Traces as a Solution to Pessimism and Modeling Costs in WCET Analysis

Jack Whitham and Neil Audsley

July 1st 2008
Goal of this work

Experiment with CPU modifications that allow an increase in guaranteed throughput versus a simple CPU, without also increasing:

- pessimism
- static modeling costs
- safety
Goal of this work

Experiment with CPU modifications that allow an increase in \textit{guaranteed throughput} versus a simple CPU, without also increasing:

- \textit{pessimism}...
  - Can we get almost exact results?
  - Even for complex CPUs, e.g. superscalar out-of-order?
Experiment with CPU modifications that allow an increase in *guaranteed throughput* versus a simple CPU, without also increasing:

- *pessimism*...
  - Can we get almost exact results?
  - Even for complex CPUs, e.g. superscalar out-of-order?

- *static modeling costs*...
  - Can we use a CPU as its own static model without compromising *safety*?
Requirements

We consider CPU modifications that:

- allow speculative and superscalar out-of-order operation,
We consider CPU modifications that:

- allow speculative and superscalar out-of-order operation,
- but make such operation fully predictable,
Requirements

We consider CPU modifications that:

- allow speculative and superscalar out-of-order operation,
- but make such operation fully predictable,
  - so timings can be determined safely by measurement,
We consider CPU modifications that:

- allow speculative and superscalar out-of-order operation,
- but make such operation fully predictable,
  - so timings can be determined safely by measurement,
  - and so the WCET analysis model won’t include any pessimistic assumptions.
How to proceed

Two strategies have been considered:

- **Trace scratchpads**, in previous work.
  - Microcode is used to implement predictable execution.

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

- **Virtual traces**, in ongoing work.

Both share the common paradigm of a *trace*.
What is a trace?

1. executable code that is functionally equivalent to some machine code,

2. a timing model that gives precise information about path timings through that code.
What is a trace?

1. **Executable code** that is functionally equivalent to some machine code,
2. a **timing model** that gives precise information about **path timings** through that code.

(a) trace entrance

BB1 ▸ BB3 ▸ BB6 ▸ trace exit
trace exit 20 clock cycles

BB2 ▸ BB5

(b) trace implemented using microcode

microcode BB1,3,6

trace exit 45 clock cycles

BB4

exit returns to machine code

BB4
What’s a virtual trace?

A virtual trace attempts to deal with issues that are specific to trace scratchpads and their use of microcode, such as

- need for a custom CPU with writable microcode store,
- need for a CPU-specific compiler to generate microcode,
- poor code density of microcode.

Solution: replace microcode with a virtual trace that controls a conventional but constrained dynamic CPU scheduler.
How is it implemented?

1. Regard the CPU dynamic scheduler as a decoder: 
   \textit{machine code + virtual trace} → \textit{microcode}

2. Handle all events that could change execution times.
Benefits

- Use speculative and superscalar out-of-order execution predictably.
Benefits

- Use speculative and superscalar out-of-order execution predictably.
- The execution time of any path through any program is precisely known.
Benefits

- Use speculative and superscalar out-of-order execution predictably.
- The execution time of any path through any program is precisely known.
- Use \{your favorite static WCET approach\} for analysis.
Benefits

- Use speculative and superscalar out-of-order execution predictably.
- The execution time of any path through any program is precisely known.
- Use \{your favorite static WCET approach\} for analysis.
- CPU is the model: use measurements to determine execution times for paths.
A loop to sum the elements of an array \( p \):

```c
for (i=0; i != n; i++)
{
    c += *p;
    p ++;
}
```

This loop has one branch which is assumed taken. The trace *main path* includes \( L \) unrollings of the loop body.
Result

With $L = 20$:
Costs

- Peak throughput is reduced!
Costs

- Peak throughput is reduced!
  - This is no better than static branch prediction.
Costs

- Peak throughput is reduced!
  - This is no better than static branch prediction.
  - Memory effects (forwarding) are a drain on throughput.
Costs

- Peak throughput is reduced!
  - This is no better than static branch prediction.
  - Memory effects (forwarding) are a drain on throughput.
- Deterministic memory is assumed.
Costs

- Peak throughput is reduced!
  - This is no better than static branch prediction.
  - Memory effects (forwarding) are a drain on throughput.

- Deterministic memory is assumed.

- WCET analysis models of a program will be more complex, since there may be more than one way to execute a basic block, e.g. due to unrolling.
Preliminary Results

![Bar chart showing execution times for various programs. The x-axis represents different programs: bubble, cnt, compress, crc, edn, fdct, fir, insertsort, jfdctint, matmult, ndes, ns. The y-axis represents execution time in clock cycles. The chart compares in-order execution (green) and virtual trace execution (red) for each program.](chart.png)
How to reduce costs

Two possible strategies to improve the effectiveness of traces:

1. Consider some branches as both “taken” and “not taken” by converting some code to the *single-path paradigm*.
   - For each branch, WCET analysis will tell us if this will reduce the overall WCET or not.

2. Embed guarantees about forwarding if possible.
   - Difficult in the general case.
Virtual traces improve on scratchpad traces in many ways.
Virtual traces improve on scratchpad traces in many ways.

But how does the paradigm compare to the pessimism and modeling costs of more conventional approaches? Are the CPU modifications really worth doing?

All questions and comments are welcome!

Further information:
http://www.jwhitham.org.uk/pubs/
Virtual traces improve on scratchpad traces in many ways.

But how does the paradigm compare to the pessimism and modeling costs of more conventional approaches? Are the CPU modifications really worth doing?

What would the WCET community like to see next?
Conclusion

- Virtual traces improve on scratchpad traces in many ways.
- But how does the paradigm compare to the pessimism and modeling costs of more conventional approaches? Are the CPU modifications really worth doing?
- What would the WCET community like to see next?
- All questions and comments are welcome!
Conclusion

- Virtual traces improve on scratchpad traces in many ways.
- But how does the paradigm compare to the pessimism and modeling costs of more conventional approaches? Are the CPU modifications really worth doing?
- What would the WCET community like to see next?
- All questions and comments are welcome!
- Further information:
  http://www.jwhitham.org.uk/pubs/