Implications of Program Phase Behavior on Timing Analysis.

Archana Ravindar Y
Y. N. Srikan
Department of Computer Science and
Automation
Indian Institute of Science
Bangalore, 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 program components on the target architecture and combine them using static analysis of the program. Measurements involve instrumentation and need to be least intrusive so that the accuracy of estimated WCET is not affected. Several programs exhibit phase behavior, wherein program dynamic execution is observed to be composed of phases. Each phase being distinct from the other, exhibit 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 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 of each phase. We describe a WCET analyzer built on this model which targets two different architectures and is evaluated against Chronos, a well known static WCET analyzer. Compared to Chronos, the proposed method provides estimates that are tighter by 13% for an architecture containing only an L1 instruction cache and tighter by 196% for an architecture containing L1 instruction and data cache and an unified L2 cache.

Categories and Subject Descriptors
C.3 [Special-Purpose and Application-Based Systems]; Real-time and embedded systems; C.4 [Performance of Systems]: Measurement Techniques

General Terms
Structural analysis, software phase markers, binary instrumentation, measurements

Keywords
WCET, program phase behavior, cycles per instruction (CPI), profiling

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; knowing WCET of individual processes can produce effective schedules and improve resource management. Factors such as unknown worst case input, influence of underlying hardware architecture, impact of dynamic interactions among different components during program execution make WCET estimation challenging. An estimate is safe if it is greater than or equal to actual WCET. An estimate is tight if it is within a few percent of actual WCET.

Traditionally there have been two major schools of thought regarding WCET analysis. Static WCET analyzers estimate WCET for a given architecture without actually running the program[1]. Measurement based WCET 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. Measurement based method is simple and can model target architecture better than static analysis. 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. Static methods intrinsically model the effect of the worst case path and architectural state and hence can guarantee safety. 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.

Measurement based WCET analyzers are mainly characterized by the number of instrumentation points placed in the program. To achieve an accurate estimate with less instrumentation is a non-trivial task [5]. Higher number of instrumentation points can make the measurement process intrusive affecting accuracy of measurements. In this paper, we propose a new way of measuring programs so that the number of instrumentation points is kept low without...
Our approach is based on the observation that several programs exhibit phase behavior\cite{7, 8, 9}. The dynamic behavior of such a program can be divided into phases during its execution. Each phase exhibits relatively homogeneous behavior with respect to architectural metrics like average cycles per instruction (CPI), L1 data cache misses, branch predictor misses and so on. Distinct behavior is seen across different phases. A program could either comprise of a single phase as shown in Figure 1. For example, Matmul\cite{Table 2} is predominantly made of a single loop repeatedly accessing a fixed set of data. Alternatively it could comprise of a number of phases. For example, Bitcount\cite{Table 2} is composed of a set of functions each performing a single simple task, as shown in Figure 2.

Phase behavior is used for architectural simulation effort reduction\cite{9, 11} 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 typically comprise 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 classify programs into phases considering the program structure and patterns of instruction execution. Each classification method views the program with a specific granularity. In this work, we choose to use static code regions to define phases as in \cite{14, 11} rather than using an arbitrarily selected fixed-size granularity that can go out of sync with the natural period of the program. Selecting code to identify phases is more dependable as code executed has an important influence on architectural behavior (Figure 2). Classifying the program thus, makes phases architecture independent. Once we map phases to a code region, we can easily model a timing equation for that region.

In this work, we employ code structural analysis \cite{14} 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\cite{15}. 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 highly retargetable as one can measure per-phase CPI of the program on a different architecture without having to re-do phase marking. This makes our approach highly attractive to use for developers building a large system and who need a quick WCET estimate of his/her programs on a set of architectures even if its approximate. Most of the processors of today have performance counters available in their hardware that ensure accurate measurement of several program metrics like CPI.

The proposed method assumes single-threaded programs that execute without preemption. Although the proposed method can model any general uniprocessor architecture, 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 popular
static WCET analyzer Chronos[16]. 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 by 196% than Chronos for architecture B. The contributions of this work can be summarized as follows.

- This paper demonstrates that program phase behavior has important implications on timing analysis. Phases can be used in measurement based WCET analyzers to reduce instrumentation overhead without affecting accuracy of the resultant WCET estimate.

- A model for estimating WCET of a program in terms of its phases is proposed. WCET of a phase is estimated as a product of static worst case instruction count and worst case CPI.

- We formulate an integer linear programming problem (ILP) to estimate worst case instruction count of a phase. Many static WCET analyzers use the ILP framework to estimate WCET[11]. Phases are identified by code structural analysis[14].

- The WCET analyzer is implemented based on this model for two different architectures and evaluated by running it on standard WCET benchmarks. The estimates are compared with Chronos, a popular static WCET analyzer for the two architectures. The proposed method is observed to give much tighter estimates compared to Chronos for architectures that have a data cache.

<table>
<thead>
<tr>
<th>Name</th>
<th>Architectural configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arch A</td>
<td>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, No Data cache</td>
</tr>
<tr>
<td>Arch B</td>
<td>Issue, decode &amp; commit width=1, RUU size=8, Instn cache 8KB L1 direct mapped, In-order issue, Data cache 8KB L1 2-way set associative, Unified 64KB 8-way associative L2 cache, 2 level branch predictor, Fetch Queue size=4</td>
</tr>
</tbody>
</table>

Table 1: Architectural configurations used for experimentation.

2. PROPOSED SOLUTION

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

\[
WCET = (WIC) \times (WCPI)
\]  

(WIC = Worst case instruction count): is statically determined by analyzing program CFG.

(WCPI = Worst case CPI): The average CPI of a program for a given input i, CPIi, is computed as a ratio of total number of cycles taken for execution to total number of instructions executed. WCPI is defined as Max(CPIi), across all test inputs, i. 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

\[
WCET = \sum_{j=1}^{p} (T_j \times WIC_j \times WCPI_j)
\]  

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 high level organization of the proposed solution is as described in Figure 3. The number of phases \( p \) in the binary is determined using code structural analysis[14]. Static analysis is then performed for each code region corresponding to a phase to determine \( WIC_j \) and \( T_j \) of equation (2). To complete the equation, we substitute cycles per instruction (CPIj) for each phase \( j \), got by direct measurement of the program on a large number of test inputs on the target architecture. 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.](image)

2.1 Phase Identification

Code structure analysis[14] 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. Each edge stores average number of instructions executed along that path(A), coefficient of variation in instructions executed each time this edge was traversed \( (COV_{inst}) \), 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 pre-determined threshold are considered as candidates for software phase markers. This is to ensure that each phase

\footnote{In this work, we do not consider recursive programs.}
is long enough. Those candidate edges that also show minimum COV$_{inst}$ 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 everytime we enter this path making it a valid software phase marker edge.

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.

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. For programs that execute a few thousand instructions, a minimum phase length threshold of 100 instructions is used.

Phase markers are typically edges representing call-loop boundaries. Hence they can be easily mapped on to binaries. Since phases are architecture-independent, we can see that phase markers obtained by analyzing alpha binaries with ATOM[17] hold good for MIPS R3K PISA binaries as shown in Figure 5. The original algorithm identifies instructions where a phase change is likely to occur. We modify the original algorithm to number phases as they occur. Execution of a program thus marked produces a phase sequence that indicates the order in which instructions belonging to different phases were executed. The phase sequence encountered for each program considered in this paper is shown in Table 2. Depending on the structural complexity of a program, multiple phase sequences are possible. The WCET for such a program is computed as a maximum of the WCETs of all possible phase sequences. Most of the programs considered here are simple and hence exhibit only one phase sequence irrespective of input.

2.1.1 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 (now two different procedures) 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 impending timing analysis. An example is Crc that has two phases, one for each clone and average CPI for each phase is about the same.

2.2 Estimating WIC

This step involves computation of worst case instructions of the code region representing a phase by statically analyzing the program CFG. 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,

$$\text{Maximize } \sum_B (N_B \cdot W_B)$$  

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.
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 $B$, 
\[ \Sigma_{B' \rightarrow B} (E_{B' \rightarrow B}) = N_B = \Sigma_{B \rightarrow B'} (E_{B \rightarrow B'}) \] (4)

Where, $E_{B' \rightarrow B} (E_{B \rightarrow B'})$ is an ILP variable denoting number of times control flows through the CFG edges $B' \rightarrow B$ ($B \rightarrow B'$). 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 [18] 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.
\[ E_{i \rightarrow j} \leq L \] (5)

### 2.2.1 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 [19] 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.

![Illustration of Branch-Branch (BB) conflicts, Assignment-Branch (AB) conflicts. BB={n1→n3,n2→n4}. AB={n3,n4→n5}.
](image)

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, $E_{i \rightarrow j}$ and $E_{m \rightarrow n}$, 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).

\[ BB \text{ conflict: } E_{i \rightarrow j} + E_{m \rightarrow n} = 1 \] (6)
\[ AB \text{ conflict: } N_B + E_{i \rightarrow j} = 1 \] (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.

### 2.3 Estimating WCPI

This step determines worst case CPI of a phase by taking measurements. The CPI of a phase is measured by sampling the phase\(^4\) at large intervals of instructions and averaging the samples. If the COV of CPI is very less within a phase, we can afford to take fewer samples without affecting the accuracy [11]. Which means, very less instrumentation is required within such a phase. WCPI or worst case CPI 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, \(p\),

\[ WCP_{pi} = \text{Max}_i (CPI_i) \] (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 [20] and Malardalen WCET benchmarks [22]. All programs are compiled to MIPS PISA binaries with -O2 static flags. We use Simplescalar v3.0 for measuring CPI of programs across a large number of inputs. The inputs are chosen so as to satisfy coverage criteria at the level of statements, decisions, conditions and modified condition/decisions [23]. 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 as 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. We choose Chronos for comparison, as the target architecture is common for both. \(aT[21]\) is a commercial static WCET analyzer widely used in the industry. However it does not work with Simplescalar. Currently, we are in the process of modifying Simplescalar to support ARM7 binaries as that is one of the targets supported by \(aT\).

### 3.1 Percentage COV of CPI

A phase is said to be well selected if it exhibits minimum variance in CPI. The percentage COV of CPI for most of the single-phase programs is observed to be within a few percent as shown in Figure 7 for both architectures. These programs are dominated by loops that exhibit repetitive behavior resulting in the CPI becoming stable. PCC 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. For multi-phase programs, Figure 8

\(^4\)CPI for a phase is the measured CPI when the code region corresponding to the phase is executed.
3.2 Accuracy of WCET Estimation

A common approach to evaluate a new WCET analyzer is to 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, and WCPI, which is estimated dynamically during execution. The safest formula would be WCET\(_1\), with WCET\(_2\) or WCET\(_3\) can be used.

Table 2: Benchmarks and their inputs.

<table>
<thead>
<tr>
<th>Benchmark and Description</th>
<th>Number of Inputs</th>
<th>Phase Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble</td>
<td>500 sets of lines</td>
<td>P1 P2</td>
</tr>
<tr>
<td>bitc[20]</td>
<td>500 vectors</td>
<td>P1 P2 P3 P4 P5 P6</td>
</tr>
<tr>
<td>bs[22]</td>
<td>20K (key, vector)combinations</td>
<td>single</td>
</tr>
<tr>
<td>bsdb[22]</td>
<td>500 vectors</td>
<td>single</td>
</tr>
<tr>
<td>cint[22]</td>
<td>500 vectors</td>
<td>P1 P2</td>
</tr>
<tr>
<td>cint[22]</td>
<td>500 matrices</td>
<td>P1 P2</td>
</tr>
<tr>
<td>dij[20]</td>
<td>500 graphs</td>
<td>single</td>
</tr>
<tr>
<td>edn[22]</td>
<td>500 signals</td>
<td>P1 P2 P3 P4 P6 P7</td>
</tr>
<tr>
<td>fft[20]</td>
<td>500 signals</td>
<td>single</td>
</tr>
<tr>
<td>ft[20]</td>
<td>500 signals</td>
<td>P1 P2</td>
</tr>
<tr>
<td>ins[22]</td>
<td>500 vectors</td>
<td>single</td>
</tr>
<tr>
<td>jan[22]</td>
<td>500 combinations of (a,b)</td>
<td>single</td>
</tr>
<tr>
<td>lms[22]</td>
<td>500 signals</td>
<td>single</td>
</tr>
<tr>
<td>lns[22]</td>
<td>500 matrices</td>
<td>P1 P2 P3 P4 P6 P7</td>
</tr>
<tr>
<td>mat[22]</td>
<td>500 matrices</td>
<td>single</td>
</tr>
<tr>
<td>nsh[22]</td>
<td>Simulates an extended petrinet, (\text{dummy}_i) is an input parameter.</td>
<td>single</td>
</tr>
</tbody>
</table>

For each \( p \), \( ACP_{PI} = \text{Avg} \forall i(CPI_i) \) (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.

\[
\begin{align*}
\text{WIC} = \text{SWIC} & \quad \text{WCPI} = \text{WCET}_1 & \quad \text{WCET} = \text{ACPI} \\
\text{WIC} = \text{MIC} & \quad \text{WCET} = \text{WCET}_2 \\
\text{WCET}_1 \geq \text{WCET}_2, \text{WCET}_3, \text{WCET}_4, \text{WCET}_5 \geq \text{WCET}_4 \\
\end{align*}
\]

The safest formula would be WCET\(_1\), with WCET\(_2\) or WCET\(_3\) can be used.

We now discuss results for architecture A. Programs Binary search, Fir, Janne complex, Lms, Matmul exhibit a
**Table 3:** Accuracy of WCET estimate got by the proposed method and Chronos on architectures A and B.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>WCET1/M</th>
<th>WCET2/M</th>
<th>WCET3/M</th>
<th>WCET4/M</th>
<th>Chronos/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>box</td>
<td>1.02</td>
<td>1.02</td>
<td>1.0</td>
<td>1.0</td>
<td>1.99</td>
</tr>
<tr>
<td>bte</td>
<td>1.17</td>
<td>1.12</td>
<td>1.04</td>
<td>1.2</td>
<td>1.09</td>
</tr>
<tr>
<td>bs</td>
<td>1.01</td>
<td>1.01</td>
<td>1.0</td>
<td>1.0</td>
<td>1.02</td>
</tr>
<tr>
<td>bus</td>
<td>1.02</td>
<td>1.02</td>
<td>1.0</td>
<td>1.0</td>
<td>1.06</td>
</tr>
<tr>
<td>cnt</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.02</td>
</tr>
<tr>
<td>crc</td>
<td>1.04</td>
<td>1.03</td>
<td>1.0</td>
<td>1.0</td>
<td>1.02</td>
</tr>
<tr>
<td>dij</td>
<td>1.12</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.02</td>
</tr>
<tr>
<td>edn</td>
<td>0.99</td>
<td>0.99</td>
<td>1.0</td>
<td>1.0</td>
<td>1.09</td>
</tr>
<tr>
<td>ft</td>
<td>1.01</td>
<td>1.01</td>
<td>1.0</td>
<td>1.0</td>
<td>1.09</td>
</tr>
<tr>
<td>fr</td>
<td>1.06</td>
<td>1.06</td>
<td>1.0</td>
<td>1.0</td>
<td>1.09</td>
</tr>
<tr>
<td>ins</td>
<td>0.99</td>
<td>0.99</td>
<td>1.0</td>
<td>1.0</td>
<td>1.09</td>
</tr>
<tr>
<td>lms</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.09</td>
</tr>
<tr>
<td>nms</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.09</td>
</tr>
<tr>
<td>ntr</td>
<td>1.01</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.09</td>
</tr>
<tr>
<td>nwd</td>
<td>1.01</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.09</td>
</tr>
<tr>
<td>pack</td>
<td>0.99</td>
<td>0.99</td>
<td>1.0</td>
<td>1.0</td>
<td>1.09</td>
</tr>
<tr>
<td>insch</td>
<td>1.02</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.09</td>
</tr>
</tbody>
</table>

**single homogeneous phase throughout execution.** These programs are predominantly loop oriented with little variation seen in instruction count and CPI across different inputs. Hence the resulting estimates using any of the four formulæ are close to the observed maximum, $M$ and are tighter than their corresponding Chronos counterparts. Programs Bezier, bitcount, Cnt, Crc, Edn, Fft, and Lud exhibit distinct phases during execution and perform as well or better than Chronos. Bitcount, Bub perform poorer than Chronos with the usage of WCPI(WCET$_1$, WCET$_2$) but fare better with ACPI(WCET$_3$, WCET$_4$). Dijkstra, Insertion sort, Lud and nsch perform poorer than Chronos with the usage of SWIC(WCET$_1$, WCET$_2$) and but fare better with MIC(WCET$_3$, WCET$_4$).

We now discuss results for architecture A. Programs Binary search, F1r, Junne-complex, Lms, Matmul appear to perform similarly on architecture B as architecture A. The reason is that these programs are composed of simple structure and exhibit very stable CPI within a run and across runs with different inputs. Hence the resulting behavior is more due to the program property than due to architecture. Programs that display high variation in CPI across inputs, like bitcount, bsb also show a corresponding increase in estimated WCET. Other programs that exhibit distinct phases, Bezier, Cnt, Crc, Edn, Fft, and Lud show a very marginal increase in the accuracy of WCET estimation when compared to architecture A. Programs Dijkstra, Insertion sort and nsch exhibit similar improvement with MIC. Bitcount and Bub encounter greater improvement using ACPI instead of WCPI as the variation seen in architecture B is greater than architecture A.

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, Ftt and nsch for architecture B.

Assuming WCET$_2$ is used, the proposed method produces estimates that are tighter by 13% compared to Chronos for architecture A and by 96% compared to Chronos for architecture B.

4. RELATED WORK

Most of the existing measurement based WCET analyzers measure execution time of smaller parts of a program on the target architecture before combining them methodologically, taking program structure into account. Corti et al [24] proposes an analytical model that estimates execution time of a basic block using values of several event counters that track instruction and data cache misses, pipeline stalls, branch mispredictions etc. The analytical equation is limited by availability of performance counters for various events. Unlike [24] we measure only CPI.

The theoretical WCET might only occur very rarely in systems. Hence Bernat et al[2] proposed to estimate WCET probabilistically by instrumenting programs and generating a trace using which the time taken by program components can be determined. The trace also contains frequency information which are combined probabilistically using timing schema[2] to estimate WCET. Programs are instrumented at the object level using hardware support to avoid intrusion[25]. However object level tracing has many issues- mapping measurements back to the source code is difficult. Other issues include limitations of hardware in recording all branches and a huge rate of trace generation[25]. The proposed method however takes advantage of phase behavior inherent in many programs which enables it to instrument over arbitrarily large windows of instructions. The trace in our case is average CPI sampled over thousands of instructions and hence very lightweight.

Wenzel et al[3] partitions programs into segments to manage complexity of measurement. The accuracy of WCET is sensitive to number of paths per segment[27]. With large segments it becomes necessary to use automatic test data generation methods. In the proposed method, input test data is generated based on the coverage of program statements including phase marker instructions, conditions, decisions, modified condition/decisions[23]. In-order to ensure accuracy of WCET is not affected by sparse instrumentation, Betts et al[5] proposes the use of instrumentation point graphs (IPG), a more powerful form of CFG. The location of instrumentation points determines IPG structure, accuracy of WCET and amount of trace data generated and hence has
to be determined very carefully. Our method uses program structure analysis that is a well tested method[14] to recognize repeated patterns of instructions executed leading to phases. The variation of CPI within a phase is much more stable and organized compared to variation of CPI across phases. The number of phases determine the number of instrumentation points to be placed. Within a phase, CPI can be sampled at arbitrarily large instruction windows without creating an impact on accuracy of estimated WCET.

Kumar et al.[26] apply a modified version of the structural analysis algorithm[14] to identify program execution 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. The proposed method marks phases in a program and estimates its 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 measured average CPI.

Seshia et al[4] formulate a game between the estimation algorithm(player) and the execution environment(adversary) to estimate WCET. The player seeks to estimate the length of any path in a program whereas the adversary sets parameters to thwart the player. The game needs to proceed in several rounds for the player to learn enough about the environment to be able to accurately predict path lengths with high probability. Basis paths of a program form the unit of measurement; determining them is the most expensive part of this analysis. Due to the non-availability of these measurement based tools in the public domain, we are not able to provide quantitative comparisons with our proposed method.

5. CONCLUSIONS AND FUTURE WORK

This paper demonstrates that phase behavior has important implications on timing analysis. Many programs are composed of phases, each phase being distinct from the other. The behavior of architectural metrics like CPI within each phase is far more stable when compared across phases. 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. Phases are marked in the program using code structural analysis that identifies repeated sequence of instructions executed across a large number of profile runs. Phases thus marked are independent of architecture, making the proposed method retargetable.

We describe a WCET analyzer, implemented for two different architectures. Compared to Chronos, a well known static WCET analyzer, the proposed method is observed to improve tightness of WCET estimates by 13% for an architecture with just an L1 instruction cache and by 196% on an architecture with an L1 instruction and data cache along with an L2 unified cache.

Currently, the phase classification algorithm does not give bounds on the variation of the CPI within a phase, although it is observed to be well within a few percent for most programs. We intend to modify the phase classification algorithm to identify finer variations in CPI at the same time being able to map back to the binary. Then we will be able to give bounds on the estimated WCET. The proposed method estimates WCET to be the sum of WCET of its constituent phases. This might cause an overestimation in cases where there exists an infeasible path that cuts across more than one phase. The occurrence of such a situation is rare as each phase identified in this work, represents a cohesive unit of execution. Nevertheless, this information can be made available to code structural analysis so that classification can take this into account.

6. REFERENCES