$ is nested within the loop $L2$. $L1$ has a maximum iteration count from 10 to 20, and $L2$ has a maximum count 1 to 10. Then the worst case iteration for $L1$ is 20 and 10 for $L2$, hence the maximum iteration counts become 200. If this situation can not actually occur, then the real worst
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>23,376</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>347</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>

case iteration counts must be smaller than 200. This causes over estimation. The Dequant function is an example of combination of overestimation of branch conditions and loop iteration counts.

Referring to HybridTransform and ClearBuffer, the loop iteration counts also affects the result in another aspect. Necessarily the execution time is overestimated as mentioned. But if the overestimated execution time is within a loop with a large iteration count, the gap increases. Both functions have a huge loop inside.

Due to the variance of the processor pipeline’s bus cycle and memory latencies, the execution cycle count of each instruction is also bounded with the best case and the worst case. Therefore, there are two bound inputs for computing - instruction’s execution counts and loop iteration counts. This information is combined with the control flow information and control dependence of a program so that we can estimate the worst and best case execution time.
Chapter 6

Conclusion and Future Work

The WCET and BCET analysis are essential and useful when allocating code and data into scratchpad memory regardless of the method used for partitioning, selecting and loading of code and data. This study provides and tests the detailed steps using well-known program analysis graphs.

The control flow graph offers a starting point for the whole analysis 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. The post-dominance tree helps determine the control dependence between the basic blocks. Since unstructured code makes the overall process much more complicated, this study considers only structured code. Unstructured code is identified by node labeling and double-painting technique. Considering the control flow graph and the post-dominance tree simultaneously, the control dependence graph is obtained, 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, the real target system and its tool chains for developing of application are developed. For testing, an application is also ported into the target system. While developing the development environment, on-chip debugger openOCD is modified to support the real target system, and integrated into eclipse together with GNU ARM tool chain for user facility. With the real target system and test benchmark, the best and worst execution cycles are measured as well as loop iteration counts.

The experimental results show that the estimated worst case and the best case execution times bound the real execution times. This is because the study takes a conser-
ervative estimation of execution time of each instruction and over estimation through control flow.

This study contributes by developing a tool chain for OKI ML674000 test board and developer’s toolkit. The tool chain for program analysis string from GNU ARM assembler to static analysis tool for ARM(ARMSAT) is also a contribution. Existing program analysis graphs are used to get the worst and best case execution times.

The worst and the best case execution time are not sufficient information for optimizing the usage of scratchpad memory, but just necessary information. Providing parametric information of data variables along the worst and the best case execution path is also needed. 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 bounds.
Bibliography


