Forming Virtual Traces for WCET Analysis and Reduction

Jack Whitham and Neil Audsley
Real-Time Systems Group
Department of Computer Science
University of York, York, YO10 5DD, UK
jack@cs.york.ac.uk

Abstract

It is notoriously difficult to model superscalar out-of-order CPUs for the purposes of worst-case execution time (WCET) analysis, which can force the use of simpler CPUs in hard real-time systems. To address this problem, it has been suggested that traces could be used to capture the timing properties of a complex CPU operation scheduler as it runs a sequence of basic blocks. In previous work, traces have been implemented using application-specific microcode.

This paper proposes restrictions to a dynamic superscalar out-of-order CPU to implement virtual traces. These have the same timing properties as the traces in previous work, but microcode is not used. Instead, CPU modifications implement the same functionality. This allows traces to be used throughout a program because space requirements are minimal. To take advantage of this, a new allocation algorithm is proposed and evaluated for virtual traces.

1 Introduction

Worst-case execution time (WCET) analysis is an important part of hard *real-time systems* design. It is used within schedulability analysis to ensure that tasks are feasible, i.e. deadlines will always be met. WCET analysis is highly effective when applied to programs running on simple CPU models, but useful results are not so easily obtained for modern CPU designs. Makers of WCET analysis tools must contend with (1) the high cost of making exact models of complex CPUs [6], (2) the possibility of anomalous timing behavior [19], and (3) the problem of making analysis feasible without increasing pessimism [22]. (A *pessimistic* WCET value is an overestimate of the true WCET.)

A key capability that is not well supported by analysis is *out-of-order execution* [5]. Modeling approaches make simplifying assumptions [7] or avoid speculative execution [16] and this introduces pessimism. Architectural modifications are one solution: CPU execution can be constrained to follow *traces* when executing code that makes a significant contribution to the WCET, e.g. loops. Traces implement acyclic paths within a program, and are usually encoded as explicitly parallel operations with assumptions about the execution path, allowing speculative and out-of-order execution [4]. The timing behavior of a trace is precisely known, since speculative execution is planned during compilation [5]. Using traces, a predictable CPU can support out-of-order execution without two of the problems associated with WCET analysis, since (1) the CPU modeling cost is low because trace execution times can be measured directly and (2) there is no possibility of timing anomalies because no dynamic resource allocation decisions are made [19]. In principle, traces provide the twin benefits of *simple and accurate WCET analysis* and *reduced execution time*.

This paper makes two contributions. Firstly, previous work [21] is extended to model *virtual traces*. These provide a new way to implement traces that is (1) easier to accommodate into existing CPU designs, (2) does not require the storage of many thousands of bits of microcode, and (3) provides the benefits of traces throughout a program. The timing analysis model is unchanged from previous work, but the execution model for the program is changed to support the new paradigm. Secondly, a new allocation algorithm is proposed for virtual traces. It assumes that traces can be allocated throughout a program, so that every basic block is part of a trace. This is possible with virtual traces because the storage requirements are minimal. Together, the algorithm and the virtual trace model are used to evaluate the new implementation paradigm; the discussion also describes problems that might be introduced by virtual traces.

The paper is structured as follows. Section 2 summarizes previous work and gives the model of a trace. Section 3 introduces virtual traces. Section 4 proposes and evaluates an allocation algorithm for virtual traces. Section 5 describes related work and section 6 concludes.

2 Traces and WCET Analysis

This section explains the formal model for traces. Previous work [21] proposed that a CPU could be extended with a *trace scratchpad* which operated as a writable control store. Programs could run on this CPU without using the trace scratchpad, but the WCET of a program could be reduced by using the scratchpad to store application-specific microinstructions. An automatic process was proposed for selecting these microinstructions: it reduced the WCET of a program by building traces along paths that

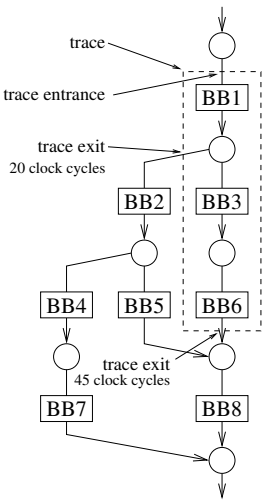

Figure 1. Example of a trace. The trace includes three basic blocks and has two exits. The execution time from the entrance to each exit can be calculated if CPU operation is suitably constrained.

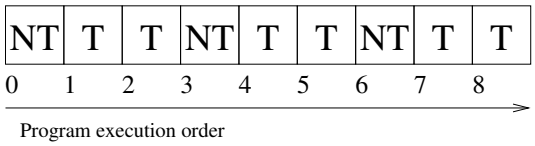

Figure 2. A virtual trace is represented by an encoded sequence of up to L branch predictions. Each item is information about a conditional branch along the main path of the trace, telling the operation scheduler the direction to be optimized. Predictions can be not taken (NT) or taken (T).

were frequently executed in the worst case. The rest of the program was executed as conventional machine code. This paper uses traces in a different way, allocating them throughout a program, but the same model of a trace is retained:

1. A trace replaces sequential machine code in one or more basic blocks, forming part of a path through the program.

2. A trace always begins execution with the CPU microarchitecture in a known state: about to execute basic block e, which is known as the *entrance*. Traces have exactly one entrance, i.e. they are *superblocks* [5].

3. A trace requires a precisely known number of clock cycles to reach each one of the n exits from the entrance. An exit is taken when a branch condition is evaluated as True or the main path's end is reached: exits lead to another trace or machine code execution.

4. The path to exit i from entrance e is denoted as $P_{e,i}$ for WCET analysis purposes: $P_{e,i}$ is a sequence of basic blocks. The time taken is $t(P_{e,i})$.

5. A trace contains up to L conditional branches along the *main path* $P_{e,0}$. Every other path $P_{e,j}$ ($j \neq 0$) also follows this path until conditional branch j is reached. Then, $P_{e,j}$ leads to an exit while $P_{e,0}$ continues. A trace has $1 \leq n \leq L+1$ exits.

6. After any exit, a transformation has been applied to the program state (i.e. the program counter, RAM, and general-purpose registers). This transformation is guaranteed to be identical to the transformation that would have been applied if the original machine code had been executed.

The WCET of a program containing traces can be analyzed using the *implicit path enumeration technique* (IPET) because every $t(P_{e,i})$ is a constant integer. IPET has the advantage of being able to accommodate any linear constraint on execution, and it can produce exact results when all constraints are known [15]. A *timing graph* (T-graph) $G = (V, E)$ is used to represent the program, with E being the set of all basic blocks, and V being the set of all the vertices that link them in the program's control flow. Introducing traces transforms the T-graph of a program, but because each trace is itself composed of basic blocks with fixed execution times, the WCET can still be computed by IPET with a few additional constraints.

However, the WCET can also be *reduced* by careful selection of traces, because speculation and out-of-order execution can be carried out within each trace. Consider Figure 1, where e = BB1, and there are two possible paths: $P_{e,0}$ = [BB1, BB3, BB6] and $P_{e,1}$ = [BB1]. If it is known that the basic block sequence $P_{e,0}$ is executed on the *worst-case execution path* (WCEP), which is the path through the program that produces the WCET, then the WCET may be reduced by speculating that BB1 is likely to be followed by BB3 and BB6, and executing operations from those subsequent basic blocks early. This captures more of the instruction level parallelism inherent in the program [18]; the cost is that paths to other exits ($P_{e,1}$, in this case) will take longer. Similar trace formation techniques have been used for decades to reduce *average case execution time* (ACET). In [21], it was applied within *some* parts of a program to reduce the WCET. In this paper, it is applied *throughout* a program.

3 Introducing Virtual Traces

Traces provide a way to use speculation and out-of-order execution to reduce the WCET of a program, while still permitting analysis by IPET, but they are limited by the microcoded implementation. This section examines a

Trace scratchpad approach (in [21])	Virtual trace approach (in this paper)
Scheduling	
Offline only: microprograms generated before analysis	Online: performed during execution
Trace execution timings ($t(P_{e,i})$)	
Obtained by analyzing microprograms	Obtained by measuring virtual trace execution on CPU
CPU design	
Fully customized design required	Minor changes to existing design required
Extra storage required	
$>$ 100 bits per microinstruction	O(L) bits per virtual trace
WCET analysis method	
IPET	IPET

Table 1. Contrasting approaches.

better way to implement the same functionality, using less space and requiring much less custom hardware.

The *function* of the traces in previous work was to ensure that the activities of the operation scheduler would be known during WCET analysis. There at least two ways to do this. Firstly, one can decide on a schedule before analysis and then enforce that schedule using a custom microprogram: this is the *trace scratchpad approach*. Secondly, one can restrict the inputs to the dynamic operation scheduler within a CPU so that it operates in the same way as an offline operation scheduler. This is the *virtual trace approach*, because traces are not explicitly encoded as microinstructions and are not visible in the program. They exist only for WCET analysis and reduction purposes. The only part of a virtual trace that is actually visible to the CPU is the sequence of branch predictions that encode it (Figure 2). Both approaches require modifications to a CPU but the latter requires fewer changes (Table 1).

An important feature of the virtual trace approach is that the WCET analysis method is shared with the trace scratchpad approach. This is because both approaches use the same timing model (section 2).

3.1 Changes for a CPU core

To implement virtual traces, the behavior of the dynamic operation scheduler in the CPU must be restricted. For the following discussion, the CPU is assumed to be a superscalar out-of-order operation scheduler that makes at most one prediction at each branch. The function of this component is to assign incoming machine code instructions to CPU execution units. In a superscalar CPU, there are several execution units, and the scheduler attempts to keep all of them busy at all times. Previous work has looked at modeling such devices so that the internal behavior of the operation scheduler can be predicted [6, 16, 19]: this work uses restrictions so that the behavior does not *need* to be predicted because it can be expressed in terms of the trace model (section 2). Investigation indicates that there are seven sources of *noise* that can affect the behavior of a typical dynamic scheduler such that execution time might change:

1. The *previous state* of the operation scheduler (due to earlier executions) affects the resource allocations used to execute the next basic block. Execution might be *stalled* by earlier operations that are still being processed, and previous resource allocations might even lead to timing anomalies [19].

2. Executing *variable duration instructions* will affect future resource allocations. This is a data-dependent effect.

3. *Cache stalls* occur when data or instructions are not in cache. These disrupt the pipeline just like variable duration instructions and the previous scheduler state because the scheduler continues to execute instructions as it waits for the stall to complete.

4. *Branch predictions* are produced as operations are fetched, often generated by a heuristic mechanism. They allow the scheduler to make an assumption about the target address of each branch. Regardless of whether a prediction is correct or not, incorporating the information into the scheduler is enough to affect timing.

5. *Branch misprediction squashes* are generated when a branch operation is executed and the associated prediction is found to be wrong. A *squash* event is generated, causing all speculative executions beyond the branch to be discarded. Fetching starts again from the correct address. The time taken to do this depends on what is executing and what has been executed. Branch misprediction squashes are particularly problematic when they occur out of program order because more than one can be active simultaneously.

6. *Memory mispredictions* occur when the scheduler has incorrectly assumed something about memory accesses, e.g. that two store operations access different addresses. The result is a squash event, much like a branch misprediction.

7. *Exceptions* are generated when an instruction cannot be executed, e.g. division by zero or null pointer dereference. The result is something like a branch misprediction, as execution jumps to an exception handler.

The above list is not exhaustive: some CPUs may include additional sources of noise, e.g. memory management subsystems. But all of the sources in the list can be handled by avoidance or by incorporation into analysis.

For example, cache stalls can be avoided by using scratchpads to store instructions [13] and data [17]. In this paper, scratchpads are used for both so that the scheduler can be considered in isolation.

An avoidance strategy is used to handle variable duration instructions and memory mispredictions. It is assumed that all instructions execute in constant time and that memory operations are issued in an order that is guaranteed to be safe by changing the memory disambiguation logic in the CPU, so store operations always execute in program order, and load operations cannot be reordered across a store operation. These restrictions prevent stalls and squashes from those sources. It is also assumed that exceptions do not occur. Three sources of noise remain: the previous state, branch predictions and branch misprediction squashes. These can't be avoided, but they can be handled as follows:

- The *previous state* needs to be *synchronized* to a known state before each virtual trace begins. In the trace scratchpad approach this is assured implicitly by the start of the microprogram. For virtual traces, it has to be assured explicitly by draining the pipeline. At the beginning of each virtual trace, a new hardware component called the *virtual trace controller* (VTC) stalls instruction fetching until the pipeline is empty.

- *Branch predictions* must come from the virtual trace (Figure 2). These predictions describe the main path $P_{e,0}$.

- *Branch misprediction squashes* need to be incorporated into the trace model. Fortunately they are already supported. For every trace, there is one main path $P_{e,0}$ with no mispredictions and $n \leq L$ other paths $P_{e,j}$ with exactly one misprediction ($0 < j \leq n$). The only consequence of this from the perspective of the CPU design is that *branch operations must be executed in program order*, since that will prevent two or more squash events being active at the same time. This can be assured through the existing mechanism that manages dependences between instructions.

The CPU is integrated with the VTC as shown in Figure 3. The VTC is a simple state machine. During normal operation, the VTC has the following functions: (1) ensure that the pipeline is synchronized before the beginning of each virtual trace (by stalling the fetch process), (2) guiding execution along the main path by providing appropriate branch predictions, and (3) fetching the next virtual trace from memory when an exit is reached. The virtual trace is stored within the *virtual trace register* (VTR). During measurement, the VTC is also used to obtain $t(P_{e,i})$ for each virtual trace. A test harness forces execution to begin at a specific address and follow a specific path while the VTC takes measurements.

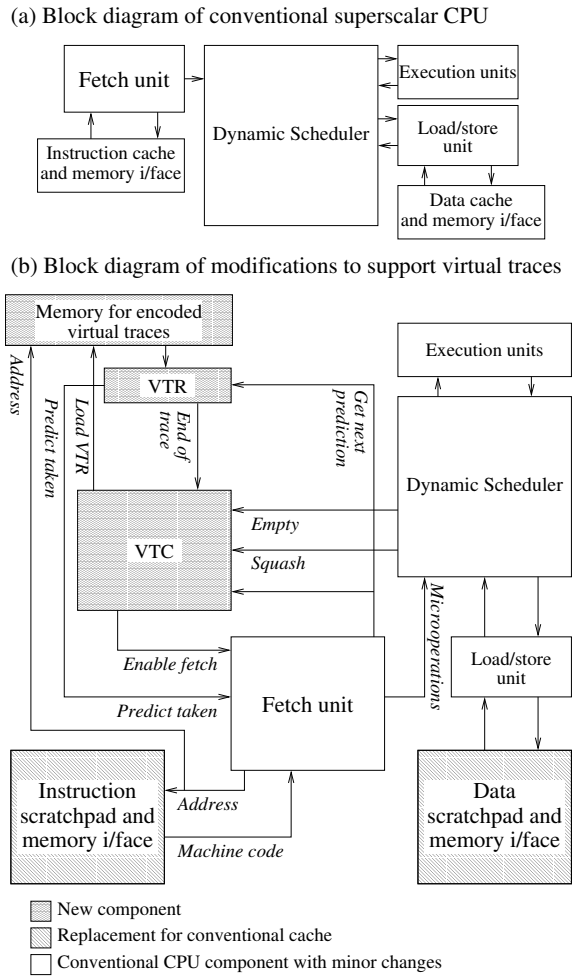

Figure 3. Block diagram of a CPU core, (a) before and (b) after support for virtual traces is added. Virtual traces are implemented by the virtual trace controller (VTC) and temporarily stored in the virtual trace register (VTR).

The key property introduced by these changes is that execution within each trace always follows one of the paths $P_{e,i}$. For all its complexity, the dynamic scheduler is still a deterministic machine. It is made predictable by constraining the inputs and resynchronizing to a known state.

3.2 Concerns about Virtual Traces

Virtual traces appear to be an ideal solution for supporting speculative execution within WCET analysis. However, new problems may be introduced:

- *Reduction in Peak Throughput* - the restrictions to the operation scheduler in a CPU are certain to cause a reduction in the maximum possible throughput. The pipeline needs to be synchronized (i.e. drained) at the beginning of each virtual trace. Additionally, the

changes to memory disambiguation reduce throughput: store operations must execute in program order so that there are no address conflicts. However, virtual traces do not reduce performance. It is true that the changes reduce the peak throughput relative to a conventional CPU design, but peak throughput is not a concern in hard real-time systems design. It is the worst-case throughput, represented by the WCET, that affects the design decisions. The argument is that virtual traces enable the WCET to be reduced to a greater extent than comparable techniques for predictable CPU design, which restrict optimizations to a single basic block [16], assume a simple pipeline model [1], or make heavy use of predication [14].

- *Deterministic Memory Assumption* - cache stalls are eliminated by the use of instruction and data scratchpads. These allow memory requests to complete in a known time period: they are deterministic, so they do not introduce noise into the schedule. Virtual traces will not work correctly with conventional caches unless the whole pipeline is stalled immediately as soon as a cache miss occurs. This would be hard to arrange in a modern CPU design due to the propagation delay of the stall signal. Scratchpad allocation is a challenging problem that is the subject of ongoing work [13, 17].

4 Virtual Trace Formation

Since virtual traces can be used throughout a program, virtual trace allocation is not a constrained optimization problem like trace scratchpad allocation [21]. This section proposes and evaluates a new trace formation algorithm for virtual traces.

Trace formation algorithms require information about the flow of execution within a program. For ACET reduction, this information is obtained using a *path profile* which indicates likely sequences of basic blocks [5]. This is useful for choosing the paths that contribute most to the average execution time. Unfortunately, whenever WCET reduction optimizations are applied, the WCEP may change [17], so the same approach cannot be directly used to optimize the WCEP. Specialized algorithms are required in order to account for this.

4.1 Static Branch Prediction

A trace (virtual or otherwise) includes implicit branch predictions (taken/not taken) for every conditional branch on its path, so algorithms for assigning static branch predictions can solve part of the problem of trace formation. The Bodin and Puaut algorithm [3] (Figure 4) assigns branch predictions to conditional branches in a program G with the goal of minimizing the WCET.

Initially, all conditional branches are assumed to be mispredicted, which means that each one will incur a "misprediction penalty" whether it is taken or not taken.

```
procedure Set_Predictions(G):
    converged = False
    (V, E) = G
    for e ∈ E:
        e.prediction = None
    end for
    while not converged:
     // Step 1: WCET estimation
X       Z = Calculate_WCET(G)
     // Step 2: Issue static branch predictions along the WCEP
        converged = True
        for e_a ∈ E:
            if f(e_a) ≠ 0:
     // Basic block e_a is on the WCEP
                (v_0, v_1) = e_a
                if |{v_2|(v_1, v_2) ∈ E}| > 1:
     // Basic block e_a is followed by a conditional branch
                    for (v_1, v_2) ∈ E:
                        e_b = (v_1, v_2)
                        if f(e_b) > f(e_a.prediction):
                            e_a.prediction = e_b
                            converged = False
                        end if
                    end for
                end if
            end if
        end for
    end while
end procedure
```

Figure 4. Bodin and Puaut algorithm for static branch prediction [3].

During each iteration of the algorithm, the WCEP is evaluated, which produces an execution count $f(x)$ for each edge $x \in E$, indicating the number of times that basic block is executed in the WCEP. Branches along the WCEP that lack a prediction are marked as *taken* or *not taken*, with the goal of minimizing the number of mispredictions on that path. In subsequent iterations, all branches on the WCEP may already have predictions assigned. When this happens, the algorithm stops.

Results reported in [3] indicate that the algorithm converges in one or two iterations for every example tested. WCET reductions of 5% to 21% are obtained relative to the conservative strategy of assuming that conditional branches are always mispredicted. The authors note that the algorithm is not optimal, and no attempt has been made to see how close to optimality it actually is.

4.2 Adapting Static Branch Prediction

The WCET-oriented static branch prediction algorithm (Figure 4) solves part of the problem of trace formation by indicating the direction to be optimized by traces. This section describes how branch predictions can be turned into traces (virtual or otherwise) by adapting the algorithm from Figure 4.

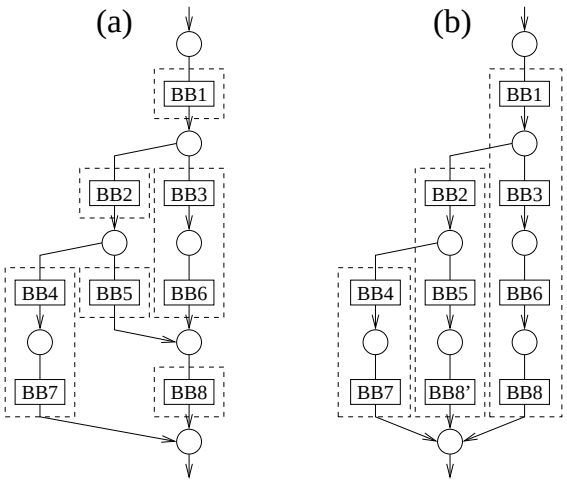

Figure 5. Results of the trace formation algorithm applied to Figure 1, (a) before and (b) after branch predictions are added. (a) shows *trivial traces*, with no more than two exits each. (b) shows longer traces, which include more basic blocks and thus allow more instruction-level parallelism to be exploited, reducing WCET further. To avoid the need to start a new trace at BB8, *tail duplication* is used to create a copy: BB8'.

The simplest way to assign traces to a program is just to map each basic block onto a trace. These would be *trivial traces* (Figure 5(a)), containing at most one conditional branch and having at most two exits. This is identical to the initial state of G in Set_Predictions where branches are always assumed to be mispredicted. As the algorithm iterates, branch predictions are added. These allow the size of each trace to be increased, incorporating multiple basic blocks (Figure 5(b)). This increases the opportunity for instruction-level parallelism in each trace, and also reduces the effects of the overhead introduced by entering or leaving a trace. Consequently, the WCET is reduced.

Figure 6 gives the Form_Traces algorithm, which forms trivial traces at each basic block before predictions become available. Then, as predictions are made by Figure 4, they are used to build longer traces. Each trace formation beginning at basic block e may stop when any of the following conditions occurs: (1) an unpredicted branch is reached, (2) L conditional branches have been processed, or (3) e is reached for the n-th time, and the total number of branches in the loop is large enough that L would be exceeded before e is reached for the $(n+1)$-th time. These formation rules permit loop unrolling and tail duplication. Tail duplication is restricted only by the limit L. The final condition ensures an integer number of loop unrolls.

```
function Form_Traces(G, L):
    (V, E) = G' = copy(G)
     // Get the roots of graph G'
    queue = {(v0, v1)|(v0, v1) ∈ E ∧ ¬∃(v2, v0) ∈ E}
    done = ∅ ; Θ = ∅
    while |queue| ≠ 0:
        x = e = pop(queue)
        if e ∈ done:
            continue  // repeat pop
        end if
        Pe = [ ] ; stop = False ; visited = ∅
        exit_num = size = 0 ; unrolling_to = None
        while stop = False:
     // add this basic block to the paths through the trace
            Pe = Pe + [x]
             // find next basic block
            (v0, v1) = x ; next = {(v1, v2)|(v1, v2) ∈ E}
            visited = visited ∪ {x}
            if |next| = 2:
                 // basic block x is followed by a branch
                if x.prediction = None:
                     // halt: no more predictions
                    stop = True
                else:
                     // predict direction of WCEP
                    x = x.prediction
                    exit_num = exit_num + 1 ; size = size + 1
                    if exit_num = L:
                        stop = True
                    end if
                     // build another trace in the other direction
                    queue = queue ∪ (next − {x})
                end if
            else:
                x = pop(next)
            end if
            if x ∈ visited:
                 // loop detected: check rule 3
                if unrolling_to ∈ [x,None]:
                     // loop entry point
                    if exit_num + size ≥ L:
                         // too many unrolls
                        stop = True
                    end if
                    unrolling_to = x ; size = 0
                end if
            end if
        end while
         // update todo and done sets
        queue = queue ∪ {x} ; done = done ∪ {e}
         // add the trace
        Θ = Θ ∪ {Generate_Trace(Pe)}
    end while
    return G'
end function
```

Figure 6. Algorithm for trace formation.

Form_Traces is applied to G immediately before Calculate_WCET. Line **X** of Set_Predictions (Figure 4) changes to:

Z = Calculate_WCET(Form_Traces(G, L))

The WCET of a program containing traces made by Form_Traces can be determined using the extended IPET model described in [20, 21]. Form_Traces, Calculate_WCET and Set_Predictions can all be applied to virtual traces and microprogrammed traces.

4.3 Experimental Assumptions

The efficacy of virtual traces can be determined by reusing the experimental platform from [21] as it includes all of the components that are needed:

- Generate_Trace: a procedure to obtain $t(P_{e,i})$ for each of the paths through a trace T_e. This procedure is an operation scheduler, generating a microprogram for the trace and obtaining the timings as a side effect. The exact details of this procedure are not important, since it fits the trace model given in section 2.
- Calculate_WCET: a procedure to obtain the WCET Z for a T-graph that includes any number of traces. This procedure carries out WCET analysis using IPET.

In conjunction with Form_Traces and Set_Predictions, these provide everything that is needed to evaluate the new trace formation algorithm by modeling virtual traces. The results include the implicit assumption that the traces will be executed on the MCGREP-2 CPU (described in [20]), since the path timings are obtained by analysis of MCGREP-2 microprograms, but similar results would be obtained for any CPU including a CPU using virtual traces and a VTC.

In order to reuse the MCGREP-2 platform, it must be assumed that the trace scratchpad size is unlimited, so that any number of traces can be built by Form_Traces. Secondly, technical limitations force the assumption that procedure call and return instructions exist in their own traces, preventing traces spanning two procedures. If the restriction were relaxed, greater WCET reductions might be possible.

4.4 Experiments

The experiments use a subset of the Mälardalen WCET benchmarks [10] as sample programs for Set_Predictions, with line **X** changed to pass the program G through Form_Traces (Figure 6) before the call to Calculate_WCET. The Mälardalen benchmarks are used because (1) they include a range of functions that might be implemented on an embedded real-time system, and (2) they are provided with the behavioral constraints required to perform WCET analysis. Hence, Set_Predictions and Calculate_WCET can be used.

The ideal value of L is unknown, so multiple values are tested from 1 to 16. $L = 1$ provides a useful baseline since it permits only trivial traces, so all conditional branches are assumed to be mispredicted. For each program G and each value of L, the experiment executes Set_Predictions and obtains the WCET.

Program	$L = 4$	$L = 8$	$L = 12$	$L = 16$
bs	0.672	0.672	0.672	0.672
bubble	0.292	0.244	0.243	0.242
cnt	0.680	0.635	0.623	0.618
compress	0.605	0.604	0.602	0.602
crc	0.565	0.374	0.310	0.287
div	0.698	0.680	0.676	0.675
duff	0.845	0.755	0.755	0.755
edn	0.633	0.538	0.511	0.498
expint	0.668	0.623	0.611	0.605
fdct	0.814	0.784	0.728	0.728
fibcall	0.720	0.720	0.720	0.720
fir	0.695	0.673	0.667	0.663
insertsort	0.699	0.640	0.585	0.546
janne_complex	0.721	0.721	0.721	0.721
jfdctint	0.686	0.648	0.638	0.634
matmult	0.678	0.637	0.627	0.622
ndes	0.669	0.664	0.663	0.662
ns	0.329	0.283	0.273	0.240

Table 2. WCETs of benchmark programs after Set_Predictions, normalized against the result for $L = 1$.

4.5 Evaluation

Firstly, it is observed that Set_Predictions with trace formation results in a reduction in the WCET. Table 2 shows the WCETs for the benchmark programs after processing, normalized against the result for $L = 1$ (trivial traces). It is clear that the algorithm provides a real benefit. Figure 8 represents the same data in graph form. There is a dramatic improvement from $L = 1$ to $L = 4$ in all cases. Many programs also benefit significantly when L is increased to 8. Beyond that point, further increases are less helpful: the WCET becomes defined by the conditional branches that leave long traces. Significant improvements for $L = 12$ and $L = 16$ are only seen for a small minority of the programs (insertsort, crc, ns). These contain simple inner loops that are guaranteed to execute many times in the worst case.

Virtual traces are better than simply assuming branches are always mispredicted, but they are most beneficial in cases where the WCEP is significantly worse than other paths. This is true in crc and ns, where all execution paths include an inner loop that executes many times. It is also true in bubble and insertsort, where the WCEP includes a "swap" operation as part of the sort algorithm, because in one case the swap is needed (and the trace helps) and in the other case the swap is not needed (and although the trace does not help, less code needs to be executed).

```
function Post_Optimizer(G):
    conditionals = {(v0, v1)|(v0, v1) ∈ E ∧
        |{(v1, v2) ∈ E}| = 2 ∧ (v0, v1).prediction ≠ None}
    improvement = True ; i = 0 ; bestp = None
    Z0 = Calculate_WCET(G)
    while improvement:
        Zi = Z0 ; best = None
        forall x ∈ conditionals:
            // flip direction of prediction
            save = x.prediction
            (v0, v1) = x ; next = {(v1, v2)|(v1, v2) ∈ E}
            x.prediction = pop(next - {x})
            // test for improvement
            Z1 = Calculate_WCET(G)
            if Z1 < Zi:
                Zi = Z1 ; best = x
                bestp = x.prediction
            end if
            // restore
            x.prediction = save
        end for
        if best ≠ None:
            Z0 = Zi ; best.prediction = bestp
            improvement = True ; i = i + 1
        end if
    end while
end procedure
```

Figure 7. Post_Optimizer attempts to improve the predictions made by Set_Predictions in order to reduce WCET.

This suggests that it might be worth eliminating some of the conditional branches and replacing them with predicated operations, so that one path through a trace implements several actual paths through the program. A superblock can be extended in this way using if-conversion: the result is known as a *hyperblock* [9]. The *single-path paradigm* [14] uses predication throughout a program to reduce all paths to a single one. This simplifies WCET analysis but might also increase the WCET of some programs, which would be avoided if such path merging was only performed as a local optimization. However, support for predication would force extensions to be made to the operation scheduler.

4.6 Improvements to the Algorithm

The optimality of the Set_Predictions algorithm has not previously been studied [3]. It is known not to reconsider any of the predictions that it makes, so suboptimal decisions are fixed forever. The algorithm does consider the possibility that the predictions might change the WCEP but it responds to this only by adding more predictions along the new WCEP. (This is used for ndes, where an additional iteration is needed.) This suboptimality is a particular concern for traces because the extra cost of a branch misprediction is potentially higher. Therefore, this section considers reviewing predictions after they are made: this might lead to a lower WCET.

Program	$L = 4$			$L = 16$		
	i	NW	%ch	i	NW	%ch
cnt	1	0.679	0.1%	1	0.618	0.1%
compress	3	0.604	0.2%	2	0.601	0.2%
edn	2	0.629	0.5%		n/a	
expint	1	0.668	0.0%	1	0.605	0.0%
fibcall	1	0.616	16.8%	1	0.616	16.8%
janne_complex	2	0.675	6.7%	2	0.675	6.7%
matmult	2	0.677	0.1%	2	0.622	0.1%
ndes	4	0.669	0.1%	5	0.661	0.2%
ns	3	0.305	7.9%	2	0.198	21.3%

Table 3. WCET reduction achieved by Post_Optimizer after i iterations, expressed as a normalized WCET (NW) and as a percentage of the starting WCET from Table 2. Benchmarks are omitted from this table where no improvement was found.

In order to experiment with this possibility, a second algorithm was implemented to improve the results of Set_Predictions (Figure 7). Post_Optimizer temporarily switches the direction of each branch prediction in turn, then evaluates the WCET. If any WCET reduction was found, the change resulting in the lowest WCET is committed and the algorithm repeats. Thus, Post_Optimizer is a hill-climbing search for the best set of branch predictions. It is slow because it considers changes to *every* conditional branch whenever an improvement can be made to *any* conditional branch. However, this is justified because the purpose is to evaluate Set_Predictions against the improvements that could be made if computation time is not an issue.

The WCET reductions obtained by Post_Optimizer are usually minimal (Table 3). Post_Optimizer was only able to improve 9 out of the 18 benchmarks, and then only by a fraction of a percentage in 6 of those cases. This suggests that Set_Predictions already works well enough for most programs, even though it uses a simple strategy. Each of the three cases where Post_Optimizer led to a significant improvement were found to be due to a situation where adding traces caused a subtle change in the WCEP at a frequently-executed point in the program. For example, in ns, the branch at the end of an inner loop is originally assumed to be taken. But adding traces transforms the inner loop to the extent that it is actually better to predict it as not taken. Similar changes are applied to fibcall and janne_complex. This suggests that Post_Optimizer is worthwhile in some cases, and that it is therefore worth trying to reduce its time cost. One way to do this would be to consider only n conditionals with the highest val-

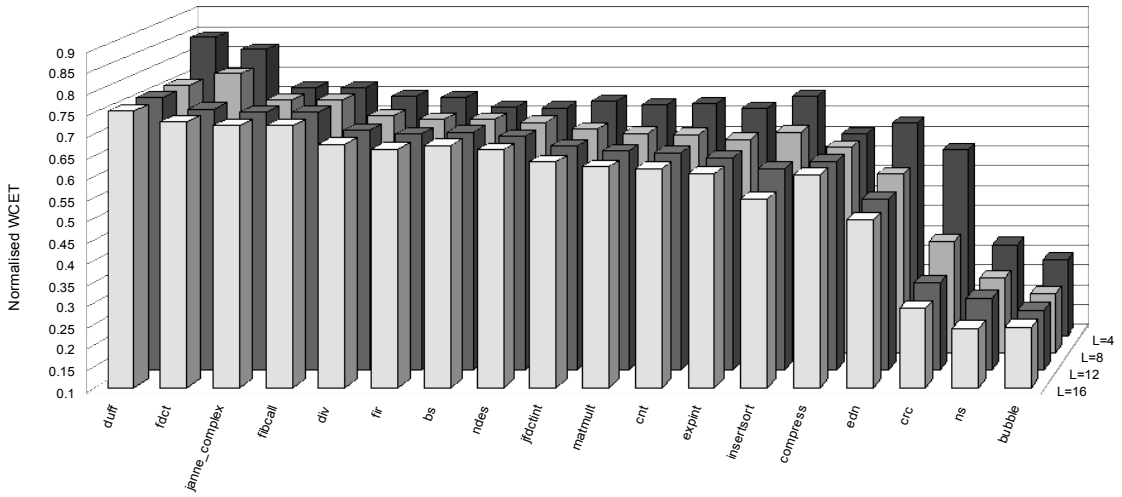

Figure 8. Data from Table 2 in graph form. The X axis has been sorted in order of average WCET reduction.

ues of $f(x)$, since these have the most significant effect on the results. Similar WCETs would have been obtained if Post_Optimizer was limited to considering only the top 10 conditionals.

5 Related Work

CPUs with dynamic superscalar out-of-order operation schedulers pose significant challenges for WCET analysis because execution times can depend on execution history. Many of the components of a CPU have to be modeled accurately in order to obtain WCET estimates that are *safe* (not underestimates) and *tight* (close to the true WCET): depending on the architecture, these may include caches, the scheduler and other parts of the memory subsystem. In the WCET analysis community, caches have been modeled with a reasonable degree of success [11] using knowledge about execution flow. But the models are not applicable to every practical cache design, and must be redesigned for every CPU in order to account for interactions with other CPU components, which may themselves depend on execution history [6]. An alternative to cache analysis is to replace caches with predictable components such as scratchpads [13, 17].

The scalability of pipeline modeling techniques is also limited by the possibility of timing anomalies [8] which may occur if dynamic resource allocation decisions are being made [19], which is possible in most superscalar out-of-order pipelines. This is not a problem for virtual trace analysis since anomalies always occur in the same way for each path through the trace, so each $t(P_{e,i})$ is constant. Earlier work has concentrated on simplifying complex pipelines to eliminate timing anomalies [16], constraining dynamic behavior to prevent their effects [1], or attempting to account for the possible effects of timing anomalies with pessimistic assumptions [7]. This paper differs from these approaches in that it permits superscalar out-of-order operation as part of the analysis without introducing pessimism.

Virtual traces constrain operation in order to permit measurement, since the use of measurement can remove the need to make an accurate model of a CPU. This is related to the motivation for measurement-based WCET analysis approaches [2, 12], but the key difference is that statistical methods are not needed when virtual traces are used because there is no unpredictable operation. *Some* constraints are applied to reduce unpredictability in [12] (e.g. arranging the CPU pipeline into a "worst case" state prior to measurement) but the constraints applied for virtual traces eliminate *all* unpredictability.

Virtual traces are related to the use of static branch prediction, which has previously been suggested as a way to simplify WCET analysis [3]. Virtual traces include sequences of static branch predictions, but the way in which they are used is very different. In [3], a fixed time penalty is added to every T-graph edge that represents a mispredicted branch, and timing anomalies are ignored by assuming worst-case behavior in all cases. This is pessimistic, but there is no way to guarantee particular behavior because the dynamic scheduler may start in any state. In contrast, virtual traces always begin in the base state and always return to that state within L branch predictions. Thus, the scheduler behavior is guaranteed, and so there is no pessimism in the low-level model of the CPU.

6 Conclusion

This paper has described and discussed virtual traces, which are viewed as an architectural solution to the problem of modeling superscalar out-of-order CPUs for WCET analysis. The paper has described and implemented an algorithm (Set_Predictions) for forming virtual traces in a

program, based on previous work on static branch prediction, and evaluated it using a model built using previous work. The results show that the WCET can be reduced using virtual traces, and limiting virtual trace sizes to 8 branch predictions is sufficient for most of the programs considered on the MCGREP-2 CPU platform, although some programs benefit from higher limits.

The greatest benefits of traces are seen in programs where one path is significantly worse than the others, suggesting that local usage of the single-path paradigm may be helpful. This paper has also attempted to improve the quality of the results of Set_Predictions by post optimization, but found that improvements are not usually possible, i.e. Set_Predictions works well, although a heuristic such as Post_Optimizer will produce improvements in some cases.

7 Acknowledgments

Thanks to the anonymous reviewers for their helpful suggestions.

References

[1] A. Anantaraman, K. Seth, E. Rotenberg, and F. Mueller. Enforcing Safety of Real-Time Schedules on Contemporary Processors Using a Virtual Simple Architecture (VISA). In *Proc. RTSS*, pages 114–125, 2004.

[2] G. Bernat, A. Burns, and M. Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Comput.*, 1(2):179–194, 2005.

[3] F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real time systems. In *Proc. ECRTS*, pages 33–40, 2005.

[4] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30(7):478–490, 1981.

[5] J. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2004.

[6] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE*, 91(7):1038–1054, 2003.

[7] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Syst.*, 34(3):195–227, 2006.

[8] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proc. RTSS*, page 12, 1999.

[9] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. MICRO*, pages 45–54, 1992.

[10] Malardalen WCET research group. WCET Benchmarks (accessed 18 October 07). `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2007.

[11] F. Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2-3):217–247, 2000.

[12] S. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Technische Universiat Munchen, 2002.

[13] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. DATE*, pages 1484–1489, 2007.

[14] P. Puschner. Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In *Proc. ECRTS*, Technical Report, Jun. 2002.

[15] P. Puschner and A. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Syst.*, 13(1):67–91, 1997.

[16] C. Rochange and P. Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Proc. CF*, pages 307–314, 2005.

[17] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. RTSS*, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.

[18] D. W. Wall. Limits of Instruction-Level Parallelism. Technical Report WRL-93-6, DEC Western Research Laboratory, 1995.

[19] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proc. Int. Conf. Quality Software*, Sep. 2005.

[20] J. Whitham. Real-time processor architectures for worst case execution time reduction. PhD Thesis YCST-2008-01, University of York, 2008.

[21] J. Whitham and N. Audsley. Using trace scratchpads to reduce execution times in predictable real-time architectures. In *Proc. RTAS*, pages 305–316, 2008.

[22] R. Wilhelm. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. *LNCS*, 2937:309–322, 2004.