Design and Implementation of a Simulator for Precise WCET Estimation of Multithreaded Programs

Original Scientific Paper

P. Padma Priya Dharishini
Department of Computer Science and Engineering
Ramaiah University of Applied Sciences
Bangalore, India
padmapriya.cs.et@msruas.ac.in

P. V. R. Murthy
Department of Artificial Intelligence and Data Science
Nitte Meenakshi Institute of Technology
Bangalore, India
pvr.murthy@nmit.ac.in

Abstract *– Significant attention is paid to static analysis methods for Worst Case Execution Time Analysis of programs. However, major effort has been focused on WCET analysis of sequential programs and only a little work is performed on that of multithreaded programs. Shared computer architectural units such as shared instruction cache pose a special challenge in WCET analysis of multithreaded programs. The principle used to improve the precision of shared instruction cache analysis is to shrink the set of interferences, from competing threads to an instruction in a thread that may be accessed from shared instruction cache, using static analysis extended to barriers. An Algorithm that address barrier synchronization and used by the simulator is designed and benchmark programs consisting of both barrier synchronization and computation task synchronization are presented. Improvements in precision upto 20 % are observed while performing the proposed WCET analysis on benchmark programs.*

Keywords*: Worst Case Execution Time; Shared Instruction Cache Analysis; Multithreaded Program, Multicore Architecture*

1. INTRODUCTION

Today Real Time Embedded Systems (RTES) are vastly used in avionics, automotive and tele-communications domains. In RTES, correctness of a system not only depends on its logical behavior but also computation time. In hard real time systems, missing deadlines can cause catastrophic damage. Multicore architectures are used in RTES domain due to their high processing power and concurrency in applications. For example, in night view assist multi-threaded in automotive environment, reading data from sensors, processing video streams and raising warning when an obstacle is detected on road happen concurrently.

Schedulability analysis is used to verify the capabilities of RTES to meet deadlines. All schedulability analyses assume that upper bound of execution of each program i.e., Worst Case Execution Time (WCET) is known. However, deriving tight and safe WCET of a program on multicore architecture is difficult because of shared hardware resources such as cache memory, buses and Input/Output. There may be unpredictable delays in the execution of the program due to contention at shared resources. One of the main factors of unpredictability is due to cache memory. There are two or three levels of cache memory placed between core and main memory to bridge the gap of high-speed processor with low-speed main memory. L1 cache memory is the smallest cache memory close to the processor and it is private for each core, while larger L2 is shared between cores. In the case of an L1 cache miss (requested memory block not in L1 cache memory) then, memory block may be fetched from higher level of memory hierarchy(L2 cache). The memory access latency to be computed due to conflicts from other cores resulting in the removal of memory block from shared L2 instruction cache plays a critical role in the estimation of precise WCET of a multithreaded program. The problem of estimating worst case latency in turn to estimate WCET of a multithreaded program is motivated in this paper for shared instruction caches.

A static analyzer is designed in [1] using Hoare's Communicating Sequential Processes (CSP) [2] to compute WCET of a multithreaded program and is based on synchronized parallel processes arising from synchronization calls to wait() and notify(). A reference thread is one for which WCET is being estimated and instruction accesses in it encounter competition for shared instruction cache from parallel processes in other threads. The conflicts for any instruction I in reference thread are encountered only from the identified parallel processes that run parallel with I. A gap identified in the static analyzer [1] is that it does not deal with barrier synchronization processes. This paper extends Interference Partitioning algorithm in [1] to address the class of programs using barrier synchronization as well. User defined abstractions are linked to the program code using PragMatics approach in [3]. The approach is based on an annotation language comprising of all features to address individual loops, application context and function calls with optimization awareness. However, pragMatics does not support recursive applications. The structure of parallel program along with its target platform is considered to obtain tight contention delays in [4]. The main drawback of the approach is that it is limited to blocking communication. Fork-join parallel model is employed in [5], in contrast, the proposed method employs fork-join, Single Program Multiple Data (SPMD), Multiple Program Multiple Data (MPMD), producer - consumer model in parallel programming. An Integer Linear Programming (ILP) based approach is proposed in [6] that maximizes the WCET of a program. It is also proposed that algorithmic approaches scale better for larger programs than ILP based approach. A parallel execution graph is employed in [7] to explore all possible execution interleavings of a parallel task and an exclusion criterion is proposed to prove that certain interleavings can never occur to make precise and feasible WCET analysis of parallel periodic tasks. Communication between the tasks in concurrent software is through message passing and life time estimates of concurrently executing tasks on multicore are improved progressively in [8]. Automatic timing analysis of parallel applications is performed in [9] by considering synchronization stall time associated with each instruction and each basic block in Control Flow Graph (CFG) for WCET estimation process. The approach considers a simple time predictable architecture to estimate WCET. Loop bounds are provided as user annotations to the WCET analyzer [10]. WCET computation of a multithreaded program is proposed in [11] and communication edges are introduced between threads in a multithreaded program in micro architectural modelling phase of WCET estimation.

The instructions that can cause or suffer from timing interferences are extracted in [12]. Based on the extracted instructions, the real time tasks are separated into a sequence of time intervals. The ILP solver uses the time intervals to minimize the WCET of the application. Concurrent execution of programs is simulated to cause conflicts resulting in the eviction of memory block from the shared instruction cache, being accessed by program in reference thread in [13]. A hardware mechanism is proposed to reduce the number of interfering accesses by forcing certain accesses to bypass shared cache. WCET analysis of parallel code can be performed using UPPAAL model checker [14]. The approach in [14] considers granularity at instruction level that increases the size of the state space compared to the basic block level granularity. Scheduling model for real-time tasks is presented in [15] and concurrency during task execution is not considered explicitly. In contrast, in the proposed approach in our paper, concurrency among the threads is considered explicitly. Interference Partition (IP) Algorithm [16], computes WCET of a multithreaded program by considering partial order information [17] of the multithreaded program based on wait and notify synchronization. IP Algorithm partitions the Control Flow Graph (CFG) each thread of a multithreaded program into parallel processes P_{m_i} (m is process id and i is thread id) based on partial order information derived using wait/notify synchronization primitives. The partitioning enables computation of a precise WCET of the multithreaded program. The research question addressed in this paper are:

- What parameters need to be considered during WCET analysis of a multithreaded program to provide precise estimates of WCET to designers of Real-time embedded applications?
- How can shared instruction cache memory be modelled by a WCET analyser for precise WCET estimation?

The main contributions are

- Extension of the interference partitioning algorithm in [16] for multithreaded programs to incorporate barrier synchronization calls
- Investigation of the effectiveness of the extended interference partitioning algorithm on benchmark programs adapted from Malardalen [18] benchmark suite
- WCET estimates of multithreaded programs with barrier synchronization calls and computation task specific synchronization calls (using wait() and notify())
- Parameters such as Number of conflicts, Conflict ratio, Overestimation ratio, Precision Improvement in Number of conflicts and Precision Improvement in WCET are proposed for performance evaluations

2. WCET ESTIMATION OF MULTITHREADED PROGRAMS

Typical WCET estimation framework of sequential program mainly comprises of three phases [10]: program flow analysis to obtain structural and functional constraints from the control flow graph of the program, micro-architectural modelling to obtain WCET of each basic block by considering underlying architectural features like cache, pipeline, branch prediction etc. and WCET calculation phase to obtain WCET of the program by maxi-

mizing the objective function comprises of execution time and execution count of each basic block. The above WCET estimation framework is not quite appropriate for WCET estimation of a multithreaded program because of complex interactions between threads in the program, mapped to different cores. A novel method is proposed and incorporated into simulator to obtain WCET of a multithreaded program implemented to run on a multicore architecture with shared instruction cache, by reducing the set of interferences to be considered to a minimal safe subset, from an interacting thread during the execution of an instruction in a reference thread.

Existing IP algorithm [16] does not deal with barrier synchronization processes and it is extended to obtain minimal safe subset of interferences from interacting threads using barrier synchronization primitives. All the functions from n threads of a multithreaded program have to reach the barrier before they proceed and barrier node counts the arrival of all the threads and once all the threads arrived it issued the proceed messages [19]. Real time embedded applications perform computation in k phases. The requirement of the application is that all the threads need to begin the computation of the ith phase, $i<=k$, only if i-1th phase is completed by all threads. For such an application typically barrier synchronization is used and all K barriers will be designed and programmed. It may be considered that the computation processes, in threads following the i-1th barrier until the ith barrier, are parallel to each other. In fact, the above-mentioned parallel processes are special kind of synchronized parallel processes [1]. The parallel processes inside a barrier may also perform a computation task specific synchronization using wait and notify synchronization primitives at a lower level while there is higher level parallelism between threads using barrier synchronization processes.

Definition of computation Processes

Computation processes arise when two threads interact using wait() and notify() synchronization calls. We refer to them as synchronization parallel processes in the paper. There is an order between processes imposed by the partial order wait<notify. There exist code regions (synchronized parallel processes) in the two threads that may be parallel to each other. Here synchronization calls are used in threads simply to wait and notify and not for barrier synchronization. What is important for the Interference Partitioning algorithm is that synchronized parallel processes in two threads compete with each other for a shared resource such as the shared instruction cache.

For example, $BSP_{2_2} \parallel BSP_{2_3}$ in Fig.1.b are identified as computation processes because they interact using wait() and notify() synchronization calls. There is order between processes imposed by the partial order wait < notify.

Definition of computation task specific synchronization calls

A computation task specific synchronization calls are defined with respect to two interacting threads. The interaction is based on synchronization using calls to wait () and notify (). It is assumed that the there is no notification loss as the program is validated before WCET analysis is performed. Therefore, for corresponding synchronization calls, wait < notify. Computation task specific synchronization calls identify not only which computation in a thread happens before the computation in another thread but also which computations happen in parallel.

For example, $BSP_{2_2} \parallel BSP_{2_3}$ are identified as computation task specific synchronization calls in Fig.1.b. In general, a multithreaded program may contain multiple barriers as shown in Fig.1.a. for a group of threads to synchronize on completion of tasks one after the other. When it comes to barrier synchronization parallel processes, two or more threads may participate in barrier synchronization and they cross the barrier for further computation together irrespective of the relative speeds until they reach the barrier. From the perspective of the Interference Partitioning algorithm, if k threads participate in the barrier, for a reference thread, (k-1) barrier synchronization parallel processes compete for the shared instruction cache along with the reference thread.

Definition of barrier synchronization parallel processes

Barrier synchronization parallel processes, two or more threads may participate in barrier synchronization and they cross the barrier for further computation together irrespective of the relative speeds until they reach the barrier. From the perspective of the Interference Partitioning algorithm, if k threads participate in barrier synchronization, for a reference thread participating in the barrier synchronization, (k-1) barrier synchronization parallel processes compete for the shared instruction cache.

For example, $BSP_{2_1} \parallel BSP_{2_2} \parallel BSP_{2_3}$ are identified as barrier synchronization parallel processes in Fig.1.a. If there are no task specific synchronization calls between two barrier lines, entire code region between the two barrier lines for each thread is a competing process that can cause shared instruction cache interferences to code of other threads in between the barrier lines. All the functions from n threads of multithreaded program need to reach the barrier before they proceed further. The barrier nodes shall keep track of the arrival of all the participating threads and once all the threads arrive at the node or barrier line, the threads proceed beyond.

Definition of Parallel Processes

Parallelism may exist between threads that simply arises out of no particular order between executions of regions of code in the threads, which we in general term as parallel processes. Parallel processes also compete for the shared instruction cache.

Competing processes refer to barrier synchronized parallel processes or synchronized parallel processes or only computation processes that run parallel with a corresponding process in the reference thread T_i. Competing processes cause conflicts to an instruction I in the

reference thread. The IP algorithm creates the mapping of the competing processes for all barrier synchronization parallel processes in Fig.1.a. as shown in Table 1.

Table 1. Mapping of competing processes for each Barrier Synchronization Parallel Processes BSP_{m_i}

Barrier Synchronization Parallel Processes BSP_{m_i}	Competing processes
BSP_{1_1}	$BSP_{1_2} \|\| BSP_{1_3} \|\| BSP_{1_4}$
BSP_{2_1}	$BSP_{2_2} \|\| BSP_{2_3} \|\| BSP_{1_4}$
BSP_{3_1}	$BSP_{3_2} \|\| BSP_{3_3} \|\| BSP_{1_4}$

In general, a thread is a composition of computation processes interacting using wait() and notify() calls, barrier synchronization processes and simply parallel processes. The interference partitioning algorithm addresses the problem of determining competing processes for a reference thread for each of the three categories of processes. Micro-architectural modelling uses competing processes to identify conflicts to any barrier synchronization parallel processes BSP_{n_j} to compute WCET of each basic block referred as node in this paper.

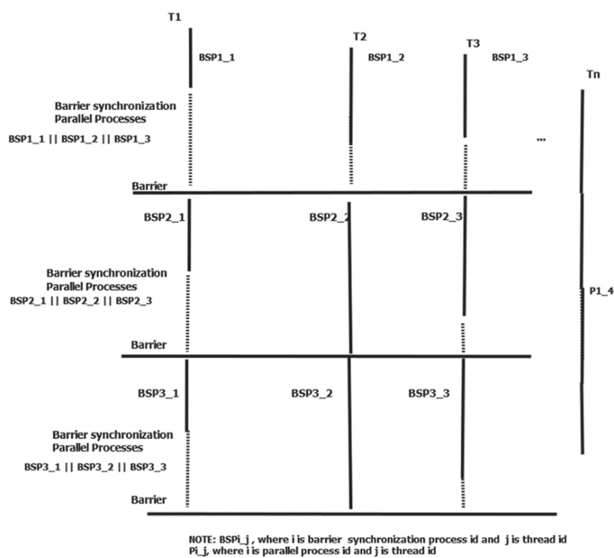


Fig. 1. a. Identified barrier synchronization parallel processes

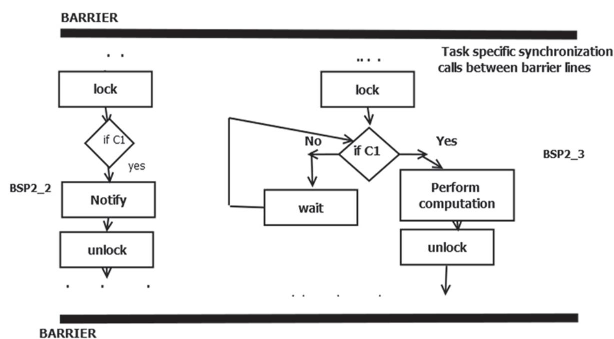


Fig. 1. b. Task specific synchronization calls between two barrier synchronization parallel processes

Fig.1. Multithreaded program with Barrier Synchronization Parallel Processes

WCET calculation uses Implicit Path Enumeration Technique (IPET) that combines program flow information along with its execution time of each basic block to compute WCET of each thread of multithreaded program. The subsection 2.1 explains mathematical representation of cache mapping function in detail.

2.1. CACHE MAPPING FUNCTION

The motivation for discussing Cache Mapping Function is to determine the instruction accesses, in shared instruction cache, made by a thread for which WCET is being estimated(reference thread) that are potentially evicted due to interferences from competing threads. The Cache Mapping Function is used by the Interference Partitioning algorithm. In this paper, barrier synchronization processes cause interferences that can evict instructions in shared instruction cache required by the reference thread. This paper extends Interference Partitioning algorithm to multithreaded programs using barrier synchronization. The three commonly used cache architectures are direct mapped cache, fully associative cache and A-way set associative caches [20]. An A-way set associative mapping architecture contains A cache lines for all S cache sets. Each cache line is capable of holding LS consecutive bytes of a memory block. Direct mapped cache, is a 1-way set associative cache where a cache block can appear in only one place in cache memory. Fully associative cache, is a A-way set associative cache where a cache block can be placed anywhere in the cache memory having only one set. A cache line may be valid (containing a memory block) or invalid (currently free).

This paper considers Harvard architecture i.e., $L1$ instruction cache is separated from $L1$ data cache and $L2$ shared instruction cache is a share resource for all cores.

- Cache size CS: Represents cache memory size in bytes
- Block size or Line Size LS: Represents number of bytes to be loaded in to cache for each memory access
- LS_L1: Represents line size of level 1 cache memory
- LS_L2: Represents line size of level 2 cache memory
- Associativity A: Accessed memory block can be placed in A cache lines in cache memory
- Cache Set S: $S=\{s_1, s_2, \ldots, s_{(CS/LS)/A)}\}$ A cache set s_i is a sequence of cache lines where memory blocks are stored
- S_L1: Represents number of cache sets in level 1 cache memory
- S_L2: Represents number of cache sets in level 2 cache memory
- Number of cache lines CL : $CL = \{l_1, l_2, \ldots, l_A\}$
- Memory block: Sequence of consecutive instructions based on block or line size

The set of ages for A-way set associative caches are $A= \{0,1,2,\ldots, A\text{-}1\}$. The block replacement method considers only the age of the memory block.

The most recently used memory block is given age 0 and least recently used memory block is given maximal age A-1. For each cache miss, the accessed block is placed in a particular cache set based on cache architecture with age as 0, age of all other memory blocks in particular cache set is increased by 1 and memory block with age A-1 is evicted from cache memory. For access to a memory block that is currently in cache memory with age a, its age is changed to 0 and the ages of memory blocks lesser than a are increased by 1 and ages of memory blocks greater than a remains the same. Instructions in each memory block are classified as Always Hit (AH), Always Miss (AM) or Not Classified (NC) based on Cache analysis using Abstract interpretation (AI) [20]. Abstract Interpretation based cache analysis does not require execution of the program to study cache behavior of the program; through appropriate abstraction, cache behavior for a program can be inferred using static analysis [20]. Each memory block is mapped to a particular cache set in $L1$ cache memory based on cache mapping function CMF_L1. The equation 1 and 2 shows the Cache Mapping Function of $L1$ and $L2$ cache memory respectively.

$$CMF_L1=((Memory\ address\ /\ LS_L1)\ \%\ S_L1 \quad (1)$$

Similarly, mapping function of shared L2 instruction cache is,

$$CMF_L2=((Memory\ address\ /\ LS_L2)\ \%\ S_L2 \quad (2)$$

The Instruction I mapped to cache set s_i having Cache Hit Miss Classification ($CHMC$) categorized as AH for shared $L2$ instruction cache is affected by a set of instructions { I_1', I_2',... ,I_k' } from interacting threads that mapped to same cache set s_i [20][21][22]. To compute where to place the memory block in $L1/L2$ cache memory, the following notations are used. Suppose instruction I_{7_1}: addiu $29,$29,-72 in Fig.3. of reference thread is stored at the memory address 0x400220. Each memory block is mapped to a particular cache set in $L1$ cache memory based on cache mapping function CMF_L1 and it is used to compute the instruction's location in $L1$ cache memory having cache size CS as 256 bytes, line size LS_L1 as 16 bytes and Associativity A as 1. Another parameter in CMF_L1, which is number of cache sets S_L1 in $L1$ cache memory, is computed using ($CS/LS_L1/A$) of $L1$ cache memory. The instruction in memory address 0x400220, is mapped to cache set number 2 of $L1$ cache memory.

$$CMF_L1=((Memory\ address\ /\ LS_L1)\ \%\ S_L1$$

$$CMF_L1=((0x400220\ /\ 0x10)\ \%\ 0x4$$

Interference Partition Algorithm for Barrier Synchronization

Each reference thread T_i is viewed as a composition of barrier synchronization parallel processes, communicating with other barrier synchronization parallel processes in interacting threads T_j in Fig.1.a. In general, a thread is a composition of computation processes interacting using wait() and notify() calls, barrier synchronization processes and simply parallel processes. The interference partitioning algorithm addresses the problem of determining competing processes for a reference thread for each of the three categories of processes. The Interference Partitioning Algorithm accepts as input a Message Sequence Chart (MSC) representation of the Communicating Sequential Processes (CSP) specification of the multithreaded program. An MSC representation consists of lifelines for threads in the program as shown in Fig1.a. Interactions between threads are through computation task specific synchronization calls (wait(), notify()) or barrier synchronization calls. A partial order based on wait < notify is constructed from the multithreaded program while transforming it to an equivalent CSP specification. The Interference Partitioning Algorithm (addressing Barrier Synchronization) traverses through MSC representation looking for synchronization calls step by step. The $WCET$ analyser identifies computation task synchronization parallel processes based on synchronization calls wait() and notify(). The $WCET$ analyser identifies barrier synchronization region in each thread using barrier initialization and barrier related calls. Simply parallel processes are identified by the $WCET$ analyser based on partial order through which neither less than nor greater than relation is observed for parallel regions. It may be noted that for uniformity, $WCET$ analyser considers a sequential process in a thread to be parallel to an empty process in an interacting thread. In this way all processes are considered to be parallel.

Algorithm for Interference Partitioning identifying barrier synchronization processes

While there exists next set of syncCalls in Thread(T)

```
{
    listOfSyncCalls = getNextSetOfSynchronizationCalls(T);
    if listOfSyncCalls contains barrierSyncCalls
        barrierSyncProcesses=identifyNextBarrierSynchronizatio
        Processes(T);
    else if listOfSyncCalls contains computationTaskSyncCalls
        computationSyncProcesses=
        identifyNextSynchronizedParallelProcesses(T);
    else if (listOfSyncCalls is empty)
        onlyComputationProcesses=identifySolelyComputation
        Processes(T);
    CreateMappingOfCompetingProcessesToCurrentProcessIn
    Thread(T,barrierSyncProcesses,computationSyncProcesses,
    onlyComputationProcesses)
}
```

The Interference Partitioning Algorithm that identifies competing Barrier Synchronization processes is an extension of the basic Interference Partitioning Algorithm [16] that deals only with Computation Task Synchronization calls. The algorithm considers an abstract view of the multithreaded program as a Message Sequence Chart(MSC) as shown in Fig.1.a. The first thread T_1 may be considered a reference thread T for which

$WCET$ is being estimated and with other threads competing for shared resources such as the shared instruction cache. The same procedure is applicable for each and every thread(as a reference thread). The Interference Partitioning algorithm uses thread T as the argument or input and the competing process set is determined for each process in T. The algorithm is applied on each thread to estimate the worst case latency in accessing shared instruction cache with competition of access from other threads. A benchmark program is considered to explain the estimation of worst case latency in accessing shared instruction cache which in turn is used in $WCET$ estimation of each thread. Fig.1.a. shows generic structure of a benchmark multithreaded program with functions from Malardalen benchmark programs[18]. The Control Flow Graph (CFG) of each thread is constructed from the assembly code of the multithreaded program. After constructing individual CFGs of threads, the procedural call graph of the program is traversed to construct a global flow graph called Transformed Control Flow Graph ($TCFG$).

The existing approaches to determining conflict set in accessing shared instruction cache during the $WCET$ analysis of a multi-threaded program are not quite exploiting the order and concurrency information between regions of code in threads [10] [13]. The Interference Partitioning algorithm uses the order and concurrency information inferred from partial order of execution of threads. As a consequence, a larger conflict set is used while accessing shared instruction cache when partial order between threads is not used. On the contrary, conflict set that Interference Partitioning algorithm uses, by exploiting partial order information between threads, is only a subset of the conflict set used without partial order information.

The instruction sequence of BSP_{2_2} and BSP_{2_3} in Fig.1.b is shown in Fig. 2. In the case of IP algorithm, conflicts arising for any instruction I_{i_2} in BSP_{2_2} are from any instruction in BSP_{2_3} ,mapped to the same cache set S_l. In contrast, in existing approaches [10][13], conflicts are from all the instructions in the entire program region. In Fig. 2. the same is shown for instruction I_{k_2} in BSP_{2_2}. The parallel process or code region BSP_{2_3} is a subset of the entire code region and hence the conflict set generated using partial order information is smaller.

Let BSP_{m_1} is the region of code in T_1 between barrier sync lines i and i+1. Fig. 3. shows CFG of a simple barrier synchronization parallel process BSP_{m_1} along with Cache Hit and Miss Classification table ($CHMC$) of all instructions in BSP_{m_1} following $L2$ cache analysis. The instruction I_{2_1} in BSP_{m_1} is categorized as AH in $L1$ instruction cache memory. Therefore, I_{2_1} will never access shared $L2$ instruction cache. The instruction I_{7_1} in BSP_{m_1} is categorized as AH in L2 instruction cache memory. Therefore, I_{7_1} will be affected by accesses to instructions made by threads running on other cores referred to as conflicts. Let BSP_{m_1} is the region of code in T_i between barrier sync lines i and i+1.

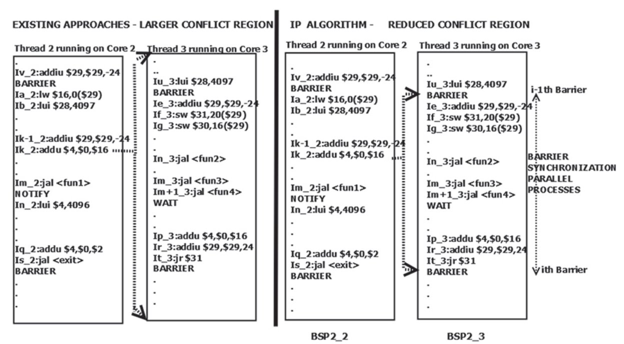


Fig. 2. Conflict Region for Existing approaches and IP Algorithm

Definition of conflicts in IP Algorithm

The conflicts for an instruction I accessed by thread T_i, mapped to cache set S_l, that belongs to any barrier synchronization parallel process BSP_{m_1} are from the instruction set $\{I_1', I_2', ..., I_p'\}$, mapped to same cache set S_l and that belongs to competing processes of BSP_{m_1}.

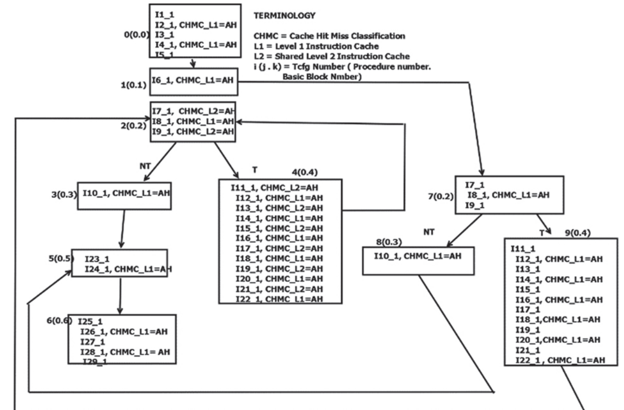


Fig. 3. Control Flow Graph of BSP_{m_1}

In the existing approaches, the conflicts for an instruction I in T_i mapped to cache set S_l is from entire program region of T_j mapped to same cache set S_l. For example, as shown in Table 2, the conflicts for instruction I_{13_1} in T_1 mapped to cache set s_1 in shared $L2$ instruction cache are from all the instructions in T_2 mapped to same cache set s_1. In IP Algorithm, the conflicts for any instruction are obtained based on partial order information derived using barrier synchronization primitives. Let BSP_{m_1} and BSP_{n_2} are barrier synchronization processes that belong to the same barrier region in threads T_1 and T_2 respectively. Therefore, the conflicts for instruction I_{13_1} in Fig.3. of barrier synchronization parallel process BSP_{m_1} of T_1 mapped to cache set s_1 in shared $L2$ instruction cache are from instructions $\{I_1', I_2', ..., I_p'\}$ in BSP_{n_2} that belongs to competing process in T_2 mapped to same cache set s_1.

IP algorithm performs inter thread shared instruction cache analysis by considering conflicts only from the instructions in barrier synchronization parallel processes BSP_{n_2} that run parallel with BSP_{m_1}. Based on those analyses, the age of each instruction is updated.

The Age Update Function (AUF) for any instruction I in shared instruction cache analysis is

$$AUF{:}Age(I) = Age(I) + conflicts$$

If age of instruction I is greater than or equal to associativity of shared $L2$ instruction cache memory, then instruction I that is currently in cache memory is categorized as NC for shared $L2$ instruction cache accesses. For example, the instruction I_{7_1} in Fig. 3. is categorized as AH following the application of AUF and I_{9_1} is categorized as Not Classified (NC) as shown in Table 2 for IP algorithm. The reduction of conflicts for each instruction leads to reduction in number of consolidated number of conflicts for each node which in turn leads to reduction in number of consolidated conflicts for a parallel process and finally leading to a reduction in consolidated number of conflicts of a thread in a multithreaded program. The above leads to precision improvement in statically estimating $WCET$ for a multithreaded program that may use barrier synchronization. This is made possible as an abstract view of parallelism in threads is not at whole thread level in our $WCET$ analyser but at smaller process level which is arising from code in barrier synchronization regions in threads.

Table 2. Number of conflicts and worst case latency

Instruction: Address	Cache Set Number	Age	Number of Conflicts		Cache Hit/Miss Classification		Worst Case Latency in Clock Cycles	
			IP Algorithm	Existing Approaches	IP Algorithm	Existing Approaches	IP Algorithm	Existing Approaches
I_{7_1}:400220	2	1	2	13	AH	NC	7	37
I_{9_1}:400230	3	1	3	13	NC	NC	37	37
I_{11_1}:400240	0	1	2	14	AH	NC	7	37
I_{13_1}:400250	1	1	3	14	NC	NC	37	37
I_{15_1}:400260	2	1	2	13	AH	NC	7	37
I_{17_1}:400270	3	1	3	14	NC	NC	37	37
I_{19_1}:400280	0	1	3	14	NC	NC	37	37
I_{21_1}:400290	1	1	3	14	NC	NC	37	37

Number of Conflicts

As discussed, the conflicts for an instruction I in T_i, mapped to cache set S_l that belongs to BSP_{m_i}, are from the set of instructions $\{I_1', I_2', ..., I_p'\}$, mapped to the same cache set S_l, which belongs to BSP_{n_j} (i.e. competing process of BSP_{m_i} in T_j.

Number of conflicts Caused by a Competing Thread

The number of conflicts encountered by a node n in a barrier synchronization parallel process BSP_{m_i} is the sum of number of conflicts encountered by each instruction $\{I_1', I_2', ..., I_t'\}$ in n. Therefore, the number of conflicts of a barrier synchronization parallel process BSP_{m_i} is the sum of consolidated number of conflicts of each node $\{B_1, B_2, ..., B_k\}$ in BSP_{m_i}. Hence, the consolidated number of conflicts of a thread T_i is the sum of number of conflicts of each barrier synchronization parallel process $\{BSP_{1_i}, BSP_{2_i}, \ldots BSP_{q_i}\}$ in T_i. It may however be noted that precision improvement in Worst Case Latency in accessing shared instruction cache takes place as the worst case execution time of each instruction, as simulated, becomes more precise due to reduction in conflicts. Thus, reduction in consolidated number of conflicts caused by a competing thread using IP algorithm is just an indication of the superiority of the approach even when barrier synchronization is used.

As a consequence of reduced number of conflicts for instruction I, $CHMC$ of I remains AH in shared $L2$ instruction cache that leads to reduced worst case latency of instruction. There are a few instructions having reduced number of conflicts with $CHMC$ categorized as NC due to its age in shared $L2$ instruction cache that leads to maximum worst case latency of instruction. Table 3 shows the number of conflicts of an instruction I, node n containing I, barrier synchronization parallel process BSP_{m_i} containing node n and instruction I, thread T_i of I associated with its $WCET$ for both approaches.

Table 3. Number of Conflicts and $WCET$ of I_{7_1} and I_{9_1}

Inst Id	Parameters		IP Algorithm	Existing Approaches
I_{7_1}	Number of Conflicts	Instruction	2	13
		Node	5	26
		Barrier synchronization parallel process	21	109
		Thread	511	4365
	WCET in Clock Cycles	Instruction	7	37
		Node	45	75
		Barrier synchronization parallel process	2571	15300
		Thread	2910490	3506290
I_{9_1}	Number of Conflicts	Instruction	3	13
		Node	5	26
		Barrier synchronization parallel process	21	109
		Thread	511	4365
	WCET in Clock Cycles	Instruction	37	37
		Node	45	75
		Barrier synchronization parallel process	2571	15300
		Thread	2910490	3506290

Number of conflicts as is being talked about is only an indirect pointer to precision improvement. $WCET$ estimate depends on better estimate of worst-case time for each instruction. Number of consolidated reductions in conflicts from a competing thread to shared instruction cache is an indication and explanation on why $WCET$ estimate for a thread improves. It is also evident from Table 3 that reduction in Number of conflicts of an instruction I does not necessarily leads to reduction in $WCET$ of I. In this paper, $L1$ cache miss latency is assumed as 6 clock cycles and 30 clock cycles for $L2$ cache miss latency.

Conflict ratio

The next parameter considered to evaluate the performance of IP algorithm is Conflict ratio. Conflict ratio of a node n in barrier synchronization parallel process BSP_{m_i} computed for IP algorithm is always lesser than or equal to Conflict ratio of a node n in barrier synchronization parallel process BSP_{m_i} of existing approaches.

Definition of Conflict ratio

Conflict ratio of a node n in barrier synchronization parallel process BSP_{m_i} is calculated by dividing number of conflicts of a node n by the total number of instructions in n. Similarly, conflict ratio of a barrier synchronization parallel process BSP_{m_i} in thread T_i is calculated by dividing number of conflicts of a barrier synchronization parallel process BSP_{m_i} by the total number of instructions in barrier synchronization parallel process BSP_{m_i}. Likewise, conflict ratio for a thread T_i is calculated by dividing number of conflicts of a thread T_i by the total number of instructions in T_i.

Over Estimation Ratio of WCET

CMP-SIM simulator (a multi-core extension of simple scalar tool set [23]), used to evaluate the accuracy of the static analyzer experimentally. All the experiments are performed in 2-cores with different architectural parameters. The estimated $WCET$ obtained using IP algorithm is compared with the simulated $WCET$.

The simulated $WCET$ of the program is highly underestimated than actual $WCET$. The worst-case input of some benchmarks is difficult to obtain because of branching and other complex mathematical calculations. The over estimation ratio of existing approaches is computed as $WCET_{Existing\ Approaches}/WCET_{Observed\ WCET}$ similarly, overestimation ratio of IP algorithm is computed as $WCET_{Interference\ Partition\ algorithm} / WCET_{Observed\ WCET}$.

Precision Improvement in Number of conflicts

The reduction in number of conflicts is considered as one of the major parameters of performance evaluation. The precision improvement in Number of conflicts is computed as $((Number\ of\ conflicts_{Existing\ Approaches} - Number\ of\ conflicts_{Interference\ Partition\ algorithm}) / Number\ of\ conflicts_{Existing\ Approaches}) * 100$. The precision improvement in number of conflicts varies from 60-90%, this is mainly due to minimal safe subset of conflicts from an interacting thread during the execution of an instruction in a reference thread.

Precision Improvement in WCET

The precision improvement in $WCET$ is computed as $((WCET_{Existing\ Approaches} - WCET_{Interference\ Partition\ algorithm}) / WCET_{Existing\ Approaches})*100$. The precision improvement in $WCET$ varies from 15-20%, this is due to shared $L2$ instruction cache hits inside loops. Though there is a huge precision improvement upto 90 % in number of conflicts, the precision improvement in $WCET$ is 20% and the reason for the same is discussed in section 3.

3. RESULTS AND DISCUSSION

The simulator multi-core chronos [24] [25] is extended to incorporate Interference Partition Algorithm for barrier synchronization. Multi-core chronos tool is extended to make it aware of threads with synchronization information, that is, to identify barrier synchronization parallel processes to be used by the IP algorithm.

Design of Simulator

Multi-core Chronos Simulator [24][25] is extended to keep track of the code regions in other threads that compete for shared instruction cache through conflicts or interferences as an instruction in a thread T is being accessed from the shared instruction cache. $WCET$ is being estimated for thread T and hence simulator needs to consider shared instruction cache misses encountered during the simulation of execution of thread T due to competing instruction accesses by other threads from shared instruction cache. The code regions in other threads that compete for shared instruction cache are Barrier Synchronization Processes, if the code regions along with the instruction under access in T are engaged in barrier synchronization. The code regions may be synchronizing processes(tasks) in two or more threads using calls to wait() and notify(). The code regions along with the instruction under access in thread T from shared instruction cache may be simply parallel processes without being engaged in any any form of synchronization. The Control Flow Graphs along with partial order information of the input multithreaded program are transformed into Hoare's CSP from which a Message Sequence Chart is visualized. Competing processes for each instruction in a thread are computed by the Interference Partitioning algorithm and are fed as input to the extended simulator. The Interference Partitioning Algorithm aids the simulator determine an abstract set of competing accesses to shared instruction cache as an instruction in thread T is accessed. The simulator can decide whether an access is a shared instruction cache miss based on the abstract set of competing accesses. A simulator that does not use Interference Partitioning algorithm handling barrier synchronization processes can only consider set of competing accesses to be the entire code regions of competing threads. On the contrary, our simulator based on Interference Partitioning Algorithm uses a much more precise set of competing accesses to shared instruction cache.

A simplified version of the typical multi-core architecture is assumed where each core has a small private $L1$

cache and comparatively larger $L2$ instruction cache, shared by all the cores. The access latency of shared $L2$ instruction cache is higher than that of $L1$ cache. The execution time of a multithreaded program is increased by the impact of the interfering shared cache accesses running on other cores. Performing cache analysis for a multithreaded program on a multicore architecture statically is a non-trivial task. It is essential for a real-time embedded application to obtain tighter *WCET* estimates precise analysis of latency due to accesses to shared cache. To evaluate the performance of existing approaches and IP algorithm, a few parameters are proposed and considered for analysis. The proposed parameters are

- Number of conflicts
- Conflict ratio
- Overestimation ratio
- Precision Improvement in number of conflicts
- Precision Improvement in WCET

Number of Conflicts

Fig. 4. shows number of conflicts of benchmark program for both IP algorithm and existing approaches. It is evident that number of conflicts in IP algorithm is always lesser than or equal to number of conflicts in the existing approaches due to reduced minimal subset of conflicting region. This leads to more precise latency computation for an instruction accessing shared $L2$ instruction cache memory.

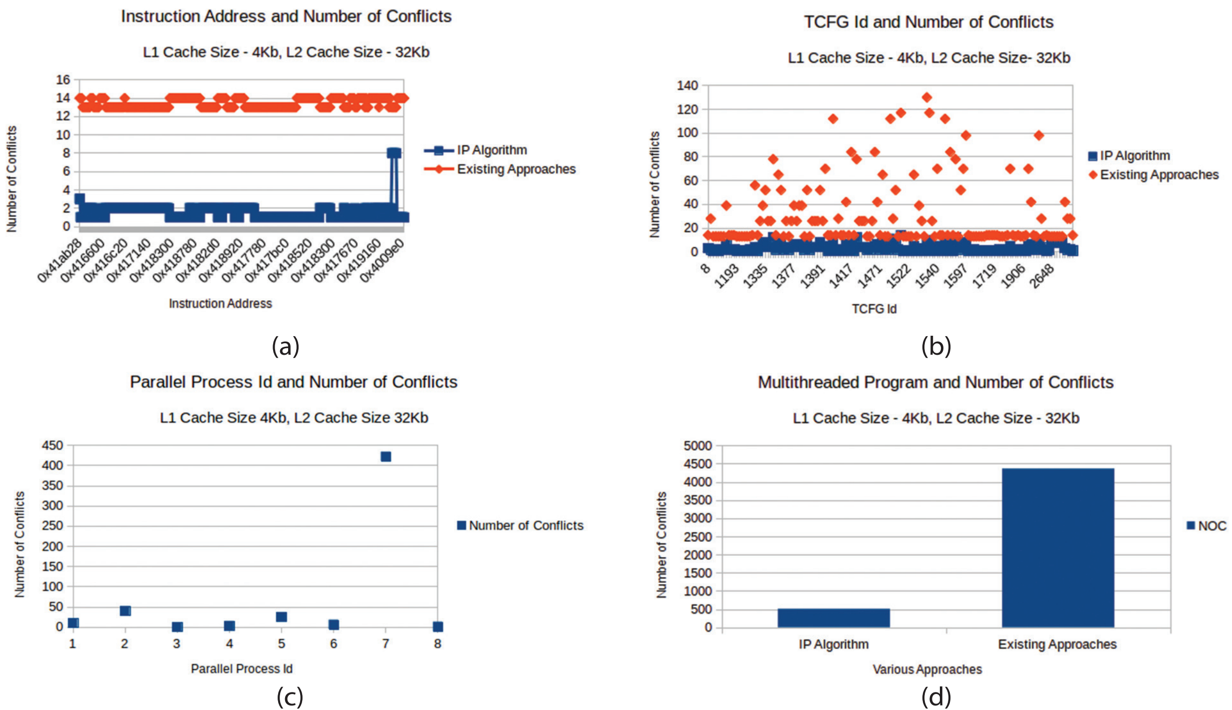
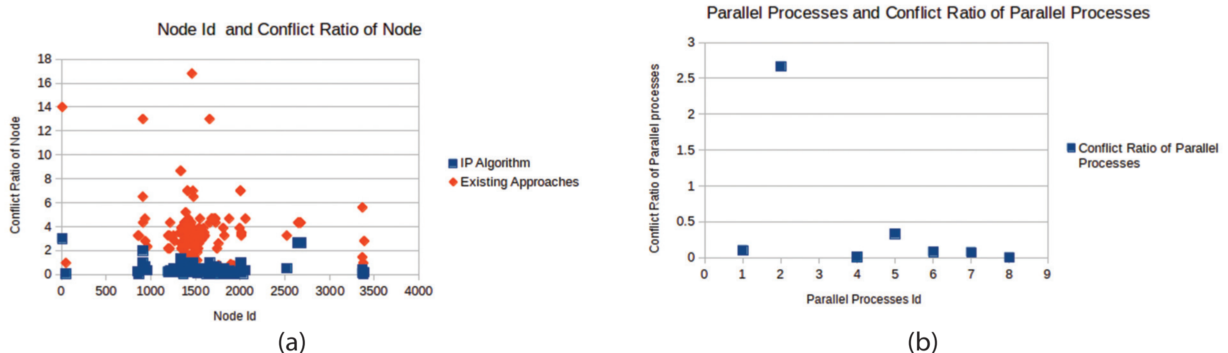
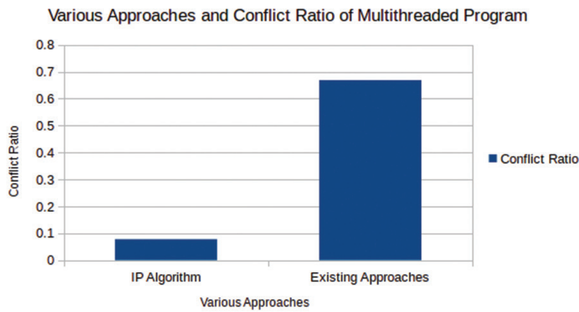


Fig. 4. Number of conflicts of benchmark program; a) of each instruction, b) of each node, c) for each Parallel Process, d) for various Approaches

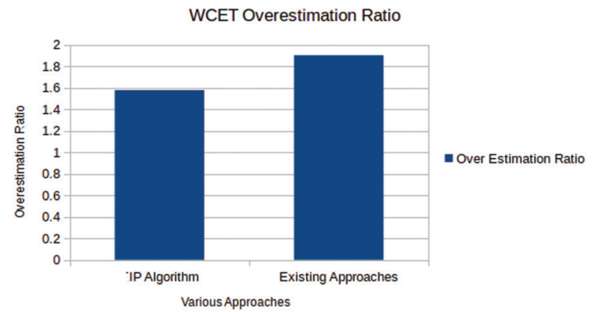
Conflict Ratio

Consider that an instruction I in node n of barrier synchronization parallel process BSP_{m_i} in T_i can have a maximum of x conflicts from interacting thread T_j for IP algorithm and let it be y for existing approaches and it is proved experimentally that $x<=y$. As shown in Fig. 5.b., the conflict ratio of BSP_{2_1} is slightly higher than conflict ratio of other parallel process in T_i, this is due to the fact that the conflicts for BSP_{2_1} are from parallel process having long calculation sequence and more number of branching statements.





(c)



(d)

Fig. 5. Conflict Ratio of multithreaded program; a) of a node, b) of a parallel process, c) of a multithreaded program, d) *WCET* of overestimation ratio

Overestimation Ratio of WCET

The main reason for reduction in overestimation ratio is due to impact of IP algorithm on architectural parameters of cache memory. IP algorithm reduces the number of conflicts that leads to a significant reduction in number of shared $L2$ instruction cache misses. Compulsory misses remain misses even with an infinite cache memory and possible way to reduce compulsory misses is by larger block size, but larger block size increases conflict misses due to fewer cache lines/blocks.

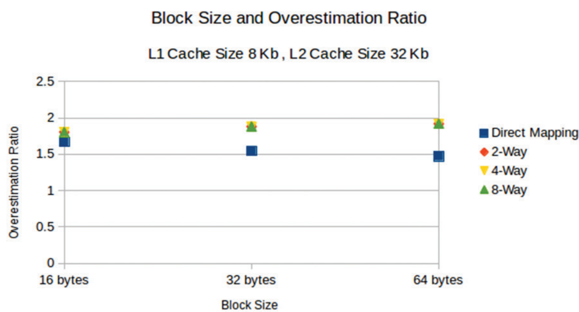


Fig. 6 .a. Block Size and Overestimation Ratio

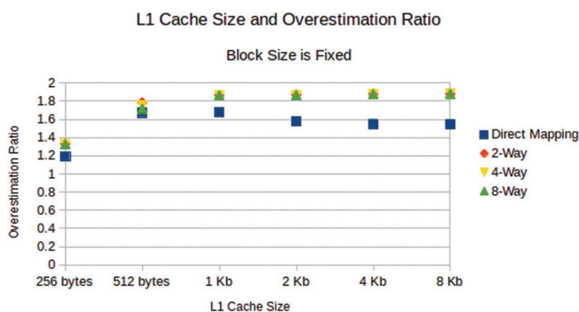


Fig. 6. b. Associativity and overestimation ratio

Fig. 6. Impact of block size and associativity on overestimation ratio

Fig. 6.a. shows the impact of various block sizes on overestimation ratio of *WCET*. One possible way to reduce conflict Misses is to have n-way set associative mapping. In n-way where $n>1$, set associative mapping cache memory, each set has n cache blocks so there are less chances of conflict between two addresses mapped to same cache set. It is evident from Fig. 6.b. that for n-way set associative mapping where $n>1$, overestimation ratio is same. It is also observed that smaller block sizes do not take maximum advantage of spatial locality that results in a greater number of compulsory misses as shown in Fig. 6. a.

Precision Improvement in Number of conflicts and WCET

Though there is a huge precision improvement up to 90 % in reduction in number of conflicts, the precision improvement in *WCET* of a multithreaded program is only up to 20% which is still significant but not commensurate with the former. The reason for the same is discussed in this subsection. Significant improvements are observed when barrier synchronization parallel process size is considerably greater than that of $L1$ cache size. This is because under the stated condition, interferences to shared instruction cache from competing processes are significantly less than those from interacting threads. This is a direct consequence of static identification of barrier synchronization parallel processes in interacting threads. If the size of a barrier synchronization parallel process in a thread is similar to the size of L1 instruction cache, then the need to use shared instruction cache may be quite less for the execution of the barrier synchronization parallel process in the thread. *WCET* precision improvement varies based on cache architectural parameters and benchmark characteristics. In Table 4, a few more benchmark results are shown by varying $L1$ cache size. Greater the number of threads, higher the number of conflicts, causing more cache misses, resulting in an increase in *WCET* estimate with greater imprecision. In contrast, the increase in *WCET* estimate using IP algorithm remains smaller by a fraction when compared to existing approaches. For the benchmarks when run on for 4-core architecture, IP algorithm gave lower *WCET* estimate over existing approaches, with the average precision improvement of 10%. It is noticed that, as the degree of parallelism in threads increases, there is reduction in percentage of precision improvement.

Table 4. Precision Improvement in Number of conflicts and WCET

Test Cases	Benchmarks Characteristics	$L1$ cache Size	Precision Improvement in Number of Conflicts (%)	WCET (Precision Improvement %)
TC1	Inner loop dependent on outer loop, Array and Matrix calculation	256 bytes	65.89%	19.58%
		512 bytes	60.1%	16.4%
		1 KB	55.8%	15.7%
TC2	Input dependent loops, Nested IF statement, Long calculation sequence, Automatically generated code	256 bytes	91.38%	21.8718%
		512 bytes	88.25%	18.1%
		1 KB	80.1%	16.8718%
TC3	Input dependent loops, Automatically generated code	256 bytes	92.69%	22.20%
		512 bytes	88.6%	19.7%
		1 KB	82.4%	16.8%
TC4	Multiple calls to same function, Nested Function calls	256 bytes	88.57%	24.6%
		512 bytes	83.9%	21.5%
		1 KB	78.2%	20.12%

4. CONCLUSION

Worst Case Execution Time Analysis of real-time embedded applications is a challenging task. In this paper, Interference partitioning (IP) algorithm is extended to obtain minimal safe subset of interferences from interacting threads using barrier synchronization primitives. Computation task specific synchronization inside barrier synchronization processes is also identified by IP algorithm. Investigation of the effectiveness of the extended interference partitioning algorithm on benchmark programs adapted from Malardalen benchmark suite is performed. Parameters such as Number of conflicts, Conflict ratio, Overestimation ratio, Precision Improvement in Number of conflicts and Precision Improvement in WCET are proposed for performance evaluations. There is a huge precision improvement upto 90 % in reduction in number of conflicts and the precision improvement in WCET is upto 20% due to IP algorithm.

5. REFERENCES:

[1] P. P. P. Dharishini, P. V. R. Murthy, “Static Analyzer for Computing WCET of Multithreaded Programs using Hoare's CSP”, Proceedings of the 15th Innovations in Software Engineering Conference, February 2022, pp. 1-12.

[2] C. A. R. Hoare, “Communicating sequential processes”, Proceedings of the Communications of the ACM, Vol. 21, No. 8, 1978, pp. 666-677.

[3] S. Schuster, P. Wägemann, P. Ulbrich, W. Schröder-Preikschat, “Annotate once–analyze anywhere: context-aware WCET analysis by user-defined abstractions”, Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, 2021, pp. 54-66.

[4] B. Rouxel, S. Derrien, I. Puaut, “Tightening contention delays while scheduling parallel applications on multi-core architectures”, ACM Transactions on Embedded Computing Systems, Vol. 16, No. 5s, 2017, pp.1-20.

[5] A. Alhammad, R. Pellizzoni, “Time-predictable execution of multithreaded applications on multicore systems”, Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, 24-28 March 2014, pp. 1-6.

[6] K. Nagar, Y. N. Srikant, “Precise shared cache analysis using optimal interference placement”, Proceedings of the IEEE 19th Real-Time and Embedded Technology and Applications Symposium, Berlin, Germany, 15-17 April 2014, pp. 125-134.

[7] T. Kelter, P. Marwedel, “Parallelism analysis: Precise WCET values for complex multi-core systems”, Science of Computer Programming, Vol. 133, 2017, pp. 175-193.

[8] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, V. Suhendra, “Timing analysis of concurrent programs running on shared cache multi-cores”, Real-Time Systems, Vol. 48, No. 6, 2012, pp. 638-680.

[9] H. Ozaktas, C. Rochange, P. Sainrat, “Automatic WCET analysis of real-time parallel applications”, Proceedings of the 13th International Workshop on Worst-Case Execution Time Analysis. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[10] R. Wilhelm et al. "The worst-case execution-time problem—overview of methods and survey of tools", ACM Transactions on Embedded Computing Systems, Vol. 7, No. 3, 2008, pp. 1-53.

[11] D. Potop-Butucaru, I. Puaut, "Integrated worst-case execution time estimation of multicore applications", Proceedings of the 13th international workshop on worst-case execution time analysis, 2013.

[12] T. Carle, H. Cassé, "Reducing timing interferences in real-time applications running on multicore architectures", Proceedings of the 18th International Workshop on Worst-Case Execution Time Analysis, 2018, pp. 1-11.

[13] D. Hardy, T. Piquet, I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches", Proceedings of the 30th IEEE Real-Time Systems Symposium, Washington, DC, USA, 1-4 December 2009.

[14] A.Gustavsson, A. Ermedahl, B. Lisper, P. Pettersson, "Towards WCET analysis of multicore architectures using UPPAAL", Proceedings of the 10th international workshop on worst-case execution time analysis, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[15] D. Casini, A. Biondi, G. Buttazzo, "Analyzing parallel real-time tasks implemented with thread pools", In Proceedings of the 56th Annual Design Automation Conference, Las Vegas, NV, USA, June 2019, pp. 1-6.

[16] P. P. P. Dharishini, P. V. R. Murthy, "Precise Shared Instruction Cache Analysis to Estimate WCET of Multithreaded Programs", Proceedings of the IEEE 18th India Council International Conference, Guwahati, India, December 2021, pp. 1-7.

[17] G. Coulouris, J. Dollimore, T. Kindberg, G. Blair, "Indirect Communication", Distributed systems: Concepts and Design, Fifth Edition, Addison-Wesley, 2011.

[18] J. Gustafsson, A. Betts, A. Ermedahl, B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future", Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik,2010.

[19] K. S. Namjoshi, "Are concurrent programs that are easier to write also easier to check", Proceedings of the Workshop on Exploiting Concurrency Efficiently and Correctly, 2008.

[20] H. Theiling, C. Ferdinand, R. Wilhelm, "Fast and precise WCET prediction by separated cache and path analyses", Real-Time Systems, Vol. 18, No. 2, 2000, pp. 157-179.

[21] D. Hardy, I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches", In Proceedings of the 29th IEEE Real-Time Systems Symposium, Barcelona, Spain, 30 November - 3 December 2008, pp. 456–466.

[22] M. Lv, N. Guan, J. Reineke, R. Wilhelm, W. Yi, "A survey on static cache analysis for real-time systems", Leibniz Transactions on Embedded Systems, Vol. 3, No. 1, 2016.

[23] T. Austin, E. Larson, D. Ernst, "SimpleScalar: an infrastructure for computer system modeling", Computer, Vol. 35, No. 2, 2002, pp. 59-67.

[24] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, H. Falk, "A unified WCET analysis framework for multicore platforms", ACM Transactions on Embedded Computing Systems, Vol. 13, No. 4s, 2014.

[25] X. Li, Y. Liang, T. Mitra, A. Roychoudhury, "Chronos: A timing analyzer for embedded 72 software", Science of Computer Programming, Vol. 69, No. 1, 2007, pp. 56-67.