Implications of Program Phase Behavior on Timing Analysis

Archana Ravindar Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science
Bangalore-560012, India
{archana,srikant}@csa.iisc.ernet.in

Abstract

Knowledge about program worst case execution time*(WCET) is essential in validating real-time systems and helps in effective scheduling. One popular approach used in industry is to measure execution time of program components on the target architecture and combine them using static analysis of the program. Measurements need to be taken in the least intrusive way in order to avoid affecting accuracy of estimated WCET. Several programs exhibit phase behavior, wherein program dynamic execution is observed to be composed of phases. Each phase being distinct from the other, exhibits homogeneous behavior with respect to cycles per instruction(CPI), data cache misses etc. In this paper, we show that phase behavior has important implications on timing analysis. We make use of the homogeneity of a phase to reduce instrumentation overhead at the same time ensuring that accuracy of WCET is not largely affected. We propose a model for estimating WCET using static worst case instruction counts of individual phases and a function of measured average CPI. We describe a WCET analyzer built on this model which targets two different architectures. The WCET analyzer is observed to give safe estimates for most benchmarks considered in this paper. The tightness of the WCET estimates are observed to be improved for most benchmarks compared to* Chronos*, a well known static WCET analyzer.*

1. Introduction

The goal of *worst case execution time* (WCET) analysis is to compute the longest execution time of a program on a given architecture. WCET analysis is critical in real-time system design where programs are expected to meet stringent performance goals. It is also valuable to systems that use dynamic task scheduling; The WCET of individual processes can be used to produce effective schedules and improve resource management. *Safety* and *Tightness* are desirable traits of a WCET estimate. A *safe* estimate is always greater than or equal to actual WCET. A *tight* estimate is within a few percent of actual WCET.

Traditionally there have been two schools of thought regarding WCET analysis. Static analyzers estimate WCET for a given architecture without actually running the program[1]. The static analyzer intrinsically models the effect of the worst case path and architectural state and hence can guarantee safety. However the architectural model of a static analyzer can be quite complex to build and re-target.

Measurement based analyzers measure smaller program components like basic blocks[2] or program segments[3] or paths[4] etc. These measurements are methodically combined to yield the final WCET. However it becomes difficult to guarantee safety as only finite measurements are taken and it is intractable to take into account the effect of all possible *inputs* on all possible program *paths* under all possible architectural states. For this reason, measurement based methods are more suited for *soft real-time* systems that do not have hard deadlines to adhere to. Such systems are typically driven by human perception and hence can afford to miss a few deadlines without causing noticeable change in system behavior.

The measurements in such a system need to be made in the *least intrusive* way in-order to avoid causing any impact on the accuracy of estimated WCET. Achieving an accurate estimate with less instrumentation is a non-trivial task[5]. In this paper, we propose a simple mechanism of measuring programs so that the number of instrumentation points is kept low without compromising on the accuracy of the estimate.

Our approach is based on the observation that several programs exhibit *phase behavior*[6, 7]. The dynamic behavior of such a program can be divided into phases during its execution. Each phase being distinct from other, exhibits relatively homogeneous behavior with respect to architectural metrics like average cycles per instruction(CPI), L1 data cache misses, branch predictor misses amongst many

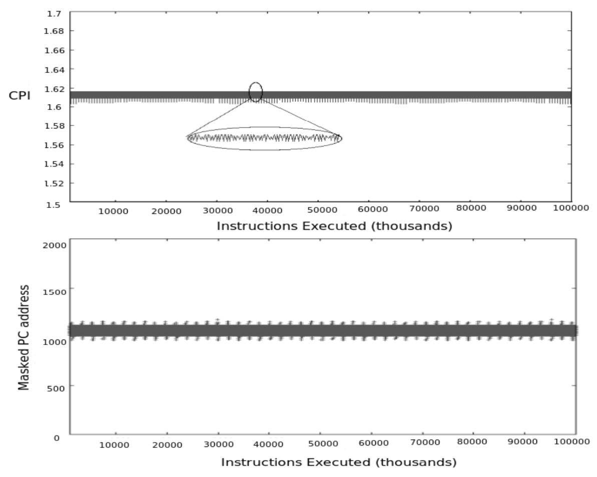


Figure 1. Variation of CPI and Program counter address values with respect to time for a single run of Matmul PISA binary.

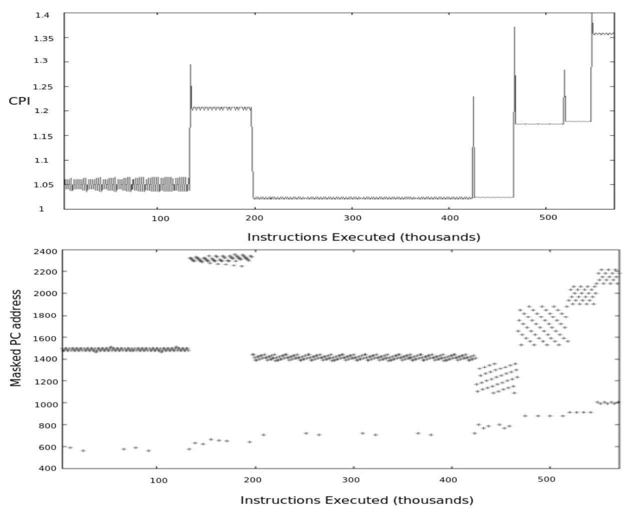


Figure 2. Variation of CPI and Program counter address values with respect to time for a single run of Bitcount PISA binary.

others. A program like *Matmul* is observed to exhibit a single phase as shown in Figure 1. *Matmul* is predominantly made of a single loop repeatedly accessing a fixed set of data. A program like *Bitcount* is observed to contain a number of phases as shown in Figure 2. *Bitcount* is composed of a set of functions each performing a single simple task. These programs are described in Table 2. Figures 1 and 2 have been plotted by sampling CPI and program counter address (masking its most significant bits), for every 1000 instructions executed.

Phase behavior is used for architectural simulation effort reduction[7, 8] apart from other applications like power and energy control, memory optimization etc. Our objective is to show that phase behavior has important implications on program timing analysis as well. We build on the observation that program CPI remains relatively stable within a phase. That is, the coefficient of variation(COV) of CPI within a phase is very less as compared to the COV of CPI across phases. Hence we can measure CPI at the phase level to effectively characterize timing of program phases and hence the whole program. Accounting for phase behavior helps alleviate instrumentation overhead compared to other measurement based approaches as phases are typically composed of thousands of instructions.

A program can be classified into phases in different ways depending on the parameter used for classification. In this paper, we divide the program static region into phases taking into account patterns of instruction execution[9],[8]. This method is in better sync with the natural period of the program and more dependable as code that is executed has an important influence on architectural behavior(Figure 2). Classifying the program thus, makes the phase pattern repeat consistently across most architectures rendering the technique retargetable. An important advantage of mapping a phase to a code region is that it is easier to come up with a timing model for that region.

We use code structural analysis [9] to mark phases in the binary. The CPI for every phase is measured by running the program with a large number of inputs. Measurements are taken using the cycle accurate simulator, *Simplescalar*[10]. The worst case CPI is defined as a function on measured CPI. The worst case number of instructions that can be executed within a phase is determined by static analysis of the program control flow graph (CFG). The WCET of a phase is then computed as a product of worst case CPI and worst case instruction count. The WCET of the whole program is computed as sum of WCETs of the individual phases. If the program has only one phase, WCET is simply a product of worst case instruction count and worst case CPI.

The proposed method is simple to implement and is retargetable, which makes it highly attractive to use for developers building a large system and need a quick WCET estimate of programs on a set of architectures even if its approximate. Most of the processors of today contain performance counters that ensure accurate measurement of several program metrics like CPI. In this work, programs are assumed to be single-threaded that execute without preemption. Although any general uni-processor architecture can be modeled, we target the WCET analyzer for two different architectures (Table 1) and test the method on a large set of standard WCET benchmarks(Table 2). The accuracy of the estimated WCET is evaluated by comparing it with a well known static WCET analyzer, *Chronos*[11]. For most programs, the proposed method is observed to give safe estimates. On an average, the proposed estimates are observed to be tighter by 13% than *Chronos* for architecture *A* and tighter by 196% than *Chronos* for architecture *B*.

Base configuration	Issue, decode and commit width=1, RUU size=8, Instn cache 8KB L1 2-way set associative, 2 level branch predictor, Fetch Queue size=4, In-order issue
Cache for Arch. A	No Data cache
Cache for Arch. B	Data cache 8KB L1 2-way set associative, Unified 64KB 8-way associative L2 cache

Table 1. Architectural configurations used for experimentation.

The remainder of this paper is organized as follows. Section 2 describes the main technique followed by its evaluation in Section 3. Related work is discussed in Section 4. The conclusions and future work are discussed in Section 5.

2. Proposed Solution

For a program, exhibiting predominantly a *single phase*, WCET is computed as,

$$WCET = (WIC) * (WCPI) \quad (1)$$

WIC or *Worst case instruction count* is statically determined by analyzing program CFG.
WCPI or *Worst case CPI* is the maximum of average CPI of a program observed across a large number of inputs. The CPI within a phase is expected to be fairly stable, hence average CPI is used in characterizing the execution time of a phase for a single input. We consider the warmup CPI separately in our calculations.
For a program, exhibiting multiple phases, WCET is computed as,

$$WCET = \Sigma_{(j \in 1 \; .. \; p)} \, (T_j * WIC_j * WCPI_j) \quad (2)$$

Where, *p* is the number of phases occurring during program execution, T_j is the number of times phase *j* occurs in the worst case, WIC_j is the worst case instruction count of code region corresponding to phase *j*, $WCPI_j$ is the worst case CPI of phase *j*.
The phases occurring in the binary are first identified using *code structural analysis*[9] as shown in Figure 3. Static analysis is then performed for the code region corresponding to each phase to determine WIC_j and T_j. The cycles per instruction (CPI_j) for each phase *j* is obtained by direct measurement of the program with a large number of test inputs on the target architecture. It is worthy to note that estimation of WIC for each phase and measurement of WCPI can be done in parallel. We now describe each step in detail, beginning with phase identification.

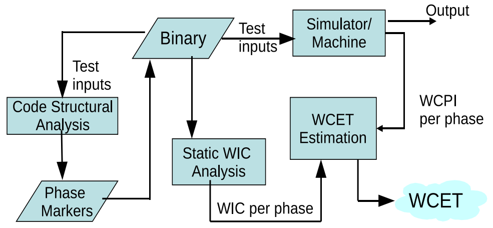


Figure 3. High level architecture of the proposed solution.

2.1. Phase Identification

Code structure analysis[9] takes the application binary as input and builds a *dynamic hierarchical call-loop graph* using profile information. A call-loop graph is a directed graph, whose nodes represent either a procedure call or a loop. It is termed hierarchical as the edges store hierarchical execution information along the path from call and loop nodes and are hence said to abstract path information in some sense. This graph is analyzed to locate instructions in the binary that accurately identify start of unique stable behaviors across different inputs. Such instructions are termed as *software phase markers*.

Each loop is associated with two nodes- loop head and loop body, to differentiate between loop invocation and each iteration of the loop respectively. Each call is associated with one node for non-recursive calls, two nodes for recursive calls. In this work, we do not consider recursive programs. Each edge stores average number of instructions executed along that path(A), coefficient of variation in instructions executed each time this edge was traversed (COV_{instn}), maximum number of instructions executed along that path (N_{max}) and total number of times the edge was traversed(C).

After the graph is constructed using profile information, all edges whose average instruction count exceeds a predetermined threshold are considered as candidates for software phase markers. This is to ensure that each phase is long enough. The phase length depends on the length of the application itself. For programs that execute a few thousand instructions, a minimum phase length threshold of 100 instructions could be used. Those candidate edges that also show minimum COV_{instn} finally qualify as software phase markers. This means that each time, such an edge is traversed, the amount of instructions hierarchically executed is more or less the same. That proves our assumption that we are seeing a faithful repetition of a phase every time we enter this path making it a valid software phase marker edge. The code region corresponding to a phase *p*, is represented

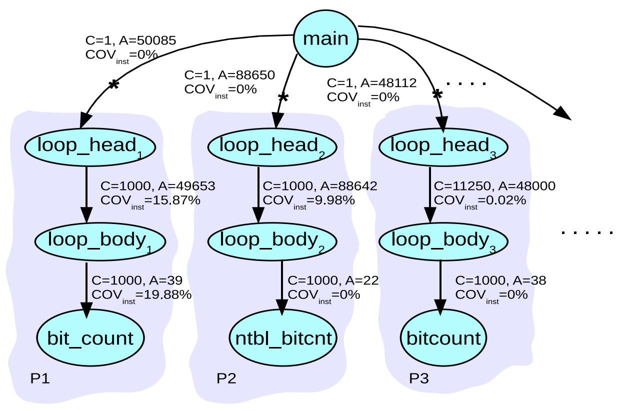


Figure 4. Hierarchical Call-loop graph for Bitcount: *C* is the number of times, each edge is traversed. *A* is the average number of hierarchical instructions executed each time the edge is traversed. COV$_{inst}$ is the hierarchical instruction count coefficient of variation. P1, P2, P3.. are phase numbers.

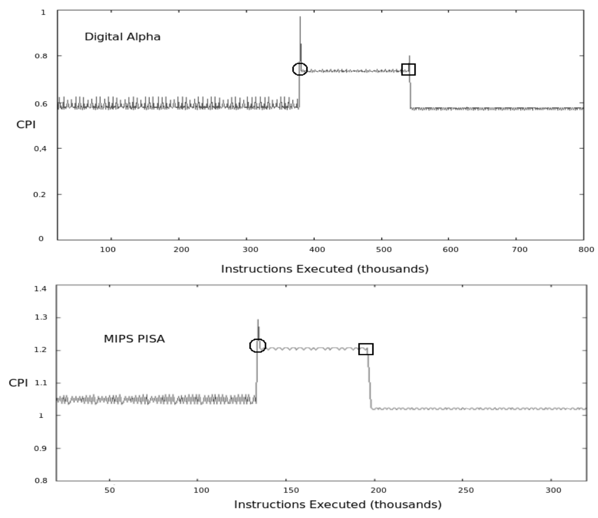


Figure 5. Time varying CPI graphs with phase markers for bitcount for an Alpha executable. The phase markers were selected from the call loop profile graph from the Alpha binary, were mapped back to source code level and then used to mark the MIPS PISA binary.

by the region between the phase marker edge of *p* and the phase marker edge of the following phase occurring in code.

A program is said to be composed of a *single phase* if its hierarchical call loop graph contains exactly one edge that satisfies these properties and that edge encompasses the whole program. Programs that cannot be classified into phases using this algorithm are also viewed as single-phase programs. However such programs depict a high degree of variance in their CPI throughout execution. *nsch* (Table 2) is an example.

Figure 4 depicts a part of the dynamic hierarchical call loop graph constructed for *Bitcount* that is run for 1000 iterations. The edge marked with an asterisk indicates that it satisfies the condition of a large enough average instruction count and small enough coefficient of variation in CPI and hence has been selected as a valid software phase marker edge. The phase marker edges picked by the algorithm for one input are observed to work well for other inputs as well[9]. The number of phase marker edges defines *p* in Equation (2).

Phase markers are typically edges representing call-loop boundaries and hence can be easily mapped on to binaries. Since phases are marked based on instruction execution patterns, we can see that *phase markers obtained by analyzing alpha binaries with ATOM[12] hold good for MIPS R3K PISA binaries* as shown in Figure 5. We modify the original algorithm that identifies instructions where a phase change is likely to occur, to also number phases as they occur. Execution of a program thus marked produces a phase sequence indicating the order in which instructions belonging to different phases are executed. The phase sequence encountered for each program considered in this work is shown in Table 2. Most of the programs considered here are simple and hence exhibit only one phase sequence irrespective of input. Programs with complex structure can exhibit multiple phase sequences. In such a case, WCET is computed as a maximum of the WCETs of all possible phase sequences. Phase based timing analysis for complex programs is in progress.

2.2. Context Sensitivity

A program analysis is termed as context sensitive if it differentiates two instances of a procedure occurring at two different contexts. In this work, we perform procedure cloning and treat each call instance as a separate call. This might cause the algorithm to assign different phase numbers to two call instances of the same procedure even if their CPI behavior is similar. This has an effect of increasing the number of phases but has no bearing on correctness of the subsequent timing analysis. Example: *Crc* has two phases, one for each clone and average CPI for each phase is about the same.

2.3. Estimating WIC

This section describes the computation of worst case instruction count of a phase which is done by a static analysis of the program CFG pertaining to the code region of a phase. We formulate an integer linear programming(ILP) problem for this purpose. ILP is used by many static WCET analyzers to estimate WCET[1]. Each basic block *B* in the CFG is associated with an integer variable N_B, denoting total execution count of basic block *B*. The static worst case instruction count of the CFG is then given by the linear objective function,

$$\textit{Maximize}\ \ \Sigma_{\forall B},\ \ (N_B * W_B) \tag{3}$$

Where, W_B is a constant denoting the number of instructions of a basic block. The linear constraints on N_B are developed from flow equations based on the CFG. Thus for basic block *B*,

$$\Sigma_{B' \to B}\ \ (E_{B' \to B})\ =\ N_B =\ \ \Sigma_{B \to B"}(E_{B \to B"}) \tag{4}$$

Where, $E_{B' \to B}$ ($E_{B \to B"}$) is an ILP variable denoting number of times control flows through the CFG edges B'$\to$ B (B $\to$ B"). If an edge happens to reside within a loop, the loop iteration bound (L) limits the number of times an edge can execute. The bounds can either be got by automatic loop bound detection techniques [13] or provided manually by an expert. For this work, we assume iteration bounds are given for all loops in the CFG. The corresponding linear constraint is specified as follows.

$$E_{i \to j} <= L \tag{5}$$

2.4. Infeasible Paths

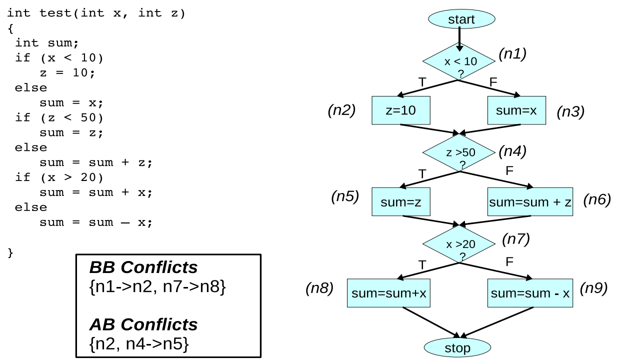
An infeasible path is one which can never occur in any valid execution of the program. Weeding out infeasible paths helps compute a much tighter WCET estimate. We follow the approach used in Vivy et al[14] and identify branch-branch conflict pairs and assignment-branch conflict pairs. A branch-branch(BB) conflict pair is a set of branch induced paths that can never occur together. Similarly an assignment-branch(AB) conflict pair is an assignment and a branch path that can never occur together. Figure 6 shows a simple example of an AB and a BB conflict that can occur.

Infeasible paths are modeled as additional linear edge constraints and are added to our linear system of equations-(3),(4) and (5). Two branch edges that figure in a BB pair, say, $E_{i \to j}$ and $E_{m \to n}$ have a linear constraint as shown in Equation (6). Similarly an assignment (node) and a branch edge that figures in an AB pair have a linear constraint as shown in Equation (7).

$$\textit{BB conflict}:\ E_{i \to j} + E_{m \to n} = 1 \tag{6}$$

$$\textit{AB conflict}:\ N_B + E_{i \to j} = 1 \tag{7}$$

Alternatively, WIC can be estimated statically by viewing the CFG as a weighted directed graph with basic blocks as nodes, W_B as edge weights and computing weighted longest path in the graph.



```
int test(int x, int z)
{
 int sum;
 if (x < 10)
    z = 10;
 else
    sum = x;
 if (z < 50)
    sum = z;
 else
    sum = sum + z;
 if (x > 20)
    sum = sum + x;
 else
    sum = sum – x;

}
```

BB Conflicts
{n1->n2, n7->n8}

AB Conflicts
{n2, n4->n5}

Figure 6. Illustration of Branch-Branch (BB) conflicts, Assignment-Branch (AB) conflicts.

2.5. Estimating WCPI

This section describes estimation of worst case CPI of a phase by measurement. The CPI of a phase is measured by sampling the phase at large intervals of instructions and averaging the samples. The samples for a phase are measured when the code region corresponding to that phase is executed. If the COV of CPI is very less within a phase, we can afford to take fewer samples without affecting the accuracy[8]. Which means, very less instrumentation is required within such a phase. Worst case CPI (WCPI) is defined as a function of measured per-phase CPI. For single-phase programs, WCPI is computed as a maximum of the observed overall program CPI across a large number of inputs, *i*. For programs containing multiple phases, *p*,

$$For\ each\ p,\ \ WCPI_p = Max_{\ \forall i}(CPI_p) \tag{8}$$

Warmup is an essential component of program execution that refers to the initial stage when all the architectural structures get filled in. The warmup CPI is typically higher than the stable program CPI. For programs executing millions of instructions, the effect of warmup can be ignored. The dynamic instruction count of the programs considered in this work range from a few thousand up to few millions. Hence for single-phase programs, CPI calculation considers the warmup stage as well. For multi-phase programs, we consider the warmup as a special phase and add the warmup cycles separately to our estimated program execution time.

3. Experimental Evaluation

We perform our experiments for a large set of benchmarks (Table 2) taken from *Mibench*[15] and *Mälardalen WCET benchmarks*[17]. All programs are compiled to

Benchmark and Description	Number of Inputs	Phase Sequence
Bezier (*bez*): Draws a set of 200 lines of 4 reference points on a 800X600 image.	500 sets of lines	P1 P2
Bitcount (*bitc*): Performs bit operations on a 1K bit-vector, 1000 times[15].	500 vectors	P1 P2 P3 P4 P5 P6
Binary Search (*bs*): Search for a key in a 10K number vector[17].	20K (key, vector) combinations	single
Bubble sort (*bub*): Sort an array of size 3K[17].	500 vectors	single
CRC (*crc*): Cyclic redundancy check on a 16KB char vector[17].	500 vectors	P1 P2
CNT (*cnt*): Counts positive numbers in a 200X200 matrix[17].	500 matrices	P1 P2
Dijkstra (*dij*): Finds 100 shortest paths in a graph of 200 vertices using dijkstra's algorithm[15].	500 graphs	single
EDN (*edn*): Implements set of signal processing algorithms[17].	500 signals	P1 P2 P3 P4 P5 P6 P7
FIR (*fir*): Finite impulse response filter over a signal of size 400[17].	500 signals	single
FFT (*fft*): Fast fourier transform on a wave of size 16K[15].	500 signals	P1 P2
Insertion Sort (*ins*): Sort a 3K number vector[17].	500 vectors	single
Janne_complex (*jan*): A a nested loop program, *a*, *b* are input parameters	500 combinations of *a*,*b*	single
LMS (*lms*): adaptive signal enhancement[17].	500 signals	single
LUD (*lud*): LU decomposition algorithm for a 200X200 matrix[17].	500 matrices	P1 P2 P3 P4
Matmul (*mat*): Matrix multiplication of two 200X200 matrices[17].	500 matrices	single
Nsch (*nsch*): Simulates an extended petrinet, $dummy_i$ is an input parameter[17].	$dummy_i = 32$, 500 starting states	single

Table 2. Benchmarks and their inputs.

MIPS PISA binaries with -O2 -static flags. We use *Simplescalar v3.0*[10] for measuring CPI of programs across a large number of inputs. The inputs are chosen so as to satisfy wide coverage at the level of statements, decisions, conditions and modified condition/decisions [18]. Invalid inputs and inputs that produce very short sequence of instructions are pruned away from calculations. We test the WCET analyzer for two different architectures, shown in Table 1. We sample programs at every phase marker instruction in addition to sampling every 1K instructions within a phase to note CPI. We have experimentally verified that the sampling interval within a phase can be varied arbitrarily without causing any impact on WCPI of the phase. *Chronos*[11], a well known static WCET analyzer also models the MIPS architecture and is hence chosen for comparison purposes. *aiT*[16] is a commercial static WCET analyzer widely used in the industry. Currently, we are in the process of modifying *Simplescalar* to work with ARM7 which is one of the targets supported by *aiT*.

3.1. Percentage COV of CPI

A good phase is said to exhibit minimum variance in CPI. The percentage COV of CPI for most of the single-phase programs is observed to be within 4% as shown in Figure 7 for both architectures. These programs are dominated by loops that exhibit repetitive behavior resulting in the CPI becoming stable. *nsch* is dominated by execution of a large number of branches that results in a large COV in instructions executed and hence makes phase identification difficult. The per-phase COV of CPI for most multi-phase programs is observed to rarely exceed 2% as shown in Figure 8. Had these programs not been classified into phases, they would exhibit much higher variance in CPI (shown as *No phase*) in the same figure. *Edn* exhibits highest *No phase* variation in CPI as it is composed of seven phases, each exhibiting a different average CPI. Similarly the variance in per-phase COV of CPI reduces after phase classification for programs on architecture B and hence not illustrated here. It

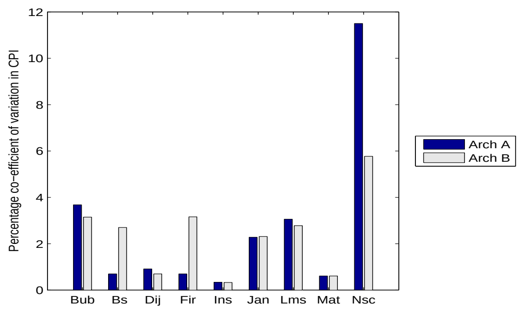


Figure 7. Percentage coefficient of variation of CPI for single-phase programs during execution on both architectures.

is this property of low variance in CPI that ensures accuracy of estimated WCET using the proposed method.

3.2. Accuracy of WCET Estimation

To evaluate the WCET analyzer, we compare estimated WCET with maximum observed cycles, *M*, got by running the program with a large number of inputs that ensure high path coverage. Estimated WCET is said to be safe if the ratio WCET/M is always greater than or equal to 1. The closer the ratio is to 1, tighter is the estimated WCET.

The proposed method splits WCET into two factors- WIC and WCPI. Worst case IC that is estimated statically, SWIC, could intrinsically be associated with a certain amount of pessimism. This is especially true for programs involving complex control flow, conditions driven by values computed at runtime etc. We can thus compute a softer estimate by using maximum observed instruction count, MIC, instead of SWIC in such cases. The second factor, WCPI is the *maximum* CPI observed across all inputs. There might be programs in which WCPI and WIC might not occur at the same time in any run. In such cases, a much softer esti-

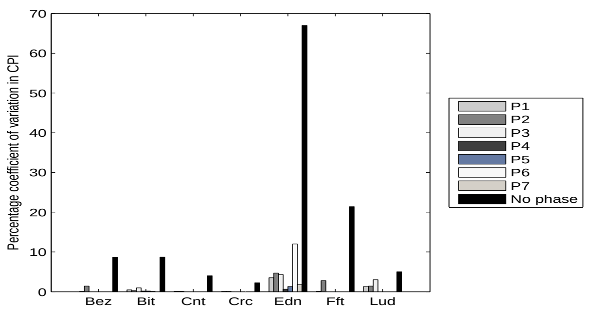


Figure 8. Per-phase percentage coefficient of variation of CPI of multi-phase programs on architecture A. (percentage coefficient of variation of CPI when phase classification is not made is also shown, as *No phase*).

mate can be got by considering ACPI-*Overall average* CPI observed across all inputs. For multi-phase programs with *p* phases, run with different inputs, *i*,

$$For\ each\ p,\ \ ACPI_p = Avg\ \forall i(CPI_p) \tag{9}$$

Depending on which of {SWIC, MIC, WCPI, ACPI} are used in timing equations (1) and (2), we have four formulae to estimate WCET as follows.

	WCPI = WCPI	**WCPI = ACPI**
WIC = SWIC	$WCET_1$	$WCET_2$
WIC = MIC	$WCET_3$	$WCET_4$

$WCET_1 >= \{WCET_2, WCET_3, WCET_4\}$, $WCET_3 >= WCET_4$

The safest formula would be $WCET_1$, with $WCET_4$ being the softest. If tightness is desired, either $WCET_2$ or $WCET_3$ can be used.

We now discuss results for architecture A. Programs *bs, fir, jan, lms, mat* exhibit a single homogeneous phase throughout execution with little variation seen in instruction count and CPI across different inputs. Hence the resulting estimates using any of the four formulae are quite close to *M* and are tighter than their corresponding *Chronos* counterparts. Programs *bez, bitc, cnt, crc, edn, fft*, and *lud* exhibit distinct phases during execution and perform as well as or better than Chronos. *bitc, bub* are overestimated compared to *Chronos* with the usage of WCPI($WCET_1$, $WCET_3$) but better estimated with ACPI($WCET_2$, $WCET_4$). *dij, ins, lud* and *nsch* are overestimated than *Chronos* with the usage of SWIC($WCET_1$, $WCET_2$) and but estimated better with MIC($WCET_3$, $WCET_4$).

We now discuss results for architecture B. Programs *bs, fir, jan, lms, mat* exhibit similar behavior on architecture B, hence the resulting WCET estimate using all four formulae are close to *M*. Programs like *bitc, bub* that display high variation in CPI across inputs, also show a corresponding increase in estimated WCET. Other programs that exhibit distinct phases, *bez, cnt, crc, edn, fft*, and *lud* show a very marginal increase in the WCET estimate when compared to architecture A. Just as in case of A, *dij, ins* and *nsch* are estimated better with MIC. Similarly *bitc* and *bub* are estimated better using ACPI instead of WCPI.

It can be observed that the gap between *M* and estimated WCET increases in case of *Chronos* on architecture B. This is due to the address analysis method used by *Chronos* for modeling data cache misses. Most of the programs under consideration involve vectors. During static WCET estimation, all addresses of a vector can equally reside in the data cache at any given point of time hence one has to conservatively assume accesses resulting in misses in absence of any information about runtime behavior. The effect is more so if vector size is large as that will increase the number of addresses. *Chronos* goes out of memory while analyzing *crc, dij, fft* and *nsch* for architecture B.

Assuming $WCET_1$ is used, on an average, the proposed method produces estimates that are tighter by 13% compared to *Chronos* for architecture A and by 196% compared to *Chronos* for architecture B.

4. Related Work

One of the earliest attempts in WCET analysis that uses performance counters was by Corti et al [19]. An analytical model is proposed that estimates execution time of a basic block using values of several important performance counters in the processor. The analytical equation is limited by availability of performance counters for various events. Unlike [19] we measure only CPI.

Most existing measurement based analyzers partition the program into smaller components like basic blocks[2], set of instructions[20], segments[3] or paths[4] to mitigate measurement overhead. They measure the execution time of these smaller components on the target architecture. Finally these execution times are combined taking the program structure into account to give the final WCET estimate. The proposed method also partitions the programs into components called phases. But the partitioning is based on observed instruction execution patterns and the intention is to group instructions exhibiting repetitive behavior into phases. This results in homogeneous behavior of the program within the phase. The phase change boundaries act as primary locations of instrumentation points. The homogenity of a phase allows us to place instrumentation points at arbitrarily large intervals within the phase thus helping us build a least intrusive measurement based WCET analyzer. Due to the non-availability of the measurement based tools in the public domain, we are not able to provide quantitative comparisons with our proposed method.

Kumar et al[21] apply a modified version of the structural analysis algorithm[9] to identify program execution

	WCET$_1$/M		**WCET$_2$/M**		**WCET$_3$/M**		**WCET$_4$/M**		**Chronos/M**	
	A	B	A	B	A	B	A	B	A	B
bez	1.02525	1.022601	1.009366	1.007426	1.025170	1.022521	1.009287	1.007348	1.017806	1.442967
bitc	1.130792	1.176671	1.059946	1.040402	1.075727	1.121183	1.008257	0.991226	1.067949	1.056883
bs	1.024415	1.012289	1.00011	1.000376	1.024415	1.012289	1.00011	1.000376	1.018388	1.266322
bub	1.087442	1.134203	1.024356	1.018872	1.073948	1.120129	1.011645	1.006229	1.0347	2.828077
cnt	1.00027	1.037123	1.002633	1.045158	0.972994	1.029021	0.952365	1.036927	1.072939	6.465426
crc	1.03839	1.031823	1.037938	1.031823	1.006924	1.001553	1.006495	1.001553	1.056714	*out of memory*
dij	5.325616	5.343056	5.325616	5.343056	1.030021	0.951454	1.030021	0.951454	5.20885	*out of memory*
edn	1.06132	1.00012	1.027345	0.976902	1.06132	1.00012	1.027345	0.976902	1.071338	1.28467
fft	1.013292	1.013729	1.013292	1.013729	0.999191	0.999414	0.999191	0.999414	1.036276	*out of memory*
fir	1.061563	1.061614	1.061205	1.059632	0.999972	1.000017	0.999632	0.998152	1.188546	2.140695
ins	3.313866	3.392809	3.248790	3.345844	0.999757	0.993932	0.981754	0.980939	3.253679	10.576999
jan	0.999182	0.999365	0.999182	0.999365	0.999199	0.999381	0.999199	0.999381	1.001164	1.025573
lms	1.000110	1.011009	1.000063	1.000063	0.999021	1.009957	0.999021	0.999021	1.036911	2.030692
lud	5.441254	5.461342	5.431489	5.431267	1.231167	1.215432	1.200245	1.200065	6.061123	5.443267
mat	0.999986	0.999983	0.999986	0.999983	0.999984	0.999981	0.999984	0.999981	1.000535	7.590843
nsch	3.469862	6.311587	2.58134	4.431003	0.9511967	0.942345	0.921990	0.931256	4.970352	*out of memory*

Table 3. Accuracy of WCET estimate got by the proposed method and *Chronos* on architectures A and B.

contexts that has highest influence on the soft real-time behavior of an application. Specifically, they identify those contexts that vary the most, across different inputs by observing variance in number of instructions executed got by profiling. However the proposed method uses phases of a program to decide instrumentation points and estimates WCET of the whole program by summing the estimated WCET of individual phases.

5. Conclusions and Future Work

This paper demonstrates that program phase behavior has important implications on timing analysis. The homogeneity of a phase allows us to instrument programs at the phase level and at arbitrarily large intervals within a phase without compromising on the accuracy of WCET. The paper proposes a model to estimate WCET as the sum of WCET of individual phases. The WCET of each phase is computed as a product of static worst case instruction count and a function of average CPI. Compared to *Chronos*, a well known static WCET analyzer, the proposed method on an average, is observed to give WCET estimates that are 13% tighter on an architecture without a data cache and 196% tighter on an architecture with a L1 data cache and L2 unified cache.

As part of future work, we intend to modify the phase classification algorithm in order to be able to give bounds on the variation of CPI within a phase and hence on estimated WCET. We also intend to modify the algorithm to detect infeasible paths that can cut across phases.

6. Acknowledgements

The authors would like to thank Matthew Jacob Thazhuthaveetil, Kapil Vaswani and Rupesh Nasre for their valuable suggestions on the initial drafts of this paper. We would also like to thank the anonymous reviewers for their helpful comments and feedback.

References

[1] R. Wilhelm et al. *The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools.* ACM Transactions on Embedded Computing Systems, 7(3), April 2008.

[2] G. Bernat, A. Colin and S. Petters. *pWCET: a Tool for Probabilistic Worst Case Execution Time Analysis of Real-Time Systems.* Technical Report YCS-2003-353, University of York, England, UK.

[3] I. Wenzel, R. Kirner, B. Rieder and P. Puschner. *Measurement-Based Worst-Case Execution Time Analysis.*, SEUS 2005.

[4] S. A. Seshia and A. Rakhlin. *Game-Theoretic Timing Analysis.* ICCAD 2008.

[5] A. Betts and G. Bernat. *Tree-Based WCET Analysis on Instrumentation Point Graphs.* ISORC'06.

[6] A. Dhodapkar and J.E. Smith. *Managing multi-configuration hardware via dynamic working-set analysis.* ISCA 2002.

[7] T. Sherwood, E. Perelman, G. Hamerly and B. Calder. *Automatically characterizing large scale program behavior.* ASPLOS 2002.

[8] W. Liu and M. C. Huang. EXPERT: Expedited Simulation Exploiting Program Behavior Repetition. ICS 2004.

[9] J. Lau, E. Perelman and B. Calder. *Selecting software phase markers with code structure analysis.* CGO 2006.

[10] `http://www.simplescalar.com`

[11] `http://www.comp.nus.edu.sg/∼rpembed/chronos/download.html`

[12] A. Srivastava and A. Eustace. *ATOM: A System for building customized program analysis tools.* PLDI 1994.

[13] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. van Englen. *Supporting timing analysis by automatic bounding of loop iterations.* Real-Time Systems(18).

[14] V. Suhendra, T. Mitra, A. Roychoudhry and T. Chen. *Efficient Detection and Exploitation of Infeasible Paths for Software Timing Analysis.* DAC'06.

[15] `http://euler.slu.edu/∼fritts/mediabench`

[16] `http://www.absint.com`

[17] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

[18] A. Dupuy and N. Levenson. *An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software.* DASC 2000.

[19] M. Corti, R. Brega and T. Gross. *Approximation of Worst-Case Execution Time for Preemptive Multitasking Systems.* LCTES 2000.

[20] A. Betts and N. Merriam and G. Bernat. *Hybrid measurement-based WCET analysis at the source level using object-level traces.* WCET 2010.

[21] T. Kumar, R. Cledat, J. Sreeram and S. Pande. *A profile-driven statistical analysis framework for the design optimization of soft Real-Time applications.* ESEC/FSE 2007.