WCET Analysis of Component-Based Systems using Timing Traces

Adam Betts
Mälardalen University
School of Innovation, Design, and Engineering
Västerås, Sweden
adam.betts@mdh.se

Amine Marref
Department of Computer Science
Umm Al-Qura University
Makkah, Saudi Arabia
ajmarref@uqu.edu.sa

***Abstract*—Construction of a Real-Time System (RTS) out of a number of pre-fabricated pieces of software, otherwise known as components, is a pervasive area of interest. Typically, only relocatable object code of the component is shipped to the customer, so that it can later be linked into the overall application. Source code is therefore withheld, and disassembling of the object code is normally disallowed to protect intellectual property. Both of these restrictions complicate, or even prevent, state-of-the-art Worst-Case Execution Time (WCET) analysis of the RTS since most techniques are grounded on their availability in order to generate a complete program model. The alternative solution — widespread in industrial circles — is to record the largest end-to-end execution time of the RTS under functional testing, but this underestimates the actual WCET, in the general case.**

This paper shows how to obtain a safer WCET estimate of a RTS composed of components using time-stamped traces of program execution. In effect, the data needed in the WCET computation (program model, execution times, execution bounds) are derived exclusively from parsing of the traces. Experiments indicate that, once simple coverage metrics have been obtained, the calculated WCET estimate bounds the actual WCET. Moreover, where instrumentation (which produces the time-stamped traces) is placed with respect to program structure has a significant bearing on the accuracy of the computed WCET estimate.

I. Introduction

A **Real-Time System** (RTS) is an embedded system for which precise operation also depends on timing constraints. Depending on the type of application, failure of a RTS has a wide range of possible consequences, from a jittery video streaming application to the delayed release of an airbag. It is therefore critical — sometimes even safety critical — that some analytical process has been undertaken to verify the temporal properties of a RTS before its eventual dispatch into the real world.

The design of a RTS revolves heavily around a model known as a task schedule, which allots the CPU resource to executing tasks, assuming access to the **Worst-Case Execution Time** (WCET) of each task. However, determining the **actual WCET** is not trivial because software and hardware properties both cause variation in execution times. On the one hand, software typically has an exponential number of paths, and on the other hand, the time taken for each instruction depends on the features present in the hardware architecture, e.g. cache. For these reasons, **WCET estimates** are sought in which a common requirement is to bound the actual WCET so that the task schedule is not compromised. Yet, simply providing a safe upper bound is tempered by the desire for accuracy because RTS resources are limited and need to be maximised accordingly. The epitome of **WCET analysis** is to therefore compute a WCET estimate that is the actual WCET.

Techniques to perform WCET analysis can broadly be categorised as follows [1]:

1) **End-to-End:** Under quite extensive and demanding test conditions, the **High Water-Mark Time** (HWMT) is obtained, which is the longest *observed* end-to-end execution time. The underlying premise is that the testing regime is representative of real system operation and that, with enough testing, the HWMT lies in close proximity to the actual WCET. However, acknowledging that there could be some underestimation, the HWMT is subsequently augmented using engineering wisdom from previous projects: this augmented value is then considered to be the WCET estimate.
2) **Static Analysis (SA):** Rather than run the software on target, SA constructs two different models: a **program model**, which represents small segments of the software; and a **processor model**, which synthesises the functional and temporal behaviour of the hardware. Execution times of the program segments are gleaned from the processor model and, together with flow information bounding loops and constraining the set of feasible execution paths, finally combined by a calculation engine [2]–[4] operating on the program model, resulting in a WCET estimate.
3) **Hybrid Measurement-Based Analysis (HMBA):** The steps are similar to those of SA, except it collects the execution times of program segments via **instrumentation points** (ipoints) as the software runs on target. Processor modelling is therefore bypassed, but the program model is retained as the measured data still needs to be combined in the calculation stage.

Both SA and HMBA rely on access to the source code or the ability to disassemble the binary in order to construct the program model. However, both of these assumptions are challenged when a RTS is composed of a number of pre-fabricated pieces of software called components, which is a continuously increasing trend in the embedded domain [5]. Only the relocatable object code of a component is shipped to the client so that it can later be linked into the overall application. Since the source code of the component is unavailable, and because disassembling of its object code

is typically disallowed to protected intellectual property, the complete program model cannot be built. As a result, existing SA and HMBA techniques to WCET analysis are neutralised and, according to the above taxonomy, the only alternative to obtain a WCET estimate is through end-to-end measurements.

However, the end-to-end approach has a number of shortcomings. First, the HWMT could severly underestimate the actual WCET. This is because, to date, no coverage metrics have been proposed which stress execution time variability caused by loop iterations, context-dependent execution, or hardware effects; that is, all *de facto* test-vector generation mechanisms stop once functional coverage metrics (e.g. MC/DC [6]) have been satisfied. Second, the safety margin added to the HWMT is clearly *ad hoc*: there is no guarantee that the actual WCET is bounded, but there could also be severe underutilisation of system resources if, in fact, the HWMT equals the actual WCET but the safety margin is too excessive.

The crux of the problem is that the end-to-end approach simply treats the system as a black box, whereas SA and HMBA need more information about system internals and are therefore white box by nature. In this paper, we present a technique to perform WCET analysis on a component-based RTS that is effectively a grey-box HMBA. Rather than rely on the disassembly or the source code, our approach builds the program model on the fly from a set of **timing traces** which have been generated by an instrumented[1] program during the test phase. Given that no algorithmic properties of the system under analysis are exposed during this construction, any restrictions on reverse engineering of the object code are implicitly observed. Moreover, another advantage of our approach is that it is immediately applicable to *any* code construct, whereas some tools and techniques cannot handle particular structures, e.g. recursion or function pointers, since the program model is *statically* constructed. Following are the concrete contributions:

- We show how to deliver a safer WCET estimate than end-to-end approaches without the disassembly or source code. This is done by constructing the **Instrumentation Point Graph** (IPG) [7] on the fly from a set of timing traces and at the same time collecting information needed in the WCET calculation, e.g. execution times of its atomic units of computation; these data are subsequently combined through the **Implicit Path Enumeration Technique** (IPET) [3] to produce a WCET estimate. Furthermore, how the code is instrumented is completely orthogonal to the presented approach.
- Our evaluation compares the impact of different instrumentation policies (see Section II) on the WCET estimates of a collection benchmarks [8] — to the authors' best knowledge, no such comparison has previously been disseminated. Results indicate that the locations of instrumentation with respect to program structure can have significant bearing on the precision of the WCET estimate and also on the overheads associated with tracing.
- We demonstrate that, for the programs under analysis, our WCET estimate bounds the actual WCET once basic block[2] coverage has been satisfied. In contrast, the HWMT obtained by using a **Genetic Algorithm** (GA) as the test driver does not manage to bound the actual WCET

The rest of the paper is organised as follows. Section II discusses mechanisms available to extract timing traces, describes existing options to instrument a program, and outlines our assumptions. Next, Section III provides more details about the IPG and the IPET, including a thorough examination of the limitations. The evaluation framework and results are presented in Section IV. A comparison with previous work appears in Section V, before conclusions are drawn in Section VI.

II. Tracing and Instrumentation

Our HMBA relies on the ability to generate a sequence of tuples (i, t) — called a timing trace — where i is the identifier of an ipoint and t its time of observation. The set of timing traces are collated into a **trace file**. How a timing trace is generated and stored depends on the type of tracing mechanism available:

- On-target: A tailored ipoint routine is inserted into the source/object code at particular locations. Upon execution an ipoint causes the target to produce a timestamp; the timing trace is stored in a memory buffer to be downloaded on program completion. The advantage of this approach is that porting to new architectures is relatively straightforward. However, because ipoints are compiled into the executable, the **probe effect** manifests: normal (i.e. without instrumentation) register and cache usages are displaced, a timing penalty is incurred, and overall code size increases.
- External capture device: An ipoint still exists in the executable but it writes its identifier to an I/O port. The port is monitored by an external capture device, e.g. a logic analyser, which timestamps ipoints off target as they appear and also serves to store the timing trace. Penalties associated with the probe effect are minimised because the ipoint routine can be reduced to a few instructions. However, the target must have available and accessible pins to emit the data, which is not always practical with more advanced CPUs. It may also be necessary to disable the cache so that the data written by the ipoint routine is observed on the bus.
- Debug interface: Nexus [9] and Embedded Trace Macrocell [10] technologies trace program flow discontinuities, i.e. conditional and unconditional jumps, in a transparent fashion. That is, the probe effect is completely eliminated. Timestamps are generated on target and the trace data are

[1]We emphasise that the term ''instrumented" umbrellas software and hardware probes; the exact means by which timing traces can be extracted are explored in detail in Section II.

[2]Basic blocks consist of consecutive instructions in which flow of control enters at the beginning and leaves at the end.

either written to an on-chip trace buffer for subsequent download, or exported directly in real time through an external port. However, bandwidth remains the major technical obstacle because the port or debugger must keep pace with the rate at which trace data are produced; otherwise, blackouts arise in which parts of a timing trace are overwritten and essentially lost.

- Simulation: Cycle-accurate simulators, such as SimpleScalar [11], allow individual instructions to serve as ipoints whereby the simulator provides the timestamp. The timing trace is written to a file on the host, which is a distinct advantage given the storage capabilities of desktop computers. Furthermore, the probe effect is eliminated and there are no issues with blackouts. The downside is that creating a cycle-accurate simulator reduces to constructing a precise SA processor model, which can be difficult, or even impossible, particularly for advanced CPUs. (This is due to unpredictable hardware accelerators and imprecise manuals.)

The tracing mechanism only serves to gather a timing trace: it is the job of an **instrumentation profile** (iprofile) to determine *where* the ipoints should reside. Here we review three common iprofiles applicable to the **Control Flow Graph** (CFG) of a program, which are later used in Section IV to evaluate our HMBA:

1) In many techniques proposed by SA and HMBA, the basic block is the atomic unit of computation because the CFG is the program model derived from the disassembly. The **basic block** iprofile therefore considers the first instruction in a basic block as an ipoint: it has previously been used in the evaluation of another HMBA technique [12]. It is the most coarse-grained iprofile (besides tracing every instruction) and is easily supported by simulation.
2) As noted above, hardware debug interfaces trace conditional and unconditional jumps. To mimic this tracing solution, the **branch** iprofile uses each branch target as an ipoint. This iprofile is primarily of interest due to the elimination of the probe effect.
3) The pre-dominator tree of a CFG models the pre-dominance relation between basic blocks: a basic block u pre-dominates a basic block v if all paths from the entry of the CFG to v pass through u [13]. The **pre-dominator** iprofile [14] instruments the leaves of the pre-dominator tree. It hails from the coverage domain where the aim is to reduce trace file size whilst still enabling coverage metrics to be measured; hence, it is typically sparser than either of the basic block or branch iprofiles.

A. Assumptions

Our HMBA assumes the following with respect to tracing and instrumentation:

1) The component can generate a timing trace. Either the supplier inserts ipoints into the source/object code using an iprofile, or the client has access to a debug interface; the latter is preferred due to elimination of the probe effect.
2) Identifiers of ipoints are unique. When an ipoint identifier corresponds to an address, the assumption clearly holds. Alternatively, it can easily be enforced by a tool, e.g. `RapiTime` [15], which automatically inserts ipoints into the program.
3) There are two identified ipoints which delimit the start and end of a new program run. These delimiters allow our analysis to deduce where each new run begins in the trace file (otherwise it will consider the entire trace file as a single execution) and also enable the HWMT to be extracted.

It is also important to stress that *this work does not attempt to quantify the impact of the probe effect on WCET estimates*.

III. System Model

The problem addressed in this paper is how to compute a WCET estimate for a program comprised of a number of components, assuming that neither the source code nor the disassembly of these components are available. As detailed in the previous section, we assume that a tracing mechanism is available to extract a trace file from the instrumented program. This section describes how our HMBA calculates a WCET estimate from the trace file. Section III-A first shows how the IPG is built on the fly, before Section III-B presents how to derive an **Integer Linear Program** (ILP) from the IPG and the timing information, thus resulting in a WCET estimate.

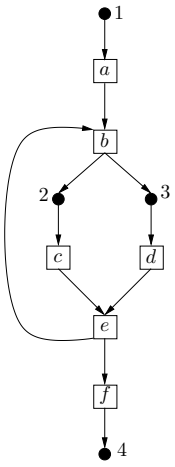
A. Instrumentation Point Graph

The IPG is a program model representing the transitions among ipoints at the intermediate code level. Its vertices are therefore ipoints and each edge contains the functional code executed when a particular transition is followed. In contrast to a CFG, therefore, *the edges* of an IPG are its atomic unit of computation rather than its vertices.

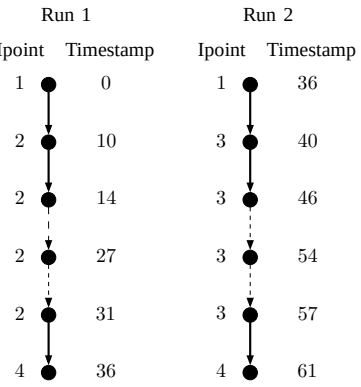
Previous work [7] has shown how to build the IPG *statically* given the CFG and the relative locations of ipoints in the CFG. But clearly this presupposes the ability to disassemble or derive the CFG during compilation from the source code. Here we instead explore how to construct the IPG from a trace file; to this end, we use an illustrative example as the mechanism itself is straightforward.

Consider the example CFG shown in Fig. 1(a), which has been instrumented with the pre-dominator iprofile. Although the pre-dominator tree is not displayed, note that c, d, f are its only leaves since they do not pre-dominate any other basic blocks — these basic blocks have been instrumented with ipoints $2, 3, 4$, respectively. Further observe that a has been instrumented with ipoint 1 and that ipoint 4 appears after the execution of f rather than before it, both of which ensure the instrumentation is compliant with the third assumption detailed in Section II-A.

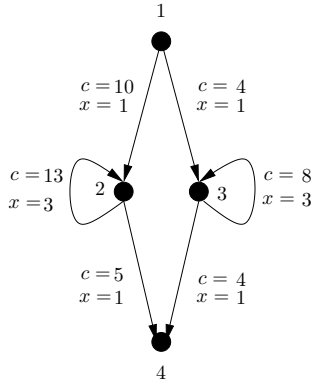
Two different timing traces through the CFG are given in Fig. 1(b), which also shows how the IPG is constructed on the



(a) CFG Instrumented with Pre-Dominator Iprofile.



(b) Two Timing Traces.



(c) IPG with Execution Times (c) and Execution Bounds (x) of Edges.

Fig. 1: Example Used to Demonstrate IPG Construction.

fly during trace parsing. Solid edges depict new transitions inserted into the IPG, whereas dashed edges depict the traversal of an existing transition. Note that, because we assume that ipoint identifiers are unique (see Section II-A), issues with a non-deterministic automaton are avoided; that is, the parser can *always* uniquely determine the exact transition to insert or traverse with a simple one token lookahead scheme. Execution times and execution bounds of IPG edges are also retrieved during this construction. For instance, on the first observation of edge $2 \rightarrow 2$, its WCET is set to $14 - 10 = 4$, which is overwritten on its second observation since $27 - 14 = 13 > 4$; that is, the WCET of an IPG edge is its maximum observed execution time across all timing traces. Similarly, the execution bound of an IPG edge is its maximum number of traversals in any particular run. In this example, three traversals of edge $2 \rightarrow 2$ are observed in the first run, which is committed when ipoint 4 is observed, as this is the ipoint delimiting successive timing traces. The resultant IPG is shown in Fig. 1(c), together with the execution time (c) and the execution bound (x) of each edge obtained from trace parsing.

There are a couple of issues with construction of the IPG in this manner that warrant further discussion. First, there is a reliance on the test harness to sufficiently stress program execution such that the data (IPG edges, execution times, execution bounds) extracted from trace parsing are reliable. In general, any property derived from timing traces which is subsequently used in the WCET calculation can potentially cause underestimation. For example, observe in the CFG of Fig. 1(a) that ipoint transitions $2 \rightarrow 3$ and $3 \rightarrow 2$ are structurally possible but, unless triggered by the test harness and subsequently written to a timing trace, will not be inserted into the IPG. Complicating the matter still further, transitions $2 \rightarrow 3$ and $3 \rightarrow 2$ might not be possible when considering the semantics of the code. However, timing coverage is an issue for *any* approach based on measurements and the exact conditions under which underestimation arises in HMBA is an open research question which is outside the scope of this paper. It is important to emphasise that the comparison in this paper is between end-to-end approaches and our grey-box HMBA; in this respect, Section IV demonstrates that the WCET estimate produced by the latter benefits much sooner from simple coverage metrics than does the former.

Second, information concerning the loops — and their nesting relation — in the IPG is missing since the trace parser only builds the IPG structure. Devoid of loop-nesting data, the trace parser cannot detect when a particular loop has iterated *relative* to an outer loop, nor when it exits. The implication is that only a *global* execution bound on IPG edges can be retrieved, and not one *local* to an outer nested loop, potentially leading to overestimation in the WCET estimate (discussed further in Section III-B). It would appear that the problem can be circumvented by identifying loops as new edges are inserted into the IPG. Previous work [7], however, has shown that the IPG resulting from arbitrary instrumentation often contains **irreducible** loops. These are basically loops with multiple entries which complicate the formation of nesting relations between cycles[3]. Although algorithms have been presented to handle irreducibility [17], [18], each of them could produce different loop-nesting data for the same IPG because there is no coherent view of what constitutes a loop in an arbitrary flow graph [16]. The upshot is that the computed WCET estimate is sensitive to the chosen loop-detection algorithm: underestimation is possible because an alternative algorithm could later be developed that results in a larger WCET estimate. In essence, *any* WCET analysis relying on such an algorithm to handle irreducibility cannot ascribe a guarantee to the WCET estimate since the cyclic properties of the program model cannot be formally proven. This is not an issue when global execution bounds are utilised, however, since the bound is relative to a single execution of the program and not to an outer loop.

[3]See [13], [16] for a detailed discussion of irreducibility.

B. The IPET

The trace-parsing stage produces the structure of the IPG and derives the execution times and execution bounds of its edges: the calculation engine is then tasked with producing a WCET estimate from these data. We use the IPET — which is basically an objective function to be maximised subject to a number of constraints — since it can easily model arbitrary control flow and is not therefore hindered by the irreducibility of an IPG. Puschner-Schedl [3] showed how to build such a model with an ILP, although recent work [19] has considered constraint logic programming as an alternative. Here we briefly describe the ILP with respect to an IPG $I = \langle \mathsf{I}, E \rangle$, where I is its set of ipoints and $E \subseteq \mathsf{I} \times \mathsf{I}$ its set of edges.

The objective function, to be maximised, is $Z = \sum_{u \to v \in E_I} wcet(u \to v) * f(u \to v)$, where $wcet(u \to v)$ is the longest observed execution time of the IPG edge $u \to v$ as obtained from the trace parser, and $f(u \to v)$ is its non-negative execution count. Program structural constraints represent the basic structure of the IPG, effectively preserving flow at each vertex. Capacity constraints, on the other hand, bound both the minimum and maximum execution count of each IPG edge, otherwise the maximisation of the objective function is ∞ since every $f(u \to v)$ can be assigned ∞. In our model, upper capacity constraints are execution bounds derived from trace parsing. Solving this model via standard (integer) linear program solvers returns both a WCET estimate and a setting of the execution count for each IPG edge in the worst case. In this way, all paths are implicitly considered since the solver attempts different assignments to the execution counts in determining the worst case [20]. In general, we cannot determine the *exact* longest path from the execution counts because the order of execution is missing.

When the execution times and upper capacity constraints on the decision variables are safe, the solution to the ILP always returns an upper bound on the actual WCET. As proven by Puschner-Schedl, however, initial reduction to a circulation problem does not precisely characterise the set of execution paths through the flow graph — in this case the IPG — potentially leading to overestimation.

In particular, the ILP models a number of "self-contained" circulations, i.e. loops. As the ILP solver can satisfy all upper bounds on capacity constraints simultaneously, an inner circulation C becomes disconnected from the longest path selected through its outermost circulation *unless* this path always includes C. More formally, the flow graph induced by the execution counts is not strongly connected, and the path is therefore structurally infeasible. For example, in Fig. 1(c), there are two inner circulations, $2 \to 2$ and $3 \to 3$, which are enclosed in the outer circulation encapsulated by the edge $4 \to 1$. Observe that the longest *acyclic* path through the outer circulation is $1 \to 2 \to 4$; therefore, when the ILP solver also satisfies the upper capacity constraint of 3 on $2 \to 2$, i.e. $f(2 \to 2) = 3$, no pessimism arises since 2 appears on this path. However, the solver will also set $f(3 \to 3) = 3$ because it does not violate any other constraint and it must maximise the objective function. Clearly, overestimation ensues because the paths through 2 and 3 cannot be taken simultaneously.

Puschner-Schedl termed this the **disconnected circulation problem** and proved that relative capacity constraints are its solution. A relative capacity constraint models the execution count of a variable with respect to its enclosing loop, rather than, or in addition to, the global execution count. However, as discussed in Section III-A, the simple construction of the IPG by the trace parser does not provide loop information and we only have access to global execution counts. This is one source of overestimation in our model, and its severity depends on the shape of the IPG. We believe this can be overcome by guaranteeing reducibility in the IPG generated from an iprofile, for example, by positioning ipoints in the headers of all loops. This allows the trace parser to derive the necessary local loop bounds, which can be fed accordingly into the ILP — how this is realised is considered beyond the scope of this paper.

IV. Evaluation

This section is split into three parts: Section IV-A explains our experimental framework, Section IV-B describes the set of programs under analysis, and finally, Section IV-C disseminates our results.

A. Experimental Set-up

Our HMBA requires timing traces. For this purpose, we employed the SimpleScalar toolset [11] because such information can be extracted in a relatively straightforward manner. The exact stages of our framework are depicted in Figure 2 for which a detailed description now ensues[4].

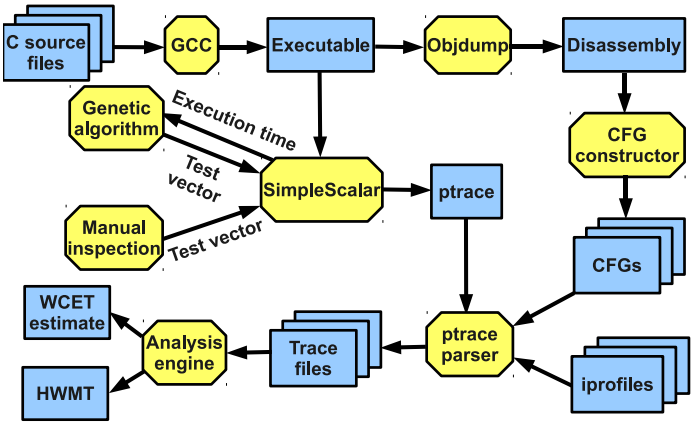


Fig. 2: Overview of Experimental Framework.

Each program under analysis was compiled using a GCC cross compiler that produces executables for the Portable Instruction Set Architecture (PISA, the SimpleScalar derivative

[4]Note that the steps corresponding to disassembling and instrumentation are only of relevance to this evaluation as they are our means to obtain timing traces.

of the MIPS-IV instruction set); level two of optimisation (i.e. using the $-O2$ flag) was selected to increase the speed of execution on SimpleScalar. The binary was then loaded onto `sim-outorder`, the SimpleScalar utility that simulates an out-of-order processor. Our configuration of the CPU was very simple, comprising instruction and data caches of 128 bytes each and a single-fetch in-order pipeline. Therefore, there is little jitter in execution times due to hardware effects, and deducing the worst-case **Test Vector** (TV) for each benchmark was subsequently straightforward.

We used a GA to generate TVs for the binary since previous work [21]–[23] has shown its relative success in generating a HWMT which lies in close proximity to the actual WCET. Therefore, we could compare our WCET estimate with the HWMT produced by a state-of-the-art end-to-end approach. Our implementation of the GA uses two-point crossover, elitism, a crossover rate of 0.9, and a mutation rate of 0.01. The fitness function in the GA is the execution time as returned by `sim-outorder`, which allows for quick evaluation of the fitness of each chromosome. The number of generations and the size of the population varied according to the particular benchmark (see Section IV-B).

Each run of the executable on `sim-outorder` with a single TV produces a `ptrace`, which is a detailed time-stamped history of each instruction as it progresses through each pipeline stage[5]. When the TV produced by the GA is *unique*, i.e. it has not previously been generated, our framework manipulates the `ptrace` to output three timing traces, one each for the iprofiles detailed in Section II; each timing trace is subsequently concatenated onto its corresponding trace file. This means that we avoid duplicating timing traces in the trace file since the GA often generates congruent TVs, particularly as evolution increases.

Clearly, in order to transform the `ptrace` into timing traces, we needed the CFGs of the program under analysis to know which addresses correspond to ipoints. This was done by disassembling the binary (using the `objdump` utility) and then constructing the CFGs from the disassembly. Note that we would normally assume the instrumentation step has already been undertaken, or that there is access to a hardware debug interface or simulator, since the central assumption of this paper is that disassembling of the object code is forbidden.

Upon completion of testing, the trace files generated for each iprofile were passed to the analysis engine. As detailed in Section III, processing of a single trace file results in on-the-fly construction of the IPG, extraction of the data needed in the WCET calculation, and generation of the ILP. In order to plot how the HWMT and our WCET estimate changes with an increasing number of TVs, we performed a WCET calculation after *each* timing trace and not only when the entire trace file had been processed. A capacity constraint to the ILP was added for each cycle-inducing edge in the IPG with the maximum execution count as retrieved at that point by the trace parser. In typical WCET analyses, these edges are those identified by a loop-detection algorithm. But as the model detailed in Section III does not detect IPG loops, a capacity constraint is added for any IPG edge categorised as a back edge during a depth-first search.

Finally, for each of the benchmarks, we determined the worst-case TV by manual inspection in order to extract the actual WCET.

B. Benchmarks

The benchmarks under investigation are taken from the Mälardalen suite [8], which are used by many groups in WCET analysis to evaluate their tools. In this evaluation they are particularly appealing since the worst-case TVs are easy to deduce.

Due to space restrictions, we cannot present the analysis of every benchmark in the Mälardalen suite; Table I instead lists the chosen programs under analysis, together with properties relevant to the test stage. Programs `bubblesort` and `insertsort` were selected because previous work [23] has shown the difficulty encountered by a GA in finding its worst-case TV (i.e. the reverse-sorted array). The choice of `janne_complex` was motivated by its complex input-dependant execution behaviour [24] and, similarly, `expint` contains an inner loop which only triggers when its second parameter is 1, potentially complicating the unguided search of a GA. For the sorting programs, the size of the population and the number of generations were both set to 100 because they ensure sufficient diversity in the TVs without significantly impeding the turnaround time of the test stage; in addition, they are generally suited to a wide range of optimisation problems [25]. These parameters were reduced to 25 each for `expint` and `janne_complex` because their input space is much smaller. Using a $100 * 100$ evolution will likely cause exhaustive input-space coverage and therefore discovery of the actual WCET, which is unrealistic in the general case as testing typically covers a small subset of the input space. By reducing the evolution process to $25 * 25$, the testing set-up is more realistic, maintaining a reasonable ratio between the covered input space and the overall input space. Table I also presents the number of *unique* TVs that the GA produced in comparison to the total number of TVs.

Table II displays properties of interest with respect to the iprofiles. Observe that the basic block (respectively predominator) iprofile is the densest (respectively sparsest), which is also reflected in the trace file sizes. Further note that each iprofile requires ipoints positioned at the start and end the program in order to determine the HWMT — this explains why the basic block iprofile always has $n + 1$ ipoints, where n is the number of basic blocks, because of the additional

[5]It is important to stress that, for each generated TV, the binary had to be reloaded onto `sim-outorder`, the implication of which is that the initial hardware state is always the same. This is unrealistic in an industrial setting where a test harness typically cycles through its TVs, invoking the program without resetting the hardware; that is, the hardware state remaining from one run is inherited by the next run. Our intention was to simulate such an environment, but fundamental limits with SimpleScalar prevented its implementation, namely the inability of `ptrace` to handle large pipeline trace files.

TABLE I: Properties of Benchmarks Selected for Analysis.

Program	#Parameters	Input Range	#Generations	Population Size	#Unique Test Vectors	#Total Test Vectors
`bubblesort`	10	[1..100]	100	100	1017	10,000
`expint`	2	[0..100]	25	25	39	625
`insertsort`	10	[1..100]	100	100	949	10,000
`janne_complex`	2	[1..30]	25	25	34	625

TABLE II: Instrumentation Properties for Benchmarks: Trace File Sizes and Number of Ipoints.

Program	#Basic blocks	Iprofile					
		Basic block		Branch		Pre-dominator	
		Trace File Size (bytes)	#Ipoints	Trace File Size (bytes)	#Ipoints	Trace File Size (bytes)	#Ipoints
`bubblesort`	9	1,898,234	10	1,468,974	7	1,051,258	6
`expint`	34	95,476	35	82,642	23	32,506	18
`insertsort`	7	1,033,058	8	633,854	6	522,704	4
`janne_complex`	11	19,461	12	14,807	10	8,970	8

ipoint at the end of the program (see the third assumption in Section II).

C. Results

Our results are presented in Table III which shows: the HMWT obtained by the GA; the actual WCET obtained by executing the program with its worst-case TV (derived by hand); the percentage of optimism in the HWMT (with respect to the actual WCET); the WCET estimate computed from each iprofile; and the percentage of pessimism in the WCET estimate (with respect to the actual WCET). A graphical representation of how the HWMT and the WCET estimate change during testing appears in Figure 3. Note that the units of time in Table III and Figure 3 are simulation cycles.

There are five interesting observations from these results. First, the GA did not manage to expose the actual WCET for any of the benchmarks, although the margin of underestimation is minimal. This underlines the fragility of any end-to-end approach: it is completely dependent on the test harness finding the worst-case TV.

Second, our WCET estimates bound the actual WCETs very quickly, irrespective of the iprofile. In the case of `bubblesort` and `insertsort`, the actual WCET is bounded immediately, after the first TV. On the other hand, for `expint` and `janne_complex`, this occurs after the third and second TV, respectively. On further investigation, we discovered that underestimation occurs in these instances because some of the basic blocks remain uncovered; as soon as 100% basic block coverage is attained, the WCET estimate bounded the actual WCET. This is precisely the benefit of HMBA: rather than rely on TVs to trigger long end-to-end execution times, the measured execution times of small program segments can be pieced together with path-specific information. Therefore, the test harness need not find a TV causing all loops to iterate through their maximum bound *simultaneously* whilst also, at the same time, trigger the WCET of blocks of code in the *same* run. By using a program model, HMBA instead multiplexes this information from the set of timing traces before carrying out the WCET computation. Obviously, coverage remains an issue since the units of computation must be stressed adequately enough to represent worst-case behaviour, although how this is realised is an open research question.

Third, the WCET estimates computed from the basic block and branch iprofiles are equivalent for each benchmark. However, the key difference is the latter is generally sparser than the former, as demonstrated by the trace file sizes. For example, Table II shows that, for `insertsort`, there is a 38.6% reduction in trace file size between the basic block and branch iprofiles. This is an important consideration in HMBA, given that trace parsing is often its biggest bottleneck [7]. Therefore, these results suggest that the sparser branch iprofile reduces overheads — including the probe effect (if any) — without impacting the accuracy of the WCET estimate.

Fourth, there is considerable overestimation given that these programs are small and simple. For example, the WCET estimate of every benchmark suffers from at least 50% pessimism for both the basic block and branch iprofiles. Two reasons underpin this problem. First, as detailed above, the ILP only contains capacity constraints for cycle-inducing edges in the IPG, and not for *every* edge. Including the latter would provide the most accuracy as the feasible execution paths through the IPG would be further constrained, although it is more susceptible to underestimation because the data are collected entirely from measurements. Second, after manually inspecting the ILPs generated for the IPG in conjunction with the trace file, we found that there was significant overestimation in the execution times of IPG edges: our ILP simply models a *single* execution time for each IPG edge, thereby ignoring different execution times arising from, for example, cache effects. In theory, the model could include the execution time profile [26] of each IPG edge; that is, a variable in the ILP for each of its observed execution times together with respective capacity constraints. Marref [27] has presented a mechanism to deduce these type of constraints from timing traces and how to subsequently model them in a constraint logic programming model. Obviously, this burdens the test harness further because it must ensure that the execution time profile is representative of its worst-case behaviour. (Note that the lack of relative capacity constraints in the ILP, as highlighted in Section III-B, is not a cause of overestimation

TABLE III: Analysis Results for Benchmarks.

Program	HWMT	Actual WCET	Optimism	Iprofile					
				Basic block		Branch		Pre-dominator	
				WCET Estimate	Pessimism	WCET Estimate	Pessimism	WCET Estimate	Pessimism
`bubblesort`	1008	1028	2.0%	1818	76.8%	1818	76.8%	2310	124.7%
`expint`	10721	10956	2.1%	16644	51.9%	16644	51.9%	12414	13.3%
`insertsort`	1113	1175	5.3%	1799	53.1%	1799	53.1%	2124	80.7%
`janne_complex`	367	398	7.8%	743	86.7%	743	86.7%	705	77.1%

for these benchmarks. We checked the IPGs induced by the execution counts and discovered that all were structurally feasible paths.)

Fifth, the WCET estimate produced by the pre-dominator iprofile is larger than the basic block and branch iprofiles for `bubblesort` and `insertsort` but smaller for `expint` and `janne_complex`. Since the *same* TVs were used to produce each trace file, the only conclusion to be drawn is that the iprofile affects the precision of the WCET estimate. We therefore examined the execution times of IPG edges and their execution counts (as set by the ILP solver) since these are the only variables between different iprofiles. The tighter WCET estimates for `expint` and `janne_complex` are explained by more execution context being included on IPG edges. That is, since the pre-dominator iprofile is sparser in terms of ipoints, basic blocks appear on *multiple* IPG transitions and the WCETs of basic blocks benefit from the execution history incorporated on those edges. For example, the WCET of basic block b in Figure 1 is likely to be higher on transition $1 \rightarrow 2$ than $2 \rightarrow 2$ because of cache misses. In contrast, the basic block and branch iprofiles incorporate less execution history in their units of computation (because they are denser) and are more pessimistic as a result. The looser WCET estimates for `bubblesort` and `insertsort`, on the other hand, arise due to insufficient modelling in the ILP. We summed the execution counts of *basic blocks* by observing on which IPG edges they appear. This revealed that the ILP of the pre-dominator iprofile returned over-approximations, whereas those returned for the basic block and branch iprofiles were precise. These findings suggest that sparsity of instrumentation is desirable, to reduce tracing overheads and to allow better accuracy in the units of computation, but that positioning of ipoints with respect to program structure is crucial to avoid problems in the ILP. Closer investigation of this issue, however, is considered as future work.

V. Related Work

A comparison with SA is not relevant here since it requires the source code or the disassembly to build the complete program model, both of which are assumed absent in this paper in particular components of the RTS. This section therefore reviews end-to-end approaches and HMBA paradigms which are the closest to our approach.

Wegener *et al.* [22], [23] first proposed using a GA to stress the end-to-end execution time of a program. Their results indicate that the obtained HWMT lies closer to the actual WCET than does the WCET estimate computed by a SA technique, but that underestimation exists. One drawback is that the fitness function only considers the execution time, disregarding other factors which could later lead to a long execution time, such as the number of cache misses. Khan *et al.* [21] considered including multiple criteria, e.g. hardware effects and loop bounds, into the fitness function with a view to recommending which criteria suit which systems. Ermedahl *et al.* [24] have investigated generating the worst-case TV for a particular program. Our approach also uses testing to trigger long execution times, but the WCET estimate is derived from a subsequent calculation operating on a program model, and is therefore not reliant on the test harness finding the absolute worst-case TV, as our results demonstrated.

Similarly, Williams [28] employs a form of limited path coverage [29] using `PathCrawler` [30] to find the actual WCET of the program, suffering from obvious scalability problems. Our work avoids scalability issues by instead deriving execution times and execution counts of program segments before combining this information globally to obtain the WCET estimate.

In terms of HMBA, three works are closest to our approach. First, Bernat *et al.* [26], [31] obtains an Execution Time Profile (ETP), rather than an integer value, for each program segment. An ETP is obtained through measurements and represents the frequency of execution for a particular execution time. The calculation engine is then able to combine dependent ETPs that occur from hardware effects which have not been captured in the measurement stage. Second, Colin *et al.* [12] generate timing traces through SimpleScalar, parse them to derive WCETs of basic blocks, and finally compute a WCET estimate using the standard timing schema [2] of an Abstract Syntax Tree (AST). Third, Marref [19] has presented predicated WCET analysis, which deduces execution time effects between basic blocks from timing traces. These execution time effects are later injected into the calculation as constraints on execution paths, consequently tightening the WCET calculation. Our work differs from all three: the Bernat and Colin approaches need very specific ipoint placement and are restricted to source-level analysis due to adoption of the AST; whereas basic blocks serve as the unit of computation in the Marref method and, consequently, the CFG is the underlying program model for which the disassembly is needed.

VI. Conclusion

Composing a real-time system out of software components is a pervasive area of interest, both in academia and in industry. When a component is shipped to a client, however,

source code is withheld and the relocatable object code cannot usually be disassembled, to protect intellectual property. Under these circumstances, state-of-the-art WCET analysis tools and techniques are neutralized because they are unable to construct a program model.

This paper showed how to derive a safer WCET estimate than end-to-end approaches using hybrid measurement-based analysis. Rather than statically derive any data needed in the WCET calculation, they are instead gathered during parsing of time-stamped traces of execution which have been produced by an instrumented program. How these traces are produced is orthogonal: our approach does not place any restrictions on the type of instrumentation deployed and thus supports probe-free tracing as supplied by hardware debug interfaces. However, we evaluated the accuracy of the WCET estimate when different instrumentation profiles are deployed. Our results indicate that achieving minimal coverage, i.e. all basic blocks, is sufficient to bound the actual WCET for the programs under analysis, whereas a state-of-the-art end-to-end approach failed to do so. We also found that branch tracing mechanisms offer the same precision as basic block tracing mechanisms but with reduced trace overheads. Furthermore, simple changes in the position of instrumentation points can dramatically change the accuracy of the WCET estimate.

Future work will investigate ways in which to reduce the pessimism by modelling the execution times and execution counts of program segments in a more powerful way. As the position of instrumentation is also key to the analysis, we intend to consider where instrumentation points should be placed to ensure more accurate WCET estimates. Finally, we will consider more rigorous coverage metrics than those proposed in the functional domain, given their inability to take timing into account.

ACKNOWLEDGEMENTS

This work is supported by the Swedish Foundation for Strategic Research (SSF) through the Research Centre for Predictable Embedded Software Systems (PROGRESS).

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem–Overview of Methods and Survey of Tools," *ACM Transations on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, 2008.

[2] C. Y. Park and A. C. Shaw, "Experiments With A Program Timing Tool Based on Source-Level Timing Schema," *IEEE Computer*, vol. 24, no. 5, pp. 48–57, May 1991.

[3] P. Puschner and A. Schedl, "Computing Maximum Task Execution Times - A Graph-Based Approach," *Real-Time Systems*, vol. 13, no. 1, pp. 67–91, 1997.

[4] F. Stappert, A. Ermedahl, and J. Engblom, "Efficient longest executable path search for programs with complex flows and pipeline effects," in *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems (CASES'01)*. New York, NY, USA: ACM Press, 2001, pp. 132–140.

[5] F. Lüders, S. Ahmad, F. Khizer, and G. Singh-Dhillon, "Use of software component models and services in embedded real-time systems," in *Proceedings of the* 40^{th} *Hawaii International Conference on System Sciences*, January 2007.

[6] J. J. Chilenski and S. P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, September 1994.

[7] A. Betts, "Hybrid Measurement-Based WCET Analysis using Instrumentation Point Graphs," Ph.D. dissertation, University of York, November 2008.

[8] Mälardalen University WCET project homepage, http://www.mrtc.mdh.se/projects/wcet, May 2010.

[9] The Nexus 5001^{TM}Forum, http://www.nexus5001.org, May 2010.

[10] ARM development tools, http://www.arm.com, May 2010.

[11] D. Burger and T. Austin, "The simplescalar tool set, version 2.0," University of Wisconsin, Madison, Technical Report CS-TR-1997-1342, 1997.

[12] A. Colin and S. M. Petters, "Experimental evaluation of code properties for WCET analysis," in *Proceedings of the 24th Real-Time Systems Symposium (RTSS'03)*, December 2003.

[13] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[14] M. M. Tikir and J. K. Hollingsworth, "Efficient Instrumentation for Code Coverage Testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, July 2002.

[15] Rapita Systems Ltd., http://www.rapitasystems.com/, May 2010.

[16] G. Ramalingam, "On Loops, Dominators, and Dominance Frontiers," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24, no. 5, pp. 455–490, September 2002.

[17] ——, "Identifying Loops in Almost Linear Time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 2, pp. 175–188, March 1999.

[18] V. C. Sreedhar, "Efficient Program Analysis using DJ Graphs," Ph.D. dissertation, McGill University, 1995.

[19] A. Marref, "Predicated Worst-Case Execution-Time Analysis," Ph.D. dissertation, York, UK, 2009.

[20] Y. S. Li, S. Malik, and A. Wolfe, "Efficient microarchitecture modeling and path analysis for real-time software," in *Proceedings of the IEEE Real-Time Systems Symposium*, 1995, pp. 298–307.

[21] U. Khan and I. Bate, "WCET analysis of modern processors using multi-criteria optimisation," in *Proceedings of the* 1^{st} *International Symposium on Search Based Software Engineering (SSBSE'09)*, May 2009.

[22] J. Wegener and M. Grochtmann, "Verifying timing constraints of real-time systems by means of evolutionary testing," *Real-Time Systems*, vol. 15, no. 3, pp. 275–298, 1998.

[23] J. Wegener and F. Müeller, "A comparison of static analysis and evolutionary testing for the verification of timing constraints," *Real-Time Systems*, vol. 21, no. 3, pp. 241–268, 2001.

[24] A. Ermedahl, J. Fredriksson, J. Gustafsson, and P. Altenbernd, "Deriving the worst-case execution time input values," in 21^{st} *Euromicro Conference of Real-Time Systems (ECRTS'09)*, July 2009.

[25] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1989.

[26] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *Proceedings of the* 23^{rd} *Real-Time Systems Symposium (RTSS'02)*, December 2002.

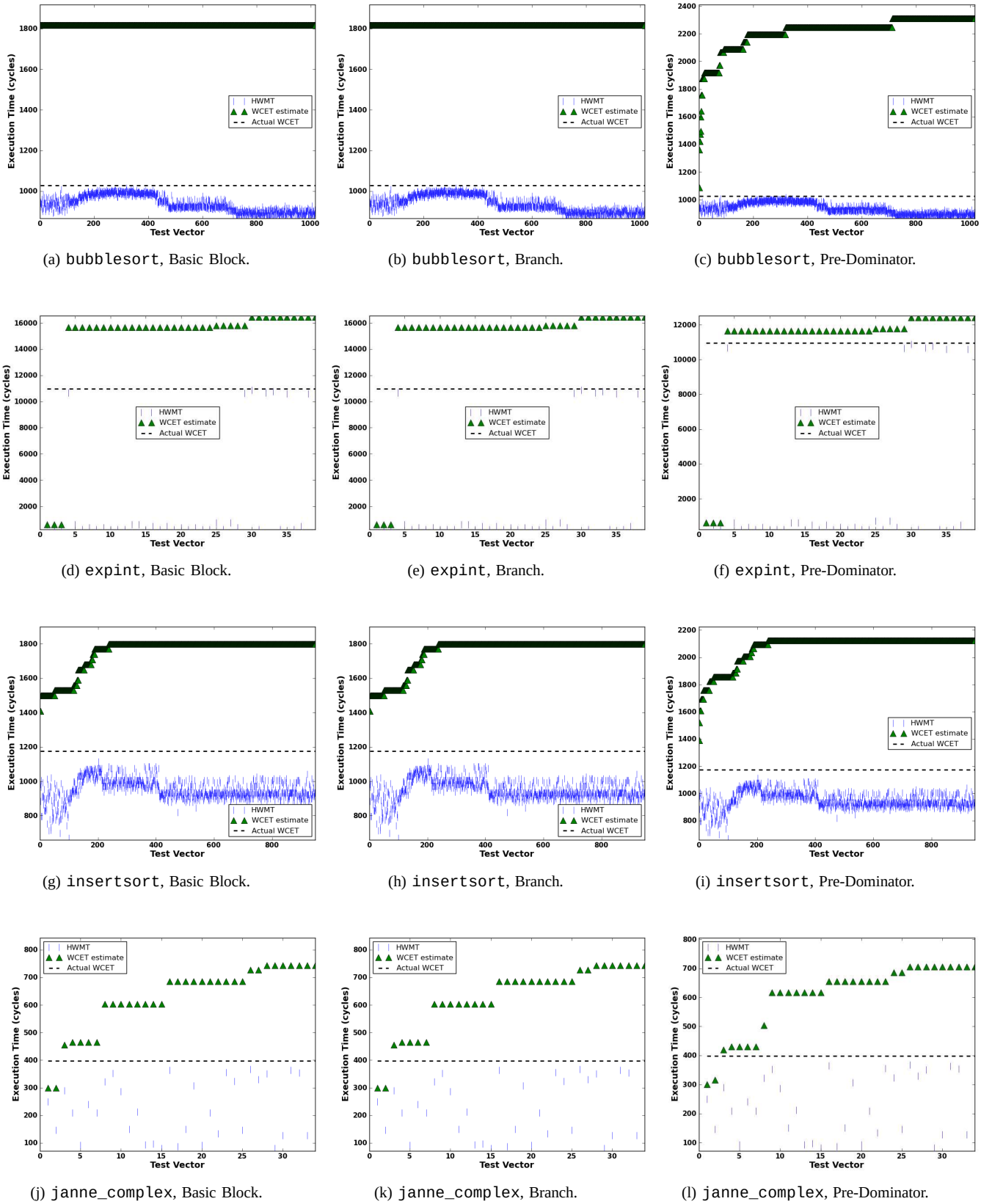
[27] A. Marref and G. Bernat, "Towards Predicated WCET Analysis ," in *Proceedings of the* 8^{th} *International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2008.

[28] N. Williams, "WCET measurement using modified path testing," in *Proceedings of the* 5^{th} *International Workshop On Worst-Case Execution-Time (WCET) Analysis in conjunction with the* 17^{th} *Euromicro International Conference on Real-Time Systems*, 2005.

[29] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[30] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *Proceedings of the* 5^{th} *European Dependable Computing Conference (EDCC'5)*, 2005, pp. 281–292.

[31] G. Bernat, M. Newby, and A. Burns, "Probabilistic timing analysis: an approach using copulas," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 179–194, 2005.



(a) `bubblesort`, Basic Block. (b) `bubblesort`, Branch. (c) `bubblesort`, Pre-Dominator.

(d) `expint`, Basic Block. (e) `expint`, Branch. (f) `expint`, Pre-Dominator.

(g) `insertsort`, Basic Block. (h) `insertsort`, Branch. (i) `insertsort`, Pre-Dominator.

(j) `janne_complex`, Basic Block. (k) `janne_complex`, Branch. (l) `janne_complex`, Pre-Dominator.

Fig. 3: How the HWMT and WCET Estimate Change as the Genetic Algorithm Evolves Test Vectors for each Benchmark, Iprofile Combination.