View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarBank@NUS

TWO CONCRETE PROBLEMS IN TIMING ANALYSIS OF EMBEDDED SOFTWARE

HEMENDRA SINGH NEGI

(B.Tech., IIT Bombay)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgement

I would like to thank my supervisors **Dr. Tulika Mitra** and **Dr. Abhik Roychoudhury**, for their invaluable guidance and support. Without their help, it was almost impossible for me to complete this work.

I would also like to thank my lab-mate Pan Yu for his help in getting me various tools and softwares required. I would also like to thank all my friends for their consistent support and encouragement.

Finally, I would like to thank my parents who stood by me during the hardest time in my life and provided me with encouragement and confidence to complete the work.

Hemendra Singh Negi

Contents

Summary

The design of real-time systems requires a timing guarantee to be given on the execution time of the tasks running in the system. The timing guarantees for the tasks can be given in the form of worst case execution time (WCET) of a program. An upper bound on the WCET of a program can be given by static analysis methods. The problem of determining WCET of a program by static analysis methods has to be solved at the following two levels (1) Programming language level, to determine the longest path in the program and (2) Micro-architectural level, to take into account the effect of features such as pipeline, cache and branch prediction. At the programming language level it is required to detect the infeasible paths in the program and use that information for giving a tight bound on the WCET of the program. At the micro-architectural level, the presence of caches in a real-time system with multiple tasks, results in additional delay in the execution time of the task due to preemption by a higher priority task. Such delays are called as cache related preemption delay (CRPD). It is important to derive an upper bound on the CRPD for the schedulability analysis of tasks running in a real-time system.

In our work we have targeted static analysis both at the programming language level and micro-architectural level. At the programming language level, we have proposed a constraint propagation based technique to determine certain infeasible paths present in a loop in the program. We have also proposed a WCET Analysis technique which uses the infeasible path information to give a tight upper bound on the worst case execution time of a loop. Our experimental results show that our infeasible path detection technique could even detect some of the infeasible paths which are hard to detect from existing infeasible path detection techniques and our WCET computation technique gives tight bounds on the WCET of a program. Further, we have proposed a code transformation based idea for reducing the number of paths in a program that has to be considered during the WCET analysis. Reducing the number of paths is very advantageous for path based WCET analysis techniques which in general are exponential on the number of paths in a program.

We have also proposed a technique to model caches in order to determine a tight bound on the CRPD of tasks. Our technique performs path analysis of both high priority and low priority tasks. Furthermore, we compute all possible states of cache, when the lower priority task gets preempted by the higher priority tasks and when the higher priority task is completed. This is more accurate than the existing set based analysis techniques which estimate the cache states by inferring the set of memory blocks which may exist in each cache block, and thus leading to over-estimations.

List of Tables

List of Figures

Chapter 1

Introduction

The world is moving fast towards embedded systems. A lot of equipments used in the day to day life for e.g. washing machine, mobile phones etc. are embedded systems. An embedded system is classified as *real-time embedded system* if the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced [46]. Further, a real-time system can be divided into two classes: 1) *Hard Real-Time System:* which has to meet strict deadlines. 2) *Soft Real-Time System:* which do not require the same degree of determinism and the task running in it retain some diminished value after its deadline so it should still be executed. Predictability of hard real-time systems is very important as missing of deadline by a task in hard real-time system may cause havocs. In this report we concentrate upon the predictability of real-time system in reference to hard real-time systems.

1.1 Design Issues For Real-Time Systems

The two main issues [1] in the design of real-time systems are: (i) *predictability:* it should be possible to give an upper bound on the execution time of a task, and (ii) *performance:* the real-time system should be fast.

One of the main issue in the design of real time system is: providing a timing guarantee (predictability) for the tasks running in it. The obstacles to the predictability of real-time systems are:

[1]Note that another important issue in the design of real-time system is *power* i.e. a real-time system should consume less power. Detail description of this issue is beyond the scope of this report

1. Large set of inputs: since in most of the cases the set of inputs to an application program in real-time systems could be very large, hence it is not feasible to test all the input cases to determine how much time the application will take to execute in the system.

2. Unpredictable hardware: In the presence of unpredictable components such as caches and pipeline it is hard to derive the exact amount of time taken by a task to run in the system (In this report, we will only concentrate on predictability issues related to caches).

Predictability for real-time system can be achieved either by *static analysis techniques* or by *hardware/software approaches*. Since, it is very difficult and time consuming and in some cases almost impossible to test all the input cases to determine the time taken by a task running in a system. Therefore, static analysis methods which analyze the program statically to give an upper bound on the time taken by a task are very beneficial [31, 23, 36, 32, 29]. On the other hand, hardware/software approaches try to model the system in a way such that it is inherently predictable.

One of the hindrance to the predictability of real-time system is the use of unpredictable hardware such as caches. The performance gain achieved by caches in a system makes their use in real-time system inevitable. Although caches are very effective means of speeding up the memory accesses in the average case. However, the worst case behavior of applications, which is of prime importance in hard real-time systems, is hard to predict in a safe and precise way in the presence of caches, due to the presence of *intra-task* and *inter-task* interferences. Intra-task interferences occur when a task overrides its own blocks in the cache due to conflicts for cache blocks. Inter-task interferences arise in multitasking systems because of the preemptions. The modelling of caches via static analysis methods include worst case execution time (WCET) analysis in case of intra-task interferences [32, 29] and a cache related preemption delay (CRPD) analysis in case of inter-task interferences. Caches can also be modelled for real-time system by various hardware/software approaches which include cache partitioning [25, 33], cache locking [12, 52, 43] etc. In this report we present our technique to determine a tight upper bound on the cache related preemption delay.

Another issue in the design of real-time embedded system is that it should be fast in performance. A real-time system can be tuned for optimal performance with respect to caches, either by analysis of programs' memory performance [20, 13] and applying code transformation accordingly or by design and optimization of cache parameters such as line size, cache size and associativity [24, 35, 49, 30].

1.2 Overview Of The Report

In this report two techniques for the timing analysis of real-time embedded systems are presented. Chapter 2 discusses a technique to determine a tight bound on the cache related preemption delay. Chapter 3 presents a method to determine a bound on time for the execution of loops in a program. Chapter 4 discusses a code transformation based technique which could be used as a pre-processing step to other WCET calculation techniques for reducing their complexity. Finally the last chapter presents conclusion and future work prospects.

Chapter 2

Estimation of Cache Related Preemption Delay

2.1 Introduction

The running of multiple tasks in a real time system with cache results in interferences in the cache, due to the replacement of memory blocks of one task by another task at the time of preemption. Such type of interferences as said before are known as inter-task interferences. Due to the preemption of a lower priority task by a higher priority task some memory blocks belonging to lower priority task might get replaced, therefore when the lower priority task resumes its execution it has to incur some extra cost by bringing back the replaced memory blocks (if they are required) from the memory to the cache. This additional cost incurred by the lower priority task is known as Cache Related Preemption Delay (CRPD) and it occurs when the useful cache blocks of the lower priority task get replaced by the higher priority task. The useful cache blocks at any point in the program are those cache blocks whose contents would be used again later in the program before being replaced.

The estimation of CRPD is very important for the schedulability analysis of tasks. In particular, [27, 28] reports how CRPD can be used to derive accurate response times of multiple periodic tasks running on a single processor for fixed-priority preemptive scheduling. There exists other ways such as cache partitioning (for e.g. [25, 54]) to avoid the analysis of CRPD. However, such an approach results in severe degradation of performance and might also require changes to hardware or software depending upon how the partitioning is controlled. Thus, for multi-tasked real-time systems with caches, CRPD estimation is necessary to evaluate task preemption costs.

To determine a tight bound on CRPD, it is important to consider the effects of both high and low priority tasks. When a task τ_0 is preempted by task τ_1 it is necessary to consider the following:

- All the memory blocks of τ_0 which are in the cache when τ_0 is preempted may not be re-referenced after resumption.
- All the memory blocks of τ_0 which are in the cache when τ_0 is preempted may not be replaced by τ_1.
- There are several possible cache contents when τ_0 is preempted (resulting from the different paths of τ_0).
- There are several possible cache contents when τ_0 resumes (resulting from the different paths of τ_1.)
- There are several possible memory reference patterns after τ_0 resumes execution (due to different paths of τ_0).

The importance of some of these factors on CRPD is mentioned in [17]. However, no estimation technique is given and only simulation results are presented.

2.2 Related Work

In [3], Basumallick and Nilsen extended the rate monotonic analysis to take into account inter-task interferences in the the form of cache related preemption delay. Some of the works on estimating the CRPD are presented in [51, 27, 28, 17]. Tomiyama and Dutt in their approach [51] have assumed that all the cache blocks are useful at the time of preemption, thus reducing the CRPD calculation to finding out the maximum number of cache blocks used by high priority task. They have also shown that it is not necessary that the longest path in the program uses the largest number of cache blocks too. They have used an integer linear programming based approach to determine the program execution path which uses the maximum number of cache blocks. They have shown that by solving the ILP problem, it is possible to achieve a tighter upper bound on CRPD. Lee at. el. in [27] have performed a set based analysis of the cache blocks used by the preempted task. In their approach they try to find out the set of all possible memory blocks that could be present in a cache block at any program point. For their analysis they consider only the low priority tasks. The work of [27] has been extended in [28] to include the high priority task also. Dwyer and Fernando in [17] have given a simulation based approach to calculate the CRPD. They generate a live cache frame [1] distribution of an application by running it under different circumstances and thus find out the maximum number of live cache frames reached at any instant. This gives an upper bound on the CRPD. A caveat to their approach is that it must be possible to find the maximum number of live frames coexistent during the execution of an application under all possible circumstances. They have also proposed refinements to include the effect of high priority task. Busquets-Mataix has proposed an approach to analyze cache eviction cost in a multi-tasking system [11]. However, they also conservatively assume that all the cache lines used by the preempting task need to be reloaded by the preempted task when the preempted task is resumed. In a recent work by Yudong and Mooney [50], the authors have proposed a method to analyze the preemption cost caused by cache eviction in a multi-tasking real-time system. They analyzes the inter-task cache eviction behavior by calculating the intersection set of cache lines used by the preempting task and the preempted task. The authors also do a path analysis to eliminate cache lines that will not be accessed in a task from being used in the estimate. In another recent work [48], Staschulat and Ernst have presented a CRPD analysis approach which considers multiple process activations and preemption scenarios. The authors have proposed a technique which extends the CRPD approach of [38], by propagating replaced cache blocks in the control flow graph. Multiple process activations are modelled by inserting an edge from the last to the first node.

Our approach is similar to the set based approach of Lee et. al. [27]. However, with our approach we are able to compute a tighter bound on CRPD, as we store the relative occurrences of memory blocks in cache. We have later compared our results with the set based approach of [27], to show that our approach gives a tighter bound on the CRPD. Therefore, to realize the usefulness and power of our approach it is important to have a basic understanding of the set based approach of [27]. The description of the set based approach follows.

[1]A ‘live cache frame’ is a cache frame that contains a block that is accessed in the future and without an intervening eviction

2.2.1 Set Based Approach of Lee et. al

Lee at. el. in [27] have performed a set based analysis of the cache blocks used by the preempted task τ_0 before and after preemption. Some of the definitions given by them are as follows.

Definition 2.2.1 (REACHING MEMORY BLOCKS(RMBs)) *The set of reaching memory blocks of cache block c at program point p, denoted by RMB_p^c ,contains all possible states of cache block c at program point p, where a possible state corresponds to a memory block that may reside in the cache block at the point. For a memory block to reside in cache block c, first, it should be mapped to cache block c. Furthermore, it should be the last reference to the cache block in some execution path reaching p.*

Definition 2.2.2 (LIVE MEMORY BLOCKS(LMBs)) *The set of live memory blocks of cache block* c *at a program point p, denoted by LMB_p^c, contains all possible states of cache block* c *at program point p, where a possible state corresponds to a memory block that may be the first reference to cache block* c *after p.*

The iterative equations for calculating the RMBs at various basic blocks of the program are as follows.

$$\begin{aligned} RMB_{IN}^c[B] &= \bigcup_{\textit{X a predecessor of B}} RMB_{OUT}^c[X] \\ RMB_{OUT}^c[B] &= \begin{cases} gen^c[B] & \text{if } gen^c[B] \text{ is not null;} \\ RMB_{IN}^c[B] & \text{otherwise.} \end{cases} \end{aligned}$$

B, X are the basic blocks (note that a program point is taken as an exit or entry of a basic block). RMB_{IN}^c and RMB_{OUT}^c are the RMBs at the beginning and end of the basic block *B*, respectively. $gen^c[B]$ contains as its unique element the memory block that is the last reference to the cache block *c* in the basic block. $gen^c[B]$ is null if basic block *B* does not have any reference to memory blocks mapped to cache block *c*. The equations for LMB is given in a similar manner. The LMBs at various basic blocks of the program can be calculated using the following equations.

$$LMB_{OUT}^c[B] = \bigcup_{\textit{S a successor of B}} LMB_{IN}^c[S]$$

$$LMB^c_{IN}[B] = \begin{cases} gen^c[B] & \text{if } gen^c[B] \text{ is not null;} \\ LMB^c_{OUT}[B] & \text{otherwise.} \end{cases}$$

where $gen^c[B]$ is null if basic block *B* does not have any reference to memory blocks mapped to cache block *c*, otherwise contains an unique element, the memory block that is the first reference to the cache block *c* in the basic block.

A fixed point iteration algorithm is used to calculate the sets of RMB and LMB at various program points, as per the above equations. The initial conditions for RMB calculation via fixed point iteration algorithm are

$$\begin{aligned} RMB^c_{IN}[B] &= \phi \\ RMB^c_{OUT}[B] &= gen^c[B] \end{aligned}$$

and the initial conditions for LMB calculation via fixed point iteration algorithm are

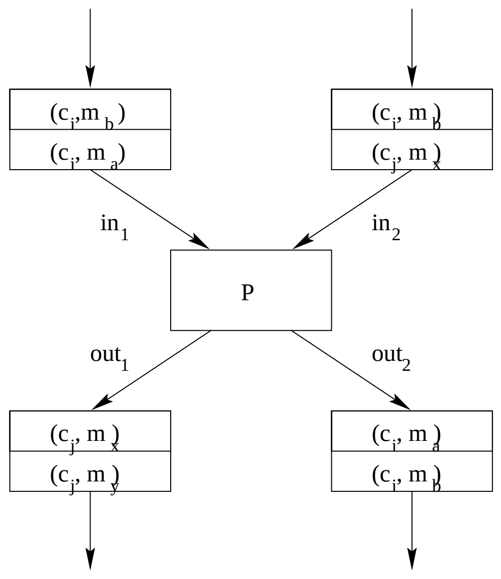
$$\begin{aligned} LMB^c_{OUT}[B] &= \phi \\ LMB^c_{IN}[B] &= gen^c[B]. \end{aligned}$$

Once the sets of RMBs and LMBs are calculated at various program points, the set of *useful cache blocks* are calculated from the intersection of RMB and LMB at various program points. A cache block *c* is useful at a program point *P* if the intersection of sets RMB and LMB for cache block *c* is not null at point *P*. The CRPD incurred, if the interrupt point is *P* would be given by the total number of useful cache blocks at point *P*.

Figure 2.1 from [27] shows an example of the set of RMB and LMB calculated at program point *P*. The cache blocks *i* and *j* are useful at point *P*, as the intersection of sets LMB and RMB is not null.

2.3 Our Technique

In our approach [38], we have refined the existing set based approach [27] to calculate a more accurate bound on the CRPD. Our technique performs path analysis of both high priority and low priority tasks. Furthermore, we compute all possible states of cache, when the lower priority task gets preempted by



RMB	LMB	State	
$\{m_a, m_b\}$	$\{m_a\}$	Useful	cache block i
$\{m_x\}$	$\{m_x\}$	Useful	cache block j

Figure 2.1: RMB, LMB and Useful states at program point P

the higher priority tasks and when the higher priority task is completed. This is more accurate than the existing set based analysis techniques [27] which estimate the cache states by inferring the set of memory blocks which may exist in each cache block.

2.3.1 Motivation

The existing set based approach [27] suffers with an overestimation as they calculate the different possible states of the cache blocks at any instant independently. Therefore, at any instant for each cache block, they give all possible memory blocks those can be present at that instant in that cache block. For example consider Figure 2.2. In Figure 2.2, there are two possible paths to P from X, but only one is possible at any time. Consider a direct mapped cache with two cache blocks. Taking the left path would result in cache state $\{< M_a, M_c >\}$ and taking the right path will result in state $\{< M_b, M_d >\}$ for the cache. [27] will represent the cache state individually as $\{M_a, M_b\}$ and $\{M_c, M_d\}$ for the cache blocks c_i and c_j, respectively. This will actually be counted as cache states $\{< M_a, M_c >\}$, $\{< M_a, M_d >\}$, $\{< M_b, M_c >\}$, $\{< M_b, M_d >\}$, later while calculating the useful cache blocks, since they do not store the relative occurrences of memory blocks in cache. In our approach at each point we actually store the different possible cache states, thus maintaining the relative occurrences of memory blocks in cache. For example, in the above case we will store the cache states as $\{< M_a, M_c >\}$ and $\{< M_b, M_d >\}$, hence

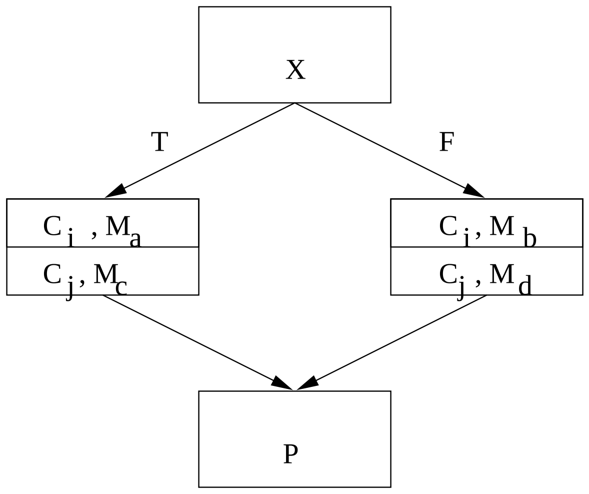


Figure 2.2: Two paths to P from X, only one of it is possible

later while calculating the useful cache blocks, we get accurate and better results than [27].

2.3.2 Approach

Let us define some of the notions first, before giving the description of our approach.

Definition 2.3.1 (Cache State) *A* cache state *represents the contents of cache at any instant. A cache state of null for any cache block denotes an empty cache block. A cache state can be imagined as a vector of size equal to maximum number of cache blocks and containing the memory addresses mapping to the cache blocks.*

In all our calculations we have assumed a direct mapped cache. For a direct mapped cache with n blocks, a cache state can be represented as a vector of n elements $c[0, ..., n-1]$ where $c[i] = m$ if cache block i holds memory block m. Otherwise, if the ith cache block does not hold any memory block we denote this as $c[i] = \perp$. At each program point we try to calculate the different possible cache states, which are called as the *set* of cache states.

Definition 2.3.2 (REACHING CACHE STATES) *The Reaching Cache States at a basic block B of a program, denoted as RCS_B, is the set of possible cache states when B is reached via any incoming program path.*

Definition 2.3.3 (LIVE CACHE STATES) *The Live Cache States at a basic block B of a program, denoted as LCS_B, is the set of possible first memory references to cache blocks via any outgoing program path from B.*

The idea is: for the low priority task, we try to calculate different possible reaching and live cache states of the whole cache, at different program point and then combining each cache state in any set (reaching/live) with every cache state in the other set (live/reaching), we get a set of useful cache states at each program point. We then report the useful cache states which has the maximum number of useful cache blocks. A cache block is useful at any program point if it will be used again during the execution of the program without its content getting replaced. At any program point P, an element R in the set of reaching cache states represent the memory blocks present in cache when the program point P is reached via some path say $p1$. And an element L in the set of live cache states represent the memory blocks which will be the first memory references via some path say $p2$ after the program point P. Hence, for any cache block if the value in R and L is same, then it represents that the content of the cache block will be used again without being replaced in the path $p1$ followed by $p2$. The maximum number of useful cache blocks obtained from our approach is lesser than that obtained from [27]. Further, we consider the high priority task and calculates the reaching cache states at the exit of high priority task. Then, by combining the different possible useful cache states at any program point in the low priority task, with the reaching cache states at the exit of the high priority task, we get the set of replaced cache states at any program point. This set of replaced cache states determines the CRPD. The maximum number of replaced cache blocks in the set of replaced cache states at any program point, gives the CRPD.
The equations for the reaching cache states and live cache states are given by.

$$\begin{aligned} RCS_B^{IN} &= \bigcup_{p \in predecessor(B)} RCS_p^{OUT} \\ RCS_B^{OUT} &= \{ r \odot gen_B | r \in RCS_B^{IN} \} \end{aligned}$$

RCS_B^{IN} and RCS_B^{OUT} are the reaching cache states at the entry and exit of basic block *B*, respectively. gen_B = [m_0,...,m_{n-1}] where $m_i = m$ if m is the *last* memory block in *B* mapping to cache block i in *B* and $\perp$ if no memory block in *B* maps to cache block i. The operation $\odot$ is defined over memory blocks as:

$m \odot m'$ = $\begin{cases} m' & \text{if } m' \neq \perp; \\ m & \text{otherwise.} \end{cases}$

and we assume that any operation $\odot$ over memory blocks can be applied to cache states (by applying the operation pointwise to its elements).

The equations for calculating the live cache states are given by:

$$\begin{aligned} LCS_B^{OUT} &= \bigcup_{s \in successor(B)} LCS_s^{IN} \\ LCS_B^{IN} &= \{l \odot gen_B | l \in LCS_B^{OUT}\} \end{aligned}$$

LCS_B^{IN} and LCS_B^{OUT} are the live cache states at the entry and exit of basic block *B*, respectively. gen_B = [m_0,...,m_{n-1}] where $m_i = m$ if m is the *first* memory block in *B* mapping to cache block i in *B* and $\perp$ if no memory block in *B* maps to cache block i. The $\odot$ operation is the same as given above.

A fixed point iteration algorithm (as given in [40]) can be used to solve the above sets of equations, to get the reaching/live cache states at each program point. Once the fixed point is reached we set $RCS_B = RCS_B^{OUT}$ and $LCS_B = LCS_B^{OUT}$. The initial assignments of variables in the two cases (reaching/live), is as follows.

$$\begin{aligned} RCS_B^{IN} &= \phi \\ RCS_B^{OUT} &= gen_B \\ LCS_B^{OUT} &= \phi \\ LCS_B^{IN} &= gen_B \end{aligned}$$

Calculating useful cache blocks. Solving the above equations would results in two sets (reaching and live) at different program points. From the intersection of each element in one set with every element in another set, we obtain a new set, called useful cache set, denoted as UCS_B. Every element of this set is an array of size equal to total number of cache blocks, and is calculated as follows. If L is a cache state in LCS_B and R is a cache state in RCS_B and combination of L and R results in an array U, in UCS_B, then U is given by:

For cache block c_i if ml_i is the memory block in c_i in L and mr_i is the memory block in c_i in R, then U[i] is given by:

$$U[i] = \begin{cases} 1 & \text{if } ml_i = mr_i; \\ 0 & \text{otherwise.} \end{cases}$$

This way at each program point P, we can get a set, representing the useful cache blocks for different possible program paths through P.

The memory space of high priority task and low priority task are disjoint hence the execution of high priority task after preemption at any program point P might replace the useful cache blocks of low priority task at P. However, it may happen that not all of these useful blocks are replaced by the high priority task. Therefore it is also necessary to know the cache state at the finish of the high priority task. Hence, a set of final cache state (*FCS*) is calculated for the high priority task. The FCS is obtained from the RCS of high priority task, at the last block (exit block). If R is a cache state in RCS_{exit} then a state F in FCS corresponding to R can be calculated as below.

$$F[i] = \begin{cases} 1 & \text{if cache block } c_i \text{ in R is nonempty ;} \\ 0 & \text{otherwise.} \end{cases}$$

Once we have both the UCS (at every point of low priority task) and FCS (RCS at exit of high priority task), we can calculate the CRPD at any point P in low priority task in the following way:

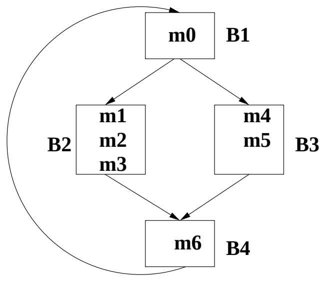
For every U in UCS_B at program point P and from every F in FCS find the number of useful cache blocks in U replaced by F, and report the maximum number of replaced useful cache blocks achieved through any combination of U and F. A useful cache block c_i is replaced, if both $U[i]$ and $F[i]$ are equal to one.

Hence by finding the maximum number of replaced useful cache blocks at any program point in low priority task, we can achieve a tight bound on CRPD.

It may appear that our approach can face exponential blowup of cache states. But this is not common in general case, because due to the limited size of the cache, different paths get merge at various basic blocks. A naive example to show this would be to assume that there is a basic block B, which has instructions mapping to all the cache blocks. Thus even if RCS_B^{IN} has more than one elements but RCS_B^{OUT} would only have a single element. This results in reduction of the exponential nature of the approach. In fact our results prove this fact and the execution time of the fixed point algorithm in general is not so high in all our test cases.

2.4 An Example

For better understanding, let us work out an example to calculate the CRPD with our technique. Consider the control flow graph (CFG) shown in Figure 2.3. The CFG consists of four basic blocks (B1-B4) and six memory blocks (m0-m6) within a loop with single if-then-else. A direct mapped cache with four cache blocks (c0-c3) is assumed. As per the CFG the gen_B of various basic blocks are as follows.



c0	m0	m4
c1	m1	m5
c2	m2	m6
c3	m3	

Figure 2.3: An example control flow graph

$$
\begin{aligned}
gen_{B1} &= [m0, \perp, \perp, \perp] \\
gen_{B2} &= [\perp, m1, m2, m3] \\
gen_{B3} &= [m4, m5, \perp, \perp] \\
gen_{B4} &= [\perp, \perp, m6, \perp]
\end{aligned}
$$

The various iterations of the fixed-point iteration algorithm to calculate the RCS_B are shown in Table 2.1. It is important to note here that some cache states might get subsumed by other cache states and are avoided for further consideration in the fixed point iteration algorithm. A cache state c' is subsumed by another cache state c if $\forall i \;\; c'[i] = c[i] \;\; or \;\; c'[i] = \perp$. Thus, in iteration 4 for basic block B3,

$$RCS_{B3}^{IN} = \{[m0, m1, m6, m3], [m0, m5, m6, \perp]\}$$

Iteration	**Basic Block**	RCS^{IN}	RCS^{OUT}
1	B1	$\emptyset$	$[m0, \perp, \perp, \perp]$
	B2	$\emptyset$	$[\perp, m1, m2, m3]$
	B3	$\emptyset$	$[m4, m5, \perp, \perp]$
	B4	$\emptyset$	$[\perp, \perp, m6, \perp]$
2	B1	$[\perp, \perp, m6, \perp]$	$[m0, \perp, m6, \perp]$
	B2	$[m0, \perp, \perp, \perp]$	$[m0, m1, m2, m3]$
	B3	$[m0, \perp, \perp, \perp]$	$[m4, m5, \perp, \perp]$
	B4	$[\perp, m1, m2, m3], [m4, m5, \perp, \perp]$	$[\perp, m1, m6, m3], [m4, m5, m6, \perp]$
3	B1	$[\perp, m1, m6, m3], [m4, m5, m6, \perp]$	$[m0, m1, m6, m3], [m0, m5, m6, \perp]$
	B2	$[m0, \perp, m6, \perp]$	$[m0, m1, m2, m3]$
	B3	$[m0, \perp, m6, \perp]$	$[m4, m5, m6, \perp]$
	B4	$[m0, m1, m2, m3], [m4, m5, \perp, \perp]$	$[m0, m1, m6, m3], [m4, m5, m6, \perp]$
4	B1	$[m0, m1, m6, m3], [m4, m5, m6, \perp]$	$[m0, m1, m6, m3], [m0, m5, m6, \perp]$
	B2	$[m0, m1, m6, m3], [m0, m5, m6, \perp]$	$[m0, m1, m2, m3]$
	B3	$[m0, m1, m6, m3], [m0, m5, m6, \perp]$	$[m4, m5, m6, m3]$
	B4	$[m0, m1, m2, m3], [m4, m5, m6, \perp]$	$[m0, m1, m6, m3], [m4, m5, m6, \perp]$
5	B1	$[m0, m1, m6, m3], [m4, m5, m6, \perp]$	$[m0, m1, m6, m3], [m0, m5, m6, \perp]$
	B2	$[m0, m1, m6, m3], [m0, m5, m6, \perp]$	$[m0, m1, m2, m3]$
	B3	$[m0, m1, m6, m3], [m0, m5, m6, \perp]$	$[m4, m5, m6, m3]$
	B4	$[m0, m1, m2, m3], [m4, m5, m6, m3]$	$[m0, m1, m6, m3], [m4, m5, m6, m3]$
6	B1	$[m0, m1, m6, m3], [m4, m5, m6, m3]$	$[m0, m1, m6, m3], [m0, m5, m6, m3]$
	B2	$[m0, m1, m6, m3], [m0, m5, m6, \perp]$	$[m0, m1, m2, m3]$
	B3	$[m0, m1, m6, m3], [m0, m5, m6, \perp]$	$[m4, m5, m6, m3]$
	B4	$[m0, m1, m2, m3], [m4, m5, m6, m3]$	$[m0, m1, m6, m3], [m4, m5, m6, m3]$
7	B1	$[m0, m1, m6, m3], [m4, m5, m6, m3]$	$[m0, m1, m6, m3], [m0, m5, m6, m3]$
	B2	$[m0, m1, m6, m3], [m0, m5, m6, m3]$	$[m0, m1, m2, m3]$
	B3	$[m0, m1, m6, m3], [m0, m5, m6, m3]$	$[m4, m5, m6, m3]$
	B4	$[m0, m1, m2, m3], [m4, m5, m6, m3]$	$[m0, m1, m6, m3], [m4, m5, m6, m3]$

Table 2.1: Computation of RCS_B for the CFG in Figure 2.3.

and $gen_{B3} = [m4, m5, \perp, \perp]$. Therefore,

$$RCS_{B3}^{OUT} = \{[m4, m5, m6, m3], [m4, m5, m6, \perp]\}$$

However, $[m4, m5, m6, \perp]$ is subsumed by $[m4, m5, m6, m3]$ and hence $RCS_{B3}^{OUT} = \{[m4, m5, m6, m3]\}$. The LCS_B can also be calculated in the similar fashion and are as follows at the fixed point.

$$\begin{aligned}
LCS_{B1} &= \{[m0, m1, m2, m3], [m4, m5, m6, m3]\} \\
LCS_{B2} &= \{[m0, m1, m6, m3], [m0, m5, m6, m3]\} \\
LCS_{B3} &= \{[m0, m1, m6, m3], [m0, m5, m6, m3]\} \\
LCS_{B4} &= \{[m0, m1, m2, m3], [m0, m5, m6, m3]\}
\end{aligned}$$

Given LCS and RCS for each basic block, the useful cache sets (UCS) can be computed.

$$\begin{aligned}
UCS_{B1} &= \{[1,1,0,1],[0,0,1,1],[0,1,1,1],[1,0,0,1]\} \\
UCS_{B2} &= \{[1,1,0,1],[1,0,0,1]\} \\
UCS_{B3} &= \{[0,1,1,1],[0,0,1,1]\} \\
UCS_{B4} &= \{[1,1,0,1],[1,0,1,1],[0,1,1,1],[0,0,0,1]\}
\end{aligned}$$

Now let us illustrate the advantage of our technique over separate analysis of each cache block [27, 28]. In that case, RCS_B and LCS_B have a set of reaching memory blocks for each cache block as shown in the following.

$$\begin{aligned}
RCS_{B1} &= [\{m0\},\{m1,m5\},\{m6\},\{m3\}] \\
RCS_{B2} &= [\{m0\},\{m1\},\{m2\},\{m3\}] \\
RCS_{B3} &= [\{m4\},\{m5\},\{m6\},\{m3\}] \\
RCS_{B4} &= [\{m0,m4\},\{m1,m5\},\{m6\},\{m3\}]
\end{aligned}$$

$$\begin{aligned}
LCS_{B1} &= [\{m0,m4\},\{m1,m5\},\{m2,m6\},\{m3\}] \\
LCS_{B2} &= [\{m0\},\{m1,m5\},\{m6\},\{m3\}] \\
LCS_{B3} &= [\{m0\},\{m1,m5\},\{m6\},\{m3\}] \\
LCS_{B4} &= [\{m0\},\{m1,m5\},\{m2,m6\},\{m3\}]
\end{aligned}$$

Let us consider RCS_{B4}. From separate analysis of cache blocks, we infer that RCS_{B4} can have four possible cache states: $[m0,m1,m6,m3]$, $[m0,m5,m6,m3]$, $[m4,m1,m6,m3]$, and $[m4,m5,m6,m3]$. However, our combined analysis of cache blocks infers that only two of these cache states are feasible. The identification of these infeasible cache states leads to decrease in the number of useful cache blocks (computed via intersection of RCS_B and LCS_B) at each program point. For example, our analysis infers at most 3 useful cache blocks for both B1 and B4 (Even though each of the cache blocks is useful along some path, all 4 of them are not useful along any path). Whereas, with separate analysis of cache blocks, we get 4 useful cache blocks for B1 and B4.

Note that we also maintain the Final Cache States (FCS) of the high priority task as a set of boolean

Program	Description
`matsum`	Summation of two 100×100 matrices
`qsort`	Non-recursive quick sort algorithm
`crc`	Cyclic redundancy check program
`sqrt`	Square root calculation
`eqntott`	Drawn from SPEC'92 integer benchmarks
`des`	Data Encryption Standard
`whet`	Whetstone benchmark
`ssearch`	Pratt-Boyer-Moore string search
`math`	Basic math within nested loop

Table 2.2: Description of benchmark programs.

vectors. This leads to further accuracy in CRPD analysis. For example, suppose we compute $FCS = \{[1,0,1,0],[1,1,0,0]\}$. This will allow our analysis to estimate the number of replaced cache blocks to be 2 leading to even tighter CRPD estimation.

2.5 Experimental Results

In our experiments we used nine different benchmarks (mostly from [32] and [22]) to present the accuracy and performance of our technique. Table 2.2 gives the description of benchmarks used by us. We used the Simplescalar architectural simulation platform [10] in the experiments. All the benchmarks are compiled to Simplescalar assembly language with modified *gcc*. A CRPD analyzer written by us accepts the assembly language code, identifies the basic blocks out of it and constructs the control flow graph (CFG) from it. Given the CFG for the low-priority and high-priority task, our analyzer implements a fixed point iteration algorithm to calculate the RCS and LCS at various program points taken at the exit of each basic block. The calculated RCS and LCS are used to compute the useful cache states (UCS) at various program points and finally the intersection of UCS and final cache states (FCS from high-priority task) is used to determine the CRPD.

We present three types of results to present the accuracy and performance of our technique, and at the same time comparing it with the set based approach.

First we present the results for CRPD analysis. Table 2.3 shows the CRPD values in terms of number of cache blocks for a direct mapped instruction cache with 32 cache blocks. `matsum`, `eqntott`, and `sqrt` are used as higher priority tasks and all others as low priority tasks. The results for actual (A in Table 2.3), set based or separate [27] (S in Table 2.3), and combined or our (C in Table 2.3), analysis of

LP Task	HP Task								
	matsum			eqntott			sqrt		
	A	C	S	A	C	S	A	C	S
qsort	19	20	24	16	22	28	18	19	26
crc	17	17	18	17	21	22	18	19	20
ssearch	19	22	23	19	25	27	21	22	25
des	22	23	24	21	24	26	22	22	25
whet	20	21	22	20	25	25	23	23	24
math	18	22	23	20	25	27	20	22	25

Table 2.3: Accuracy of CRPD analysis for a 32-block cache. A stands for actual value(by simulation), C stands for combined analysis of all cache blocks and S stands for separate analysis of each cache block.

Task	Combined	Separate
matsum	23	24
eqntott	26	28
sqrt	23	26

Table 2.4: Maximum number of cache blocks used by high priority task for a 32-block cache.

all cache blocks are presented. The maximum number of cache blocks used by high-priority tasks are shown in Table 2.4. Our analysis produces much tighter bound on CRPD, with improvement as high as 37% for some benchmarks.

Second we show the maximum number of useful cache blocks of the low-priority task at any program point in Table 2.5. In Table 2.5, A, C and S have their usual meaning as described above. Again our technique results in tighter values for useful cache blocks than the separate analysis of [27].

Third in Table 2.6 we show the number of preemption points (basic blocks) at which useful cache block count differs in our combined and separate analysis as well as the maximum of these differences. It should be noted that even though the maximum number of useful cache blocks over all preemption points

Task	# of Cache Blocks											
	8			16			32			64		
	A	C	S	A	C	S	A	C	S	A	C	S
qsort	1	1	1	14	14	14	26	28	32	51	62	63
crc	2	2	2	12	12	12	22	26	26	47	48	48
ssearch	2	6	6	14	15	16	29	31	31	59	59	59
des	0	0	0	6	12	12	30	30	30	60	60	64
whet	0	1	2	10	11	14	29	29	29	59	59	59
math	3	4	5	10	14	16	27	30	31	63	63	64

Table 2.5: Maximum number of useful cache blocks of the low-priority task at any program point for different cache sizes. A stands for actual value(by simulation), C stands for combined analysis of all cache blocks and S stands for separate analysis of each cache block.

Task	**# of Cache Blocks**							
	8		**16**		**32**		**64**	
	BB	**Diff**	**BB**	**Diff**	**BB**	**Diff**	**BB**	**Diff**
qsort	0	0	1	1	2	4	1	1
crc	2	1	0	0	0	0	0	0
ssearch	0	0	2	2	0	0	0	0
des	0	0	0	0	9	2	47	15
whet	7	1	11	4	0	0	0	0
math	2	1	4	5	2	3	1	1

Table 2.6: Comparison of combined and separate analysis for low-priority task. BB denotes the number of basic blocks at which useful cache block count differs and Diff denotes the maximum of these differences.

Task	**# of Cache Blocks**			
	8	**16**	**32**	**64**
qsort	0.003	0.004	0.022	0.082
crc	0.016	0.015	0.164	65.663
ssearch	0.007	0.025	1.613	16.168
des	0.010	0.022	0.525	55.329
whet	0.005	0.023	4.189	89.858
math	0.013	0.059	1.061	8.414

Table 2.7: Time to compute useful cache blocks for low-priority task in sec.

of low-priority task (shown in Table 2.5) might be the same in both analysis techniques, the estimated number of useful cache blocks in individual preemption points of low-priority task may be different.

We have used pentium 4, 1.7 GHz CPU with 1 GB memory for our experiments. For all benchmarks it takes less than 1.5 minute, to calculate the useful cache states. Also, the cache states are quite less as compared to expected exponential blow up of cache states. For e.g., for `qsort` with 40 basic blocks and 490 memory blocks and 8 cache blocks, the total number of live cache states are 68 and reaching cache states are 69. Table 2.7 shows the time taken by our analyzer to compute the UCS for all basic blocks in the low-priority task.

2.6 Optimization Using Binary Decision Diagrams

In our approach, we represent the cache states at a program point in a task as a set of tuples (where each tuple denotes an assignment of memory blocks to the cache blocks). To avoid an exponential blow-up in the space consumption we can represent the possible cache states at any program point implicitly as a Binary Decision Diagram (BDD) [9, 8]. A BDD is an efficient data structure for representing

c_1		c_2	
	x_1x_2		y_1y_2
M_{a1}	01	M_{a2}	01
M_{b1}	10	M_{b2}	10
M_{c1}	11	M_{c2}	11

Table 2.8: Possible encoding of memory blocks into boolean form.

a propositional logic formula. In this section we show how to translate our approach to use BDD. Although we have implemented our approach using BDD but not much gain was obtained in terms of memory usage. The possible factors for this are discussed later in the section.

Binary Decision Diagram (BDD)

Binary decision diagrams(BDD) [9, 8] are an abstract representation of boolean functions. An Ordered BDD (OBDD) can be obtained from a BDD, by imposing some of the restriction on the ordering of the variables used by the boolean function, such that the resulting form is canonical. An OBDD is a directed acyclic graph with internal nodes corresponding to the variables over which the function is defined and the terminal nodes labelled by the function values 0 and 1.

Encoding of our approach into boolean Form

For the purpose of using OBDD we first have to encode our variables and structures into boolean form. Let us assume we have two cache blocks say c_1 and c_2, and let the memory blocks (only these are mapped) M_{a1}, M_{b1} and M_{c1} gets mapped to cache block c_1 and memory blocks (only these are mapped) M_{a2}, M_{b2} and M_{c2} gets mapped to cache block c_2. We can encode the mapping of memory blocks to cache block in boolean form. For eg. let us have two boolean variables representing the memory block residing in each cache block, further assume x_1 and x_2 represent the memory block residing in cache block c_1, and y_1 and y_2 represent the memory block residing in cache block c_2. We can encode the memory blocks into boolean values. A possible set of encoding is shown in Table 2.8.

The 00 encoding is used to represent an empty cache state. Now for instance at any program point P, let the set of cache states is $\{< M_{a1},M_{a2} >,< M_{c1},M_{a2} >\}$. Hence this set of cache states can be represented as a boolean function *F* (which is a disjoint sum of product cover) of the form.

$$F = \bar{x_1}x_2\bar{y_1}y_2 + x_1x_2\bar{y_1}y_2 \tag{2.1}$$

where each product term represents the possible cache state at the program point P. The function F can then be reduced to

$$F = x_2\bar{y_1}y_2 \tag{2.2}$$

Thus the representation of cache states as boolean function can be used for a more compact representation.

Our representation

For our purpose we assign each basic block with four boolean functions corresponding to RCS_B^{IN}, RCS_B^{OUT}, LCS_B^{IN} and LCS_B^{OUT}, which are stored as BDDs. The previously described LCS and RCS analysis can then be done by manipulating these boolean functions. The initial values for ${RCS_B}^{OUT}$ and ${LCS_B}^{IN}$ can be obtained by just encoding the cache state gen_B in the same way as shown above. The recursive equations for the RCS and LCS analysis can then be modified in the following way.

- The union operation ($\bigcup$) for RCS or LCS analysis would just become the + (boolean OR) operations over various boolean functions representing the different ${RCS_p}^{OUT}$ or the ${LCS_s}^{IN}$ respectively.

- The $\odot$ operation for RCS analysis can be represented in boolean form in the following way.
  ${RCS_B}^{OUT}$ = $f \odot gen_B \mid f$ is a boolean function for ${RCS_B}^{IN}$
  $f \odot gen_B$ can be calculated as follows: Let gen_B = $\{c_1, c_2$,...,$c_i\}$ and let each memory block mapping to a cache block c_i is encoded with *j* variables x_{i1},...,x_{ij}, hence c_i would have a particular set of values(0 or 1) coming from each boolean variable, depending upon the encoding of memory block which is present in the cache block. Now for each non-empty cache block c_i (i.e. at least one of x_{i1},...,x_{ij} has a non-zero value), do the following:
  For each variable x_{ik} first convert all the instances of x_{ik} in *f* into don't-care and then do the boolean "and" of *f* with x_{ik} if x_{ik}=1 in c_i, otherwise do the boolean "and" of *f* with $\bar{x_{ik}}$ if x_{ik}=0 in c_i. To reduce a variable *x* into don't-care in a boolean function *f*, first *f* is projected with x=0 and then it is projected with x=1. A boolean "Or" for the two projection results in a new function *f'* from *f* with *x* reduced to don't-care.
  Let us consider an example with two cache blocks to show this.
  Let at any instance f = $\bar{x_1}x_2y_1\bar{y_2}$ + $x_1\bar{x_2}\bar{y_1}y_2$ and let the gen_B be $\{$<01,00>$\}$, this means that the

cache block 2 is empty, therefore the $f \odot gen_B$ can be calculated in the following way: first take x_1, convert all instances of x_1 in *f* into don't-care, hence *f* will become;

f = $x_2 y_1 \bar{y_2}$ + $\bar{x_2} \bar{y_1} y_2$

and now do boolean "and" of *f* with $\bar{x_1}$ as the value of x_1 in gen_B is 0. hence the *f* would become

f = $\bar{x_1} x_2 y_1 \bar{y_2}$ + $\bar{x_1} \bar{x_2} \bar{y_1} y_2$

following the same terminology for x_2 the resulting f would be

f = $\bar{x_1} x_2 y_1 \bar{y_2}$ + $\bar{x_1} x_2 \bar{y_1} y_2$

The $\odot$ operation for the LCS analysis can also be represented in the same way as in RCS analysis as shown above.

How does BDD serves us

The above manipulation of boolean functions is done through OBDD. For this purpose the boolean functions used by us are represented as nodes of OBDD and the transformation functions applied to the boolean functions are then treated as manipulation of the subtree, rooted at the node representing those functions, in a OBDD. Although the OBDD representation of the boolean function may have size exponential in the number of variables, many useful functions have more compact representation. Since, in our approach we store all possible cache states (corresponding to different paths) at a program point, therefore our technique might possibly suffer from exponential blow up. The use of OBDDs to represent intermediate functions representing various cache states might prove helpful as OBDD could represent many functions compactly.

We have used the CUDD package (Release 2.3.1)[15] by University of Colorado in our implementation. The CUDD package provides a large set of functions to manipulate Binary Decision Diagrams. The CUDD package is used as a black box i.e. only the exported functions of the package are used. Although our implementation proves the accuracy of encoding our approach using OBDD but the memory usage can not be compared directly, as the CUDD package itself uses a large amount of memory to maintain its various data structures. Table 2.9 compares the maximum number of Reaching and Live cache states obtained via normal implementation with the maximum number of nodes in the BDDs during the execution of fixed point iteration algorithm using CUDD, for 8 and 16 cache blocks. The columns *LCS* and *RCS* represent the maximum number of Live and Reaching cache state obtained in normal implementation. And the columns *LCS nodes* and *RCS nodes* represent the maximum number of nodes used in BDD

Benchmark	Basic Blocks	LCS		RCS		LCS Nodes		RCS Nodes	
		8	16	8	16	8	16	8	16
qsort	40	68	95	69	109	2661	5363	2505	5171
des	117	236	411	257	469	11796	42434	12239	29653
whet	52	175	506	136	367	5763	14676	4889	11081
ssearch	76	202	479	247	467	6366	14995	7079	17101
crc	66	219	356	186	306	6253	14755	5643	14008
math	22	99	702	81	869	1773	8015	1681	9526

Table 2.9: Comparison of memory usage between normal and BDD representation of cache states

during the execution of fixed point iteration algorithm. It can be observed that a large number of nodes in BDD are maintained. The size of BDD depends upon a lot on number of variables and their ordering, hence a proper encoding of memory blocks into boolean variables and their ordering can help in better performance of OBDDs.

2.7 Conclusion

The determination of Cache Related Preemption Delay (CRPD) is important for the schedulability of tasks in real-time system. CRPD is the additional delay incurred by the low-priority task owing to additional cache misses introduced by preemption. We have provided an accurate analysis of CRPD by maintaining the cache states possible at any program point via various paths in a program. Further we have considered both low and high priority tasks in our approach. Our experiments show that our approach results in tighter bound on CRPD than the existing approaches.

One possible concern regarding our analysis technique is a blow-up in space consumption. As observed in the previous section, none of our benchmarks suffered from an exponential space blow-up due to our decision to represent cache states (instead of the content of each cache block separately).

BDD can be used to reduce the space consumption. To use BDD we have to encode our approach into a boolean form (a way to do this is presented in previous sections). A proper encoding and ordering of boolean variables is very important to derive full advantage from BDD. Although we have implemented our approach using BDD but we have not tried optimization using various encodings and orderings, since none of our benchmarks suffered with space blow ups.

Chapter 3

Timing Analysis of Loop Behaviors

3.1 Introduction

A real-time system requires that some timing guarantee should be given for the tasks running in it. A bound on the time, taken by an application can be provided by static analysis of programs in the form of worst case execution time (WCET) of the program. The problem of determining WCET of a program by static analysis methods has to be solved at the following two levels [53]: (1) Programming language level, and (2) Micro-architectural level, to take into account the effect of features such as pipeline, cache and branch prediction [36, 32, 29]. There are three main approaches for calculation of WCET at the programming language level: *path based*, *tree based* and *implicit path enumeration technique (IPET)*. In *path based* approach, the longest path is discovered from the start to end of a program. In a *tree-based approach* the final WCET is generated by a bottom-up traversal of a tree, generally corresponding to a parse tree of the program, using rules defined for each type of compound program statement to determine the execution time of the statement. In *IPET*, program flow and low-level execution time are modelled using arithmetic constraints

The previous chapter presented an analysis of caches to determine the delay caused in the execution of a task due to preemption (by higher priority task). In this chapter we present a program path analysis technique to determine the WCET of loops in a task. In particular, we try to identify certain infeasible paths [1] spanning across loop iterations, which are hard to detect via existing path analysis techniques and

[1]A path (from start to end) in a program is referred as infeasible (or false) if it can never be executed regardless of the input data [34].

then use the infeasible path information to get a tight bound on the WCET of a loop. We first describe the types of infeasible paths along with some techniques on how to detect them. We then present our technique to detect infeasible paths and use the infeasible path information to give a bound on the WCET of a loop in a program.

3.1.1 Types of Infeasible Paths

```
1   for ( i := 0; i < limit; i++)
2   {
3       if ( i < 3 )
4               u := 0;
5       if ( i > 3 )
6               u := 1;
7   }
```

Figure 3.1: Infeasible paths due to branch correlation

The knowledge about infeasible paths in a program can be used to give a tighter bound on WCET. There could be various types of infeasible paths possible in a program. There could be infeasible paths because of the correlation between branches. For example, in Figure 3.1, $\langle \texttt{3,4,5,6} \rangle$ is an infeasible path because $\langle \texttt{3,4} \rangle$ implies that the outcome of branch at line number 3 is true therefore the outcome of branch at line number 5 can not be true, hence $\langle \texttt{5,6} \rangle$ can not be executed. The idea in such types of infeasible paths is to detect the effect of outcome of a branch on the outcome of another branch. Another type of infeasible paths can occur due to the effect of assignment of a variable on a branch. For e.g. Figure 3.2 shows how the assignment of variable v to value 5 makes $\langle \texttt{3,4,6} \rangle$ an infeasible path. Detection of such types of branch correlation and assignment effect based infeasible paths has been studied in [6, 23].

```
1   for ( i := 0; i < limit; i++)
2   {
3       v := 5;
4       if v < 6 then
5           u := 1;
6       else ...
7   }
```

Figure 3.2: Infeasible paths due to effect of assignment on branch

There could exists a different type of branch correlated infeasible paths where the outcome of a

```
1  if u = 0 then
2           v := 1;
3  if w = 0 then
4           x := 1;
5  if (u = 0 and w = 0) then
6           y := 1
7  else     y := 2;
```

Figure 3.3: Correlation of a branch outcome with a conjunction of other branches

```
1   sumeven := 0;
2   for (j:=0; j <= limit; j++)
3   {
4       if (j % 2 == 0) then
5               sumeven = sumeven + j;
6   }
```

Figure 3.4: Infeasible paths across iteration

branch is dependent upon outcome of more than one branch visited earlier in the program flow. For example in Figure 3.3 the outcome of the branch at line 5 depends upon the outcome of branches at lines 1 and 3. Therefore, $\langle$`1,2,3,4,5,7`$\rangle$ is an infeasible path. However, such types of infeasible paths are hard to detect because they can not be obtained by considering the direct effect of one branch on another, rather they require the combined knowledge about the outcomes of several branches. In Figure 3.3 it is easy to determine the infeasible path $\langle$ `1,2,3,5,6` $\rangle$ because it could be easily reasoned out that the falsehood of the branch condition in line 3 forces the condition in line 5 to be false.

Other type of infeasible paths which can be present in a program include ones that span over multiple iterations of a loop. For example consider the code to calculate the sum of even numbers, as shown in Figure 3.4. If the path $\langle$`3,4,5,6`$\rangle$ is taken in some iteration of the loop then it is not possible to take it again in the consecutive iteration. Information about such types of infeasible paths (which span over multiple iterations) can be utilized to give a tighter bound on the timing of the loops. In this chapter we present our technique which uses information about such types of *iteration-spanning* infeasible paths to get a tight bound on the WCET of a loop. From now on we will use the term *iteration-spanning infeasible paths* to refer such types of infeasible paths.

3.2 Infeasible Path Detection Technique

We propose a constraint propagation based infeasible path detection technique to detect the infeasible paths within a loop. With our technique we could not only detect infeasible paths based upon direct correlation between two branches or a branch and an assignment statement, but also those infeasible paths which originate due to the combined effect of more than one conditional branches on some conditional branch as in Figure 3.3 or those which span over multiple iterations of a loop. Our constraint propagation technique is based upon the following two main ideas:

1. Propagating the set of constraints in the backward direction (in CFG) via weakest precondition calculation.

2. Checking for satisfiability of the set of constraints using a constraint solver.

3.2.1 Technique

Consider a bounded loop L with k branches within the loop structure. Assume there are n basic blocks $B_1, ..., B_n$ in the control flow structure of the loop. Assume there are m variables $v_1, ..., v_m$ constituting the set V. The infeasible path detected will be a sequence over the alphabets $\{B_1, ..., B_n\}$. Every visit of the basic block B_i is annotated with a set of constraints C_{i_x} over the variables in V, x is used to differentiate between different visits of the basic block B_i. The constraint propagation algorithm works in a backward breadth-first-search traversal way. At reaching basic block B_i through any path, it tries to solve the constraints available in C_{i_x} using some constraint solver (an external constraint solver can be used). The constraint solver will result in a *FALSE* answer if there exists no set of values for variable in V such that C_{i_x} satisfies, and the path in which it returns *FALSE* is not further pursued for constraint propagation. Else, the algorithm proceeds in the following way: For each predecessor B_{ij} of B_i, a set of constraints $C_{i_x j}$ is calculated via weakest precondition [2] of C_{i_x} w.r.t. the statements in B_i. There is another condition for termination of algorithm: A path is not pursued further in the algorithm if the sequence of basic blocks in it implies a pre-defined maximum unrolling of loop say M. Hence, the two conditions for the termination of algorithm are: A path is stopped from being pursued further if the set

[2]The condition that characterizes the set of all initial states such that activation will certainly result in a properly terminating happening leaving the system in a final state satisfying a given post-condition is called “ the weakest pre-condition corresponding to that post-condition” [16]

of constraints in it at any instant can not be satisfied by any set of values for V or the sequence of basic blocks in the path implies the desired maximum unrolling of loop.

3.2.2 Example

Consider the CFG shown in Figure 3.5(A). The working of infeasible path detection algorithm on the example is shown in Figure 3.5(B). The infeasible path detection algorithm starts from the false edge of basic block 1, propagating the constraints backward via weakest precondition calculation. The constraints at the exit of each block is shown next to that block in Figure 3.5(B). The path $\langle 1, 3, 4, 0, 1, 3 \rangle$ is detected as infeasible path, as the set of constraints at the end of path $\langle 1, 3, 4, 0, 1, 3 \rangle$ is unsatisfiable for any set of values for variables.

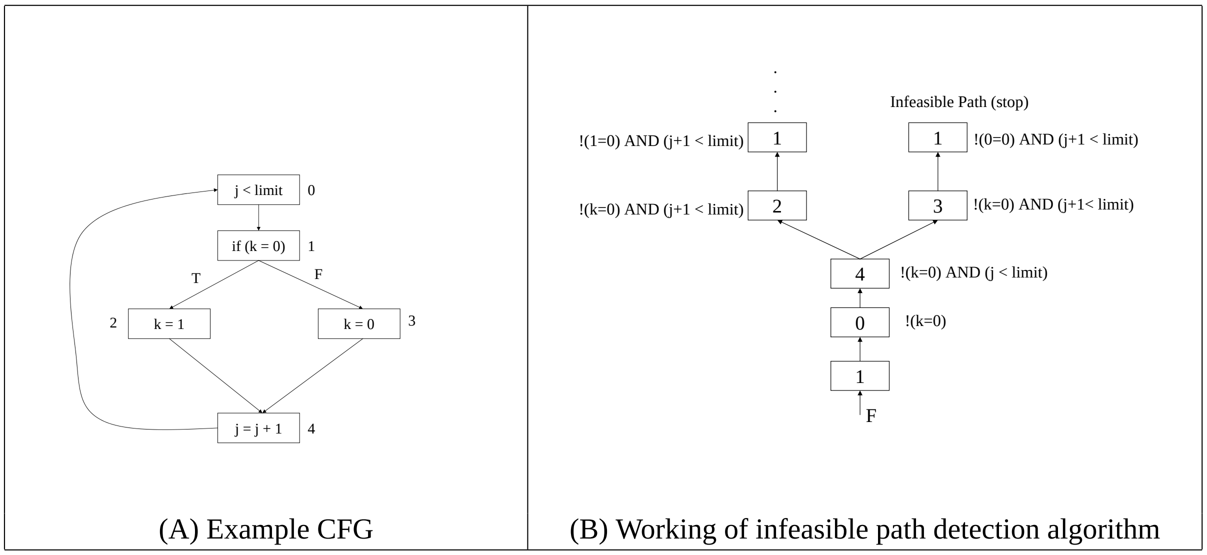


Figure 3.5: Example CFG and working of infeasible path detection algorithm

3.2.3 Implementation And Results

We have implemented our technique using C++. We have used *Simplify* [14] by Compaq to check the satisfiability of constraints. Let at any instant, the set of constraints stored at a node are: $c_1, ..., c_n$. Therefore to check the satisfiability, we are required to check the following predicate formula: $\exists(v_1, ...v_m)(c_1 \wedge c_2 \wedge ...c_n)$. Instead we check the following predicate formula :$\forall(v_1, ...v_m)\neg(c_1 \wedge c_2 \wedge ...c_n)$. If the return value for this is $TRUE$ then it states that there does not exists any set of variables such that the formula

$(c_1 \wedge c_2 \wedge ...c_n)$ is true. Hence, an infeasible path is detected. For weakest precondition calculation we use the methods described in [16].

We have used three benchmarks (corresponding to three different types of infeasible paths which could exist in a program) to check the validity of our approach. We were successfully able to detect the infeasible paths for the example shown in Figure 3.5(A). Here, if a branch outcome is true in one iteration it can not be true in the successive iteration. We were also able to detect all the infeasible paths for the code shown in Figure 3.3. The infeasible paths due to the correlation between one branch and the combined effect of other two branches were detected too, which can not be detected by other existing branch correlation based infeasible path detection approaches. Further we used the code shown in Figure 4.2(D) (from Chapter 4) and we were able to detect all the infeasible paths spanning over a pre-defined number of iteration. We have later used the code shown in Figure 4.2(D) (from Chapter 4), along with the infeasible path information derived from our infeasible path detection technique, as a benchmark for our WCET analysis technique.

3.3 WCET Calculation Technique

In this section we present our technique to calculate the WCET of a loop. We use the information about iteration-spanning infeasible paths in our technique. In this section we first describe the simplified version of our technique along with the motivation behind it. We then present the general form of our technique.

3.3.1 Basic Technique

In the simplified form of our technique let us assume that the bound on start and end iterations for every feasible path through the loop is $[1, I]$, where I is the loop bound. Also assume a set p of paths (with their WCET) between the start and end of each iteration, and a set of infeasible sequences (each of length $k+1$) of paths, where each element of the sequence is drawn from p. For clarity, the elements (paths) of set p will be referred as $ipath$ (iteration path) and a sequence of $ipath$ will be referred as $wpath$ (whole program path).

We now present a WCET analysis technique which only considers the infeasible patterns and assumes that I is a multiple of k (i.e. $I = k * c$, where c is some integer greater than zero). Later, we show how to modify this technique to use the bounds on iteration numbers for each path and handle cases where I is not a multiple of k.

A naive way to find a tight bound on WCET will be to use an exhaustive search with complete loop unrolling. Basically, the search will enumerate all legal $ipath$ sequences of length *I*. The WCET value of the loop is equal to the maximum WCET value among the sequences. If the information about the infeasible paths is exact this method will generate an exact WCET value. In our analysis technique we also compute the exact WCET value, but in a much more efficient manner than the exhaustive search.

The intuition behind our approach is as follows. Suppose after *i* iterations we have seen the following sequence of paths (one path each iteration), p_1, p_2,...,p_{i-k}, ...,p_i. Therefore, the possible path that could be taken in the $i+1^{th}$ iteration would be one of those, which does not form an infeasible sequence of path with the previously seen sequence of paths, i.e. p_{i-k+1},...,p_{i+1} should not be an infeasible sequence. And similarly for the path taken in $(i+2)^{th}$ iteration and so on. This way we can determine the set of possible sequence of paths for the next k iterations. Therefore, if we divide the total number of iterations into blocks of *k* iteration each, we can decide after each block the set of next possibly taken block.

In our basic technique the whole procedure to determine the WCET for the loop can be divided into four steps:

1. Generate the set of nodes (blocks) with each node representing a sequence of paths of length *k* and having a weight equal to the sum of the WCET of the paths in the sequence.

2. Create transitions (directed edges) from each node to other nodes, representing the possible sequence that could be taken next after the present node's sequence.

3. Optimization of the *transition graph* say 'G', generated from nodes and edges in step 1 and 2, in order to reduce the number of edges and nodes in the graph.

4. Dynamic programming algorithm to calculate the longest path of I/k nodes in the graph G.

We now elaborate each of the four steps.

1) The set of nodes is constructed by generating all possible sequences of length k, where each position of the sequence can be one of the p *ipaths*. Hence there could be p^k such sequences and therefore same number of nodes in the graph: each node representing one sequence.

2) The edges of the graph are constructed by considering various pairs of nodes say N1 (sequence $p_1p_2...p_k$) and N2 (sequence $q_1q_2...q_k$) and checking the following:
for i going from 1 to k check if $p_ip_{i+1}...p_{i+(k-i)}q_1...q_i$ is a feasible sequence. If any value of i gives an infeasible sequence, we do not create an edge between N1 and N2 otherwise if none of the sequences is infeasible then we create a directed edge from N1 to N2.

3) The *transition graph* generated from the above two steps can be reduced in terms of number of edges and nodes by the following optimization scheme:

- Put all the nodes that have the same set of outgoing edges into one group say S. Now if there is a node which has an outgoing edge to more than one member of a group just keep the edge which is to the maximum weighted node and delete the edges to the nodes with lower weights. The intuition behind this is: Let a node n have transitions to elements of a group S, whose all elements have the same set of outgoing edges. Since all elements of set S have the same set of outgoing edges, therefore after a node in S is taken, the sequences of paths with maximum weight (and same number of nodes in it) that could be seen from any node in S will have the same weight. Hence, only keeping the transition with maximum weight from n to a node in S, will still result in sequences of maximum weight.

- If there is a node X with no incoming edges (Figure 3.6) then for every outgoing node say Y attached to X (X$\rightarrow$Y), check if there is any node Z such that (Z$\rightarrow$Y), and w(Z) $\geq$ w(X), then delete the edge (X$\rightarrow$Y).

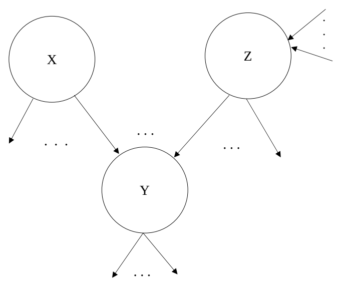


Figure 3.6: X has no incoming edge

Let at any instant the maximum weight sequence S of nodes has the starting node as Y. Therefore adding one or more nodes at the beginning of S such that the new sequence will be of maximum weight is not possible by adding X before Y in S, because if only one node has to be added then it should be Z or some node with weight greater than Z and if more than one nodes are added before Y then again X can not be added because once X is added before Y in S then no further node can be added before X as X has no incoming edge. Hence, (X→Y) will never be used in determining maximum weight sequence of nodes.

- If there is a node X with no outgoing edge (Figure 3.7) then for every incoming node say Y attached to X (Y→X), check if there is any node Z such that (Y→Z), and w(Z) $\geq$ w(X), then delete the edge (Y→X).

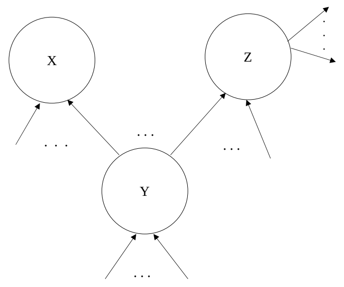


Figure 3.7: X has no outgoing edge

Let at any instant the maximum weight sequence S has the last node as Y. Therefore adding one or more nodes after Y in S such that the new sequence will be of maximum weight is not possible by adding X after Y, because if only one node has to be added then it should be Z or some node with weight greater than Z and if more than one nodes are required to be added after Y then again X can not be added because once X is added after Y in S then no further node can be added after X as X has no outgoing edge. Hence, (Y→X) will never be used in determining maximum weight sequence of nodes.

- If there is a node with no incoming edge and no outgoing edge then delete the node.

4) The Dynamic Programming algorithm iterates in steps of I/k. The equation for the dynamic programming algorithm for calculating the longest path in the graph with I/k nodes is as follows:

$$f(i,x) = max_{\{\forall y: y \rightarrow x\}}(f(i-1,y) + w(x)$$

where *i* is a step. *w(x)* denotes the weight of node *x*.
For every node *x*, for a particular step *i*, the algorithm tries to find out the longest (in terms of weights) path of *i* nodes ($i \times k$ iterations) ending at the node *x* by taking the maximum of the longest path for *i-1* steps of the incoming-edge neighbors of x and by adding the weight of node x to it. Note that we have assumed that I is a multiple of k. Later in the refined approach we show how to handle cases when I is not a multiple of k.

The basic technique assumes that all feasible paths in a loop have the same start and end iterations (equal to that of loop) between which they can be possibly taken. However, in general it is not so. For, e.g. in Figure 3.1, the path $\langle$`2,3,4,5,7`$\rangle$ can not be taken after i is greater than 3. Such type of information is useful to give a tight bound on WCET of loops. However, there are some important modifications that has to be done to the basic technique to incorporate the individual start and end of each path. The following section present the refined technique to take into account the start and end information for each path in determining the WCET of the loop.

3.3.2 Refined Approach

In the refined approach we handle the *iteration based constraints* (i.e. constraints on the index of a loop, which determines whether a path can be executed or not in a particular iteration number) in order to get a tight bound on WCET of loop. The introduction of iteration based constraints require the following modifications in the basic technique.

- A start and end for each node in the transition graph should be provided
- An additional check on the validity (possible sequence of paths representing the node) of node based upon the start and end information of the paths.
- Modification in the algorithm for creating edges between different nodes.
- Modification in the Dynamic Programming algorithm for computing the WCET of the loop.

Let us now explain the approach in detail.

3.3.3 Problem Statement

Input: Given the following set of inputs:

- A loop of I iterations.
- Set p of all possible paths $p_1, p_2, ...$ between the start and end of each iteration.
- WCET of each path in p.
- Start and end iteration for each path in p. It is the range of iterations during which each path can be taken.
- A set of infeasible sequences of paths with the longest sequence of length $k + 1$.

Output: The WCET for the loop.

3.3.4 Method

We use a transition graph along with Dynamic Programming (DP) algorithm to determine the WCET of a loop. Each node *x* in the graph comprises of the following four elements:

1. A sequence P_x of paths given by p_1, p_2,...,p_k, and representing the sequence of paths taken in *k* consecutive iterations.

2. A weight w_x, given by $w_x = \sum_{i=1}^{k} WCET(p_i)$.

3. A start P_x^s of node, stating the iteration at which the node is activated.

4. An end P_x^e of node, stating the iteration at which the node is terminated.

Our technique to determine the WCET of a loop can be divided into the following three steps. It is important to note that the optimization steps as present in basic technique is not present in the refined approach, because due to the introduction of start and end for each node, it is not possible to classify nodes with same outgoing edges into one group, unless they all have the same start and end also. In general, it will be very rare that everything is same for nodes, hence it is not useful to apply the optimization approach as described in the previous section.

1. Generate the set of nodes for the transition graph ‘G’.

2. Create transitions (directed edges) from each node *x* in ‘G’ to itself and to other nodes in ‘G’, representing the sequence of paths possible after the sequence in *x* is taken.

3. Dynamic Programming (DP) algorithm to calculate the WCET of the loop.

3.3.5 Creating Nodes

The first step in creating nodes for the transition graph ‘G’ is to generate all possible sequences P_x of paths of length *k*, where $k + 1$ is the length of the longest infeasible sequence of paths. Each P_x corresponds to a node *x*. If P_x contains an infeasible sequence of paths then discard the node *x*. The weight w_x of node *x* is calculated as given before. The next step is to calculate the start and end of each node and also discard those nodes which are not valid. A node is *invalid* if it represents an infeasible sequence of paths in it. For example in Figure 3.8, *ba* is an infeasible sequence of paths, because once path *b* is taken path *a* can never be taken, hence a node containing a sequence *ba* is invalid and should be discarded.

```
if (i < 5)
        path a;
else
        path b;
```

Figure 3.8: *ba* is an infeasible sequence of paths

The start and end of each node in the transition graph can be determined in the following way: Consider a node x with sequence P_x of paths $p_1 p_2 ... p_k$. Let each path p_i has a start (p_i^s) and end (p_i^e) iteration as determined by the path analyzer.

Calculating the start: The start iteration P_x^s for node x is given by.

$$P_x^s = \max_i (p_i^s - (i-1)) \tag{3.1}$$

Calculating the end: Similarly the end iteration P_x^e for node x is given by.

$$P_x^e = \min_i (p_i^e + (k-i)) \tag{3.2}$$

Checking the validity of node

Once the start P_x^s and end P_x^e for each node has been calculated, it is important to check the validity of the node. A node x is invalid if the start and end for some path p_i (in the sequence P_x), calculated on the basis of start and end for node x, does not satisfy the actual start (p_i^s) and end (p_i^e) of p_i. In other words, a node x is invalid if for any of its constituent paths p_i the following constraints are violated.

$$p_i^s \leq P_x^s + (i-1) \leq p_i^e$$

$$p_i^s \leq P_x^e - (k-i) \leq p_i^e$$

These invalid nodes are removed from the transition graph and the corresponding path sequence is added to the set of infeasible paths. **Note:** It is possible that there are some nodes which are valid but never reached in the Dynamic Programming algorithm. An example of it is presented in the later section.

3.3.6 Creating Transitions

Once all the possible nodes are created, every pair of nodes in 'G' is checked for creating a transition (edge) between them. There is a transition (directed edge) from a node x_1 to node x_2 if the following conditions are satisfied.

1. The periods (start to end) of node x_1 and x_2 overlap and there is no infeasible path sequence generated from the concatenation of path sequence of x_1 with x_2 (i.e. P_{x_1} with P_{x_2}).

2. The periods of nodes do not overlap but the start of x_2 is one greater than the end of x_1 (i.e. $P^s_{x_2}$ = $P^e_{x_1}$+1), and there is no infeasible path sequence generated from the concatenation of path sequence of x_1 with x_2.

3.3.7 Dynamic Programming Algorithm

The Dynamic Programming algorithm for calculating the WCET of a loop takes the transition graph 'G' as an input and at a specified iteration *i* (where i is a multiple of k), it calculates for every node *x*, a valid path of length *i+k* (which is same as (i+k)/k nodes) and maximum weight and with *x* as the last node in the path. The Dynamic Programming equation for calculating the above such paths can be written as:

$$\begin{aligned} f(i,x) &= max_{\{\forall y : y \rightarrow x, y \in valid(x,i-1)\}} f(i-1,y) + w(x) \\ & \quad \text{if } valid(x,i-1) \neq \emptyset \end{aligned} \tag{3.3}$$

$$f(i,x) = invalid \text{ if } valid(x,i-1) = \emptyset \tag{3.4}$$

where *valid(x,a)* represents the set of incoming nodes of x which are valid at iteration *a*. The working of the Dynamic Programming algorithm is shown in Algorithm 1. In Algorithm 1, *start(x)* and *end(x)* are the start and end of node *x* respectively. *k* is the length of sequence of paths in each node. Assume the total iterations of the loop to be N. The input to the algorithm are transition graph and the *last_iteration*, where *last_iteration* is given by:

$$last\_iteration = N - 1 - (N\%k) \tag{3.5}$$

where $N\%k$ gives the remainder when N is divided by k. Note that the first iteration is taken as 0. Hence for a total of N iteration the loop will iterate from 0 to $N-1$.

For the remaining iterations (i.e. $N - last\_iteration - 1$) add the weight of path p_i with maximum weight to the WCET (calculated as in Algorithm 1) for calculating the final WCET. Hence the final WCET is given by:

$$FINAL\_WCET = WCET + (N - last\_iteration - 1) \times \max_i(p_i)$$

A more accurate bound on WCET can be determined by creating nodes corresponding to sequences

```
DP_ALGORITHM(transition graph, last_iteration)
$iteration = 0$ ;
$WCET = 0$ ;
for every node x do
    if $start(x) \leq 0$ and $end(x) \geq k - 1$ then
        $f(0, x) = w(x)$ ;
    else
        $f(0, x) = invalid$ ;
    end
end
for iteration ← 1 to k-1 do
    for every node x do
        $f(iteration, x) = f(0, x)$ ;
    end
end
for every node x do
    if $WCET < f(iteration, x)$ then
        $WCET = f(iteration, x)$ ;
end
$iteration = iteration + 1$ ;
while iteration ≤ last_iteration do
    for every node x do
        if $start(x) \leq iteration$ and $end(x) \geq iteration + k - 1$ then
            Calculate $f(iteration, x)$ as given in equations 3.3 and 3.4 ;
        else
            $f(iteration, x) = invalid$
        end
    end
    $current = iteration$;
    for iteration ← current+1 to current+k-1 do
        for every node x do
            $f(iteration, x) = f(iteration - 1, x)$ ;
        end
    end
    for every node x do
        if $WCET < f(iteration, x)$ then
            $WCET = f(iteration, x)$ ;
    end
    $iteration = iteration + 1$ ;
end
```

Algorithm 1: Dynamic Programming algorithm for calculating the WCET of loop

(of paths) of length $N\%k$ and assigning the start for such nodes to be $last\_iteration + 1$. However, in practice since the value of k is small (for e.g. in our benchmarks k is maximum 2), it will not be very useful to create extra set of nodes and increase the complexity of approach for a small difference in WCET bound. It should be noted that the refined Dynamic Programming algorithm iterates over the original number of iterations. However, the values for the iterative function is calculated at every k^{th} step only and is kept the same for the following $k - 1$ iterations. The algorithm can easily be optimized to iterate only on steps of k iteration. However, for better understanding and clarity it is presented as given above (iterating on original number of iterations).

3.4 Implementation And Illustration By Examples

We have implemented our technique using C++. We have used simplescalar [10] architectural simulation platform for compiling the benchmarks to simplescalar assembly language with modified *gcc*. A prototype analyzer written by us accepts the assembly language code, disassembles it, identifies the basic blocks and constructs the control flow graph (CFG). It then separates out the CFGs for the loops, and for every loop generates the various paths possible from start to end of loop in each iteration, along with the weight of each path in terms of number of instructions executed in each path. The user then provides the following inputs to the timing analyzer: (a) Total no. *N* of paths, with start and end iteration for each path (this could be obtained using a path analyzer written by us) (b) The infeasible sequences of paths, with the length of longest infeasible sequence of paths say as *k+1*. The infeasible paths could be derived using the infeasible path detection tool written by us (c) Total number of iteration for the loop. The timing analyzer tool generates N^k sequences of paths (with each sequence of length *k*), corresponding to same number of nodes. The weight of the node is the sum of weights of each path in the sequence belonging to it. Out of these nodes those are removed which contains an infeasible sequence of paths. A start and end for each node is calculated. The validity of the nodes (on the basis of start and end calculated for them) is checked next (as given in section 3.3.5) and all the nodes which are invalid are discarded and their sequences are added to the set of infeasible paths. The transitions (directed edges) for the graph are derived and the function implementing the Dynamic Programming algorithm is called with input as transition graph.

In this section some of the examples are presented to illustrate the working of our technique under

```
for(i=0; i<10; i++)
{
    if (i == 2)
    {
        S1;
        continue;
    }
    if (i < 5)
        S2;
    if ( (i >= 5) && (i <= 8) )
        S3;
    S4;
}
```

Figure 3.9: Example 1: Illustrating iteration based constraints

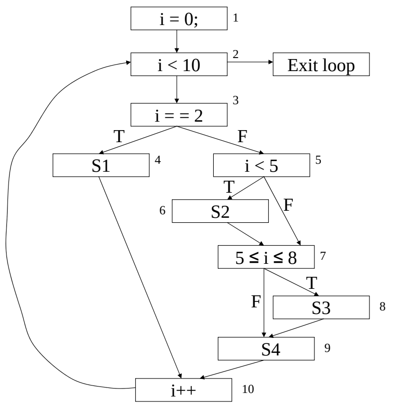


Figure 3.10: Control Flow Graph for Example 1

different types of scenarios.

3.4.1 Example 1: Iteration Based Constraints

Consider the piece of code shown in Figure 3.9. The constraints on different paths in the code are derived by comparing the index of the loop with constants, i.e. not all paths can be possibly taken on a particular iteration of the loop. This type of constraints will be referred as *iteration based* constraints from now on, borrowing the terminology from [23]. The control flow graph for example in Figure 3.9 is shown in Figure 3.10. There are 4 possible paths in each iteration of loop as given in Figure 3.11.

Note that the sequence: 2 3 5 6 7 8 9 10 is an infeasible sequence of paths, because if the result of branch at 5 is true then the result of branch at 7 can never be true. Hence, this path is not included for WCET

Path Name	Sequence of blocks executed	(start,end) iterations
a	2 3 4 10	(2,2)
b	2 3 5 6 7 9 10	(0,1) & (3,4)
c	2 3 5 7 9 10	(9,9)
d	2 3 5 7 8 9 10	(5,8)

Figure 3.11: Paths and their corresponding sequence of blocks executed

analysis. It is assumed that such type of paths have been identified and eliminated by the path analyzer. The transition graph for the example is shown in Figure 3.12. Let us assume the WCET of paths a, b, c,

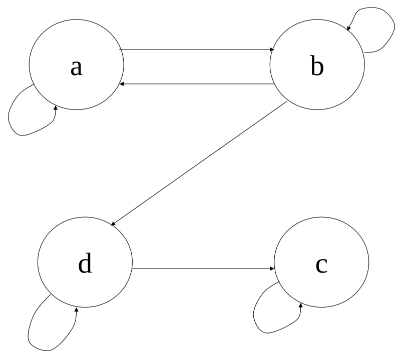


Figure 3.12: Transition graph for Example 1

d are 1, 2, 3, 4 respectively. The working of Dynamic Programming algorithm for the example in Figure 3.9 is shown in Figure 3.13. The values inside parentheses shows the start and end for each node and the value inside [] represents the weight (sum of WCET of paths) of each node. ‘x’ in Figure 3.13 represents an invalid node. ‘-’ means “same as in previous iteration”. It is used to show that the values are not computed in an iteration but just copied from the previous iteration.

3.4.2 Example 2: Effect Based Constraints

If a path *p* is taken in some iteration of the loop, it might make some paths infeasible in the following iteration. Such type of constraints are referred as effect based constraints and leads to iteration-spanning infeasible paths. Consider the code shown in Figure 3.14. *x* in the code, takes value in the form of a harmonic motion around the value 0. The values of *x* seen at line 4 of the code are (0 1 2 1 0 -1)*.

iteration	a	b	c	d
	(2,2)[1]	(0,1)(3,4)[2]	(9,9)[3]	(5,8)[4]
0	x	2	x	x
1	x	4	x	x
2	5	x	x	x
3	x	7	x	x
4	x	9	x	x
5	x	x	x	13
6	x	x	x	17
7	x	x	x	21
8	x	x	x	25
9	x	x	28	x

Figure 3.13: Working of DP algorithm for Example 1

'*' represents that the same sequence of values is repeated. If the value of *x* seen at line 4 is 0, then line number 5 is executed, otherwise line number 8 and 9 are executed. The two possible paths in every iteration of the loop can be identified as shown in Figure 3.15

```
1  x := 0; t := 1;
2  For(i := 0; i < 9; i++)
3  {
4      if ( x == 0)
5          u = 0;
6      else
7      {
8          u = 1;
9          update(x,t,temp);
10     }
11     x = x + t;
12 }

update(x,t)
{
    switch (x)
    {
       case 2 : t = -1;
                break;
       case -1: t = 1;
                break;
       default: t = t;
    }
}
```

Figure 3.14: Example 2: Illustrating effect based constraints

It should be noted that the function "update" itself has several possible paths from start to end, which can combine with the paths in the loop to result in many more paths than shown in Figure 3.15. But

Path Name	Sequence of line numbers executed	(start,end) iterations
a	3 4 5 11 12	(0,9)
b	3 4 6 7 8 9 10 11 12	(0,9)

Figure 3.15: Possible paths for Example 2

since every call to "update" will take a constant amount of time, irrespective of the path taken within it, therefore the function "update" can be treated as a block rather than breaking it down, further into paths. It is important to note here that the above shown code for "update" is not an optimal implementation for "update", rather it is written in this way to reduce the number of possible paths in each iteration of loop. A detailed description of such code transformation to simplify infeasible path detection is given in the next chapter. Based upon the possible values of *x*, let us assume that the following infeasible path sequences were identified: *bbbb*, *aa*. Since the longest infeasible path is of length 4 each node in the transition graph will consist of sequences of length 3. Also each node will have the same start and end iteration as (0,9). The transition graph for the example is shown in Figure 3.16. Assuming the WCET of

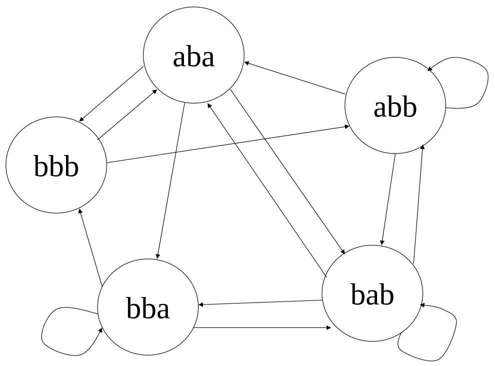


Figure 3.16: Transition graph for Example 2

path *a* as 1 and of path *b* as 3, the working of the Dynamic Programming algorithm is shown in Figure 3.17. 'x' in Figure 3.17 represents an invalid node. '-' means "same as in previous iteration". It is used to show that the values are not computed in an iteration but just copied from the previous iteration.

iteration	aba (0,9)[5]	abb (0,9)[7]	bab (0,9)[7]	bba (0,9)[7]	bbb (0,9)[9]
0	5	7	7	7	9
1	-	-	-	-	-
2	-	-	-	-	-
3	14	16	14	14	16
4	-	-	-	-	-
5	-	-	-	-	-
6	21	23	23	21	23
7	-	-	-	-	-
8	-	-	-	-	-

Figure 3.17: Working of DP algorithm for Example 2

3.4.3 Example 3: Combination of Effect Based And Iteration Based Constraints

In this section a more general example which combines both iteration and effect based constraints is presented. Consider the piece of code shown in Figure 3.18. To reduce the length of infeasible path sequence, the "update" function has been modified from its previous form of example 2. Here, the values of *x* seen at line 8 will be (0 1 2 0 -1)*. Again, since the implementation of update function is such that it takes constant amount of time on every call of it, therefore it can be considered as a block. The possible paths per iteration of loop in example 3 are shown in Figure 3.19. Based upon the values, which *x* can take in the example code, the user can specify the following infeasible paths: *ccc, bb*. Since the longest infeasible path is of length three, therefore each node in the transition graph would consist of path sequences of length two. The possible nodes and their corresponding start and ends are shown in Figure 3.20. Note that node *bb* is discarded as it contains an infeasible sequence of paths. Checking the validity of nodes results in *ba, ca* as invalid nodes. Both *ba* and *ca* will get added to the set of infeasible paths. The resulting transition graph is shown in Figure 3.21. Now assuming the WCET of path *a* as 1, of path *b* as 2 and of path *c* as 4, the working of Dynamic Programming algorithm is shown in Figure 3.22. **Note:** *ab, ac* never become a valid node, that means they can never be taken.

3.4.4 Results

In our experiments we have used the benchmarks shown in Table 3.1. The first two benchmarks are taken from [23], `check_data` is taken from [41], `fresnel, sprsin, expint` and `gaujac` bench-

```
1  x := 0; t := 1;
2  For(i := 0; i < 10; i++)
3  {
4      if (i < 4)
5          S1;
6      else
7      {
8          if ( x == 0)
9              u = 0;
10         else
11         {
12             u = 1;
13             update(x,t);
14         }
15         x = x + t;
16     }
17 }

update(x,t)
{
    switch (x)
    {
        case 2 : t = -1;
                 x = 1;
                 break;
        case -1: t = 1;
                 x = x;
                 break;
        default: t = t;
                 x = x;
    }
}
```

Figure 3.18: Example 3: Combining effect and iteration based constraints

Path Name	Sequence of line numbers executed	(start,end) iterations
a	3 4 5 17	(0,3)
b	3 4 6 7 8 9 15 16 17	(4,9)
c	3 4 6 7 8 10 11 12 13 14 15 16 17	(4,9)

Figure 3.19: Possible paths for Example 3

```
aa - (0,3)                ab - (3,4)
ac - (3,4)                ba - (4,3) - invalid
cb - (4,9)                ca - (4,3) - invalid
bc - (4,9)                cc - (4,9)
```

Figure 3.20: Nodes and their corresponding start and end for Example 3

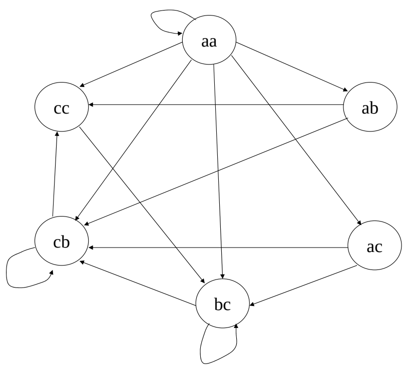


Figure 3.21: Transition graph for Example 3

it	aa (0,3)[2]	ab (3,4)[3]	ac (3,4)[5]	bc (4,9)[6]	cb (4,9)[6]	cc (4,9)[8]
0	2	x	x	x	x	x
1	-	-	-	-	-	-
2	4	x	x	x	x	x
3	-	-	-	-	-	-
4	x	x	x	10	10	12
5	-	-	-	-	-	-
6	x	x	x	18	16	18
7	-	-	-	-	-	-
8	x	x	x	24	24	24
9	-	-	-	-	-	-

Figure 3.22: Working of DP algorithm for Example 3

marks are from *Numerical Recipes in C* [42], `SHM` is same as shown in Figure 4.2(D) (from Chapter 4). `Sumoddeven` sums the odd and even indexed elements of an array. There is an additional exit condition in the loop which is input data dependent, due to which the end iteration of a path may not be equal to total number of iterations. `Summidall` sums the total and middle elements of an array. The loop has different paths with different start and end. `Wordcount` counts the number of words in a file with any two words separated by a single space. Since two words are separated by a space, the path which detect a end of word and increases the count of words can not be repeated in successive iterations. `Check_data` checks the input data for value less than zero. It has a loop which depends upon input data, hence the end

Benchmark	Description
Sumoddeven	Sums the odd and even elements of a 100 integer vector.
Summidall	Sums the middle half and all elements of a 100 integer vector.
Wordcount	Counts the number of words in a string vector of 256 characters.
Check_data	Checks if the input vector of 100 integers has an entry less than 0.
Fresnel	Computes noncomplex Fresnel integrals.
Sprsin	Converts a 10×10 integer matrix into row-indexed sparse storage mode.
Expint	Computes an exponential integral.
Gaujac	Computes the abscissas and weights of a 10 point Gauss-Jacobi quadrature formula.
SHM	The sequence of values for a variable, at a particular line number repeats constantly.

Table 3.1: Description of benchmarks used

iterations for the paths in the loop are lesser than the maximum iterations for the loop. The `Fresnel` program has a loop which takes different paths on odd and even steps. The loop in the `Sprsin` program does not take the longer path when the loop index is equal to a variable whose value is constant inside the loop. `Expint` has a loop in which the longer path is executed only when the index of the loop is equal to some variable whose value is constant within the loop. The loop in the `Gaujac` program executes different paths on different iterations. `SHM` has *iteration-spanning* infeasible paths.

The WCET calculation results for the benchmarks are presented in Table 3.2. The column *Iterations* shows the total number of iterations for the loop. The *Default WCET* refers to WCET (in terms of number of instructions executed) calculated on the basis of longest path and without considering infeasible paths or start/end for paths. The column *WCET (Path Analysis)* shows the values (in terms of number of instructions executed) computed via our approach considering the effects of infeasible paths and start/end iterations for a path. The column *WCET (separating effects)* shows the values (in terms of number of instructions executed) for WCET by separating out the effects of considering start/end for paths and infeasible paths. The *Start/End* column shows the WCET values when only the start/end information for paths was considered and the column *Inf. Path* shows the WCET values when the infeasible paths information is used and it is assumed that all the path's start/end is same as the start/end of the loop. The experimental results shows that our approach gives a tighter bound on the WCET, as it uses the information about the infeasible paths and the start and end of paths.

It is apparent that a exhaustive search over all possible path sequences will take a large amount of time to execute. For example the exhaustive search will require to generate all permutation of path sequences. And generating all path sequences of length 25, where each element of the sequence can have

Benchmark	Iterations	Default WCET	WCET (path Analysis)	WCET (separating effects)	
				Start/End	Inf. Path
		no. of inst.	no. of inst.	no. of inst.	no. of inst.
sumoddeven	100	3400	1734	1734	3400
summidall	1000	36000	30500	30500	36000
wordcount	256	9472	8064	9472	8064
check_data	100	1900	916	916	1900
fresnel	100	5200	5000	5200	5000
sprsin	10	520	476	476	520
expint	100	185200	6109	6109	185200
gaujac	10	45090	44805	44805	45090
SHM	100	2200	2002	2200	2002

Table 3.2: Results showing WCET prediction

two possible values, requires around 400 seconds in a Pentium 4, 2.4 GHz machine. However, WCET calculation by our technique for all the benchmarks took less than 0.01 seconds in a Pentium 4, 2.4 GHz machine.

From our experimental results, we identify that most of the benchmarks have a single type of constraints associated with the paths. Separating out the effects of considering start/end for paths and infeasible paths for the benchmarks shows that all the benchmarks have constraints on the paths which are derived either only by comparing the index of loop with constants or due to the effect which execution of some path, in an iteration, makes on the execution of other paths in the successive iterations. For e.g. the benchmarks, `sumoddeven`, `summidall`, `check_data`, `sprsin`, `expint` and `gaujac` have constraints on the paths based upon just comparing the index of loop with constants and the benchmarks, `wordcount`, `fresnel` and `SHM` have constraints on the paths based upon just on the effect which execution of some path, in an iteration, makes on the execution of other paths in the successive iterations.

3.5 Related Work

Determining WCET of a program by static analysis methods is a well studied problem. Li et. al. in [31] have given a method to determine the WCET of a program by implicit path enumeration using Integer Linear Programming (ILP).

An important strategy to reduce the bound on WCET is: identifying the infeasible paths in a program and then eliminating them from consideration while calculating the WCET of the program. Detection of infeasible paths and their removal is central to various type of static analyzers. Bodik et. al. in [7] have given a method to detect infeasible paths due to correlation exhibited by a conditional branch. In their analysis the authors consider correlated paths spanning procedural boundaries, as well as correlation that occur within the same procedure. The correlation is detected by performing a query propagation search in the backward direction from a conditional branch, to find assertions on program variables that indicate the correlation along paths leading to the condition. The authors have used the infeasible paths information from branch correlation for compiler optimization, by separating out paths with correlation via replication of code. Further in [6], Bodik et. al. have refined their approach to identify the shortest infeasible paths and to label the control flow graph with these paths. In [6] the authors have used the infeasible path information to show how the precision of def-use pair analysis can be improved by it. Mueller and Whalley in [37] have given a method to determine when a conditional branch can be avoided in a loop and used it for compiler optimization. In their method the authors first calculate the set of registers and variables on which a conditional branch depends and then determine if there exists a path through a loop from the point immediately after a conditional branch is encountered to the same branch without the comparison associated with the branch being effected. Once the conditional branches that has to be avoided are determined the control flow is restructured through replication to avoid these branches.

The infeasible path information can be used to determine a tighter bound on WCET of tasks. Stappert et. al. in [47] have used the infeasible path information to refine their WCET calculation method which takes into account of low-level machine aspects like pipelining and caches, and high-level program flow like loops and infeasible paths. They have used the flow fact language ([18]) to determine the infeasible paths in a timing graph based upon constraints specified through facts. The authors do a repetitive longest path search by removing the infeasible path detected each time, until a feasible longest executable path is found. Lundqvist and Stenstrom in [34] have given a instruction-level simulation approach for the detection and elimination of infeasible paths. The authors simulate all paths through the program and in this process exclude the paths that are not possible regardless of input data. To do this, they have extended traditional instruction-level simulation techniques with the capability to handle unknown data, using an element denoted *unknown*. All conditions that depend on data values that are known statically will be computed during the simulation. Hence the infeasible paths that a conditional branch could create are automatically eliminated since the branch condition is known while simulating. The elimination of infeasible paths helps in giving a tighter bound on WCET. Ermedahl and Gustafsson in [19] have given a static analysis method to automatically derive safe and tight annotations from the program semantics. The derived annotations can then be used for finding the infeasible paths. Altenbernd in [2] have given a heuristic based approach to determine false paths in a program. The author uses a branch and bound algorithm to perform the actual path search in the control flow graph. The author uses symbolic execution which is a simulated execution with partially instantiated variable values. Symbolic execution evaluates program statements in concurrence to the path search algorithm. Values are assigned on reaching certain branches and false paths are deducted during path search, based upon the knowledge of stored values of variables. Park in [41] have given a framework for timing analysis using timing schema approach along with regular expressions. In his approach the author has assumed that the infeasible paths information is being provided by the user. The author has proposed an information description language (IDL) that can be used by user to provide information about the program.

A very similar work on detection and use of infeasible paths for the timing analysis is conducted by Healey and Whalley in [23]. The authors have given a method to automatically detect infeasible paths based upon branch correlation and use that information in determining the WCET of loops. They have used an effect based technique to determine the infeasible paths in a program and used this information for calculating the WCET of a loop. They first determine how a conditional branch can be effected by an assignment to a variable and/or the outcome of another conditional branch. The effects on the conditional branches by the assignment of a variable are then exploited while traversing the basic blocks in every path of the program to determine whether the path is feasible or not.

There is a similar problem to determine the longest executable path with known false paths in the field of hardware development, where it is of interest to find the longest executable path in a network of logic gates. David et. al. in [5] have given an efficient method for removing user specified false paths from the timing graph of a circuit. They have used a node splitting based method by determining the minimum number of nodes that have to be splitted for the removal of infeasible paths. Krishna and Suess in [4], have given a method for timing analysis of circuits with known false sub graphs. Goldberg in [21] has given a method to determine the longest feasible path from the start to the end of a circuit in the presence of false (infeasible) paths.

3.6 Discussion

Detection of infeasible paths is important for giving a tight bound on the WCET of loops. There could exist infeasible paths due to correlation between various branches and also due to branch condition on index of loop. There could also be infeasible paths spanning over multiple iteration of loop. The infeasible paths originating due to direct correlation between branches are easy to detect and there are several works done in the past in this direction. However detecting infeasible paths which are dependent upon more than two branch conditions are hard to detect. Similarly it is hard to detect the infeasible paths that span over multiple iteration. Whalley in [23], have given a technique to detect iteration-spanning infeasible paths. But, Whalley's technique also depend upon direct relation between assignment and branch or branch and branch.

In this chapter we proposed a constraint propagation based approach to detect infeasible paths in the system. Our technique could even detect infeasible paths which are dependent upon more than two branch conditions. We also presented a WCET computation approach. A very similar approach to determine the WCET of loops is also given by Whalley in [23]. However, there are instances where our approach can give much tighter results than Whalley's approach. For instance, consider the example code given in Figure 3.14 and the various paths in it as given in Figure 3.15. Due to the possible values which 'x' can take, the sequence of paths executed is (abbab)*. Now, lets assume that the time taken by path a is more than the time taken by path b. According to Whalley's approach the minimum count after which path *a* can be repeated is two (i.e. a maximum of 5 iterations are possible out of 10, in which path a can be taken) and that of path *b* is one. Therefore, the bound on time taken by loop to execute as per Whalley's approach would be $(5 * time\_of(a) + 5 * time\_of(b))$, even when the infeasible path information is exact, which is certainly an overestimation, because path a can only be taken a maximum of 4 times out of 10 in reality. With our approach we can determine the exact WCET if the infeasible path information is exact.

Chapter 4

Simplifying WCET Analysis by Code Transformations

In chapter 3 we presented our infeasible path detection and WCET analysis techniques. It is apparent from our techniques that the number of paths from the start to end of a loop makes a lot of difference to the complexity of the technique. Similarly, the WCET analysis technique as in [23] and infeasible path detection technique of [6] depend on the number of paths in each iteration of loop. Therefore, it will be very useful if the number of paths in each iteration of loop could be reduced. The other motivation behind reducing the number of paths is that the timing prediction of loops via control flow (as in [23]) poses a lot of problems for timing analyzer. A lot of space is required to represent all the paths, unavailability of which might abort the timing analyzer. Moreover, a large number of paths will result in a significant increase of the execution time of the timing analyzer.

In the last chapter we discussed the various types of infeasible paths that could be present in a program/loop structure. There could be infeasible paths due to branch correlation and also there could be infeasible paths due to correlation between assignment of a variable and a branch condition. There could be other type of infeasible paths which span over multiple iteration of a loop. We named them as iteration-spanning infeasible paths. Detection of infeasible paths in a program is an important but difficult problem. A technique to detect and use infeasible path information is presented in [23]. We briefly describe their technique here to motivate how it could be benefited by our code transformation approach. In [23], the authors have used an effect based technique to determine the infeasible paths in a program and used this information for calculating the WCET of a loop. They first determine how a conditional branch can be effected by an assignment to a variable and/or the outcome of another conditional branch. The conditional branch could have one of the three types of effects: *unknown, fall-through* or *jump*. The effects on the conditional branches by the assignment of a variable are then exploited while traversing the basic blocks in every path of the program to determine whether the path is feasible or not. The reduction of paths in each iteration of a loop will reduce the complexity of technique in [23] to a great extent.

In chapter 3 we presented our WCET analysis technique. The complexity of our WCET analysis technique is also directly proportional to the number of paths between the start and end of each iteration. Therefore a code transformation technique which could reduce the number of paths between the start and end of each iteration can be very beneficial in reducing the complexity of our WCET analysis technique.

Due to the branches in a program structure, the number of possible paths in the program can grow exponentially. This makes the detection of infeasible paths quite complex. In this chapter we present a method to transform the code such that the number of paths in the program could be reduced and hence the search space for the infeasible paths is brought down. This could reduce the complexity of determining infeasible paths in a program and also result in tighter WCET. We present our code transformation based technique [39] as a pre-processing step to reduce the number of paths and hence reduce the complexity and time taken by other path based WCET analysis techniques.

4.1 Our Technique

We observe that the detection of infeasible paths is inherently exponential in terms of the number of branch constraints. Hence, we try to develop a strategy to identify which branch conditions can be removed from consideration during the detection of infeasible paths such that the complexity of the detection algorithm could be reduced and at the same time a tighter bound on the WCET could be provided. We also try to optimize the code such that the number of paths in the code can be reduced. We try to exploit the constraints generated at branch conditions to optimize the code. In this section we will illustrate our technique with the help of an example and also show how the WCET analysis as per [23] can be benefited by it.

4.1.1 Reducing number of loop paths

Consider the piece of code shown in Figure 4.1. The values of *x* in the Figure 4.1 seen at line number 4 are in the form of a simple harmonic motion around the value 0. The sequence of values seen for *x* at line number 4 are (0,1,2,1,0,-1)*. ‘*’ represents zero or more repetitions.

The control flow graph for the code in Figure 4.1 is shown in Figure 4.2(A). From Figure 4.2(A), it is apparent that there are 3 branch conditions and 8 paths in each iteration of the loop. The various possible paths for each iteration in terms of basic blocks executed are given in Figure 4.3.

```
1 x = 0; t = 1;
2 for(i = 0; i < 10; i++)
3 {
4      if( x == 0)
5          S1;
6      else
7          S2;
8      if( x == 2)
9          t = -1;
10     if( x == -1)
11         t = 1;
12     x = x + t;
13 }
```

Figure 4.1: Example code to illustrate our technique

However, it could be observed from the branch constraints that the results of branch conditions at block 3 and 6 could never be true simultaneously. Therefore block 4 can never be executed together with block 7. Moreover, both (true/false) paths from block 3 reaches block 6 and 8 where block 6 is a conditional statement and blocks between 6 and 8 could only be executed along with the false path from block 3. Also variable *x* (which is checked for condition at block 6) does not get assigned along the true path from block 3. Therefore, blocks 6 and 7 could be moved in the false path from block 3. Figure 4.2(B) shows the result of such a transformation.

Due to the transformation, the number of paths in the loop gets reduced to 6 from the initial number 8. Using the similar observation for conditional branches at blocks 3 and 8, the code can be optimized as shown in Figure 4.2(C), reducing the number of paths to 5. And finally the code can be modified to as shown in Figure 4.2(D), reducing the number of paths to 4.

The WCET analysis on the basis of the technique given in [23] will involve the following steps: determining the effect of assignments on the three branch conditions and then using this information to determine the infeasible sequence of paths. The technique will be greatly benefited by the optimization

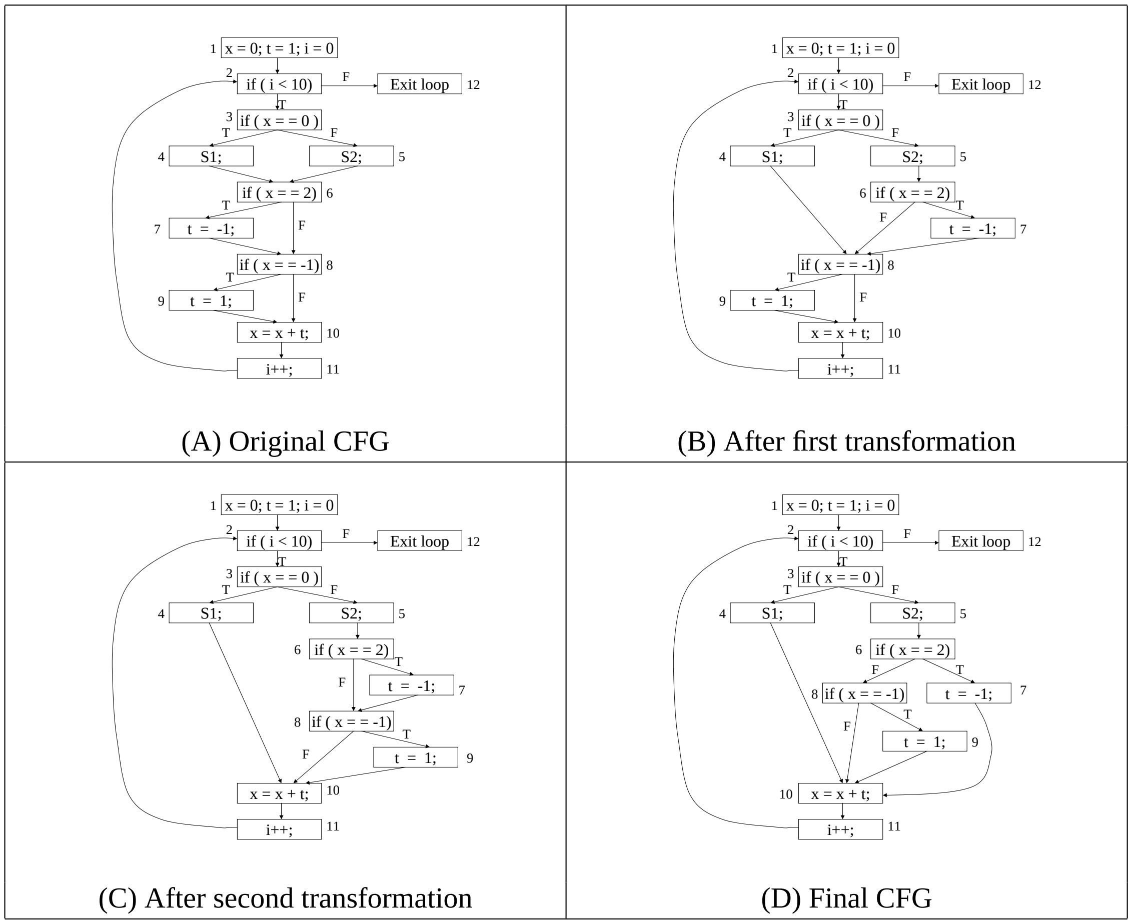


Figure 4.2: Reduction of number of loop paths in Control Flow Graph

as the number of paths are decreased and so is the complexity of the technique which traverse over the paths to determine feasibility of paths and also the sequence of paths which is infeasible in consecutive iterations.

4.1.2 Equalizing path lengths

The optimization given in the previous section will transform the original example code into an optimized code as shown in Figure 4.4. We now try to deduce a transformation for this code to further simplify the WCET analysis. For our purpose, we propose a new type of block in the CFG along with basic blocks. The new block will be called as *functional block* which will represent a function. The various paths inside such a functional block will not be considered in the WCET analysis. We will see later in

```
a : 2 3 4 6 7 8 9 10 11     b : 2 3 4 6 7 8 10 11
c : 2 3 4 6 8 9 10 11       d : 2 3 4 6 8 10 11
e : 2 3 5 6 7 8 9 10 11     f : 2 3 5 6 7 8 10 11
g : 2 3 5 6 8 9 10 11       h : 2 3 5 6 8 10 11
```

Figure 4.3: Possible paths for original loop

```
x = 0; t = 1;
for(i = 0; i < 10; i++)
{
     if(x == 0)
         S1;
     else
     {
         S2;
         if(x == 2)
             t = -1;
         else
             if( x == -1)
                 t = 1;
     }
     x = x + t;
}
```

Figure 4.4: Example code after loop path reduction

this section that a safe WCET bound can still be reached even though the number of paths considered for WCET are reduced without actually removing such paths.

We can identify the following paths, in each iteration of loop, from Figure 4.2(D).

```
a : 2 3 4 10 11            b : 2 3 5 6 8 10 11
c : 2 3 5 6 8 9 10 11      d : 2 3 5 6 7 10 11
```

The execution of loop will result in the following sequence of taken paths (abdbac)*. It is apparent that *aa, bb, cc, dd* along with *ad, abc, bdc* and many more, are infeasible sequences of paths that cannot be taken in consecutive iterations. Determining such infeasible sequences of paths with techniques as in [23] will be quite complex and computationally expensive. However, we propose the following code transformation to simplify things. The code in Figure 4.4 can be modified to the code as in Figure 4.5. The CFG for the modified code is shown in Figure 4.6

The combining of the blocks in path from 6 to 10 into *update* function and writing the *update* function in the way shown in Figure 4.5 can be very fruitful for reducing the number of paths taken in any iteration for the loop of Figure 4.4. Every call of the *update* function will take a constant amount of time due to the structure of the *update* function, hence the time taken to execute block 6 in Figure 4.6 will always be the same, irrespective of the path taken within the function. The block 6 is a *functional* block in Figure

```
x := 0; t := 1;
For(i := 0; i < 9; i++)
{
    if ( x == 0)
        S1;
    else
    {
        S2;
        update(x,t);
    }
    x = x + t;
}

update(x,t)
{
    switch (x)
    {
        case 2 : t = -1; break;
        case -1: t = 1; break;
        default: t = t;
    }
}
```

Figure 4.5: Example code after path length equalization

4.6, and a constant time can be assigned to it just like basic blocks.

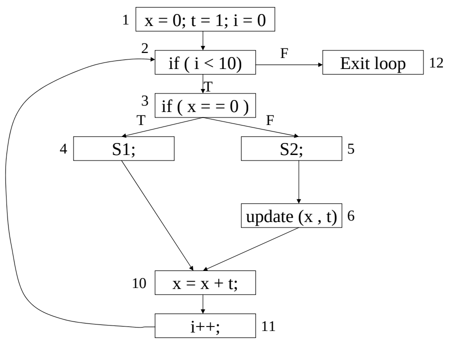


Figure 4.6: Control Flow Graph after path equalization

The transformation of code will result in the following two possible paths (from Figure 4.6) in each iteration of loop, that should be considered by the analyzer to detect infeasible sequences of paths taken in consecutive iterations.

```
a : 2 3 4 10 11          b : 2 3 5 6 10 11
```

The execution of loop will result in the following sequence of taken paths (abbbab)*, from which it is easy to identify that the infeasible sequences of paths are *aa, bbbb, abba, ababa*. The transformation results in reducing the search space for possible infeasible paths, to a great extent. Therefore the complexity of infeasible path detection as per the technique in [23] is greatly reduced and will result in a tight and safe bound on WCET. In the previous chapter we presented our constraint propagation based infeasible path detection technique. Our infeasible path detection technique is exponential in terms of number of paths present in each iteration of a loop. Hence a code transformation which could reduce the number of paths will be very beneficial to our technique (Note that in the previous chapter we used our technique to detect infeasible paths spanning over two to three iterations of a loop, in our benchmarks. However, if we try to use it for detecting infeasible paths spanning over even 5 iterations of a loop, it takes more than 2 minute for some of the benchmarks). To incorporate our code transformation technique, a proper method to handle the *functional* blocks will be required. Even though there exists other infeasible paths when the paths inside the update functions are considered, such infeasible paths can be ignored in WCET analysis as every call to update function takes constant amount of time.

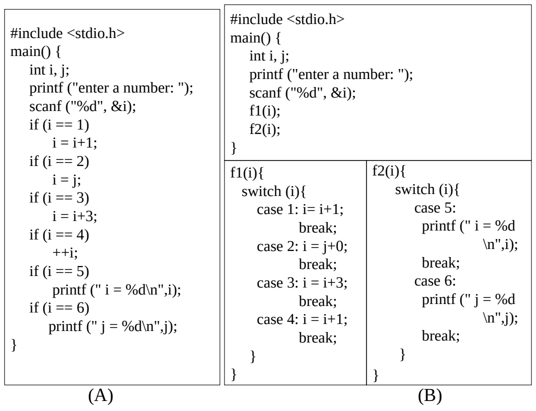
4.2 Conclusion

Detection of infeasible paths in a program is important for WCET analysis. However, it is difficult to detect all the infeasible paths in a program and moreover the search space for infeasible paths could grow exponentially in terms of number of branches in the program. Our proposed technique can not only reduce the number of paths in the program by optimization but can also consolidate a group of paths into one path as far as WCET analysis is concerned. Thus we reduce the complexity of infeasible path detection and WCET analysis.

However, due to the introduction of functions there will be decrease in performance of the system. The code transformation proposed by us will have to trade off performance with the reduction in complexity of WCET analysis.

4.3 Discussion & Future Work

Mueller and Whalley in [37] have also exploited the idea of restructuring the control flow and replicating code. However, they have used it for compiler optimization via avoiding conditional branches. Previously, Puschner in [44, 45] have also given a code transformation based approach to reduce the



```
#include <stdio.h>
main() {
    int i, j;
    printf ("enter a number: ");
    scanf ("%d", &i);
    if (i == 1)
        i = i+1;
    if (i == 2)
        i = j;
    if (i == 3)
        i = i+3;
    if (i == 4)
        ++i;
    if (i == 5)
        printf (" i = %d\n",i);
    if (i == 6)
        printf (" j = %d\n",j);
}
```

(A)

```
#include <stdio.h>
main() {
    int i, j;
    printf ("enter a number: ");
    scanf ("%d", &i);
    f1(i);
    f2(i);
}
```

```
f1(i){
  switch (i){
     case 1: i= i+1;
             break;
     case 2: i = j+0;
             break;
     case 3: i = i+3;
             break;
     case 4: i = i+1;
             break;
   }
}
```

```
f2(i){
    switch (i){
        case 5:
          printf (" i = %d
                  \n",i);
          break;
        case 6:
          printf (" j = %d
                  \n",j);
          break;
     }
}
```

(B)

Figure 4.7: Example Code: Toy6

complexity of WCET analysis. The author has proposed a single path paradigm for programs so that there could only be a single path in a program hence making WCET determination simple. Such a transformation will have to trade a lot of performance with predictability. On the other hand, with our proposed technique, the WCET analysis complexity could be reduced to a large extent without much trade off in performance. Another work by Al-Yaqoubi et. al. ([26, 1]) also describes a technique to simplify the control flow of complex loops by partitioning the control flow into sections that are limited to a predefined number of paths. Each section is then treated by the timing analyzer as a loop that iterates only once. Using the same example **Toy6** as in [1] (shown in Figure 4.7(A)), we see that our transformation (shown in Figure 4.7(B)) can reduce the number of paths in Toy6 from 64 to 1, without much increase in the code length and still giving a tight prediction for time using timing analyzer as in [23]. Function `f1` in Figure 4.7(B) can be assigned a constant amount of time (equal to any single case of the switch statement), similarly function `f2` can also be assigned a constant amount of time and both `f1, f2` are treated as functional block while calculating WCET. Hence, our approach can reduce the complexity of control flow much better than that in [26], without trading of much in terms of code length and tightness of estimation.

It should be noted that our idea for code transformation is not a timing analysis technique. It could be used as a **preprocessing** step to other infeasible path detection and timing analysis techniques such as [23, 6]. Our idea could reduce the complexity of other techniques and provide tighter bounds on WCET. Other techniques need to be modified in order to handle the functional blocks. However, *at the present* *stage we do not have a concrete technique, to determine the potential regions in the code which could be worked upon for transformation and for handling the functional blocks in WCET analysis*. In this chapter we have illustrated our idea with the help of an example to signify the benefit of such a code transformation in WCET analysis. For example, a certain type of *if* structures in the program can be optimized for reducing the paths as in the given example in this paper and also a group of basic blocks can be converted into a functional block by transforming *if* statements into a *switch* statement inside the new function. In our future work, we plan to come up with efficient methods to automatically determine potential regions for transformation and incorporate the *functional* blocks in our infeasible path detection technique.

Chapter 5

Conclusion & Future Work

In this report we presented our techniques for timing analysis of real-time systems. In chapter 2 we presented our technique to effectively determine the cache related preemption delay. With the help of our experiments we showed that our technique results in much tighter bound than other existing techniques. In chapter 3 we discussed our technique for timing analysis of loop behavior. Unlike, our CRPD technique which analyzes the micro-architectural feature, the timing analysis of loop is done at the programming language level. A tight bound on the execution time of the loop is determined by taking into account the infeasible paths within the loop. There could exists an infeasible path from start to end of each iteration and there could also exist an infeasible path spanning over multiple iteration of the loops. Both type of infeasible paths are taken care of in our technique. Chapter 4 presented a method to transform code such that the number of paths in the program, that should be considered for timing analysis, is reduced.

There are various prospects for future work in regard to techniques presented in this report. A brief description of future prospects for work is given below.

5.1 A Tighter Bound on CRPD

Consider a set of task for which the CRPD estimation is already made by existing techniques and it is determined that the set of task is non-schedulable. However, it should be noted that the current CRPD estimation techniques reports the maximum delay at any program point. Hence, there could exist other program points where the CRPD value is lower than what reported by the technique. Therefore, it is possible to schedule the set of tasks if the preempted tasks are not allowed to be preempted at program points where they incur high CRPD and at the same time the preempting tasks still meets its deadlines. Illustration of the idea with the help of an example follows.

5.1.1 Example

Consider a system of two tasks A ($C_a = 0.8, T_a = 2, D_a = T_a$) and B ($C_b = 1, T_b = 5, D_b = T_b$). Let the CRPD cost of preemption is 0.9. Let the preemption of low priority task by high priority task could be delayed by 'd'. Lets consider the schedulability of the two tasks.

$C_a < T_a - d$

$C_b + \lceil \frac{T_b}{T_a} \rceil \times C_a + CRPD \times \lfloor \frac{T_b}{T_a} \rfloor < T_b$

substituting the values for the variables.

$d < 0.8$

$1 + 3 \times 0.8 + CRPD \times 2 < 5$

$3.4 + CRPD \times 2 < 5$

$CRPD < 0.8$

For the two tasks to be schedulable the CRPD value should be less than 0.8. However, the CRPD value obtained from the static analysis is 0.9, hence the two tasks are statically determined as non-schedulable. It should be noted that the CRPD value reported is for the region where the low priority task incurs maximum CRPD. Therefore, if the low-priority task is not allowed to get preempted in the regions where it incurs CRPD greater than 0.8, the system could be made schedulable. The only issue that could arrise is that if the high priority task is not allowed to preempt then it may miss the deadline. From above equations it is clear that, even if the high priority task is delayed for 0.8 sec, it would still meet its deadline. Therefore, if the regions in low priority task which incur high CRPD has total execution time lesser than 0.8 sec, then the preemption of low priority task while executing such regions, could be avoided. And, all the tasks would still meet their deadlines. This way the above set of tasks which appear non-schedulable with given CRPD estimation could be schedulable by delaying the preemption of low-priority task at regions of high CRPD value.

5.1.2 Future work

In order to schedule the set of tasks by delaying preemptions, it is important to have a method for schedulability analysis and determining how much the preemption by various tasks could be delayed such that the set of tasks is schedulable.

5.2 Simplifying WCET Analysis

In chapter 4 we presented our approach to simplify WCET analysis by code transformations. However, our approach is presently just an idea which we showed with an example. It is important to come up with a concrete technique to solve the following issues:

1. How to determine potential regions in the code which could be converted into *functional blocks*?
2. Are there scopes other than `ifs` (which we used in our illustration in chapter 4) which could be converted into *functional blocks*? For e.g. a inner nested loop might can also be converted into a *functional block*.
3. How to incorporate the effects inside a *functional block* for detecting the infeasible paths? In fact this is a serious issue, as it might be necessary to consider the paths inside the *functional block* separately for detecting the infeasible paths and in that case the infeasible path detection will not be benefitted by transforming parts of code into *functional blocks*.

Bibliography

[1] N. Al-Yaqoubi. Reducing timing analysis complexity by partitioning control flow. Master's thesis, Florida State University, Tallahassee, FL, 1997.

[2] P. Altenbernd. On the false path problem in hard real-time programs. In *8th Euromicro Workshop on Real-Time Systems*, 1996.

[3] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *ACM PLDI Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[4] Krishna Belkhale and Alexander J. Suess. Timing analysis with known false sub graphs. In *International Conference on Computer-Aided Design (ICCAD)*, 1995.

[5] David Blaauw, Rajendran Panda, and Abhijit Das. Removing user-specified false paths from timing graphs. In *Proceedings of the 37th conference on Design automation*, pages 270–273, 2000.

[6] R. Bodik, R. Gupta, and M. Lou Soffa. Refining data flow information using infeasible paths. In *ESEC/SIGSOFT FSE*, 1997.

[7] R. Bodik, R. Gupta, and M.L. Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997.

[8] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[9] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[10] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR96-1308, University of Wisconsin-Madison, 1996.

[11] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Real-Time Technology and Applications Symposium*, pages 204–212, June 1996.

[12] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE/IEE Real-Time Embedded System Workshop (Satellite of the IEEE RealTime Systems Symposium)*, December 2001.

[13] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI'01)*, pages 286–297, 2001.

[14] Compaq. Extended static checking for java. http://research.compaq.com/SRC/esc/Simplify.html.

[15] CUDD. Colorado University Decision Diagram Package Version 2.3.1. Free software. http://vlsi.colorado.edu/ fabio/CUDD/.

[16] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1997.

[17] Harry Dwyer and John Fernando. Establishing a tight bound on task interference in embedded system instruction caches. In *Proceedings, CASES 2001*, pages 8–14, 2001.

[18] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *21st IEEE Real-Time Systems Symposium*, 2000.

[19] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of EUROPAR'97*, August 1997.

[20] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. *ACM SIGPLAN Notices*, 33(11):228–239, 1998.

[21] E. Goldberg and Alexander Saldanha. Timing analysis with implicitly specified false path. In *Int. Workshop on Timing Issues in the Specification and Synthesis of Digital Designs*, 1999.

[22] Seoul National University Real-Time Research Groups. SNU real-time benchmarks. http://archi.snu.ac.kr/realtime/benchmark/.

[23] C.A. Healy and D.B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering*, 28(8), 2002.

[24] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[25] D. Kirk. SMART (Strategic Memory Allocation for Real-Time) cache design. In *Proceedings of 10th IEEE Real-Time Systems Symp.*, pages 229–239, December 1989.

[26] L. Ko, N. Al-Yaqoubi, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. G. Harmon. Timing constraint specification and analysis. In *Software Practice and Experience*, pages 77–98, January 1999.

[27] C.-G. Lee, J. Hahn, Y.-M. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proceedings of 17th IEEE Real-Time Systems Symp.*, pages 264–274, December 1996.

[28] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Enhanced analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *IEEE Real-Time Systems Symposium*, pages 187–198, December 1997.

[29] X. Li, T. Mitra, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *Design Automation Conference (DAC)*, 2003.

[30] Xianfeng Li, Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Design space exploration of caches using compressed traces. In *18th Annual ACM International Conference on Supercomputing (ICS)*, June 2004.

[31] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 1995.

[32] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3):257–279, 1999.

[33] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 213–227, Washington - Brussels - Tokyo, June 1997. IEEE.

[34] Thomas Lundqvist and Per Stenstrom. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–15, 1998.

[35] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[36] T. Mitra, A. Roychoudhury, and X. Li. Timing analysis of embedded software for speculative processors. In *ACM Intl. Symp. on System Synthesis (ISSS)*, 2002.

[37] F. Mueller and D. B. Whalley. Avoiding conditional branches via code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 55–66, June 1995.

[38] Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS Merged Conference*, October 2003.

[39] Hemendra Singh Negi, Abhik Roychoudhury, and Tulika Mitra. Simplifying wcet analysis by code transformations. In *4th International Workshop on WCET Analysis*, June 2004.

[40] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag Berlin Heidelberg, 1999.

[41] C.Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Journal of Real-time Systems*, 5(1), 1993.

[42] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. New York: Cambridge University Press, 1992.

[43] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE International Real-Time Systems Symposium*, December 2002.

[44] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In *Proceedings of IFIP World Computer Congress, Stream on Distributed and Parallel Embedded Systems*, pages 163–172, 2002.

[45] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, January 2002.

[46] J. Stankovic. Misconceptions about real-time computing. In *IEEE Computer*, october 1988.

[47] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 132–140, 2001.

[48] Jan Staschulat and Rolf Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT 2004*, September 2004.

[49] R. A. Sugumar and S. G. Abraham. Efficient simulation of multiple cache configurations using binomial trees. Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991.

[50] Yudong Tan and Vincent J. Mooney III. Timing analysis for preemptive multi-tasking real-time systems with caches. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Volume II (DATE'04)*, February 2004.

[51] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache related preemption delay in preemptive real time systems. In *Proceedings of 8th International Workshop on Hardware/Software Codesign (CODES2000)*, pages 67–71, May 2000.

[52] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *2003 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03) , San Diego*, 2003.

[53] Reinhard Wilhelm. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. In *VMCAI 2004*, volume 2937 of *LNCS*, pages 309–322, 2004.

[54] A. Wolfe. Software-based cache partitioning for real-time applications. *Journal of Computer & Software Engineering*, 1(3), 1994.