# Predictable Out-of-order Execution Using Virtual Traces

**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

*The problem of worst-case execution time (WCET) analysis of complex CPUs is addressed in this paper using a proposed architectural modification. The virtual trace controller (VTC) constrains execution to follow only the paths that have been considered by the WCET analysis model, allowing the WCET to be determined safely by measurement. Each path has a constant execution time regardless of CPU complexity because the VTC enforces predictable operation. This paper evaluates the VTC using benchmark programs and the M5 simulator.*

*The results show that guaranteed throughput is increased for many programs using the constrained CPU model versus an idealized in-order design, indicating that the VTC can make complex CPU designs operate predictably without reducing throughput to the level of a simple CPU design. Additional results provide more information about the implications of each of the VTC features. Of all the restrictions introduced for predictability, disabling memory forwarding has the greatest effect on the maximum throughput, although conditional branches can also be significant. This paper suggests ways to improve the VTC to increase the guaranteed throughput.*

## 1 Introduction

This paper proposes and evaluates an architectural solution for three issues related to the problem of *worst-case execution time* (WCET) analysis, which is an important component of timing analysis for real-time software [23]. The issues are: (1) minimizing *pessimism* in analysis, (2) increasing the CPU throughput that can be guaranteed, and (3) minimizing CPU *modeling costs*. WCET analysis determines the longest possible execution time for a specific program on a specific CPU. Previous approaches have achieved this by modeling the program and CPU, either explicitly [14] or by using execution time measurements for sections of the code [6, 19].

Modeling and measurement are most difficult for CPUs that use speculative and out-of-order execution. These CPUs include an operation scheduler heuristic that executes instructions as early as possible, exploiting *instruction level parallelism* (ILP) [26] to increase throughput. The operation scheduler is affected by *timing noise* from many sources, causing execution times to vary unpredictably. Timing noise is any interference that might change the timing of an execution path, and it makes determining the worst case difficult. While sources of timing noise can be explicitly modeled, this is costly because of the complexity of the CPU design. Pessimism is introduced whenever it is not possible to be certain of timing, with the result that the throughput of an out-of-order CPU over an in-order design may not be guaranteed [25]. (Throughput is the effective execution speed of the CPU running the tasks; guaranteed throughput is the worst-case execution speed for those tasks.)

This paper introduces the *virtual trace controller* (VTC) as a modification for a CPU of arbitrary complexity. It constrains the CPU operation so that a program is executed as a collection of *traces* (Figure 1). In this form, each execution path has an execution time that (1) is precisely known (so pessimism is minimized), (2) can be obtained by measurement (so modeling costs are zero; the CPU hardware is the model), and (3) can take advantage of out-of-order execution to maximize throughput. The VTC guarantees that traces always execute in the same way by eliminating sources of timing noise in the pipeline. The WCET can be obtained safely and the traces can be optimized to reduce the WCET. Preemption does not affect trace execution times and is permitted on any boundary between virtual traces. It is possible to set a maximum preemption latency by limiting the trace size.

The contributions of this paper are: (1) a CPU-independent description of traces, the VTC and the process for program conversion; (2) a description of the implementation of O3+VTC (VTC extensions for the M5 Alpha simulator [7]); and (3) an evaluation of the VTC that considers ways to increase the guaranteed throughput.

The structure of this paper is as follows. Section 2 has background information on traces. Section 3 describes virtual traces and the VTC, then section 4 discusses the O3+VTC implementation, which is used to carry out some experiments in section 5. Section 6 has related work and section 7 concludes.

## 2 Background - Traces for WCET Analysis

*Traces* are executable representations of paths through a program [12]. Typically, implementing part of a program using a trace will change the execution time but not

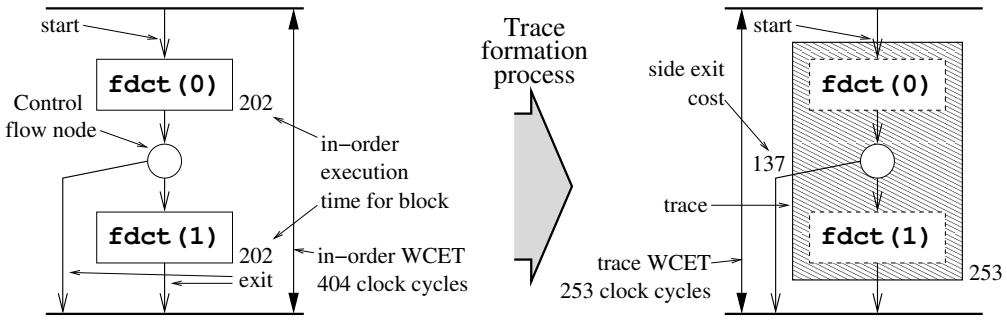


**Figure 1. Two representations of a program fragment. Left: a *control flow graph* (CFG) in which the atomic unit of code for WCET analysis is a basic block. This representation is useful when code executes in program order. Right: a *trace control flow graph* (TCFG), composed of traces. The functionality of both graphs is identical but the WCET of the TCFG is lower, because speculative and out-of-order execution is possible within each basic block *and* between the two basic blocks. Consequently more instructions are executed in parallel. Execution times are from the O3+VTC and IIO simulators (section 5); the code is from [18].**

the functionality of the program. In this work, traces are used as a framework to model speculative and out-of-order program execution. Traces have previously been used to reduce the average execution time of programs on explicitly parallel CPUs [12] such as *very long instruction word* (VLIW) machines [13]. But because trace execution can be extremely predictable, the paradigm is also useful to enable (1) WCET analysis, and (2) WCET reduction. The discussion is limited to *predictable traces* which are *traces in which every path has a fixed execution time*. (In VLIW machines, traces are not necessarily predictable.)

*Example*. Figure 1 shows two representations of a program fragment that carries out a *fast discrete cosine transform* (FDCT), an operation that is used for image compression. The *trace control flow graph* (TCFG) is composed of traces while the CFG is composed of basic blocks. The CFG runs on a conventional CPU that executes code in-order and consequently cannot achieve any less than 202 clock cycles for the 202 instructions in each `fdct` basic block. The overall WCET for the fragment is 404. However, the TCFG allows a CPU to do better without sacrificing predictability. The WCET is reduced by *biasing* speculative execution towards the *worst-case execution path* (WCEP). In this case, the WCEP runs through both `fdct(0)` and `fdct(1)`, so the *main path* of the trace is `fdct(0)` followed by `fdct(1)`. (Each trace has exactly one main path: main paths are formed as subpaths of the WCEP.) When the trace executes, the CPU is able to exploit ILP within `fdct(0)` and assume that `fdct(1)` will follow `fdct(0)`, so additional ILP can be obtained by executing operations from `fdct(1)` as early as possible. The benefit of allowing this predictable speculation is a reduction of the WCET from 404 to 253. If `fdct(1)` is not executed, then a *side exit* is taken, and the execution time is 137 instead of 202.

The TCFG is built after a conventional compiler has produced a program binary and CFG (algorithms for doing this are examined in [29]). Representing a program as a TCFG has three significant effects. Firstly, if traces are allocated correctly, the WCET can be reduced, and hence the guaranteed throughput of the CPU is increased. Secondly, in every case where the program itself can be modeled, the WCET can be determined accurately since the execution times in each trace are constant. Thirdly, it is possible to determine every trace execution time by measurement. Traces have the following timing model for WCET analysis purposes:

1. A trace replaces sequential machine code in one or more basic blocks. A trace always begins execution at an entrance $e$ (a basic block) and has $1 \leq n \leq L$ exits. Exits lead to other traces. The example in Figure 1 has $n = 2$ exits and entrance $e$ = `fdct(0)`.

2. A trace requires an exact number of clock cycles to reach each one of the $n$ exits from the entrance. The path through branch $i$ from entrance $e$ is denoted as $P_{e,i}$ for WCET analysis purposes (a basic block sequence). The time taken is $t(P_{e,i})$, a constant.

3. A trace has up to $L - 1$ conditional branches along the *main path* $P_{e,0}$. Every other path $P_{e,i}$ ($i \neq 0$) also follows $P_{e,0}$ until branch $i$ is reached, when a *side exit* is taken. $P_{e,0}$ continues to the *main exit*. The example in Figure 1 has $t(P_{e,0})$ = 253 and $t(P_{e,1})$ = 137.

Traces support WCET analysis by any well-known method; the *implicit path enumeration technique* (IPET) [15] has previously been applied to traces [28] but other methods such as tree and graph-based analysis are also applicable [4, 9, 22].

## 2.1 Implementing Traces

During program execution, additions to the CPU constrain execution, ensuring that $t(P_{e,i})$ values cannot vary in the traces that are executed. One implementation of

| | |
|---|---|
| **add** | VTC and VTR hardware. These are a small state machine (Figure 4) and an $L$-bit register (Figure 5) with almost negligible area cost. |
| **remove** | Cache updating logic is not needed because scratchpads are used instead; this may cause a reduction in the size of the on-chip memory area of around 34% [2]. |
| **add** | New scratchpad for virtual trace data - this is smaller than the instruction scratchpad since it has one $L$-bit entry per basic block instead of one 32-bit entry per instruction. |
| **remove** | Branch predictor, memory access predictor: neither are used. Both contain RAM. |

**Figure 2. The silicon area used by a virtual trace CPU may be smaller than that of a regular CPU with the same pipeline.**

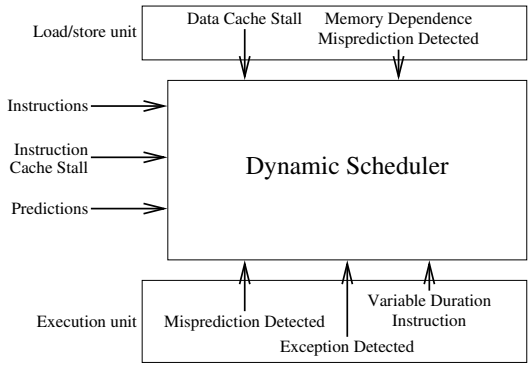


**Figure 3. Sources of timing noise that could affect the operation of a dynamic operation scheduler.**

traces uses microprograms within a *trace scratchpad* [30]. The microprograms explicitly encode the operations to be executed: the CPU resembles a VLIW machine that can (1) execute both wide (explicitly parallel) and narrow (sequential) instructions, and (2) uses scratchpad memory in place of caches. (Scratchpads are on-chip memories similar to caches, but controlled directly by programs [2]. They have been proposed as predictable replacements for caches [21].)

Although this approach can be used to reduce WCETs there are three major problems. Firstly, a specialist CPU is required. Secondly, a specialist compiler (matched to the CPU) is also needed to convert traces into microprograms for the scratchpad. Thirdly, the information density of the microprograms is poor, so a small scratchpad memory cannot hold more than a small part of a program. In [30], less than ten traces are used for each program and resource usage is far from optimal.

The third problem can be addressed by (1) combining conventional execution (for rarely-used code) with trace execution (for code that is frequently executed in the worst case), and (2) by using an allocation process to optimize the use of scratchpad space [30]. The problem can also be addressed by dynamically updating the scratchpad during execution [28], but the time taken to update the scratchpad is significant (again due to the poor information density). WCET reductions can be obtained in both cases, and hence guaranteed throughput is increased, but the problems with the approach suggest that custom microprogramming is not the best solution to the issues with WCET analysis. It requires fully custom CPUs and compilers, microprograms consume too much valuable on-chip memory space, and loading new microprograms is also costly.

## 3 Virtual Traces

In this paper, *virtual traces* are proposed as an alternative to the microcoded implementation described in section 2.1. Microprogramming is not used. Instead, virtual traces are sequences of commands for a *virtual trace controller* (VTC), which has two purposes: (1) eliminate sources of *timing noise* within the CPU, and (2) force predicted execution to proceed along the main path of the current virtual trace. The result is functionally equivalent to normal execution, but as with microprogrammed traces, timing is predictable. The new type of trace is *virtual* in the sense that its full form (microoperations) is never stored in RAM, as it passes directly from the CPU scheduler to the execution units. Virtual traces can be applied to almost any *synchronous* (single clock domain) CPU core by implementing some modifications; the benefit is a potential increase in guaranteed throughput without any need to adopt an unusual CPU architecture. The modifications may even lower the silicon cost of building a CPU core (Figure 2), although the area cost will be increased slightly if the CPU must support both dynamic and virtual trace modes of execution.

*Timing noise* is defined as any sort of interference that *could* change the execution time of a path $P_{e,i}$ through a trace. Figure 3 shows the sources of timing noise in a typical out-of-order CPU. A cache miss or an opportunity for memory forwarding could change the order in which operations become ready to execute; an exception might change the sequence of operations that can be executed. An explicit WCET analysis model would attempt to capture all the possibilities. However, a synchronous CPU core is a deterministic device, i.e. given known inputs and a known starting state, the output is also known, so timing noise can be eliminated by removing unpredictable inputs. Changes are implemented to control the following sources of timing noise:

1. *Previous pipeline state*. The theory of *timing anomalies* indicates that the previous state of the CPU can affect future execution in unpredictable ways [16], so the pipeline needs to be *synchronized* to a known state. This could be done in two ways: (1) an explicit reset function that clears all blocks from the CPU pipeline, or (2) a drain

(1) **Initializing**
When an instruction is fetched, e.g. at the start point in Figure 1, the associated virtual trace information is also fetched and stored in a new *virtual trace register* (VTR) shown in Figure 5. Virtual trace information exists for each basic block and is stored in scratchpad RAM. Virtual traces are lists of static branch predictions, laid out along an execution path rather than being placed in instruction memory. Each item refers to one branch instruction: *taken* or *not taken*.
(2) **Running**
As execution continues, the VTC feeds the fetch hardware with branch predictions from the VTR, guiding the dynamic operation scheduler along the main path. In Figure 1, the trace says that `fdct(0)` is followed by `fdct(1)`, but functionality will be preserved if the branch to the side exit is taken instead.
(3) **Resynchronizing**
When either (1) a side exit (branch misprediction) is detected, or (2) the supply of branch predictions in the VTR is exhausted, the VTC halts instruction fetching until the CPU pipeline is empty. This operation returns the pipeline to a known state; once this is done, the VTC moves to the Initializing state and execution continues.

**Figure 4. The VTC state machine enforces isolation between virtual traces. It fits into a CPU architecture as in Figure 5.**

function that allows the CPU pipeline to empty by halting the fetch process. This paper chooses the latter strategy since the implementation is simple (Figure 4). This arrangement permits timing anomalies *within* a trace, since the CPU scheduler makes dynamic resource allocation decisions [27]. However (1) they are guaranteed to affect execution in the same way since timing noise is controlled ($t(P_{e,i})$ is constant for each path) and (2) they have no effect on subsequent execution because the pipeline is resynchronized at the end of each trace.

2. *Variable Duration Instructions*. Because these are *data dependent*, they might cause some $t(P_{e,i})$ to vary, e.g. when operating on large numbers. Timing noise is eliminated by modifying the execution units to enforce constant execution times.

3. *Cache stalls*. In an out-of-order CPU, cache stalls only block the instructions that are dependent on a particular access. The CPU continues fetching, decoding and executing other instructions. Any cache effect may introduce timing noise by delaying an instruction or forcing a different execution order. In this paper, the applied solution is the use of *scratchpad memory* (section 2.1) to replace the instruction and data caches with fully predictable memories (Figure 5).

4. *Memory dependence mispredictions*. Some out-of-order CPUs speculate about the relationships between memory access instructions in order to expose more ILP. This type of speculation creates timing noise as it fails unpredictably. This is avoided by changing the operation scheduler to enforce a safe ordering on memory operations: load operations cannot be reordered across store operations, and store operations cannot be reordered at all.

5. *Branch predictions* fit into the virtual trace model. The dynamic branch predictor is replaced with data from the current virtual trace, delivered by the mechanism shown in Figure 5. The VTC state machine (Figure 4) implements branch misprediction events as side exits. Another restriction ensures that *branch operations are executed in program order*, since that prevents two or more misprediction events being active at the same time. This is done using the instruction dependence mechanism.

6. *Exceptions* occur when an instruction cannot complete normally, e.g. a load from a non-existent memory address. They can be modeled as conditional branches, but the sheer number of operations that could generate exceptions would make this sort of analysis intractable. The easiest strategy is to ignore exceptions; many programs do not use them.

Timing noise can also be introduced by preemption. This can be avoided by forcing preemption to occur only on boundaries between virtual traces, i.e. after resynchronization. Very long traces could lead to high preemption latency, but since the maximum preemption latency is known ($max(t(P_{e,i}))$), trace lengths can be limited to ensure that the latency is appropriate for the application.

## 4 Implementation using M5

This section describes the implementation of the VTC within the M5 simulator [7]. An example of its application is given in section 4.1. Section 4.2 explains how the VTC allows measurements to be used safely in place of explicit modeling.

M5 is an open-source simulator for computer architecture research [7]. It has been used by a wide variety of research projects [17] as it is easily extended to serve a new purpose. As originally published, M5 includes simulated memory systems, CPUs and hardware devices which can be connected together to make clusters of simulated computers. M5's CPU simulator can be operated in a "fast" mode where the functionality of each device is simulated but correct timing is ignored, or in a "detailed" mode where both timing and functionality are simulated precisely. This paper uses the detailed mode and adopts the O3 out-of-order CPU for the work. O3 is configured for the Alpha *instruction set architecture* (ISA), a RISC ISA that was designed for dynamic out-of-order execution. O3's default resources are outlined in Table 1.

O3 is modified to include the VTC, creating a new CPU named O3+VTC. Other M5 subsystems are unchanged since the only requirement is that they must respond with deterministic timing (section 3) and this is achieved by stalling O3+VTC whenever an external component is not ready. (This is not realistic in a real design due to propa-

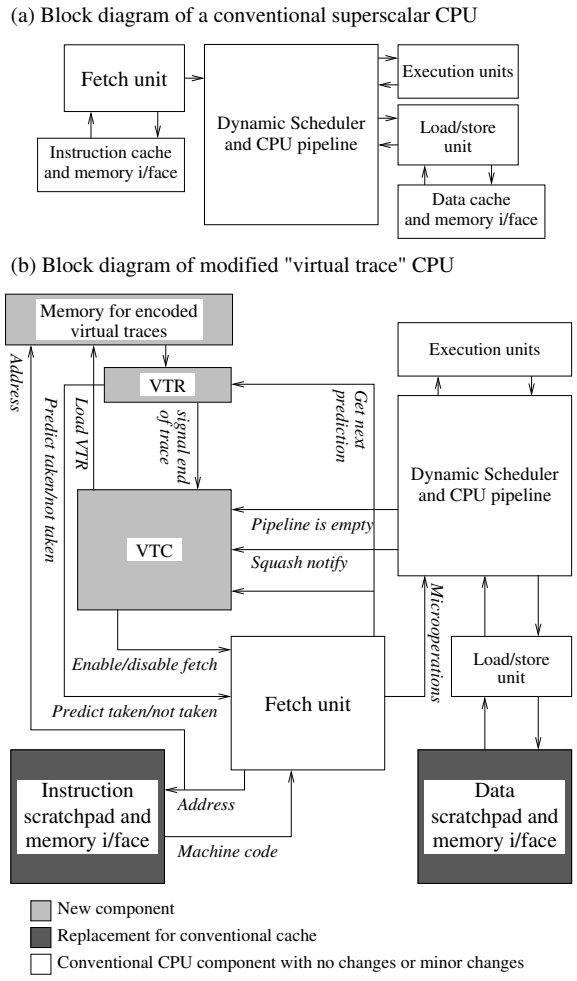


**Figure 5. Block diagram of a CPU core, (a) before and (b) after the VTC and VTR are added. The scheduler, pipeline and execution units are unchanged.**

gation delays, but it is suitable as a model for scratchpad memories [2, 21] as "caches that never miss".) The implementation takes the form of patches applied to the C++ and Python source code of M5. O3+VTC is O3 with the following changes:

**Patch P1**. *Instruction fetch unit*[1]: an interface to the VTC is added. This relays four types of message between the VTC and the *instruction fetch* (IF) unit: (1) Branch predictions (VTC → IF); (2) Empty pipeline notifications (IF → VTC); (3) Squash notifications (IF → VTC); generated when a branch misprediction is detected, and (4) Fetch enable/disable commands (VTC → IF) which allow the VTC to pause instruction fetching at the end of a virtual trace. This is used to resynchronize the pipeline for the next trace.

**Patch P2**. *Instruction decoder*[2]: an extra dummy register is added as an input and output of every branch instruction. This register holds no data. It exists only to force branch operations to execute in program order, which is necessary to preserve trace semantics.

| Resource type | Count | Latency |
|---|---|---|
| Integer ALUs | 6 | 1 |
| Integer Mult/Div | 2 | 3/20 |
| Float Add/Cmp | 4 | 2 |
| Float Mult/Div | 2 | 4/12 |
| Memory Read/Write | 2 | 1 |

**Table 1. O3 provides a large number of resources for code execution, so the limit on execution speed is imposed by the code itself (the degree of ILP) rather than the pipeline. Latencies are in clock cycles.**

**Patch P3**. *Memory dependence unit*[3]: modifications are made here to enforce a strict order on memory operations that is explicitly safe.

**Patch P4**. *Load/store queue unit*[4]: forwarding information between memory operations is disabled because it is dependent on the effective addresses being used. (This change is a matter of deactivating some functionality.) Additionally, memory exceptions are ignored.

Other changes were made in order to ensure that the memory subsystem responds deterministically (as detailed above). The modified simulator ignores *translation-lookaside buffer* (TLB) exceptions, and simulates the use of scratchpads for all storage. Furthermore, it assumes that programs communicate with the outside world via emulated Linux system calls, which are forwarded to the host operating system. Implementation also required some minor extensions to the microoperations used within O3+VTC, which are known as *dynamic instructions* in the source code. The principle change was the addition of new flags to each dynamic instruction to allow branches to be forced "taken" or "not taken" by the VTC, and to allow instructions to be marked "fake". Fake instructions execute as usual, but have no effect (unlike no-ops which are discarded by the CPU). Both of these features are used for measuring $t(P_{e,i})$ values since they allow CPU operation to be tested along any path without changing the memory or registers.

The VTC is an implementation of the state machine in Figure 4. It is used in conjunction with a driver program that can measure the execution time of every path through every trace for a specific TCFG, i.e. obtain all $t(P_{e,i})$ values. The driver program is also able to use the CPU to execute the program. It may execute as a TCFG (using traces, with the VTC enabled), or as a CFG (using a less pre-

[1] In `src/cpu/o3/fetch_impl.hh`.
[2] In `src/arch/alpha/isa/main.isa`, `src/arch/alpha/isa/branch.isa`.
[3] In `src/cpu/o3/mem_dep_unit_impl.hh`, `src/cpu/o3/lsq_unit_impl.hh`.
[4] In `src/cpu/o3/lsq_unit.hh`.

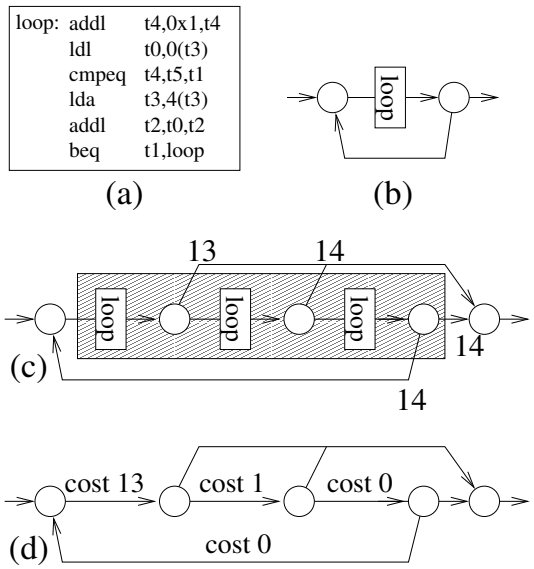


**Figure 6. A sample program (a), with its CFG (b), its TCFG (c) (using $L = 4$ to unroll the loop), and its T-graph (d). The execution timings were measured using the modified O3 CPU. Each execution timing is an integer constant, independent of data and the previous state of the CPU.**

dictable CPU, e.g. with dynamic branch prediction used in place of traces). This enables comparison and testing.

### 4.1 Example: Exact Execution Times

Figure 6(a) shows a loop that sums all of elements in an array, compiled from the following C code:

```
for (t4=0; t4 != t5; t4++) {
    t2 += *t3;
    t3 += 4;
}
```

The loop is implemented in Alpha machine code, since O3+VTC uses this ISA. The CFG, TCFG and T-graph of this code are shown in Figures 6(b), (c) and (d). The VTC driver program allows us to obtain the $t(P_{e,i})$ values for the program running on O3+VTC, which are added to the TCFG (Figure 6(c)). In the TCFG, the loop has been unrolled three times, but the unrolling has not changed the machine code as a compile-time optimization would, as it takes place within the CPU. The trace merely *models* the unrolling process, which is a by-product of speculative execution. The CPU throughput is lowest when the loop bound t5 = 1 because the ratio between $t(P_{e,i})$ and the loop bound t5 is at its greatest. Larger values of t5 make better use of CPU resources. The second unrolled iteration costs just 1 extra clock cycle, the third costs nothing. This is because those future iterations have already been executed speculatively.

Even if the degree of speculative execution is not so extreme, there is always a substantial cost to running any trace due to the time taken to resynchronize the pipeline and restart from the known state. This *minimum trace cost* is dependent on the specific implementation of traces in use, i.e. the O3+VTC pipeline and its configuration. It is analogous to the pipeline latency.

The traces in a program are planned during the TCFG building step. Within the TCFG, each trace could be a single basic block, but that would mean that the CPU would be poorly utilized as most of its time would be spent resynchronizing and filling the pipeline rather than executing: predictable, but with poor throughput. Therefore, traces are formed along paths that contribute the most to the WCET. CFG → TCFG conversion can be carried out using a variety of standard algorithms [13], which are conventionally focused on minimizing *average case execution time* (ACET). However, CFG → TCFG conversion can also be WCET-focused [29].

The TCFG → T-graph conversion (Figure 6(c) and (d)) is straightforward: the *dual form* of the TCFG is obtained (edges and vertices are exchanged). The edges of a T-graph represent executable code (parts of a trace) and are associated with a worst-case execution count $f$ and an execution cost $\gamma$. The vertices represent branches and join points. Every $t(P_{e,i})$ is obtained by measurement, then the $\gamma$ values are obtained by assigning non-negative integers to T-graph edges such that the total cost of every path $P_{e,i}$ is $t(P_{e,i})$. The T-graph and $\gamma$ values are combined with behavioral constraints, such as upper bounds on loop iterations. These can be specified as part of the source code [8], and approaches exist to derive them automatically for some programs [10]. The T-graph, $\gamma$ values and behavioral constraints make a complete program model for IPET analysis, and the T-graph is analyzed as if it were composed of basic blocks [24]. In cases where a single basic block appears in more than one trace, or several times in a single trace, the behavioral constraints apply to the sum of the number of executions of that block. The IPET equations that are used are detailed in [28, 30].

In the example, increasing $L$ would result in (1) a larger TCFG and T-graph with more loop unrolling, (2) more $t(P_{e,i})$ values, and (3) a change to the WCET. Table 2 shows the $t(P_{e,i})$ values for Figure 6(a) and $L = 20$. As before, the greatest throughput is obtained when no exits are taken: 19 iterations are executed for a cost of 30 clock cycles. The lowest throughput is obtained when the first exit is taken: 14 clock cycles for a single iteration. After that, future iterations take approximately one clock cycle until improvements tail off and timings become roughly constant due to the halted fetch process. The high start-up cost (approximately 14 clock cycles) is typical of all types of trace. It is irrelevant provided that the overall WCET is reduced, as it will be if the loop bound t5 is large.

The changes to the CPU core are designed to ensure that $t(P_{e,i})$ values are constant. Implementation errors can be debugged by repeatedly measuring execution times in

| $i$ | $t_i$ | $i$ | $t_i$ | $i$ | $t_i$ | $i$ | $t_i$ |
|---|---|---|---|---|---|---|---|
| 1 | 14 | 6 | 20 | 11 | 25 | 16 | 30 |
| 2 | 16 | 7 | 21 | 12 | 26 | 17 | 30 |
| 3 | 17 | 8 | 22 | 13 | 27 | 18 | 30 |
| 4 | 18 | 9 | 23 | 14 | 28 | 19 | 30 |
| 5 | 19 | 10 | 24 | 15 | 29 | 0 | 30 |

**Table 2. Execution times of every path through Figure 6(a) with trace length $L = 20$. $i = 0$ is the main path, $i = 1$ is the path if the first exit is taken, and so on.**

| Program | IIO | O3+VTC | | |
|---|---|---|---|---|
| | | $L = 1$ | $L = 10$ | $L = 20$ |
| bs | 92 | 272 | 82 | 85 |
| bubble | 5286 | 16011 | 8754 | 8480 |
| cnt | 3580 | 4024 | 2806 | 2787 |
| compress | 3545 | 6512 | 4222 | 4093 |
| crc | 21096 | 35583 | 21173 | 21082 |
| duff | 496 | 600 | 510 | 515 |
| edn | 97001 | 76486 | 45298 | 43227 |
| expint | 533 | 1482 | 437 | 389 |
| fdct | 3410 | 2328 | 2093 | 2100 |
| fibcall | 44 | 114 | 27 | 28 |
| fir | 2988 | 3665 | 1325 | 1206 |
| insertsort | 887 | 1728 | 739 | 676 |
| janne_complex | 348 | 728 | 316 | 299 |
| jfdctint | 3467 | 2511 | 1560 | 1509 |
| matmult | 142810 | 191662 | 128089 | 124595 |
| ndes | 40284 | 49993 | 25574 | 23294 |
| ns | 2852 | 9981 | 2815 | 1691 |

**Table 3. Execution times of benchmark programs on IIO and O3+VTC (clock cycles).**

a randomized order: the driver program includes support for this. In the case of Figure 6, this would be checked by (1) randomly choosing a path $i$, (2) measuring $t(P_{e,i})$, and then (3) selecting another path and repeating the process. If timings ever change, an error has been detected.

### 4.2 No CPU Modeling Required

This paper avoids discussion of the internal layout of the O3 CPU in this paper. This is because the details are irrelevant. The work treats O3+VTC as a black box, as there is no *need* to model the internals of the CPU because all behavior is captured by VTC measurement. The TCFG includes all possible paths through the program, each composed of members of the set of all paths $P_{e,i}$, and programs never leave that set during execution. Therefore, obtaining all $t(P_{e,i})$ values for a program provides everything that is needed by the WCET analysis model. This avoids the costly engineering of an explicit model [14], or the need to use probabilistic methods [5] since the VTC restricts the behavior that is possible.

The exact nature of $t(P_{e,i})$ values allows *exact analysis*, i.e. zero pessimism in WCET analysis. It is known that IPET computes the exact WCET value if all behavioral constraints are known [24], and the use of traces does not prevent this since traces can be converted to T-graphs. In general, pessimism is introduced from two places: (1) inaccuracies in the model of the CPU, which force the worst case to be assumed, and (2) infeasible paths in the model of the program. The first source is eliminated by the VTC: there are no inaccuracies in the timing model. The second source can be eliminated by better constraints that describe more about the program.

## 5 Experiment

O3+VTC can be used to experiment with virtual traces. This section compares O3+VTC against other CPUs, beginning with an idealized in-order CPU which is named IIO. IIO is deliberately easy to model for WCET analysis: it has the same execution units as O3+VTC, so instructions have the same latency on both CPUs (Table 1), but IIO's pipeline has no latency and there is no branch prediction. All instructions execute in program order: IIO does not use ILP. This section does not consider preemption, although it is supported in principle by the O3+VTC implementation.

To make comparisons as realistic as possible, benchmark programs are used (Table 3). The corpus is the Mälardalen WCET benchmark suite [18], which is specifically intended to support WCET-related research. It is assumed that the benchmark programs are single-path programs, i.e. that they have fixed input and therefore always execute in the same way, since this allows a single measurement to be considered as the WCET, avoiding WCET analysis. However, the findings still apply even if the programs are *not* single-path, because IPET and other methods can also be used to obtain the WCET (section 4.1). For each program, the experimental method is as follows:

1. Compile the program for the Alpha ISA, then execute it using IIO to obtain a full execution trace, listing every instruction that was executed.

2. Analyze the full trace to (1) obtain the CFG (by detecting basic block boundaries), then (2) use edge execution frequencies to add static predictions at each conditional branch, and finally (3) convert the CFG to a TCFG by forming traces around the static branch predictions. This is a way of building a TCFG without performing WCET analysis. Regardless of the method used, traces of length $\leq L$ are formed at every reachable basic block using the path described by static branch predictions.

3. Run the program on O3+VTC, obtaining (1) a clock cycle count, (2) a trace side exit count, and (3) a trace entrance count. The counting process is set up to disregard program initialization and exit code.

This method was repeated for each benchmark program with various values of $L$. Table 3 shows the execution

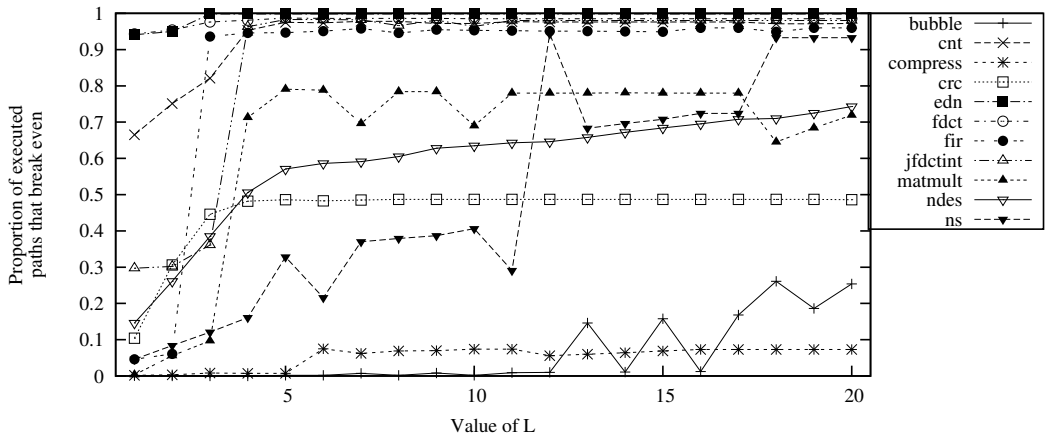


**Figure 7. Proportion of O3+VTC execution time spent in paths that break-even versus IIO. Small programs (less than 1000 clock cycles on IIO) are omitted.**

times for IIO and O3+VTC with $L \in \{1, 10, 20\}$.

### 5.1 Results

Table 3 shows that (in most cases) O3+VTC achieves better guaranteed performance than IIO where $L$ is large. When $L$ is small, IIO is usually better. This is because short traces force the VTC to resynchronize the CPU more frequently and this reduces performance. However, O3+VTC is not always better for *any* $L$. For `bubble`, IIO has a higher guaranteed throughput even when $L = 20$. Increasing $L$ leads to greater guaranteed throughput with diminishing returns.

Table 4 shows an example of the timing behavior of traces in O3+VTC versus IIO. IIO executes traces as sequential code, so the IIO timings are identical to the timings for execution of the original basic blocks. There is no cost for taking a side exit early so IIO timings are much less than O3+VTC timings *for early side exits*. For later side exits, O3+VTC is faster. In this table, the minimum trace cost is 17. O3+VTC is faster than IIO provided that exits 1, 2 and 3 are not used, since $t_{i_{\text{IIO}}} > t_{i_{\text{O3+VTC}}}$ for $i > 3$. The *break-even* point, where O3+VTC overtakes IIO, is 20 clock cycles.

| $i$ | $t_i$ O3+ VTC | $t_i$ IIO | $i$ | $t_i$ O3+ VTC | $t_i$ IIO | $i$ | $t_i$ O3+ VTC | $t_i$ IIO |
|---|---|---|---|---|---|---|---|---|
| 1 | 17 | 6 | 8 | 24 | 48 | 15 | 31 | 90 |
| 2 | 18 | 12 | 9 | 25 | 54 | 16 | 32 | 96 |
| 3 | 19 | 18 | 10 | 26 | 60 | 17 | 32 | 102 |
| 4 | 20 | 24 | 11 | 27 | 66 | 18 | 32 | 108 |
| 5 | 21 | 30 | 12 | 28 | 72 | 19 | 33 | 114 |
| 6 | 22 | 36 | 13 | 29 | 78 | 0 | 32 | 114 |
| 7 | 23 | 42 | 14 | 30 | 84 | | | |

**Table 4. Path execution times for the same code on O3+VTC and IIO. (This trace is an unrolled loop in `ndes`.)**

### 5.2 Lowering Break Even Points

Three distinct issues combine to increase the break-even point for a trace, and hence reduce the guaranteed throughput of O3+VTC. These are addressed below:

1. *$L$ may be too small.* Table 3 suggests that small values of $L$ lead to longer execution times; this is because many of the trace paths do not break-even for small $L$ (Figure 7). For example, less than 20% of the paths in `ns` break-even when $L = 4$. In this situation, most paths are actually taking longer on O3+VTC than they would on IIO. This explains the long execution times for small $L$, but something more interesting can also be observed in Figure 7: increasing $L$ doesn't *always* lead to shorter execution times. The break-even proportion for `ns` actually decreases twice for $L$ between 10 and 14. For some paths, it is better to pick a *local* small value of $L$, which suggests that *search* will be needed to find the best length for each virtual trace in order to optimize the WCET. (This effect exists because $L$ controls the degree of speculation. Setting $L$ to a smaller value forces a more conservative strategy, which might reduce ILP, but might also reduce the execution time because fewer operations would be executed. This is accommodated by the VTC because virtual traces can have any length *up to* $L$.)

2. *Traces may not accurately represent execution paths.* When execution does not follow the main path of a trace, a side exit is taken. These exits are particularly costly because the operation scheduler speculates past them on the VTC's instruction. However, the number of taken side exits for each program is independent of $L$. (Traces only have one main path, so there is no way to avoid taking a side exit by increasing the trace length.) One reason for `bubble`'s poor throughput in Table 3 is the unpredictable

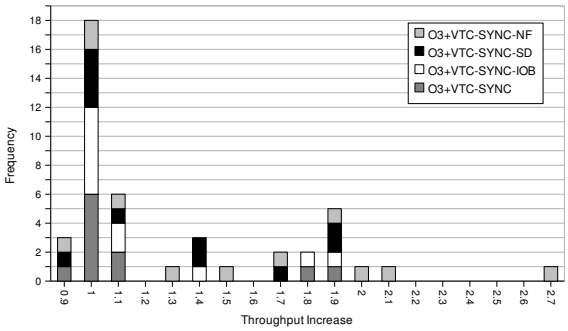


**Figure 8. Distribution of execution times following removal of some constraints. For each program, $L = 20$ and the O3+VTC execution time is defined as 1.0, so a data point in 2.1 indicates a 2.1× throughput increase. Small programs are omitted.**

| |
|---|
| **O3+VTC-SYNC** (Patch P1 omitted)<br>O3+VTC minus the requirement to resynchronize the pipeline after each trace. |
| **O3+VTC-SYNC-IOB** (Patch P2 omitted)<br>O3+VTC-SYNC minus in-order execution of branches. |
| **O3+VTC-SYNC-SD** (Patch P3 omitted)<br>O3+VTC-SYNC minus enforcement of static memory disambiguation rules. |
| **O3+VTC-SYNC-NF** (Patch P4 omitted)<br>O3+VTC-SYNC with support for dynamic memory forwarding. |

**Table 5. Additional CPUs tested for Figure 8: all of these impose fewer restrictions than O3+VTC.**

conditional branch in the inner loop: it is common for traces to end at the first or second side exit, so fewer of the executed paths break-even. One way to avoid this problem is to use predicated execution in place of branches. This could be used to convert the body of a loop into single-path code [22], which is executed extremely efficiently by traces. Another would be to locally reduce $L$ to force more conservative speculation as required.

3. *Dynamic scheduler restrictions may reduce guaranteed throughput*. The O3+VTC CPU includes many restrictions to make the operation scheduler predictable (section 4). To identify the restrictions having the greatest effect on the throughput, a second experiment was carried out in which the restrictions were disabled in turn. Table 5 lists the CPU variants used. Removing restrictions usually led to improved throughput (and variable $t(P_{e,i})$ values). Figure 8 shows the distribution of execution times as a result of relaxation. Based on Figure 8, it is observed that the execution time is increased by the following restrictions, in descending order of importance:

3.1. *No dynamic memory forwarding* (patch P4, CPU O3+VTC-SYNC-NF). Of all the restrictions, this has the biggest effect on throughput. It prevents information being moved quickly between store and load operations that use the same address. It causes a throughput reduction of up to 2.7×, with a mean of 1.55× observed across all benchmarks. Unfortunately, forwarding cannot be applied predictably within the WCET model for traces as it is data-dependent: an example of a situation in which speculative forwarding fails destructively is seen in `ns`, where the predictable O3+VTC is 8% faster than O3+VTC-SYNC-NF.

3.2. *Static memory disambiguation* (patch P3, CPU O3+VTC-SD). This causes a mean throughput reduction of 1.31× across all benchmarks, with the most significant effects on programs that make heavy use of RAM (e.g. `bubble`, `compress`, `fdct`). In most cases, programs that benefit from dynamic forwarding also benefit from dynamic memory disambiguation to a lesser extent.

3.3. *Resynchronization* (patch P1, CPU O3+VTC-SYNC). Resynchronizing the pipeline after each trace causes a mean throughput reduction of 1.17× across all benchmarks. Resynchronization is a very significant effect for every program when $L$ is small (Figure 7) but it is dwarfed by memory effects for large $L$.

3.4. *In-order branch resolution* (patch P2, CPU O3+VTC-SYNC-IOB). This restriction has no significant effect for most programs (Figure 8), but it does cause loss of throughput for programs with large numbers of unpredictable branches, e.g. `bubble`.

To summarize, memory accesses are the most significant drain on throughput from the VTC design *in addition to* the effects from a possibly suboptimal selection of $L$ and the problem of early side exits being used frequently. Therefore, efforts to improve the VTC should aim to reduce the overhead of memory accesses. Any data dependent event (e.g. a conditional branch) can be modeled as an explicit branch for WCET analysis, but this technique is unlikely to scale as a model of forwarding or memory disambiguation due to the number of possible execution paths, i.e. $O(2^n)$ after $n$ load operations that *might* take advantage of forwarding. Complexity is reduced by shortening traces, but that costs performance due to resynchronization (contrast $L = 1$ and $L = 20$ in Table 3). The most workable solution might be to mark *some* loads as *guaranteed forwarded* in the VTR if it was certain that forwarding would take place, or *guaranteed not forwarded* if two memory operations could never share an address, and hence disambiguation was known to be unnecessary. If this was applied to even a subset of loads, guaranteed throughput would improve.

## 5.3 Overall Comparison

O3+VTC can be compared directly against O3. Across all of the programs used for Figure 8, O3 executes code 1.9× faster than O3+VTC on average. This is the conse-

quence of all of the restrictions added together: the worst example is `bubble`, 3.6× slower on O3+VTC. However, throughput on O3+VTC is improved when compared to IIO. The results provide part of the basis for an eventual silicon implementation of the VTC by indicating the benefits and costs that can be expected. Minor modifications to the memory hierarchy of the CPU will be required to provide predictable memory accesses for a hardware implementation (e.g. using scratchpads or locked caches). All such implementation details are the subject of future work.

## 6 Related Work

The most similar work to this paper is by Mohan and Müller [19], who describe CheckerMode, a CPU modification which enables execution paths to be timed on a complex CPU architecture. The purpose of the work is similar: make a WCET model using measurements from the CPU. The authors also use structures that are similar to traces as their unit of WCET analysis, enabling them to account for speculative and out-of-order execution. However, there is a crucial difference between their work and this paper. Specifically, this paper uses traces to *constrain* and *control* the operation of the CPU, *as well as* using them for measurement. In contrast, Mohan's technique only uses them for measurement. It relies on being able to save and restore the CPU state, which could be loaded and unloaded from many CPUs through the debugging interface. It uses the state to reproduce execution scenarios for each path, obtaining execution time measurements which are added to the program's control flow graph. Mohan's technique does not affect the CPU's runtime operation, so it might be possible for *timing anomalies* [16] to occur between paths. This is partly accounted for by draining the pipeline during measurement, but it is not possible to be certain that this will eliminate all timing anomalies if CPU resources are being allocated dynamically [27]. Timing anomalies *cannot occur* between traces with a VTC.

The idea of using execution measurements obtained from a real CPU rather than a model is also explored in the *virtual simple architecture* (VISA) [1]. In VISA, an out-of-order CPU is allowed to run a program directly, but if block execution time measurements drop below a predicted worst case, the CPU is automatically switched into in-order execution (like IIO). This stops any timing anomalies and guarantees the predicted WCET. The flaw in VISA is that real-time programs always effectively execute in-order for the purposes of schedulability analysis [23] since the WCET must be used. This limit could be attacked by using a VTC to implement the simple mode rather than in-order execution. It could also be reduced by allowing out-of-order execution but constraining it to single basic blocks [25], which is like the VTC approach with $L = 1$.

The concept of a CPU instruction set that specifies information about timing also appears in work related to *precision timed* (PRET) machines [11], which must include both predictable CPUs and memory subsystems. Suitable memory systems include *scratchpads*, which are small memories integrated into a CPU [2]. Scratchpads can be used as highly predictable cache replacements [3], and many locked caches can be used in the same way. The issue is that instructions and data objects must be allocated scratchpad space: this may be carried out to minimize the WCET [21] or to reduce average energy consumption. Scratchpad allocations can be changed dynamically, so some algorithms attempt to partition the instructions within a program into distinct allocation regions [20]. Adding trace information to a scratchpad is one way to reduce the WCET further: scratchpad allocation algorithms have also been applied to *trace scratchpads* (storing microinstructions) [30]. But allocating *data* to a scratchpad is not an easy problem because common programming paradigms permit programs to access any data in memory at any time, and such random access is not easy to accommodate in a scratchpad. This has led some researchers to propose a migration to programming languages that force data accesses to take a more predictable form [3].

This work is also related to measurement-based WCET analysis. An execution time measurement is guaranteed to equal the WCET if the program is *single-path*, as advocated by Puschner [22]. Provided that loop bounds are known, any conditional branch can be if-converted by adding predicates to the operations that follow it. So any program that could be WCET analyzed can also be converted to a single-path form. However, there is a performance reduction due to the need to fetch and decode all of the instructions in an *if* statement (including the *not taken* branch) and a predictable memory subsystem is assumed. Bernat et al. [5] describe a way to transform statistics sampled from a CPU running a conventional (multi-path) program into a probabilistic model for that program's WCET. It deals with the timing noise problem (section 3) by trying to sample as many execution scenarios as possible; in contrast, the VTC approach *eliminates* timing noise but requires CPU modifications. Betts and Bernat [6] use an *instrumentation point graph* for WCET analysis. Instrumentation points are inserted when measurements are sampled. They divide the program's CFG into a TCFG-like arrangement where the paths between points are analogous to traces. The work has been criticized as it has the potential to produce an unsafe WCET estimate, but the probability of safety can be computed, and it can applied to any CPU without modifications.

## 7 Conclusion

This paper has discussed and evaluated the *virtual trace controller* (VTC): an architectural mechanism for WCET analysis and reduction. Previous work [30] indicates that traces enable the WCET to be calculated exactly (i.e. with no pessimism). This paper has shown that a CPU can be modeled precisely using measurements alone (section 4.1). The O3+VTC simulated model has demonstrated that (given sufficient trace length $L$) the guaranteed through-

put exceeds that of an idealized in-order CPU for *most* programs (Table 3). Further investigation reveals that it is useful to search for the best length $l \leq L$ for each virtual trace, in order to balance the main path execution time against the costs of side exits (Figure 7). Finally, experiments with alternative CPU configurations show that memory restrictions have the greatest effect on reducing throughput (Figure 8).

The results show that virtual traces are viable, and that the approach of constraining a complex CPU does work. The WCET was reduced by O3+VTC versus an idealized in-order CPU in many cases. But the results assume that the memory subsystem is predictable, i.e. that data and instructions can be allocated to scratchpad, and they show that it is important to know whether memory items could ever share an address or be forwarded. Together, these factors motivate the study of the wider research issue of writing code with known data access characteristics. Unless the CPU and WCET analysis model can be provided with more information about data accesses, the problems of data scratchpad allocation, predictable memory disambiguation and predictable memory forwarding can only be solved using a pessimistic strategy as in this paper.

## 8 Acknowledgments

Thanks to Andy Wellings for his comments on an early draft, to the delegates of the WCET workshop for their comments on our previous work, and to the anonymous reviewers for their suggestions regarding this paper.

## 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] R. Banakar, S. Steinke, B. sik Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proc. CODES*, pages 73–78, 2002.

[3] S. Bandyopadhyay, F. Huining, H. Patel, and E. Lee. A scratchpad memory allocation scheme for dataflow models. Technical Report UCB/EECS-2008-104, EECS Department, University of California, Berkeley, Aug 2008.

[4] I. Bate, G. Bernat, and P. Puschner. Java virtual machine support for portable worst-case execution time analysis. In *ISORC*, Washington, USA, Jan 2002.

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

[6] A. Betts and G. Bernat. Tree-Based WCET Analysis on Instrumentation Point Graphs. In *Proc. ISORC*, pages 558–565, 2006.

[7] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[8] R. Chapman. *Static Timing Analysis and Program Proof*. PhD thesis, 1995.

[9] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. ECRTS*, pages 37–44, 2001.

[10] M. de Michiel, A. Bonenfant, H. Cass, and P. Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *Proc. RTCSA*, pages 161–166, 2008.

[11] S. Edwards and E. A. Lee. The Case for the Precision Timed (PRET) Machine. Technical Report UCB/EECS-2006-149, EECS Department, University of California, Berkeley, Nov 2006.

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

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

[14] 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.

[15] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. DAC*, pages 456–461, 1995.

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

[17] M5. A modular platform for computer system architecture research. `http://www.m5sim.org/`.

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

[19] S. Mohan and F. Mueller. Hybrid timing analysis of modern processor pipelines via hardware/software interactions. In *Proc. RTAS*, pages 285–294, 2008.

[20] I. Puaut and D. Hardy. Predictable paging in real-time systems: A compiler approach. In *Proc. ECRTS*, pages 169–178, 2007.

[21] 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.

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

[23] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Syst.*, 18(2-3):115–128, 2000.

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

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

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

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

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

[29] J. Whitham and N. Audsley. Forming Virtual Traces for WCET Analysis and Reduction. In *Proc. RTCSA*, pages 377–386, 2008.

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