TRACES AS A SOLUTION TO PESSIMISM AND MODELING COSTS IN WCET ANALYSIS

Jack Whitham and Neil Audsley[1]

Abstract

WCET analysis models for superscalar out-of-order CPUs generally need to be pessimistic in order to account for a wide range of possible dynamic behavior. CPU hardware modifications could be used to constrain operations to known execution paths called traces, permitting exploitation of instruction level parallelism with guaranteed timing. Previous implementations of traces have used microcode to constrain operations, but other possibilities exist. A new implementation strategy (virtual traces) is introduced here.

In this paper the benefits and costs of traces are discussed. Advantages of traces include a reduction in pessimism in WCET analysis, with the need to accurately model CPU internals removed. Disadvantages of traces include a reduction of peak throughput of the CPU, a need for deterministic memory and a potential increase in the complexity of WCET models.

1. Introduction

Worst-case execution time (WCET) analysis determines the maximum execution time for a program on a particular CPU [17]. On some CPUs, the execution of each instruction within the program is independent of execution history and data inputs, making the program easy to model for WCET analysis. Methods like the *implicit path enumeration technique* (IPET) [10] can find the exact WCET in these simple cases because the execution time of each section of code can be considered independently of all the others [18]. If the execution times are not constant, then it is still possible to use an upper bound on the execution time of blocks of code, yielding a *pessimistic* (i.e. inexact but safe) WCET estimate. In IPET, the program is modeled as a flow network, and *integer linear programming* is used to determine the execution path with the maximal execution time.

Out-of-order CPUs are difficult to analyze due to their complexity, but accurate analysis is necessary because a safe upper bound is required [8]. Advanced CPUs provide higher instruction throughput by using very long pipelines, executing operations in parallel, and executing operations *speculatively* (that is, before the branch that led to them): all of these features must be modeled. The rate of code execution is dependent on history (via branch prediction [5], caches [13] and pipeline interaction effects [11]) and dependent on data values (because of memory disambiguation [12] and variable duration instructions). Consequently, the upper bound may be much greater than the typical case. In some cases, a slower and simpler CPU might have a lower WCET as well as facilitating analysis.

Previous work suggested the use of microcoded *traces* as an abstraction of CPU behavior for WCET analysis [23, 22]. Traces support speculative and parallel execution within an IPET-based WCET

[1]Real-Time Systems Group, Department of Computer Science, University of York, York, YO10 5DD, UK.
email: {jack/neil}@cs.york.ac.uk

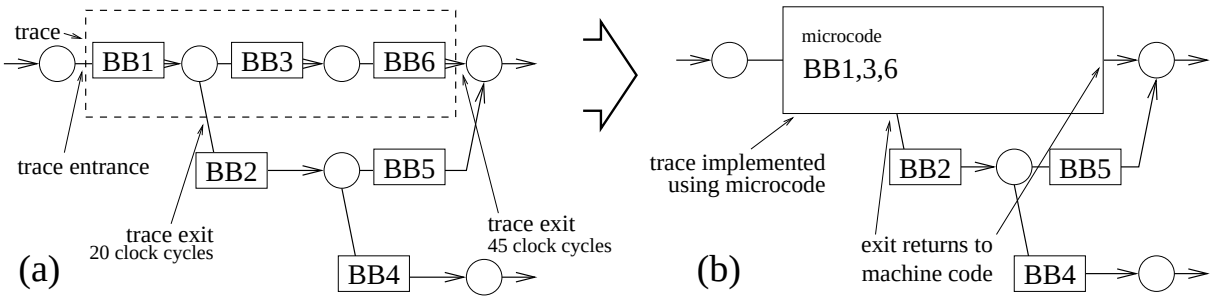

Figure 1. Trace formation in [23]: (a) basic blocks on the WCEP are selected for inclusion in a trace, then (b) their instructions are rescheduled into blocks of microcode, reducing the execution time of the path.

analysis model, potentially *without* introducing pessimism. Each trace is a sequence of basic blocks: a subpath of a program. Trace allocation algorithms try to match traces to subpaths of the *worst-case execution path* (WCEP) which maximizes a program's execution time. The WCEP might be changed by each trace allocation, so it is (1) initially estimated by assuming fixed execution times for code and performing WCET analysis, and then (2) refined by repeating WCET analysis after N trace allocations. When a path has been selected for trace formation (Figure 1(a)), a scheduler converts the basic blocks into functionally equivalent microcode (Figure 1(b)). The microcode is explicitly parallel and uses speculation to execute the operations along the main path [BB1, BB3, BB6] as quickly as possible. Hence, the local WCET of that path is reduced, and the WCET of the whole program may also be reduced if the main path is chosen correctly. The cost of adding the trace is that other paths may have increased execution time (e.g. [BB1, BB2]), so allocation algorithms must evaluate the overall WCET reduction benefits of each choice.

This paper details the motivation for traces using previous work (section 2) then describes the timing model that they enable (section 3). Then a new implementation strategy is discussed, avoiding microcode (section 4). Section 5 concludes.

2. Why use Traces?

Out-of-order CPUs exploit instruction level parallelism using a complex heuristic mechanism that attempts to run operations as soon as possible. The incoming instructions specify code for a sequential machine: an out-of-order CPU preserves in-order semantics while executing code in parallel where possible. The execution rate is limited by the data dependences in the code, the accuracy of predictions about future control flow, the resource limits of the CPU and the memory bandwidth. The first out-of-order pipelines were implemented in the 1960s using two different mechanisms to track dependences: a *scoreboard* (in the CDC 6600 [2]) and Tomasulo's algorithm (in the System 360 CPU [20]). Since then, the designs have been greatly refined [19].

As part of real-time systems design, it is necessary to compute the WCET of various programs running on the CPU. If the CPU is an out-of-order CPU, then building an accurate model will be costly [8]. The costs can be reduced by improving CPU predictability, e.g. by using locked caches [6] or scratch-pads [15] to replace unpredictable memory systems, and replacing dynamic branch predictions with static predictions [4]. But dynamic behavior still exists in the operation scheduler which is affected by history and data dependences. Accounting for all possible behaviors turns WCET computations into pessimistic estimates, so the CPU resources are under-utilized, particularly if a suboptimal con-

dition such as a *domino effect* [11] or *timing anomaly* [21] is possible. Such conditions are handled by incorporating pessimism into the CPU model [9] or by adding pipeline synchronizing instructions to the code to prevent the effect [11]: both approaches increase the WCET estimate.

To *avoid* CPU modeling, the probabilistic WCET approach has been proposed, where a statistical model of the execution time of a program is built automatically using measurements [3], but this approach is not suitable for all applications because the upper bound cannot be guaranteed. Alternatively, modeling costs can be *reduced* by constraining a complex CPU to intermediate deadlines obtained using a simpler CPU model. In VISA [1], programs execute on an out-of-order CPU that is downgraded to predictable in-order operation if an intermediate deadline is not reached. Unfortunately, this limits the WCET to that of an in-order CPU. Finally, modeling costs can be eliminated entirely by *single path programming* [16] where branches are replaced by predicated execution, since single path programs have constant execution times. Here, the WCET is limited by predication effects, since instructions in both the *if* and *else* cases of a conditional statement pass through the CPU.

The trace approach is related to all of the above. As in [3], it is observed that superscalar out-of-order CPUs are (1) difficult to model, and (2) models are pessimistic in any case, so it is best to avoid making a model of the CPU. Measurements obtained for probabilistic WCET allow statistically valid observations to be made regarding the WCET, but traces require a finite number of measurements which always cover all possible behaviors while probabilistic WCET requires a potentially unbounded number of measurements for a high degree of confidence in the WCET computation. As in [16], it is observed that removing control flow permits direct measurement. Traces remove the need to model internal control flow within the CPU (leaving program control flow) while single path programming removes both types of control flow. Finally, as in [1], an out-of-order CPU is modified to provide guarantees about timing, but the trace model accommodates speculative out-of-order execution instead of enforcing an upper timing bound.

3. What is a Trace?

In this paper, a trace is (1) executable code and (2) a timing model to represent the properties of that code. As executable code, a trace replaces sequential machine code in one or more basic blocks, forming part of a path through the program. It is *functionally equivalent* to that code, but *executes in less time*, at least along its *main path*. This is the same definition that is widely used in previous work, e.g. [7]. Traces have been previously implemented using microcode [23], but this is not necessary (section 4). An entire program could be composed of traces (as in this paper), or traces might be combined with predictable in-order execution (as in [23]). As a timing model, a trace is a subgraph of a *timing graph* (T-graph), as proposed in [18] for WCET analysis using IPET. The model is:

1. A trace always begins execution with the internal parts of the CPU in a well-defined state. The next instruction to be executed is the beginning of a basic block e, known as the *entrance*.

2. A trace has $1 \leq n \leq L + 1$ exits: when these are reached, execution may move to another trace. Figure 2(a) shows a trace with three exits.

3. A trace requires a precisely known number of clock cycles to reach each one of the n exits from the entrance. 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 $t(P_{e,i})$.

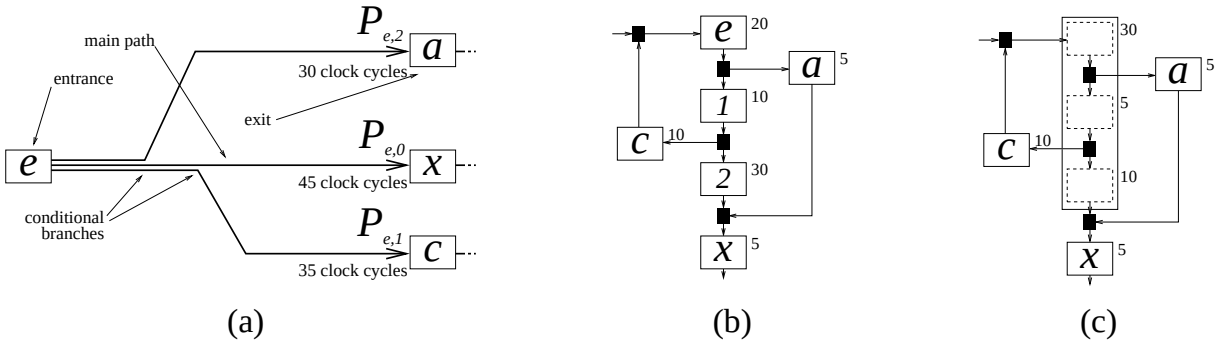

Figure 2. (a): Three paths through a trace beginning at basic block e. The paths lead to basic blocks a, x and c. (b): T-graph containing a, x and c. (c): T-graph incorporating the trace. The execution cost of each basic block is shown. The cost of the paths $e \rightarrow a$ and $e \rightarrow c$ increases slightly, but the cost of the main path $e \rightarrow x$ is decreased.

4. A trace contains up to L conditional branches along the *main path* $P_{e,0}$ ($e \rightarrow x$ in Figure 2(a)). 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.

5. An exit is taken when a branch condition is evaluated as True or the main path's end is reached.

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

The purpose of the trace is to reduce the execution cost of the main path $P_{e,0}$ by permitting speculation and out-of-order execution along this path. The cost of other paths may be reduced or increased. Because each $P_{e,i}$ is constant, it is possible to use exact IPET analysis (without pessimism): every trace is composed of "basic blocks" in microcode, each with constant execution times, permitting IPET to determine exact results [18]. The T-graph shown in Figure 2(b) is transformed to the T-graph in Figure 2(c) by the trace shown in Figure 2(a). More complex transformations are required when a trace represents an unrolled loop, because a basic block may be executed in multiple contexts [22].

4. Constraining CPU Behavior

The dynamic operation scheduler's behavior can be predicted precisely if hardware exists to (1) reset the scheduler to a known state, and (2) constrain all of the external inputs that could affect it. This can be used to implement *virtual traces*, which share the trace timing model (section 3) but use the dynamic operation scheduler in place of microcode. Figure 3 shows a diagram of a dynamic scheduler with external inputs, showing every source of *noise* that could affect execution. To fit the timing model in section 3, virtual traces must specify a *main path* through the program, and the execution time of that path (and all exit paths) must be an exact number of clock cycles. To implement virtual traces, dynamic scheduler inputs are restricted as follows:

• *Cache stalls* can be eliminated by cache locking [6] or by using *scratchpad memory* [15]. Like caches, scratchpads are on-chip memories that can allow programs to avoid slow and energy-intensive accesses to off-chip memory. But unlike caches, scratchpads are not automatically updated during program execution: they must be explicitly loaded by a program [14]. This is not as convenient as a

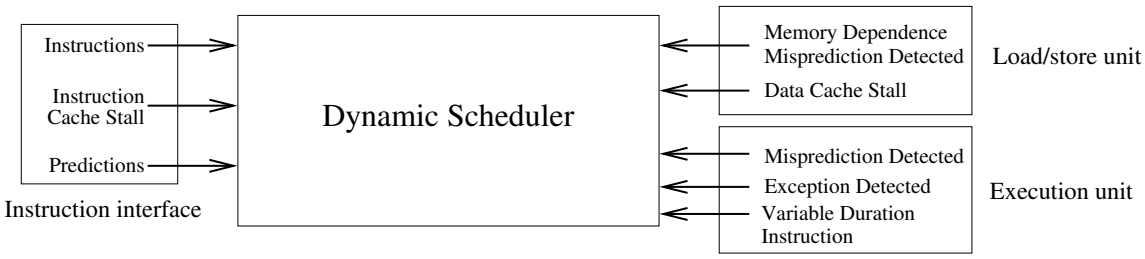

Figure 3. Sources of noise that could affect the operation of a dynamic operation scheduler.

cache but it is easy to predict the latency of a memory access. No cache modeling [13] is required, so the complexity of analysis is reduced.

• *Memory dependence mispredictions* [12] can be eliminated by enforcing a safe ordering on memory operations: load operations cannot be reordered across store operations, and store operations cannot be reordered at all. Load/store forwarding is disabled as it is data dependent.

• *Variable duration instructions* can be eliminated by forcing a fixed (upper bound) duration.

• *Exceptions* are discarded; many programs do not use them. (They could be modeled as conditional branches if necessary.)

Other inputs shown in Figure 3 are accommodated:

• *Branch predictions* fit into the trace model; the dynamic branch predictor is replaced by a representation of the trace. It (1) generates predictions so that instruction fetching follows the main path through the trace, and (2) considers the detection of a misprediction as an exit from the trace. However, the hardware must ensure that *branch operations are executed in program order*, since that prevents $n > 1$ misprediction events being active at the same time, leading to up to $2^n - 1$ possible exit conditions instead of 1. This can be done through the instruction dependence mechanism.

• *Instructions* also fit into the trace model: they are fetched along the main path, and when the end of the main path is reached, fetching is stalled. This prevents further instructions introducing noise.

The previous state of the scheduler may also have an effect on the schedule. This could be handled by (1) adding a reset function or (2) stalling the incoming instructions until the pipeline is drained.

4.1. Benefits

The CPU modifications guarantee that the operation scheduler is not affected by execution history (except within each trace) and that operation is not data dependent. This allows speculation and out-of-order execution along the main path. The speculation that occurs is always predictable. Inputs always arrive at known intervals, so the scheduler always does the same thing: following one of the paths $P_{e,i}$ through each trace.

The changes affect the load/store unit (removal of load/store forwarding) and the execution unit (removal of variable duration instructions). There are new dependences for branches and memory operations. Finally, a device is added to manage the execution of virtual traces (Figure 4). This is a

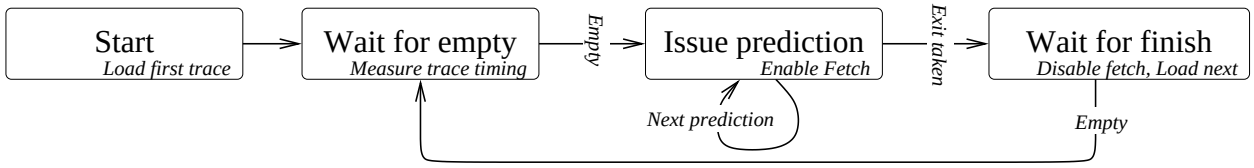

Figure 4. Virtual trace controller state machine: waits for the pipeline to empty before beginning the next trace.

simple state machine that enforces a strict order on trace execution, ensuring that each trace has fully completed before the next one begins.

This arrangement allows each $t(P_{e,i})$ value to be measured using the CPU, which is treated as a black box. Hence, CPU modeling costs are very low. Given these timing values, programs composed of traces can be represented within an IPET model as in [23]. The only limit on the number of traces is the storage space required: for virtual traces, the stored data comprises branch predictions and the length, so space requirements are minimal. As in VISA [1], the CPU modifications could be turned on and off dynamically, allowing real-time tasks to be mixed with non-real-time tasks on the same platform without any interference between the two. The approach could also be combined with single path programming [16] by using predication to remove conditional branches in frequently executed code: this could be beneficial since fewer exits would exist in each trace, and consequently opportunities for parallelism would be increased.

4.2. Costs

Previous work suggests three potential problems with traces, independent of the implementation:

• *Reduction in Peak Throughput* - this is almost certain to be lower than the peak throughput of a similarly-configured CPU optimized for ACET reduction. For example, the pipeline is emptied at every exit from a trace. Typical CPU designs would attempt to do useful work during this time, such as executing the next piece of code, but that could interfere with subsequent timing. Enforcing an order on memory operations will also reduce throughput [12].

• *Deterministic Memory Assumption* - every memory access must respond in a known time period. Cache stalls disturb the operation of the pipeline, perhaps introducing timing anomalies [21]. In an environment with multiple CPU cores, this could be particularly problematic as bus contention would also be a factor. Scratchpads could be used as a replacement for caches, but this increases the engineering difficulty of building the program [15, 14].

• *Analytical Complexity* - the IPET model becomes more complex when traces are introduced, because (1) there are more basic blocks, and (2) the new trace basic blocks are linked to the constraints on the original basic blocks [23]. Although the total number of integer linear program constraints is only increased by $O(n)$ for a program with n basic blocks, the difficulty of solving the IPET problem could still be vastly increased due to the NP-hard nature of integer linear programming problems.

5. Conclusion

This paper has explained the motivation for traces, outlined a WCET analysis model for them, and described a way to implement virtual traces by modifying a superscalar out-of-order CPU. Likely benefits and costs have been discussed.

Future work will use a simulated implementation of virtual traces to determine the exact costs of the restrictions on throughput. It would be interesting to compare these to the pessimistic assumptions that would otherwise need to be made in order to determine the WCET. Intuitively, the pessimism inherent in traces is likely to be lower than the pessimism from analysis, and consequently traces could provide higher guaranteed performance. Low CPU modeling costs are another benefit. Despite this, the disadvantages of traces (section 4.2) may be prohibitive. Further research will provide more information about the costs and benefits of the ideas.

6. Acknowledgments

Thanks to the anonymous reviewers for their helpful suggestions.

References

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

[2] Gordon Bell. CDC 6600 registers (online, accessed 3 June 08). `http://research.microsoft.com/~gbell/craytalk/sld040.htm`.

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

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

[5] Claire Burguiere and Christine Rochange. A Contribution to Branch Prediction Modeling in WCET Analysis. In *Proc. DATE*, pages 612–617, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Heiko Falk, Sascha Plazar, and Henrik Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proc. CODES+ISSS*, pages 143–148, New York, NY, USA, 2007. ACM Press.

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

[8] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE*, 91(7):1038–1054, 2003.

[9] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Syst.*, 34(3):195–227, 2006.

[10] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. DAC*, pages 456–461, 1995.

[11] Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proc. RTSS*, page 12, 1999.

[12] Andreas Moshovos. Exploiting load/store parallelism via memory dependence prediction. In *Speculative Execution in High Performance Computer Architectures*, pages 355–392. CRC Press, 2005.

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

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

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

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

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

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

[19] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. on Computers*, 39(3):349–359.

[20] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. R&D*, 11(1):25–33, 1967.

[21] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Proc. Int. Conf. Quality Software*, Sep. 2005.

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

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