Indonesian Journal of Electrical Engineering and Computer Science
Vol. 29, No. 2, February 2023, pp. 990~1005
ISSN: 2502-4752, DOI: 10.11591/ijeecs.v29.i2.pp990-1005

Analysis of benchmark program results of worst case execution time for multithreaded programs

Padma Priya Dharishini Paraman[1], Prakriya V. Ramana Murthy[2]
[1]Department of Computer and Software Engineering, Ramaiah University of Applied Sciences, Bangalore, India
[2]Department of Artifical Intelligence and Data Science, Nitte Meenakshi Institute of Technology, Bangalore, India

Article Info

Article history:

Received Jul 21, 2022
Revised Oct 13, 2022
Accepted Oct 21, 2022

Keywords:

Multicore architecture
Multithreaded program
Shared instruction cache
Static analyzer
Worst case execution time

ABSTRACT

Worst case execution time (WCET) estimation by static analyzers is being investigated with keen interest in view of their importance in designing applications for embedded systems that have real- time requirements. Recent work reported on improving precision of estimates of WCET of multithreaded programs, by improving precision of shared instruction cache analysis, shows significant improvement in WCET estimates. An abstraction of a multithreaded program as Hoare's communicating sequential processes (CSP) specification program is realized to enable higher precision in micro-architectural modelling unit of WCET analyzer of multithreaded programs. A thread is viewed as a composition of CSP. The WCET of a thread may be viewed as dependent on WCET of processes in a thread and in turn WCET of each process is the WCET of the sub-graph of basic block nodes in the process. Corresponding CSP in interacting threads, based on calls to synchronization primitives wait and notify, generate shared cache interferences to the process in a thread whose WCET is being estimated by the analyzer. A detailed study of how partitioning of a thread into processes yields higher reduction in WCET is performed on benchmark programs. Furthermore, which processes in a thread yield higher reduction in WCET is performed.

Corresponding Author:

Padma Priya Dharishini Paraman
Department of Computer and Software Engineering, Ramaiah University of Applied Sciences
Bangalore, Karnataka, India
Email: padmapriya.cs.et@msruas.ac.in

1. INTRODUCTION

Multicore processors started dominating the domain of general-purpose computing and embedded systems as single core processors reached their power limit. Multicore processors are the key component for the design of embedded systems because of their high performance, low cost and low power consumption. In real time computing, the timing requirements of programs need to be satisfied in addition to functional requirements. Therefore, real time systems must verify their timing requirements using verification or static timing analysis tools [1]. A static timing analysis tool uses an abstract model of the program to compute safe worst case execution time (WCET). The estimated WCET by static timing analysis tool shall be greater than or equal to actual WCET to ensure safety, by considering program flow and micro-architectural features such as shared cache, shared bus and branch prediction [1]. At the same time, improving the precision of statically estimated WCET is a challenge as the best possible approximation of dynamic state of interactions in architecture have to be approximated by simulator. Higher precision keeps the estimated WCET as close as possible to the actual WCET statically.

WCET analysis for sequential programs is well-studied [1]-[3] and it is complex to determine WCET of a given thread due to dynamic sharing of resources (shared cache, shared bus) between threads running on the cores. Threads communicate through shared memory and synchronize actions on shared data using critical sections. To estimate WCET, an abstract view of the multithreaded program having calls to synchronization primitives is required. A multithreaded program is seen as sequential computations that communicate with each other using calls to synchronization primitives. This is realized by transforming a multithreaded program into an equivalent Hoare's communicating sequential processes (CSP) specification program [4]. The main differences between WCET analysis of a sequential program and a multithreaded program are in accounting for time estimates of shared resources by threads (or cores) such as shared instruction cache.

Execution of a program task or thread on a core may encounter latency, for example, due to interferences resulting in removal of the required bytes of memory, from shared instruction cache, being accessed by a competing task or thread on another core. The current state of the art [3] indicates that the interferences or conflicts for an instruction in a task running on a core is from the entire program region of tasks running on other cores. A challenge in precise WCET estimate of execution time of an instruction in a thread, statically, while using shared instruction cache, is to model, as precisely as possible, the set of interferences from other threads (or cores). Li *et al.* [5], a timing analysis of concurrent programs is proposed to derive partial order for tasks that is based on send and receive messages and the paper does not discuss about WCET of multithreaded programs. A progressive refinement of the cost of execution of tasks, by iterating and identifying interferences between tasks for shared cache analysis, is employed [5]. In contrast, [6] first applies a novel interference partitioning algorithm that partitions range of interferences from entire thread into sub-ranges or partitions of interferences to improve the precision in worst case latency computation in accessing an instruction by a thread.

Potop-Butucaru and Puaut [7], during micro-architecture modelling phase communication edges between threads are introduced but in [8] the program flow analysis phase creates an abstraction of a multithreaded program as CSP specification, identifies the communication edges among threads, and collects dependency information. Touzeau *et al.* [9], memory accesses are classified as always hit (AH), always miss (AM) and unclassified (U). The developed model checker refines the unclassified memory blocks as AH or AM. Touzeau *et al.* [9] assumes that every instruction requires the same number of memory bytes, which is an unrealistic assumption. Ozaktas *et al.* [10], WCET of a parallel program is analyzed. The execution time of the program is the execution time of the longest running thread. Ozaktas *et al.* [10] considers a simple architecture, in which execution time of each instruction is a single clock cycle and for memory access, extra latency is configured and the issues in shared cache analysis are not addressed. Kelter and Marwedel [11], the problem addressed is that in multi-core systems, certain interleavings can never occur by properly analyzing shared resource accesses by tasks. This information can make WCET analysis of tasks precise and feasible. Carle and Cassé [12] extracts the instructions that can potentially cause or suffer from timing interferences. The extracted instruction separates the real time tasks into a sequence of time intervals. An integer linear programming (ILP) solver schedules the sequence of time intervals to minimize the make span time of the program. Worst case interference placement (WCIP) approach discussed in [13] results in reduction in the number of interferences because the effect of same interferences is not considered for all the shared cache hits. Therefore, a larger number of shared cache accesses are classified as hits. It is reported that ILP based approach to interference placement does not scale well for larger programs. Puaut *et al.* [14] compiler optimization that results in lower WCET is explored.

Dharishini and Murthy [6] and Dharishini and Murthy [8], an approach to reduce the set of interferences from an interacting thread during the execution of an instruction in a given thread is proposed. Parallel processes and competing processes in threads of program are identified based on the partial order information derived between interacting threads. The reduction, of interferences to shared instruction cache, is to a smaller group of instructions in a thread, termed a parallel process. The multicore chronos simulator is extended to incorporate parallel processes and competing processes enabling reduction in conflicts or interferences to shared instruction cache resulting in more precise WCET estimates. 20%-30% reduction in WCET estimates is observed using the implemented static analyzer. However, in Dharishini and Murthy [6] and [8], a detailed study of the simulation of benchmark programs is not carried out to pinpoint the processes and in particular basic blocks in sub-graphs of processes that generate interferences to shared instruction cache leading to eviction of instruction being accessed from the shared instruction cache by a thread. An attacking process (or set of such processes) that is/are larger than the current process(es) in the currently active thread evict(s) the currently required instruction from shared instruction cache with a higher probability. This is is experimentally observed during the simulation of benchmark programs. Another contribution is that in contrast with the work reported in [6] and [8] which only report simulation results on MPMD programs, current work performs transformation of single program multiple data (SPMD) multithreaded programs into Hoare’s CSP and runs simulation results on SPMD benchmark programs. The results are generalized for different types of applications: fork/join, SPMD and multiple program multiple data stream (MPMD).

The main contributions of this paper are:

– Characterization of benchmark programs is performed in terms of the conditions under which our method results in a shared instruction cache hit whereas existing methods such as in [3] encounter a shared instruction miss.
– The WCET of a thread is viewed as dependent on WCET of processes in a thread and in turn WCET of each process is the WCET of the sub-graph of basic block nodes in the process. This view enables us to study in this paper not only macroscopic differences in WCET estimates at the level of a thread between our approach and the approach in [3] but also the microscopic differences in WCET estimates or shared instruction cache misses at different levels of granularity, process, basic block and finally at the level of an individual instruction. The study indicates that in our method the WCET of any arbitrary instruction I in a thread is less than or equal to that in the method used in [3]. This is not only observed in simulation of benchmark programs but also is proven using a theoretical argument.
– An algorithm for transformation of a multithreaded program to Hoare’s CSP is designed to enable identification of parallel processes that compete with each other to access shared instruction cache.
– To show, WCET of a Thread computed using interference partition (IP) algorithm is always lesser than or equal to WCET of a Thread computed using Hardy, Piquet and Puaut approach [3], assuming that worst case execution time contribution from all other architectural units other than shared instruction cache are equal in both the approaches.

2. OVERVIEW OF STATIC WCET ANALYZER FOR MULTITHREADED PROGRAMS

The main goal of a static analyzer [8] that estimates the WCET of a multithreaded program is to analyze the input program and to perform required transformations so that statically measured execution time is as close as possible to the actual execution time of the program, when actually run. A static analyzer, shown in Figure 1 that works for different programming languages such as Java threads or POSIX threads is designed and implemented. Dharishini and Murthy [8], “Each thread (T_i) in a multithreaded program is viewed as a composition of sequential and parallel processes, communicating with processes in other threads and if there are n threads in a multithreaded program, n threads are viewed as n communicating sequential processes (CSP) [4]”.

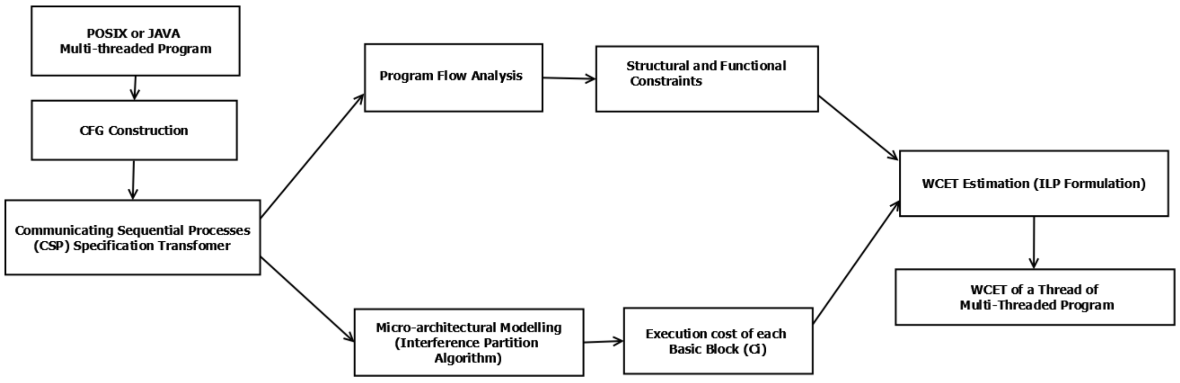

POSIX or JAVA Multi-threaded Program
CFG Construction
Communicating Sequential Processes (CSP) Specification Transfomer
Program Flow Analysis
Structural and Functional Constraints
WCET Estimation (ILP Formulation)
WCET of a Thread of Multi-Threaded Program
Micro-architectural Modelling (Interference Partition Algorithm)
Execution cost of each Basic Block (Ci)

Figure 1. Overview of static WCET analyzer of multithreaded program

3. TRANSFORMATION OF MULTITHREADED PROGRAM TO COMMUNICATING SEQUENTIAL PROCESS SPECIFICATION

An appropriate abstract view of the multithreaded program that facilitates the static analyzer that performs WCET computation with all the information required about the order of computations in threads, concurrency or parallel execution of computations in thread is created. The abstraction of the program is best provided by viewing the multithreaded program as an equivalent Hoare’s CSP specification [6]. Each thread (T_k) in a multithreaded program is viewed as a composition of parallel and sequential processes, communicating with parallel or sequential processes in other threads. The POSIX synchronization/ communication calls or barrier synchronization primitives between threads are transformed into CSP calls to receive ()/send () or equivalent synchronization primitives. Intra thread partial order information that is derived from a multithreaded program is based on the happens before relation [15]. The multithreaded program along with CFG of each thread is provided as input to the Algorithm 1 that emits inter-thread partial order information.

Algorithm 1. To emit partial order information from input multithreaded program
Input: Control flow graph (CFG) of each thread
Output: Set of tuples indicating happens before relation between intra-thread and inter-thread synchronization nodes (for each thread)

– Step 1: Let n_1, n_2, n_3, .. n_k be the sequence of nodes then, the partial order information is $n_1 < n_2 < n_3 \ldots < n_k$. Partial order information is essentially based on happen-before relation between nodes in the CFG of a multithreaded program. Basic block nodes containing instructions are viewed as events of execution of nodes leading to corresponding CSP process transitions.
– Step 2: Based on the equivalent synchronization calls to send () and receive (), inter-thread partial order information is derived from multithreaded program. Partial order information emitted is receive node in a thread < corresponding send node in the interacting thread. CSP specification transformer generates CSP processes of the multithreaded program that identifies parallel processes [8] and synchronized parallel processes.

Algorithm 2, transforms the multithreaded program into communicating sequential process (CSP) program specification equivalent to input multithreaded program based on the partial order information emitted by Algorithm 1.

Algorithm 2: To transform multithreaded program to communicating sequential processes
Input: Partial order information of multithreaded program and CFG of each thread of multithreaded program
Output: Communicating Sequential Process (CSP) program specification equivalent to input multithreaded program
A start process (P_0) is assumed at the start node of the CFG of each thread. The start node is denoted as P_0. The transformation rules are:
Rule1:
If the next node n_i is a computation node (i.e. an event or basic block node), neither a synchronization call node nor a decision node, in that case
P_i <n_i> -> P_{i+1} (P_i accepts basic block node n_i as an event transitioning to process P_{i+1} on its execution)
Rule 2:
If the next node n_i is a conditional node, in that case
P (if E_c then E_t else E_f) -> P E_t, if E_c
P (if E_c then E_t else E_f) -> P E_f, if $!E_c$
Rule 3:
If the node is receive () (Inter- thread communication primitive) then the transformation rule is
P_i <receive> -> $P_{receive}$
Similarly, If the node is send () (Inter- thread communication primitive) then the transformation rule is
P_i <send> -> P_{send}
Similarly, If the node is barrier synchronization primitive, then the transformation rule is
P_i <barrier node $_i$> -> $P_{\text{barrier reached}}$ where <barrier node $_i$ > is the node in P_i just preceding the barrier.
End

Figure 2 shows the general structure of a SPMD multithreaded program (taken from Malardalen benchmarks [16]) having medium loop size and accessing large amount of data. Each thread initializes its own portion of data and performs computation on the initialized data and waits for all the threads to complete the computation on the initialized data. Threads communicate using send () and receive () communication primitives or through barrier synchronization. For the SPMD program in Figure 2(a), its corresponding Message sequence chart (MSC) and CSP representation are shown in Figure 2(b). Each of the child threads and main thread initializes its own portion of data and performs computation on its initialized data. Threads wait at the barrier for the completion of all other threads. Figure 3 shows control flow graph of initialize function of multithreaded program. Figures 3(a) and (b) show the initialize function and CSP representation of the initialize function where basic block nodes act as events of the CSP transitions.

3.1. Micro-architectural modelling

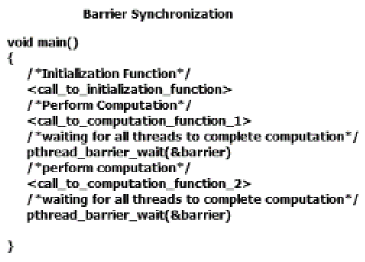
To estimate WCET of CSP processes, various units of the underlying computer components used for their execution, need to be considered. In order to model shared instruction cache, most of the research efforts consider interferences, from all instructions of an interacting thread or from the entire code, while an instruction in a thread is being executed, statically. Dharishini and Murthy [8], “interfering region of code in an interacting (competing) thread is reduced by considering interferences to shared instruction cache only from the relevant maximal region of code in the interacting thread running concurrently with an instruction under execution in a thread and such interfering maximal regions of code in threads are identified by defining synchronized parallel processes, parallel processes, concurrent partitions in threads and list of concurrent partition pairs of a given thread”.

“A synchronized parallel process is defined with respect to two interacting threads. The interaction is based on synchronization using calls to send () and receive (). The instructions in the two threads between a consecutive pair of calls to send () or receive () is a pair of interacting synchronized parallel processes. As special cases, in one or both of the threads, start node of a thread to its first call to send, receive call pair in two interacting threads may be regarded as synchronized parallel process(es). Similarly, send, receive call pair to end of one or both threads may be regarded as synchronized parallel process(es). Parallel processes refer not only to synchronized parallel processes specifically introduced to improve the precision of WCET analysis with reference to shared instruction cache but also other parallel processes that arise based on partial order information in multithreaded programs”.

Figure 4 shows the CSP representation of SPMD program with CSP processes identified. All the CSP processes from n threads have to reach the barrier before they proceed and barrier process keeps track of the arrival of all the processes and once all the processes arrive, it issues the proceed messages [17].

Based on identified synchronized parallel processes and parallel processes, the list of concurrent partition pairs, concurrent partitions and competing processes (P, -) for each parallel processes are constructed as shown in Table 1.

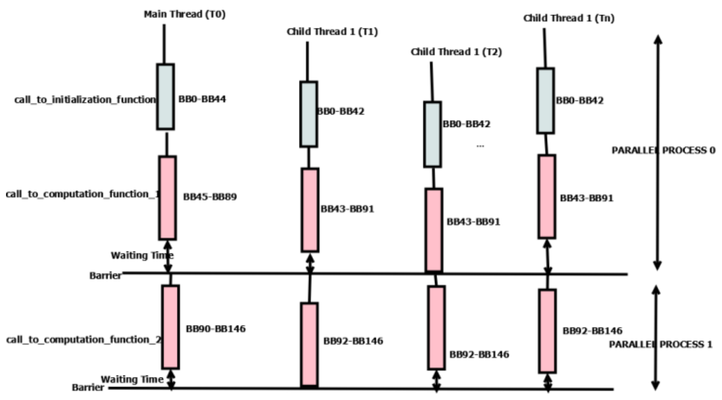
WCET of each basic block is computed by considering the interferences from the corresponding competing processes. The interference partition algorithm computes set of interferences for each instruction in a basic block by considering the interferences from the corresponding competing processes from the interacting threads. The analyzer computes the worst case latency of accessing each instruction in Thread (T) by updating shared instruction cache based on the set of interferences from competing processes. The WCET of each basic block$_i$ (C_i) is computed by including latencies of all instructions in the basic block. To determine worst case latency of accessing instructions in thread T in the presence of shared instruction cache and to compute (C_i), interference partition (IP) algorithm is proposed and implemented. The WCET estimates of each process and thread are computed using the cost-coefficients, computed for basic block nodes, in the integer linear programming formulation to maximize the WCET of the process (or thread).



Barrier Synchronization

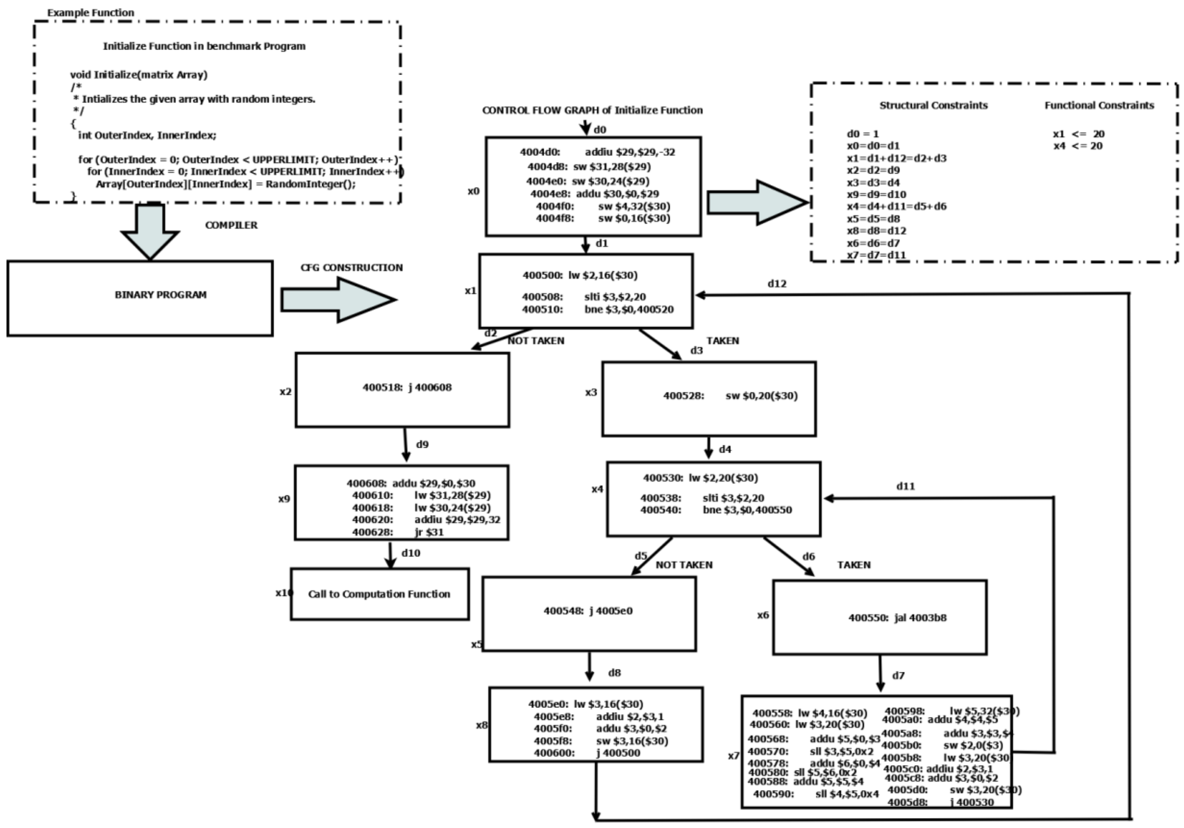
```
void main()
{
    /*Initialization Function*/
    <call_to_initialization_function>
    /*Perform Computation*/
    <call_to_computation_function_1>
    /*waiting for all threads to complete computation*/
    pthread_barrier_wait(&barrier)
    /*perform computation*/
    <call_to_computation_function_2>
    /*waiting for all threads to complete computation*/
    pthread_barrier_wait(&barrier)

}
```

(a)



(b)

Figure 2. SPMD Program, (a) C-code of SPMD program and (b) MSC of SPMD program

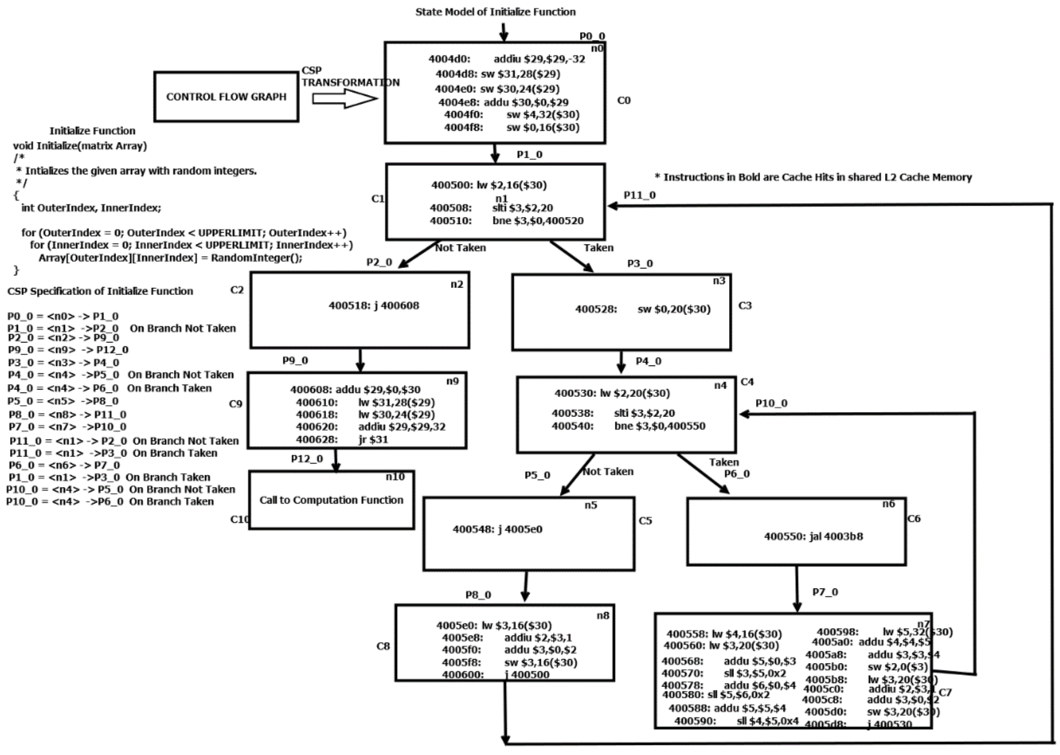


```
void Initialize(matrix Array)
/*
 * Intializes the given array with random integers.
 */
{
  int OuterIndex, InnerIndex;

  for (OuterIndex = 0; OuterIndex < UPPERLIMIT; OuterIndex++)
    for (InnerIndex = 0; InnerIndex < UPPERLIMIT; InnerIndex++)
      Array[OuterIndex][InnerIndex] = RandomInteger();
}
```

```
4004d0:    addiu $29,$29,-32
4004d8:  sw $31,28($29)
4004e0:  sw $30,24($29)
4004e8:  addu $30,$0,$29
4004f0:      sw $4,32($30)
4004f8:      sw $0,16($30)

400500:  lw $2,16($30)
400508:      slti $3,$2,20
400510:      bne $3,$0,400520

400518:  j 400608

400528:      sw $0,20($30)

400608:  addu $29,$0,$30
400610:      lw $31,28($29)
400618:      lw $30,24($29)
400620:      addiu $29,$29,32
400628:      jr $31

400530:  lw $2,20($30)
400538:      slti $3,$2,20
400540:      bne $3,$0,400550

400548:  j 4005e0

400550:  jal 4003b8

4005e0:  lw $3,16($30)
4005e8:      addiu $2,$3,1
4005f0:      addu $3,$0,$2
4005f8:      sw $3,16($30)
400600:      j 400500

400558:  lw $4,16($30)       400598:      lw $5,32($30)
400560:  lw $3,20($30)       4005a0:  addu $4,$4,$5
400568:      addu $5,$0,$3   4005a8:      addu $3,$3,$4
400570:      sll $3,$5,0x2   4005b0:      sw $2,0($3)
400578:      addu $6,$0,$4   4005b8:      lw $3,20($30)
400580:  sll $5,$6,0x2       4005c0:  addiu $2,$3,1
400588:  addu $5,$5,$4       4005c8:  addu $3,$0,$2
400590:      sll $4,$5,0x4   4005d0:      sw $3,20($30)
                             4005d8:      j 400530
```

Structural Constraints	Functional Constraints
d0 = 1	x1 <= 20
x0=d0=d1	x4 <= 20
x1=d1+d12=d2+d3	
x2=d2=d9	
x3=d3=d4	
x9=d9=d10	
x4=d4+d11=d5+d6	
x5=d5=d8	
x8=d8=d12	
x6=d6=d7	
x7=d7=d11	

(a)



Initialize Function

```
void Initialize(matrix Array)
/*
 * Intializes the given array with random integers.
 */
{
  int OuterIndex, InnerIndex;

  for (OuterIndex = 0; OuterIndex < UPPERLIMIT; OuterIndex++)
    for (InnerIndex = 0; InnerIndex < UPPERLIMIT; InnerIndex++)
      Array[OuterIndex][InnerIndex] = RandomInteger();
}
```

CSP Specification of Initialize Function

```
P0_0 = <n0> -> P1_0
P1_0 = <n1>  ->P2_0   On Branch Not Taken
P2_0 = <n2> -> P9_0
P9_0 = <n9>  -> P12_0
P3_0 = <n3> -> P4_0
P4_0 = <n4>  ->P5_0   On Branch Not Taken
P4_0 = <n4> -> P6_0   On Branch Taken
P5_0 = <n5>  ->P8_0
P8_0 = <n8> -> P11_0
P7_0 = <n7>  ->P10_0
P11_0 = <n1> -> P2_0  On Branch Not Taken
P11_0 = <n1>  ->P3_0  On Branch Taken
P6_0 = <n6> -> P7_0
P1_0 = <n1>  ->P3_0  On Branch Taken
P10_0 = <n4> -> P5_0  On Branch Not Taken
P10_0 = <n4>  ->P6_0  On Branch Taken
```

```
4004d0:    addiu $29,$29,-32
4004d8:  sw $31,28($29)
4004e0:  sw $30,24($29)
4004e8:  addu $30,$0,$29
4004f0:      sw $4,32($30)
4004f8:      sw $0,16($30)

400500:  lw $2,16($30)
400508:      slti $3,$2,20
400510:      bne $3,$0,400520

400518:  j 400608

400528:      sw $0,20($30)

400608:  addu $29,$0,$30
400610:      lw $31,28($29)
400618:      lw $30,24($29)
400620:      addiu $29,$29,32
400628:      jr $31

400530:  lw $2,20($30)
400538:      slti $3,$2,20
400540:      bne $3,$0,400550

400548:  j 4005e0

400550:  jal 4003b8

4005e0:  lw $3,16($30)
4005e8:      addiu $2,$3,1
4005f0:      addu $3,$0,$2
4005f8:      sw $3,16($30)
400600:      j 400500

400558:  lw $4,16($30)       400598:      lw $5,32($30)
400560:  lw $3,20($30)       4005a0:  addu $4,$4,$5
400568:      addu $5,$0,$3   4005a8:      addu $3,$3,$4
400570:      sll $3,$5,0x2   4005b0:      sw $2,0($3)
400578:      addu $6,$0,$4   4005b8:      lw $3,20($30)
400580:  sll $5,$6,0x2       4005c0:      addiu $2,$3,1
400588:  addu $5,$5,$4       4005c8:      addu $3,$0,$2
400590:      sll $4,$5,0x4   4005d0:      sw $3,20($30)
                             4005d8:      j 400530
```

(b)

Figure 3. Control flow graphs of threads, (a) generation of structural and functional constraints and (b) state model view of communicating sequential processes specification of a C-function

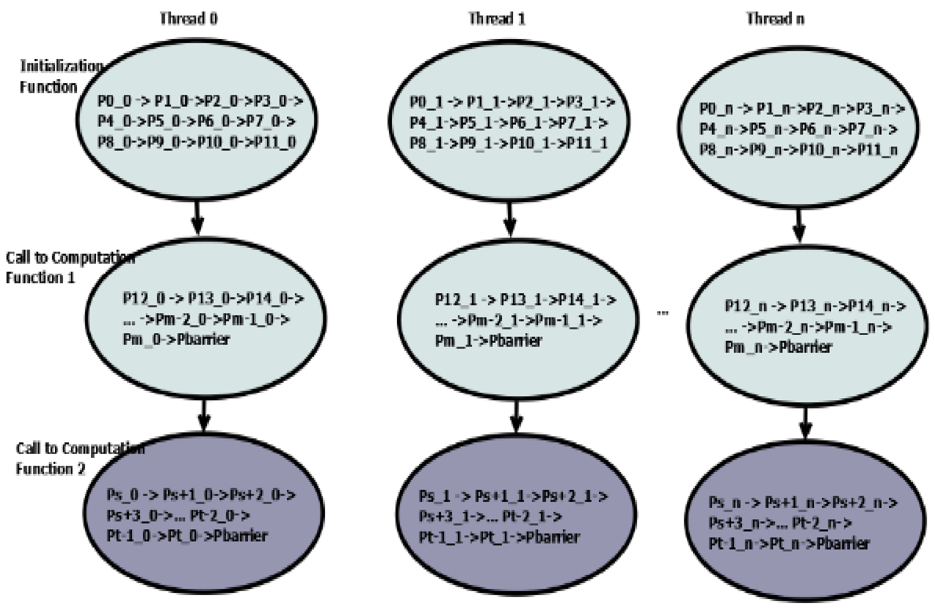

Thread 0
Thread 1
Thread n
Initialization Function
P0_0 -> P1_0->P2_0->P3_0-> P4_0->P5_0->P6_0->P7_0-> P8_0->P9_0->P10_0->P11_0
P0_1 -> P1_1->P2_1->P3_1-> P4_1->P5_1->P6_1->P7_1-> P8_1->P9_1->P10_1->P11_1
P0_n -> P1_n->P2_n->P3_n-> P4_n->P5_n->P6_n->P7_n-> P8_n->P9_n->P10_n->P11_n
Call to Computation Function 1
P12_0 -> P13_0->P14_0-> ... ->Pm-2_0->Pm-1_0-> Pm_0->Pbarrier
P12_1 -> P13_1->P14_1-> ... ->Pm-2_1->Pm-1_1-> Pm_1->Pbarrier
P12_n -> P13_n->P14_n-> ... ->Pm-2_n->Pm-1_n-> Pm_n->Pbarrier
Call to Computation Function 2
Ps_0 -> Ps+1_0->Ps+2_0-> Ps+3_0->... Pt-2_0-> Pt-1_0->Pt_0->Pbarrier
Ps_1 -> Ps+1_1->Ps+2_1-> Ps+3_1->... Pt-2_1-> Pt-1_1->Pt_1->Pbarrier
Ps_n -> Ps+1_n->Ps+2_n-> Ps+3_n->... Pt-2_n-> Pt-1_n->Pt_n->Pbarrier

Figure 4. CSP Representation of SPMD Program with identified parallel processes

Table 1. Competing processes for each identified parallel process

List of concurrent partition pairs (P_{Ti}, P_{Tj})	Concurrent Partition Pairs
List of concurrent partition pairs (P_{T0}, P_{T1})	[(P_{0_0} -> P_{1_0} -> P_{2_0}-> P_{3_0} -> P_{4_0} -> …-> P_{m_0} ->$P_{barrier}$, P_{0_1} -> P_{1_1} -> P_{2_1}-> P_{3_1} -> P_{4_1} ->… ->P_{m_1} ->$P_{barrier}$), (P_{s_0} -> P_{s+1_0} -> P_{s+2_0}-> P_{s+3_0} -> P_{s+4_0} -> …-> P_{t_0}->$P_{barrier}$, (P_{s_1} -> P_{s+1_1} -> P_{s+2_1}-> P_{s+3_1} -> P_{s+4_1} -> … ->P_{t_1}->$P_{barrier}$)]
List of concurrent partition pairs (P_{T0}, P_{Tn})	[(P_{0_0} -> P_{1_0} -> P_{2_0}-> P_{3_0} -> P_{4_0} -> … -> P_{m_0} ->$P_{barrier}$, P_{0_n} -> P_{1_n} -> P_{2_n}-> P_{3_n} -> P_{4_n} ->… -> P_{m_n} ->$P_{barrier}$), (P_{s_0} -> P_{s+1_0} -> P_{s+2_0}-> P_{s+3_0} -> P_{s+4_0} -> … ->P_{t_0}->$P_{barrier}$, (P_{s_n} -> P_{s+1_n} -> P_{s+2_n}-> P_{s+3_n} -> P_{s+4_n} -> …-> P_{t_n}->$P_{barrier}$)]
Concurrent Partitions (P_{T1})	{[(P_{0_0} -> P_{1_0} -> P_{2_0}-> P_{3_0} -> P_{4_0}-> … -> P_{m_0} ->$P_{barrier}$, P_{0_1} -> P_{1_1} -> P_{2_1}-> P_{3_1} -> P_{4_1} ->… -> P_{m_1} ->$P_{barrier}$), (P_{s_0} -> P_{s+1_0} -> P_{s+2_0}-> P_{s+3_0} -> P_{s+4_0} -> … ->P_{t_0}->$P_{barrier}$, (P_{s_1} -> P_{s+1_1} -> P_{s+2_1}-> P_{s+3_1} -> P_{s+4_1} -> … ->P_{t_1}->$P_{barrier}$)], … [(P_{0_0} -> P_{1_0} -> P_{2_0}-> P_{3_0}-> P_{4_0} ->… -> P_{m_0} ->$P_{barrier}$, P_{0_n} -> P_{1_n} -> P_{2_n}-> P_{3_n}-> P_{4_n} ->… P_{m_n} ->$P_{barrier}$), (P_{s_0} -> P_{s+1_0} -> P_{s+2_0}-> P_{s+3_0} -> P_{s+4_0} -> … -> P_{t_0}->$P_{barrier}$, (P_{s_n} -> P_{s+1_n} -> P_{s+2_n}-> P_{s+3_n} -> P_{s+4_n} -> … P_{t_n}->$P_{barrier}$)]}
CompetingProcesses (P_{0_0} -> P_{1_0} -> P_{2_0}-> P_{3_0} -> P_{4_0} ->… ->P_{m_0} ->$P_{barrier}$)	(P_{0_0} -> P_{1_0} -> P_{2_0}-> P_{3_0} -> P_{4_0} ->… ->P_{m_0} ->$P_{barrier}$, P_{0_1} -> P_{1_1} -> P_{2_1}-> P_{3_1} -> P_{4_1} ->… ->P_{m_1} ->$P_{barrier}$) … (P_{0_0} -> P_{1_0} -> P_{2_0}-> P_{3_0} -> P_{4_0} ->… ->P_{m_0} ->$P_{barrier}$, P_{0_n} -> P_{1_n} -> P_{2_n}-> P_{3_n} -> P_{4_n} ->… ->P_{m_n} ->$P_{barrier}$)
CompetingProcesses (P_{s_0} -> P_{s+1_0} -> P_{s+2_0}-> P_{s+3_0} -> P_{s+4_0} -> …-> P_{t_0}->$P_{barrier}$)	(P_{s_0} -> P_{s+1_0} -> P_{s+2_0}-> P_{s+3_0} -> P_{s+4_0} -> … ->P_{t_0}->$P_{barrier}$, (P_{s_1} -> P_{s+1_1} -> P_{s+2_1}-> P_{s+3_1} -> P_{s+4_1} -> … ->P_{t_1}->$P_{barrier}$) … (P_{s_0} -> P_{s+1_0} -> P_{s+2_0}-> P_{s+3_0} -> P_{s+4_0} -> … -> P_{t_0}->$P_{barrier}$, (P_{s_n} -> P_{s+1_n} -> P_{s+2_n}-> P_{s+3_n} -> P_{s+4_n} -> … -> P_{t_n}->$P_{barrier}$)

3.2. Interference partition algorithm

Interference partition (IP) algorithm, a part of micro-architectural modelling statically executes each instruction in a multithreaded program making required measurements such as latency in accessing instructions, in clock cycles. The steps in IP algorithm are as follows:

- 884eIdentify all parallel processes and synchronized parallel processes for all threads (Ti), based on synchronization/communication primitives.
- Construct list of concurrent partition pairs in the order of processes in Thread Ti for all parallel processes.
- Construct set of concurrent partitions where set is union of list of concurrent partition pairs.
- Identify CompetingProcesses(P), set of Concurrent pairs of the form (P,-), where '–' indicates any parallel process that is parallel to P in another thread.

– Associate each instruction in process P with all instructions in each member of CompetingProcesses(P) which can cause shared instruction cache interferences

Interference partition (IP) algorithm, a part of micro-architectural modelling statically executes each instruction in a multithreaded program making required measurements such as latency in accessing instructions, in clock cycles. The IP algorithm considers the concurrent partitions of each thread and keeps track of the parallel processes. Micro-architectural modelling, being aware of competing processes, is able to generate a reduced set of interferences from an interacting thread to an instruction in a thread, in contrast with other approaches reported [1], [18]-[20].

3.3. WCET computation

The program under analysis is represented as CFG, a directed graph whose nodes are basic blocks and an edge connects two basic blocks if the two basic blocks are in sequence or call to function or in case of branches. An ILP variable is associated with each basic block and edges to represent the number of times the basic block and edges are executed. Let x_i be the variable associated with BB_i to represent the execution count of BB_i, and d_k be the variable associated with the edges to BB_i. The execution count of the basic block is equal to the sum of incoming edges to the basic block and sum of the outgoing edges from the basic block [21] as in (1).

$$BB_i: \sum d_k, BB_i = x_i = \sum BB_i, d_k \quad (1)$$

The control flow of multithreaded program is captured as structural constraints and functional constraints and Figure 3(a) shows structural and functional constraints of initialization function of SPMD benchmark. The worst case execution cost of each basic block $node_m$ or $event_m$ (C_m) is the output of the micro-architectural modelling. The WCET of each basic block (C_m) is composed to get WCET of the each parallel processes (sub-graph of CFG) and WCET of parallel processes are composed to obtain overall WCET entire multithreaded program. Given the constant C_m and variable N_m, the execution time of a parallel process P_{k_j} corresponding to thread (T_j) may be expressed as in (2).

$$\text{Worst case Execution Time}_{k\_j} = \Sigma_m C_m * N_m. \quad (2)$$

The ILP variable N_m denotes the number of times $event_m$ in CSP process k is executed and C_m, the worst execution cost or time of each event (basic block) in process k is obtained from the micro-architectural modelling. The worst case execution time of thread j is obtained by composing the computed worst case execution time of all CSP processes. It is also proved theoretically that WCET computed using IP algorithm is always lesser than or equal to WCET computed using Hardy, Piquet and Puaut approach [3] under the assumption that the contribution from other architectural parameters are equal in both approaches.

Lemma 1: WCET of a Thread using IP algorithm<=WCET of a Thread using Hardy, Piquet and Puaut approach, assuming that worst case execution time contribution from all other architectural units other than shared instruction cache are equal in both the approaches. Let MP be a multi-threaded program. WCET(MP) using interference Partitioning algorithm is always less than or equal to WCET(MP) without using interference partitioning algorithm. $WCET(MP)_{\text{using Interference Partitioning algorithm}} <= WCET(MP)_{\text{without using Interference Partitioning algorithm}}$

Proof: to prove that $WCET(MP)_{\text{using Interference Partitioning algorithm}} <= WCET(MP)_{\text{without using Interference Partitioning algorithm,}}$ we prove that WCET(of each thread T) using interference partitioning algorithm is less than or equal to that without using the algorithm. To prove the above for each thread T, we prove that worst case latency (WCL), of accessing an instruction I in a thread T from shared instruction cache, using interference partitioning algorithm is always less then or equal to that without using the interference partitioning algorithm.

That is, WCL (accessing I in T) using interference partitioning algorithm<=WCL (accessing I in T) without using interference partitioning algorithm. For any instruction in a basic block, the set of interferences from competing threads using IP algorithm is a subset of the interferences without using the algorithm.

– Case 1: the currently accessed instruction by thread T is present in shared instruction cache. If IP algorithm is used, the probability of eviction of instruction I from shared cache is smaller than the alternative in which IP algorithm is not used. Therefore, worst case latency in accessing an instruction from a shared instruction cache using IP algorithm will be less than that without using the algorithm.
– Case 2: the currently accessed instruction by thread T is not present in shared instruction cache (not present in L1 cache either). The instruction has to be fetched from RAM both when the IP algorithm is used and when it is not. Therefore, worst case latency in accessing an instruction using IP algorithm will be equal to that without using the algorithm.

- Case 3: The currently accessed instruction by thread T is present in L1 cache itself. Therefore, worst case latency in accessing an instruction using IP algorithm will be equal to that without using the algorithm.
- Considering all the three cases mentioned above, WCL (accessing I in T) using interference partitioning algorithm<=WCL (accessing I in T) without using interference partitioning algorithm.

The above is true for all instructions in a basic block. Assuming less than or equal relation for (worst case) execution time contribution from all other architectural units other than shared instruction cache, WCET of an instruction using IP Algorithm<=WCET of an instruction using Hardy, Piquet and Puaut approach. Sum of worst case latencies of all instructions in a basic block is less than or equal to the sum of worst case latencies of all instructions using Hardy, Piquet and Puaut approach. Assuming less than or equal worst case execution time contribution from all other architectural units, other than shared instruction cache, WCET of a Basic block using IP algorithm<=WCET of a basic block using Hardy, Piquet and Puaut approach. A process in IP algorithm approach is a sub-graph of basic block nodes. The sum of worst case latencies of instructions in a sub-graph corresponding to a process using IP algorithm is always less than or equal to the sum of worst case latencies of the instructions in the corresponding sub-graphs in Hardy, Piquet and Puaut approach. Assuming less than or equal worst case execution time contribution from other architectural units besides shared instruction cache, WCET of a process (sub-graph) using IP algorithm<=WCET of the corresponding sub-graph in Hardy, Piquet and Puaut approach. The process (sub-graph) in the case of IP algorithm is actually the same as the corresponding sub-graph in Hardy, Piquet and Puaut approach. As a consequence, sum of the WCET estimates of sub-graphs corresponding to processes in a thread T using IP algorithm is less than or equal to the sum of WCET estimates of corresponding sub-graphs in Hardy, Piquet and Puaut approach. The above implies that the same inequality holds for the entire thread. Hence the lemma is proven. This is observed experimentally also through simulator runs on benchmark programs.

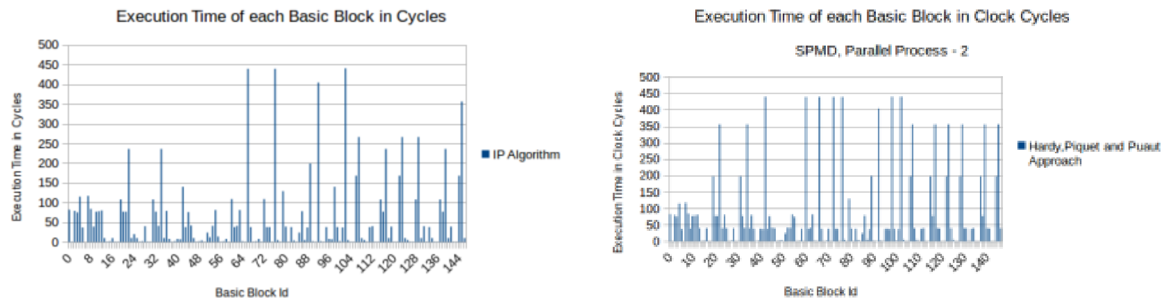
4. RESULTS AND DISCUSSION

Multicore chronos WCET analyzer [22] is used to show empirically that WCET of a thread computed using IP algorithm is always less than or equal to WCET of thread computed using Hardy, Piquet and Puaut approach. Interference partitioning (IP) algorithm is incorporated into multicore chronos tool to simulate shared instruction cache behavior. A number of benchmark programs from Malardalen WCET benchmark suite [16] are executed using the simulator. Our algorithm yields more precise WCET estimates and the approach is validated empirically on benchmark programs. A unique core is assigned to each thread of the benchmark programs. IP algorithm mainly focuses on the impact of shared instruction cache on estimation of WCET and for every shared cache miss caused due to interferences, a fixed miss penalty is assumed [23]. The simulator accounts for instruction execution cycle that includes pipelining and branch prediction as well. Hardy, Piquet and Puaut approach [3] which considers the effect of interferences on all shared cache hits is used to compare the WCET obtained using IP algorithm. The benchmark program reported shows SPMD style parallelism.

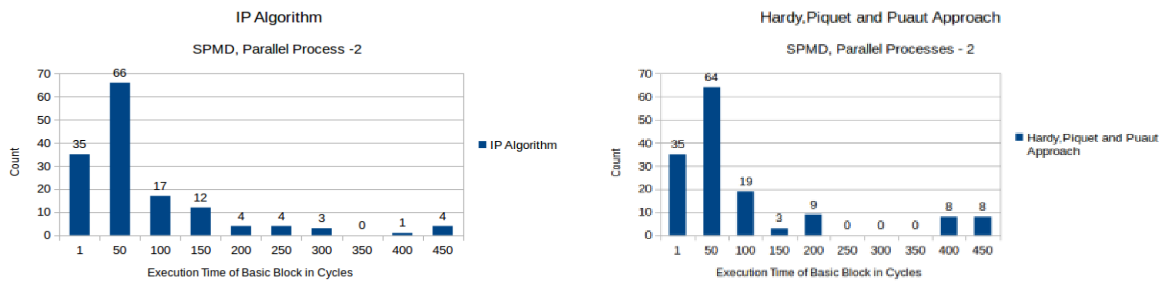
Micro-architectural modelling computes WCET of each basic block$_i$ (C_i) by considering the interferences from the corresponding competing processes. Based on Lemma 1, for any instruction in a basic block, the set of interferences using IP algorithm is a subset of the interferences using Hardy, Piquet and Puaut approach. Therefore, worst case latency in accessing an instruction from a shared instruction cache using IP algorithm is always less than or equal to using Hardy, Piquet and Puaut approach theoretically. The observation is made empirically as discussed below. The worst case latencies of all instructions in basic blocks are composed to form WCET of each basic block in both the approaches as shown in Figure 5. Figure 5(a) shows WCET of basic blocks for both the approaches using IP algorithm and Hardy, Piquet and Puaut method by considering latency while accessing shared instruction cache. It is clear that the estimated worst case execution time of basic blocks are not equal in both the approaches. In fact, it is often found that WCET of a basic block using IP approach is less than that using Hardy, Piquet and Puaut approach. The difference in execution time is mainly due to reduction in latency in accessing instructions using IP algorithm incorporated into the micro-architectural model. Figure 5(b) shows the distribution of the execution time of basic blocks in cycles with both IP algorithm and Hardy, Piquet and Puaut approach [3]. It is evident from Figure 5(b) that the distribution of WCET of basic blocks using IP algorithm and Hardy, Piquet and Puaut approaches [3] are not the same. The reason for the difference is mainly due to reduction in size of interference regions in competing threads, when micro-architectural model employs IP algorithm.

Following Intra-core L2 cache analysis [2], instructions in basic blocks are classified as always hit, always miss and not classified. Always hit instruction of L2 shared instruction cache are affected by the interferences from interacting threads that are running on other cores may lead to shared instruction cache misses that give rise to higher execution time as shown in Figure 6. With IP algorithm, the interferences for an Always Hit instruction in shared L2 cache are from the corresponding competing processes running on other cores but for Hardy, Piquet and Puaut approach interferences are from the entire address space of threads running on other cores.

Figure 6(a) shows the basic blocks with different execution times using interference partition algorithm based and Hardy, Piquet and Puaut approaches. Figure 6(b) shows the number of shared L2 instruction cache hits for basic blocks in Figure 6(a), without considering interferences from the Interacting threads running on other cores. It is clear from Figure 6(a) that the execution time of basic blocks computed using IP algorithm is always less than or equal to the execution time of basic block computed using Hardy, Piquet and Puaut approach and difference in the execution time is mainly due to shared Instruction cache interferences. The basic block 43 encounters cumulative execution time of 440 clock cycles (CC) using Hardy, Piquet and Puaut approach mainly due to shared cache misses following inter-thread shared cache analysis. This is mainly due to larger interference region. When IP algorithm is used, the execution time is 140 CC because of less number of shared cache misses following inter-thread shared cache analysis due to reduced interference region.

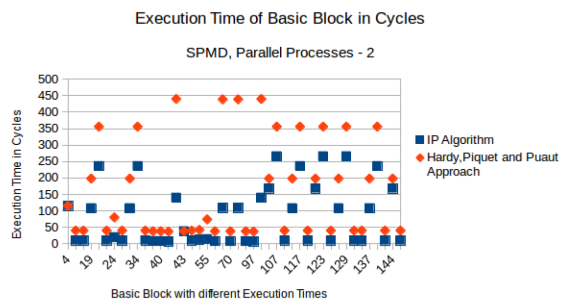


(a)

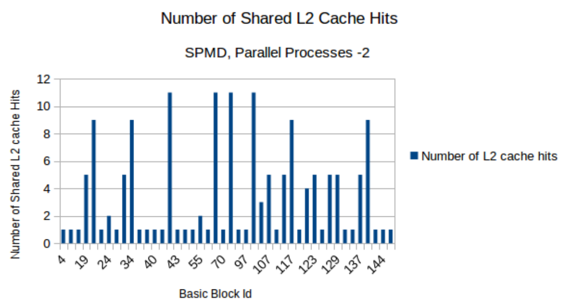


(b)

Figure 5. Execution time of each basic block; (a) using IP algorithm and Hardy, Piquet and Puaut approach and (b) distribution of execution time in cycles using IP algorithm and Hardy, Piquet and Puaut approach



(a)



(b)

Figure 6. Basic blocks; (a) with different execution times and (b) number of L2 cache hits

Assume an instruction (I) that is mapped to the cache line ($line_x$), the interferences for (I) are from the Instructions (I_1, I_2, I_3, .. , I_n) in competing processes that are mapped to the same cache line ($line_x$) . Assume that there are n interfering instructions from interacting threads that are mapped to the cache line ($line_x$). Therefore, the age of instruction (I) in shared L2 instruction cache memory has to be increased by n. If the age of the Instruction (I) after shared instruction cache analysis is less than cache associativity of shared instruction cache, then the instruction is considered as cache hit in shared L2 cache memory. For example, as shown in Table 2, the age of the instruction 400558:lw $4,16($30) in L2 shared instruction cache is 0 and there are 3 interfering instructions from competing processes that are mapped to the same cache line ($Line_{10}$). Then the age of instruction 400558:lw $4,16($30) following shared L2 instruction cache analysis is 3 and access to the instruction remains cache hit in shared L2 cache memory. On the contrary, in Hardy, Piquet and Puaut approach, there are 5 interfering instructions from competing threads that map to the same cache line ($Iine_{10}$). The age of instruction 400558:lw $4,16($30) following shared L2 instruction cache analysis is 5 and the access to the instruction encounters always miss in shared L2 cache.

Table 2. Instructions from competing processes causing interferences for L2 cache hits

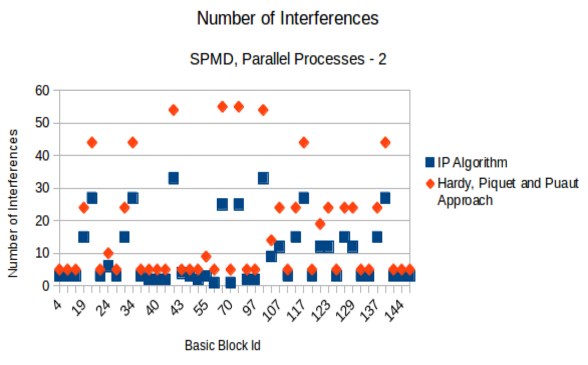
Cache configuration	Age	Interferences from competing processes	
L1:256bytes with 2-way L2: 2Kb with 4-way L2 Cache Line Number		IP Algorithm	Hardy, Piquet and Puaut approach
400558:lw $4,16($30) Line Number :10	0	400340:sw $5,28($30) 400558: lw $4,16($30) 400740: addu $6,$0,$3	400340:sw $5,28($30) 400558: lw $4,16($30) 400740: addu $6,$0,$3 400940: addiu $2,$3,1 400b40: addiu $3,$3,1
400560:lw $3,20($30) Line Number :11	0	400360: lw $4,28($30) 400560: lw $3,20($30) 400760: addu $6,$6,$2	400360: lw $4,28($30) 400560: lw $3,20($30) 400760: addu $6,$6,$2 400960: addu $29,$0,$30 400b60: addu $29,$0,$30
400580: sll $5,$6,0x2 Line Number :12	0	400380: lw $6,32($30) 400580: sll $5,$6,0x2 400780: addu $5,$5,$6	400380: lw $6,32($30) 400580: sll $5,$6,0x2 400780: addu $5,$5,$6 400980: addiu $29,$29,-16 400b80: addiu $29,$29,-16
4005a0: addu $4,$4,$5 Line Number :13	1	4003a0:lw $30,16($29) 4005a0: addu $4,$4,$5 4007a0: sll $7,$8,0x2	4003a0: lw $30,16($29) 4005a0: addu $4,$4,$5 4007a0: sll $7,$8,0x2 4009a0: sw $5,20($30) 400ba0: addu $29,$0,$30
4005c0: addiu $2,$3,1 Line Number :14	1	4005c0: addiu $2,$3,1 4003c0: sw $30,0($29) 4007c0: addu $7,$8,$9	4005c0: addiu $2,$3,1 4003c0: sw $30,0($29) 4007c0: addu $7,$8,$9 4009c0: bne $4,$0,4009d0

Figure 7 shows the interferences for each basic block in the benchmark program. Figure 7(a) shows the average number of interferences for each shared L2 cache hit in the basic block of the multithreaded program. The Average number of interferences $_{\text{Hardy, Piquet and Puaut approach}}$ is calculated as Total Number of $\text{Interferences}_{\text{Hardy, Piquet and Puaut approach}}$ / Total Number of instructions in the basic block. The Average number of $\text{interferences}_{\text{IP Algorithm}}$ for a basic block is calculated as Total Number of $\text{Interferences}_{\text{IP Algorithm}}$/ Total Number of instructions in the basic block. It is evident from Figure 7(b) that the average number of interferences for a basic block is more in the case of Hardy, Piquet and Puaut approach because of larger interference region.

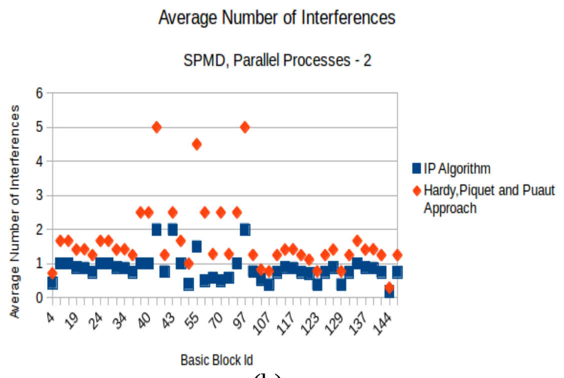
Interferences for any instruction in reference thread is mainly from the corresponding competing processes of interacting threads in the case of IP algorithm, the reason for reduction in number of interferences is that instead of the entire address space of each competing thread, only a smaller portion of the address space for each competing process in an interacting thread generates interferences to shared instruction cache. Interferences for any instruction in reference thread is mainly from the entire address space of each competing thread in the case of Hardy, Piquet and Puaut approach, generating larager number of interferences. Figure 7(a) shows the number of Interferences for both the approaches and 7.b shows the average number of interferences in the case of both the approaches.

For any Always Hit instruction in shared L2 instruction cache, if the sum of number of interferences from the interacting core instructions and age of the instruction is greater or equal to associativity of shared L2 instruction cache, then the instruction become Always Miss after inter core shared instruction cache analysis and the same for the benchmark program is shown in Figure 8. It is evident from Figure 8(a), number of shared L2 cache misses of Hardy, Piquet and Puaut approach is always greater than or equal to the number of shared L2 cache misses of IP algorithm. The miss Penalty associated with each cache miss for accessing next level cache increases the execution time of each basic block of the multithreaded program. Figure 8(b) shows the number of shared L2 instruction cache hits after intercore shared cache analysis.

Figure 9 shows the worst case execution time of the multithreaded program for IP algorithm and Hardy, Piquet and Puaut approaches. The WCET of a multi-threaded program using IP algorithm is always less than or equal to that using Hardy, Piquet and Puaut approach as shown in Figure 9(a). The difference in WCET is mainly due to the relative number of interferences and evictions of instructions from shared instruction cache and the resulting miss penalties. Figure 9(b) shows the simulated WCET computed using simple scalar cycle accurate simulator [24]. It is clear from Figure 9(b) that estimated WCET using IP algorithm is always safe and tight [25]. The precision improvement in WCET is calculated as $\text{WCET}_{\text{Hardy, Piquet and Puaut approach}} - \text{WCET}_{\text{IP Algorithm}} / \text{WCET}_{\text{Hardy, Piquet and Puaut approach}}$. The precision improvement in WCET ranges from 20% -30% for benchmarks having nested loop structure. The main difference in the worst-case execution time of both the approaches is mainly due to shared cache hits for instructions accessed multiple times. The main computation in this SPMD benchmark consists of matrix operations through a nested loop structure. For example, siumaltion of execution of Basic block 19 encounters five shared cache hits (without considering interferences from other cores) and following shared cache analysis in Hardy, Piquet and Puaut approach, all the shared cache hits become misses. Therefore, miss penalty for shared cache misses is added to latency and cumulative worst case latency of instructions in basic block 19 becomes 198 Clock Cycles. In case of IP algorithm, only two shared cache hits become misses, therefore execution cumulative worst case latency of instructions in basic block 19 becomes 108 CC. Since the execution count of basic block 19 is 400 (inside nested loop structure), the reduction of worst case latency by 90 clock cycles causes a huge impact on WCET.

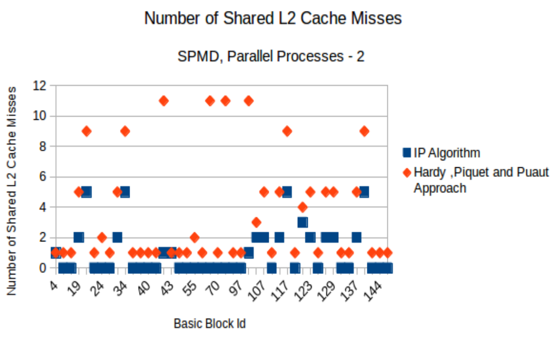


(a)

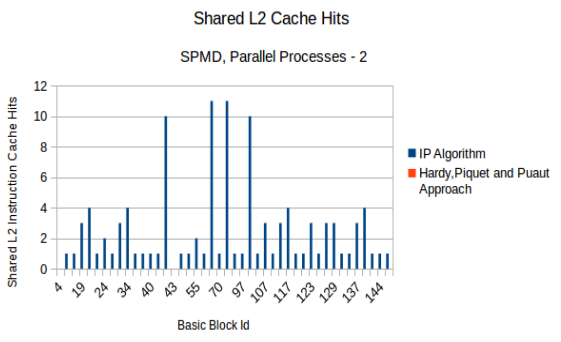


(b)

Figure 7. Total interferences; (a) both approaches and (b) average number of interferences



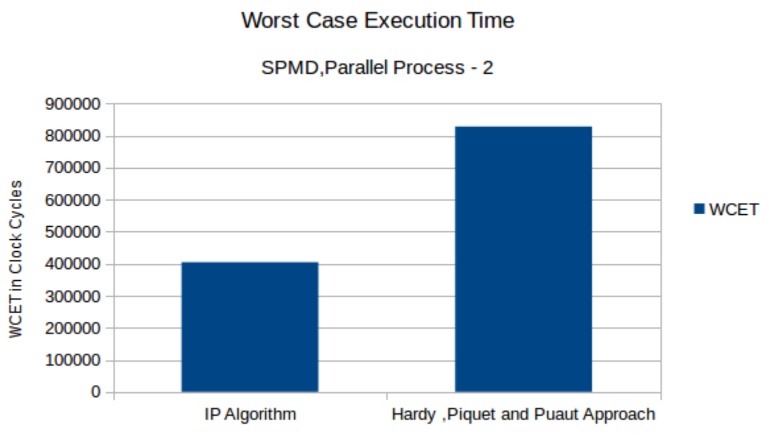
(a)



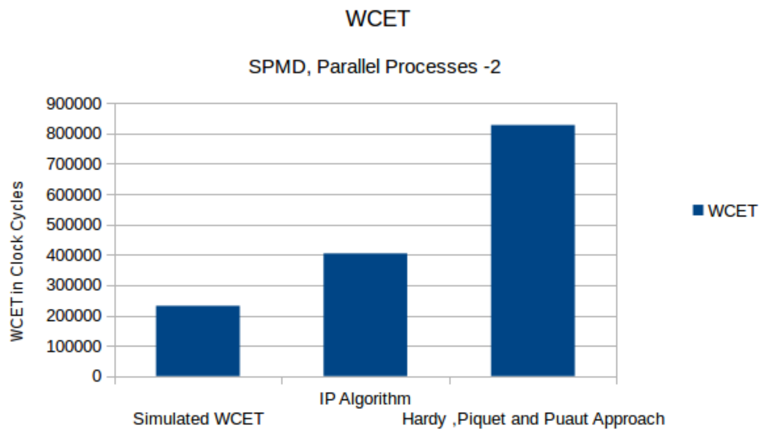
(b)

Figure 8. Cache parameters; (a) number of shared L2 cache misses and (b) number of L2 cache hits

It is also clear that temporal locality [26] plays a major role in reducing WCET of IP algorithm. The effect of other cache parameters such as block size, number of cache lines and associativity on both IP algorithm and Hardy, Piquet and Puaut approach is the same. Based on the above analysis, it is evident that the reduction in WCET of a thread is mainly due to reduction in latency in accessing instructions due to interference partition algorithm. As a consequence, there is considerable reduction in WCET of basic blocks, reduction in WCET of process (sub-graph), and reduction in WCET of thread. The impact of architectural parameters such as number of interferences, number of shared L2 instruction cache misses and number of shared L2 cache hits on parallel processes is shown in Table 3 for both IP algorithm and Hardy, Piquet and Puaut approach. Total number of interferences for any sub-graph in Hardy, Piquet and Puaut approach is always greater than or equal to total number of interferences for any corresponding parallel process in IP algorithm. This is mainly due to reduction in interference region. Similarly, the total number of shared instruction cache misses for any sub-graph$_i$ in Hardy, Piquet and Puaut approach is greater than or equal to the total number of misses for any corresponding parallel process$_i$ in IP algorithm.



(a)



(b)

Figure 9. WCET of benchmarks; (a) estimated WCET for both approaches and (b) simulated and estimated WCET

Table 3. Comparison of IP algorithm and Hardy, Piquet and Puaut approach

Total L2 Hits – 155	IP algorithm			Hardy, Piquet and Puaut approach		
	Thread 0	Parallel process 1	Parallel Process 2	Thread 0	Parallel process 1	Parallel Process 2
Total Number of Interferences	428	368	60	758	619	139
Number of Shared L2 Instruction Cache Misses	45	44	1	155	128	27
Number of Shared L2 Instruction Cache Hits	110	82	28	0	0	0

As L1 cache size increases, capacity and conflict misses of L1 cache decrease leading to reduction in the number of L2 cache accesses which results in improvement of precision in WCET estimate. Table 4 shows the same for various L1 cache sizes. The impact of IP algorithm is tested on various L1 cache sizes and various cache parameters and all the experimental results show that precision improvement in WCET ranges from 20%-30% for benchmarks. When L1 cache size increases, more memory blocks are placed in L1 cache therefore interferences has no role for those blocks. The number of shared instruction cache interferences encountered by a parallel process from competing processes is proportional to the size of each competing process. If the size of competing processes for a parallel process P is large then there will be more interferences from the corresponding competing process results in less WCET precision improvement, similarly, if the size of competing processes for a parallel process P is small then there will be very less interferences from the corresponding competing process results in higher precision improvement as shown in Table 5.

Table 4. L2 cache access count for different cache organizations

	L1 Cache Size		
Cache Parameters	256 bytes	512 bytes	1 Kb
L2 Hit Count	155	38	1
L2 Miss Count	81	81	81
L2 Not Classified Count	61	65	49
L2 Access Count	297	184	131

Table 5. WCET precision improvement for benchmark programs

Benchmark	Parallel process size	Competing process size	WCET (precision improvement %)
SPMD	76312	76568	12.58%
	69632	34800	30.20%
	21420	2108	42.88%
MPMD	15662	25234	16.8718%
	3578	381	24.66%
	8929	2904	40.0511%

5. CONCLUSION

Worst case execution time analysis (WCET) of multithreaded programs is an important problem with the advent of multi-core architectures. Our work has addressed the need to investigate at a microscopic level of simulation measurements that have an impact on precise modelling of shared architectural units such as shared instruction cache with competing concurrent computations. Micro-architectural modelling component of WCET analyzer is made more precise for modelling shared instruction cache by reducing the set of shared instruction cache interferences based on an analysis of the multithreaded program. The analysis based on interference partitioning (IP) algorithm computes a more precise set of interferences from the competing threads. However, a study of impact of competing threads on shared instruction cache in WCET analysis of multi-threaded programs at different levels of granularity has not been addressed before. The different levels of granularity considered in this paper are at a basic block level and CSP process in a thread. The contributions and main observations of the work in the paper are Interpretation of the results of applying WCET analysis of a multithreaded program from WCET estimate of a thread to WCET estimate of each CSP process in the thread to WCET estimate of each basic block in the process. For the above purpose, rules of transforming a multithreaded program, to an equivalent Hoare's CSP program based on happens before relation between inter-thread synchronization calls to send() and receive(), are specified and implemented. For the SPMD style multithreaded programs chosen as benchmark programs, let T be a thread for which WCET is being estimated. Consider a process P in T. The WCET estimate of P can be made significantly more precise by statically determining a reduced and more accurate subset of potential shared instruction cache interferences from competing threads, that is a better approximation to the actual run-time interferences. It is always the case that the set of shared instruction cache interferences from competing processes (P) using IP algorithm is only a small subset of the set of shared instruction cache interferences from competing threads (P) using existing approaches. In such a case, the precision improvement in the WCET estimate of a process P in a thread T is significant. A thread is a composition of CSP processes, hence, WCET estimate of each thread is improved in terms of precision and being closer to the actual WCET. If the set of shared instruction cache interferences from competing processes (P) is a large subset of the set of shared instruction cache interferences from competing threads (P), then the reduction in the WCET estimate of process P in a thread may not be significant. Such a case arises when inter-thread synchronization in a multi-threaded program is only sparingly based on calls equivalent to wait() and notify(). In such a class of multi-threaded programs, it is more likely that entire code regions in threads are parallel to each other generating large sets of interferences. Significant improvements in precision of WCET estimates are observed empirically on a class of programs, whether multiple program multiple data stream (MPMD), or single program multiple data stream (SPMD) or other types of parallel programs such as fork/join, that contain several inter-thread communication calls equivalent to wait() and notify(), thereby causing a number of concurrent partitions which leads to small sets of interferences. WCET estimate of a basic block using the approach based on interference partitioning algorithm reduces significantly from 20%-30%. As a part of future work, WCET analysis of shared data cache will be considered.

REFERENCES

[1] R. Wilhelm *et al*., "The worst-case execution-time problem-overview of methods and survey of tools," *Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, Apr. 2008, doi: 10.1145/1347375.1347389.

[2] D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," in *Proceedings - Real-Time Systems Symposium*, Nov. 2008, pp. 456–466, doi: 10.1109/RTSS.2008.10.

[3] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in *Proceedings - Real-Time Systems Symposium*, Dec. 2009, pp. 68–77, doi: 10.1109/RTSS.2009.34.

[4] C. A. R. Hoare, "Communicating sequential processes," in *Theories of Programming*, New York, NY, USA: ACM, 2021, pp. 157–186, doi: 10.1145/3477355.3477364.

[5] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multi-cores," in *Proceedings - Real-Time Systems Symposium*, Dec. 2009, pp. 57–67, doi: 10.1109/RTSS.2009.32.

[6] P. P. P. Dharishini and P. V. R. Murthy, "Precise shared instruction cache analysis to estimate WCET of multi-threaded programs," in *Proceedings of the 2021 IEEE 18th India Council International Conference, INDICON 2021*, Dec. 2021, pp. 1–7, doi: 10.1109/INDICON52576.2021.9691620.

[7] D. Potop-Butucaru and I. Puaut, "Integrated worst-case execution time estimation of multicore applications," *OpenAccess Series in Informatics*, vol. 30, pp. 21–31, 2013, doi: 10.4230/OASIcs.WCET.2013.21.

[8] P. P. Priya Dharishini and P. Ramana Murthy, "static analyzer for computing WCET of multithreaded programs using Hoare's CSP," in *15th Innovations in Software Engineering Conference*, Feb. 2022, pp. 1–12, doi: 10.1145/3511430.3511438.

[9] V. Touzeau, C. Maïza, and D. Monniaux, "Model checking of cache for WCET analysis refinement," *arXiv preprint 1701.08030*, Jan. 2017, [Online]. Available: http://arxiv.org/abs/1701.08030.

[10] H. Ozaktas, C. Rochange, and P. Sainrat, "Automatic WCET analysis of real-time parallel applications," *OpenAccess Series in Informatics*, vol. 30, pp. 11–20, 2013, doi: 10.4230/OASIcs.WCET.2013.11.

[11] T. Kelter and P. Marwedel, "Parallelism analysis: Precise WCET values for complex multi-core systems," *Science of Computer Programming*, vol. 133, pp. 175–193, Jan. 2017, doi: 10.1016/j.scico.2016.01.007.

[12] T. Carle and H. Cassé, "Reducing timing interferences in real-time applications running on multicore architectures," *OpenAccess Series in Informatics*, vol. 63, pp. 31–312, 2018, doi: 10.4230/OASIcs.WCET.2018.3.

[13] K. Nagar and Y. N. Srikant, "Precise shared cache analysis using optimal interference placement," in *Real-Time Technology and Applications - Proceedings*, Apr. 2014, vol. 2014-October, no. October, pp. 125–134, doi: 10.1109/RTAS.2014.6925996.

[14] I. Puaut, M. Dardaillon, C. Cullmann, G. Gebhard, and S. Derrien, "Fine-grain iterative compilation for WCET estimation," *OpenAccess Series in Informatics*, vol. 63, pp. 91–912, 2018, doi: 10.4230/OASIcs.WCET.2018.9.

[15] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, "Indirect Communication," in *Distributed systems: Concepts and Design*, 5th ed., Addison-Wesley, 2011.

[16] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen WCET benchmarks: Past, present and future," *OpenAccess Series in Informatics*, vol. 15, pp. 136–146, 2010, doi: 10.4230/OASIcs.WCET.2010.136.

[17] K. Namjoshi, "Are concurrent programs that are easier to write also easier to check?," *Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[18] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Computing Surveys*, vol. 48, no. 2, pp. 1–36, Nov. 2015, doi: 10.1145/2830555.

[19] F. Brandner and A. Naji, "Worst-case execution time analysis of predicated architectures," *OpenAccess Series in Informatics*, vol. 57, pp. 61–613, 2017, doi: 10.4230/OASIcs.WCET.2017.6.

[20] F. Guet, L. Santinelli, J. Morio, G. Phavorin, and E. Jenn, "Toward contention analysis for parallel executing real-time tasks," *OpenAccess Series in Informatics*, vol. 63, pp. 41–413, 2018, doi: 10.4230/OASIcs.WCET.2018.4.

[21] A. Roychoudhury, "Performance validation," in *Embedded systems and software validation*, 2009.

[22] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified WCET analysis framework for multicore platforms," *Transactions on Embedded Computing Systems*, vol. 13, no. 4 SPEC. ISSUE, pp. 1–29, Jul. 2014, doi: 10.1145/2584654.

[23] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," *ACM Computing Surveys*, vol. 52, no. 3, pp. 1–38, May 2019, doi: 10.1145/3323212.

[24] T. Austin, E. Larson, and D. Ernest, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002, doi: 10.1109/2.982917.

[25] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 3, no. 1, pp. 05:1-05:48, 2016, doi: 10.4230/LITES-v003-i001-a005.

[26] J. L. Hennessy and D. A. Patterson, "Fundamentals of quantitative design and analysis," *Computer Architecture : A Quantitative Approach*, pp. 2–71, 2011.

BIOGRAPHIES OF AUTHORS

Padma Priya Dharishini is a Ph.D. Research scholar and Assistant Professor at the Department of Computer Science and Engineering at M.S. Ramaiah University of Applied Sciences, Bangalore. She has received her Master's in Embedded System from SASTRA University, Tanjore. Her current interest is in the domain of Worst-Case Execution Time Analysis, Multicore Architecture, and Static Program Analysis. She can be contacted at email: padmapriya.cs.et@msruas.ac.in.

Prakriya V. Ramana Murthy obtained Ph.D. degree in 1991 from Indian Institute of Science. He is currently a Professor in the department of Artificial Intelligence and Data Science at Nitte Meenakshi Institute of Technology, Bangalore. His interests are in the areas of Program analysis and verification. He conducted research in software engineering working as Senior Member Technical Staff and as a Principal Research Scientist at Siemens Corporate Technology. He can be contacted at email: prakriyamurthy@gmail.com.