Preserving Timing Anomalies in Pipelines of High-End Processors

Sibin Mohan and Frank Mueller
Dept. of Computer Science, North Carolina State University,
Raleigh, NC 27695-7534, mueller@cs.ncsu.edu

Abstract

Many embedded systems are subject to temporal constraints that require advance guarantees on meeting deadlines. Such systems rely on static analysis to safely bound worst-case execution (WCET) bounds of tasks. Designers of these systems are forced to avoid state-of-the-art processors due to their inherent architectural complexity that results in non-determinism. Such micro-processors are typically tuned to reduce average-case execution times — at the expense of predictability. Dynamic instruction scheduling techniques, such as out-of-order (OOO) execution, are examples of features that reduce average time but are statically unpredictable at large.

This work addresses this problem by providing analysis techniques for characterizing the worst-case behavior of real-time systems on modern processor architectures. We propose minor enhancements to processor architectures that, coupled with static analysis techniques, support the derivation of safe WCET bounds. We also introduce novel pipeline analysis techniques for accurately capturing the worst-case behavior of real-time tasks, i.e.*, methods to capture ("snapshot") pipeline state and to subsequently perform a "merge" of previously captured snapshots. We prove that our pipeline analysis correctly preserves worst-case timing behavior on OOO processor pipelines. We further specifically show that anomalous pipeline effects, effectively dilating timing, are preserved by our method. To the best of our knowledge, this method of pipeline analysis and interactions between hardware/software for obtaining WCET bounds on OOO processors is the first of its kind.*

1. Introduction

Each year, billions of microprocessors are used in embedded systems [29]. This is in sharp contrast to a few hundred million desktop processors that are sold in the same time frame. From automobiles to medical equipment, thermostats to space shuttles, embedded systems are all around us yet designers of such systems are inhibited in the type of processors they can use. They are often forced to use older, less sophisticated microprocessors, even when their application domain has high computational requirements and would profit from state-of-the-art architectures. Microarchitectural complexity and lack of analysis tools for contemporary processors are often to blame for this condition. The situation is exacerbated in many embedded systems that have strict timing constraints (often expressed in the form of "deadlines") on task execution. Such systems are often referred to as "real-time systems."

Real-time systems require advance knowledge of the worst-case behavior of constituent tasks. This is to aid in the process of offline schedulability analysis. Various techniques exist to obtain worst-case execution times (WCETs) for real-time tasks. These techniques are often categorized as being either *static* timing analysis techniques [4, 7, 8, 10, 17, 18, 20, 22, 28] or *dynamic* timing analysis techniques [5, 31, 33]. Complex architectural features, such as out-of-order (OOO) processing [21] and branch prediction [26], are often beyond the reach of static analyses, mainly due to the fact that they introduce non-determinism into the task code. These issues cannot be resolved at compile time forcing real-time system designers to completely avoid the use of such processors. Dynamic timing analysis, on the other hand, has been shown to be dangerous because it may result in underestimations in WCET estimation, which can lead to fallouts dangerous to users, the environment or both [32].

In recent work, we introduced the notion of "hybrid" timing analysis [16] called the *CheckerMode* infrastructure. This method combines the best features of both static and dynamic analysis to obtain accurate WCET estimates for real-time tasks running on modern microprocessors. We proposed minor enhancements to the microarchitecture of future processors that aid in the process of timing analysis. A "checker mode" is added to processors that, on demand, captures information in the form of "*snapshots*" of processor state. This information, along with accurate timing information for each path, is sent to a software module (similar to static timing analyzers) that is then able to obtain accurate WCET values for the entire program. When timing alternate paths, information from a previously captured snapshot is "restored" onto the processor function units to reflect the state of the system when the choice between the paths was made.

In this paper, we *(a)* more formally define the semantics of a snapshot; *(b)* explain how the information in a snapshot is obtained; *(c)* illustrate how two or more snapshots are "merged", which occurs when multiple control paths "join" together and *(d) prove* that the mechanisms for capturing and merging snapshots are correct in that they retain all worst-case pipeline effects. We also explain how

our mechanisms to capture and merge snapshots are able to correctly handle "timing anomalies" [3, 12, 13, 25]. To the best of our knowledge, these techniques to capture/merge pipeline information, coupled with hardware/software interactions to accurately gauge the worst-case execution times of real-time tasks and techniques for correct handling of timing anomalies are, the first of their kind.

The remainder of this paper is organized as follows: Section 2 lists the assumptions under which we operate. Section 3 gives a brief overview of the CheckerMode infrastructure. Section 4 introduces our notion of snapshots while Section 5 explains the models used for the analysis in this paper while Section 7 explains the techniques to capture the behavior of instructions in the pipeline. This is mainly aimed at capturing structural and data dependencies in an accurate manner. Section 6 details how a snapshot is captured while Section 8 provides the context on how these snapshots can used. Section 9 discusses how two or more snapshots are merged (before a join point in the control flow) so that the processor are reset to a consistent state for the following instructions. Section 10 proves that pipeline effects that modify timing will be retained post-merge. Section 11 develops a simple mechanism to merge register files. Section 12 discusses the details of our implementation. Section 13 compares this work to related work, while Section 14 talks about our goals for future work. Finally, we conclude in Section 15.

2. Assumptions

- We constrain ourselves to analyzing the unpredictable nature of out-of-order (OOO) instruction execution in contemporary, high-end embedded processor pipelines.
- Other complexities, such as memory hierarchies (which includes caches), dynamic branch prediction, *etc.* are beyond the scope of this initial work and will be addressed in the future.
- Tasks are assumed to execute in isolation.
- The issue of preemption delays (including cache-related preemptions delays) is orthogonal to this work, but existing [23, 24] and future techniques to handle these issues can be incorporated (with minimal changes) into our framework.
- We assume that there are no loops in the tasks being analyzed. We are currently in the process of finalizing a *fixed-point* method to capture the behavior of loops in the task code, which is not explained here due to space restrictions.

Note: The process of timing analysis, in our framework, amounts to timing sequences of paths coupled with saving and restoring processor context (captured in the form of "snapshots") in a co-ordinated fashion. While this process can be lengthy, it still remains independent of program inputs and can be run overnight, even perhaps in a parallelized fashion. This is an **offline** task to be performed during systems design and/or validation. Hence, cost is secondary and *does not* affect the dynamic, run-time behavior of the system after it has been deployed. In practice, such extensive verification is generally only warranted after either extensive code changes (development, upgrades, software deployment, *etc.*) or when the hardware itself must be reconfigured/upgraded.

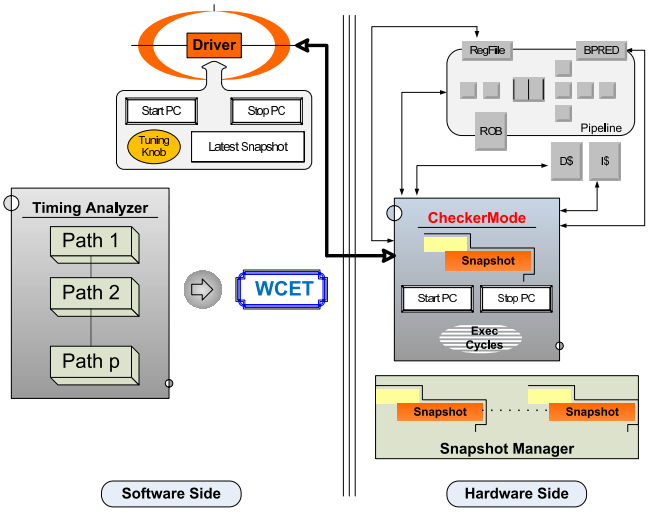

Figure 1. CheckerMode Design for High-Confidence WCET Analysis

3. CheckerMode

The CheckerMode infrastructure [16] depicted in Figure 1 provides the means to obtain accurate WCET values for real-time tasks executing on modern processor pipelines. We proposed modifications to the design of embedded processors so that, in addition to the ability to execute software normally (*deployment* mode), processors are capable of executing in a novel *checker* mode. This is to aid in the process of timing analysis.

CheckerMode provides cycle-accurate bounds on a task's WCETs by analyzing alternate execution paths in a program. In *deployment* (normal execution) mode, the processor executes along just one path following conditional branches. Which path executes may depend on input data. In *checker* mode, processors no longer proceed with conventional data-driven execution. All alternate paths that follow a conditional branch are executed, one at a time. Before executing one of the (possibly) many alternate paths, the original execution context (pipeline state, register file, caches state, etc.) is restored onto the processor to ensure that paths execute in isolation from one another. These low-level WCET results are propagated inter-procedurally in a bottom-up fashion over the combined control flow (and call graphs) until the WCET for the entire task has been obtained. **Note**: The "checker mode" is turned off or disabled when the system is in deployment mode, *i.e.*, when the system is active and performing the tasks it was designed for.

The CheckerMode framework supports the following capabilities:
(1) The processor context can be captured as snapshots.
(2) The processor can be reset to a previously captured state.

(3) The processor can be started/stopped in its execution for any pair of start/stop counter (PC) values.
(4) The timing analyzer (TA) breaks down task code into its constituent paths and also determines the start and end points for alternate execution flows, *i.e.*, points where snapshots must be captured and merged, respectively. The TA also calculates the final WCET for the entire task based on information obtained from the hardware side.
(5) The driver controls the entire framework. It controls start and stop of executions, when snapshots must be captured, and when the state of the processor must be reset to a previous captured or newly merged snapshot.
(6) Altered instruction semantics to handle values that depend on program inputs, many of which are known only at run-time.

Capturing/restoring snapshots and constraining instruction flow through the pipeline can be achieved in one of two ways, as we introduce in this paper: *(a)* by enhancing the *fetch* and *retire* stages of the pipeline so that fine control can be exercised on *when* instructions are to be fetched and from *where* in the program flow as well as when they should retire; or *(b)* by inserting nop instructions to cover "bubbles" in the program flow. The latter technique is less invasive and requires hardly any changes at the micro-architectural level. Since the driver in the CheckerMode infrastructure has overall control of the framework, it could periodically inject nops to maintain the correctness of the analysis. **Note**: We still require minor modifications to the *retire* stage of the pipeline: *(i)* to take note of when certain instructions retire and *(ii)* to ensure that instructions retire only at predetermined points in time, which is similar to injecting nop instructions between retiring instructions. We also require knowledge of structural dependencies between instructions that have been *issued*. This is to retain the worst-case behavior in the pipeline. This technique used to capture these effects is explained in more detail in Section 7.

4. Snapshots

Snapshots describe the state of the processor captured while performing timing analysis using our "hybrid" CheckerMode technique [16] to obtain the worst-case execution time for modern processor architectures. It typically consists of the state of each functional unit of the processor at a given point in time (t). This state includes, but is not limited to:

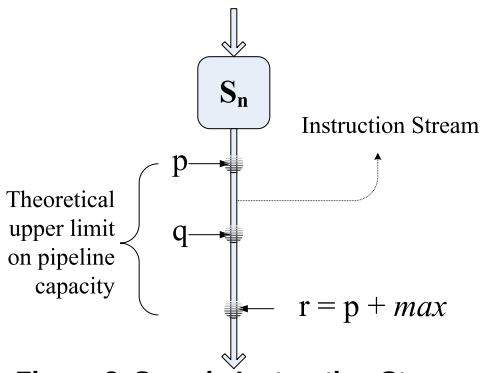

Figure 2. Sample Instruction Stream

I. **pipeline state**: in a generic sense, the state of instructions in the pipeline. Ideally, this state includes a description of *which* instructions are at *what* stage in the pipeline at time t. It also includes the contents of the register file.
II. **cache state**: the contents of the instruction and data caches at t. This information could be either *(a)* the complete cache contents or *(b)* incremental difference compared to the last snapshot. It could also be a combination of the two, where periodically we capture the state of the entire cache, but in between we only store the incremental differences (so-called deltas).
III. **branch predictor state**: similar to the cache state above: *(a)* complete branch history register and branch table contents; *(b)* delta from previous snapshot; or *(c)* a combination of the two. We are concerned with dynamic branch predictors in this work.
IV. **your favorite processor unit**: state from any additional/future processor units that needs to be captured to accurately characterize the worst-case behavior of the processor.

In this work, we focus on capturing the **pipeline** information of the processor for snapshots and not on caches, branch predictors, *etc.* Analysis of instruction caches is a solved problem, and any such analysis can be plugged into our framework to obtain better worst-case results. Analysis of data caches is a hard problem but some analysis does exist [19, 23, 30, 34], results from which can also be inserted into our framework to tighten the WCET results. We intend to analyze the branch predictor as part of future work.

While we would like to capture fine-grained details of instruction flow through the pipeline (defined above as "pipeline state"), practical difficulties prevent us from doing so. Many changes to the design and implementation of the processor will have to be carried out to attain the ability to observe every single stage of the pipeline, instructions in flight, data forwarding, etc. Hence we have devised a technique to capture pipeline information, which, in essence, achieves the effect of characterizing the state of the pipeline at the given instant. We call this the "drain-retire" (DR) technique. The DR technique is based on the idea that the only point of predictability in an out-of-order pipeline is at the retire stage. Since retire happens *in-order*, we can be sure that the retire order of instructions is deterministic. We discuss the DR technique in more detail in Sections 6 and 8.

5. Analysis Model

Figure 2 shows a section of the instruction stream that is executing through the pipeline. Let S_n be the last snapshot that was captured. Let "max" be the maximum number of instructions that can fit into the pipeline assuming that there are no dependencies between any of them. This is the theoretical upper bound for the pipeline capacity and is typically never achieved in practice – due to the existence of dependencies between instructions, which introduce bubbles in the pipeline.

If r is the most recent instruction that was fetched into

the pipeline, then let p be the instruction that was issued max cycles earlier in the instruction stream. Hence, p is the farthest instruction in the stream that can directly affect r's flow through the pipe. Instructions before p have retired, and any resulting state changes have been committed.

Figure 3 shows the pipeline model that we assume for this work. Fetch happens in-order, but multiple instructions can be fetched in the same cycle. Similarly, retire also happens in-order and multiple instructions can retire in the same cycle. Hence, when we fetch r at time t_r^F (*i.e.*, the Fetch time for instruction r), let q be the last instruction that retired one cycle earlier at time t_q^R (*i.e.*, the Retire time for instruction q). From Figure 2, we see that q must lie between:

$$p \leq q < r \tag{1}$$

Note that q is no longer in the pipeline when r is being fetched. Hence:

$$t_q^R = t_r^F - 1 \tag{2}$$

6. Snapshot Capture using Pipeline Drain-Retire (DR) Technique

Ideally, capturing a snapshot at r would involve capturing information about which instructions are in what stage of the pipeline and how long they have been/will be there. This resembles a step curve of the instructions that are in the pipeline. This is not practical as we are unable to capture the precise information in a pipeline without significant changes in silicon. Instead, we use what we call a "drain-retire" mechanism to characterize the flow of instructions in the pipeline. We take advantage of the fact that in an out-of-order pipeline the only point where determinism is guaranteed is at the *retire* stage (instructions *must* retire in-order). The algorithm to capture a snapshot using the DR mechanism is as follows:

1. Stop fetching after r.
2. Store t_q^R, the time when q retired.
3. Let execution proceed through the pipeline until r retires (*i.e.*, the pipeline drains completely).
4. Track the retire time of every instruction from q up until, and including r (*i.e.*, t_r^R).

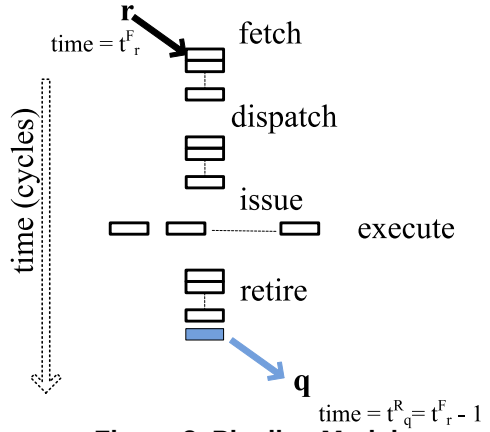

Figure 3. Pipeline Model

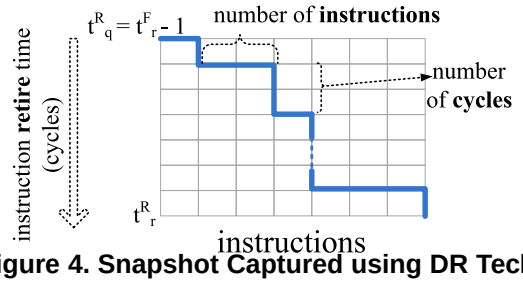

Figure 4. Snapshot Captured using DR Technique

Figure 4 shows the results of applying the above algorithm to the model and instructions described in Section 5. This figure shows the step curve obtained by tracking the retire times of all instructions following q until r retires. The vertical axis represents time while the horizontal axis represents the instructions that retire. Hence, the curve is bounded, in the time domain, by t_q^R and t_r^R with upper bound max. Unlike similar step curves for in-order pipelines, this curve is *multi-dimensional*. The horizontal axis now encodes information about groups of instructions that retire together. As the figure shows, the horizontal parts of the "step" directly represents the order and the number of instructions retiring at that particular point in time (*i.e.*, multiple instructions retiring in the same cycle). *Note*: we must also keep track of the *exact* order of instruction retirement at any given level. All of this information, combined with the "state" of the reservation stations (Section 7), now forms a pipeline *snapshot*, which is formally defined in Figure 5.

7. Capturing Structural and Data Dependencies using Reservation Stations

7.1. Structural Dependencies

Consider the situation shown in Figure 6. "a_1" and "a_2" are two *multi-cycle* instructions that require the same exe-

$$S_n = \left\{ q, t_q^R, \left\{ t_{\{i\}}^R, \{i\} \right\}, RES, RF, S_{<q} \right\}$$

where,

S_n : *snapshot at instruction* **n**

q : *last instruction to retire before* **n** *was fetched*

t_q^R : *retire cycle for* q

$\left\{ t_{\{i\}}^R, \{i\} \right\}$: *set of tuples where,*
$t_{\{i\}}^R$: *retire cycle*
$\{i\}$: *all instructions that retire at* $t_{\{i\}}^R$

RES : *state of the reservation stations immediately* **after** *instruction* **n** *has retired*

RF : *state of the register file immediately* **after** *instruction* **n** *has retired*

$S_{<q}$: *link/pointer to last snapshot before* q $(= \emptyset$ *if* S_n *is first snapshot)*

Figure 5. Definition of a Snapshot

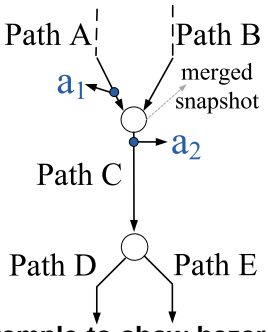

Figure 6. Example to show hazards affecting worst-case behavior

cution unit (for *e.g.*, the "floating point multiply" unit). Assume that there is only one instance of this type of execution unit in the pipeline. Now there exists a *structural dependency* between a_1 and a_2. Hence, a_2 cannot obtain access (be issued) to the execution unit before a_1 vacates it. We must retain this dependency across the join point (where alternate paths meet) because it could affect the worst-case behavior of execution that proceeds beyond it. Let us assume that path "D"'s execution time is affected by the fact that a_2 has to wait. Assume that the WCET for path "D" ($wcet_D$) increases or decreases by a factor δ depending on whether a_2 is made to wait or not, respectively. Let $wcet_E$ be the worst-case execution time for path "E". We now have the following pathological situation:

$$wcet_D - \delta < wcet_E < wcet_D + \delta$$

Hence, we see that even though "E" becomes the longer path (among "D" and "E"), it is *not* the worst-case path for the combination. The worst-case effects are seen when a_2 experiences a delay due to structural dependencies with a_1, thus delaying path "D" to exhibit a WCET of $wcet_D + \delta$. We now need a mechanism to capture these structural dependencies among instructions, especially those that lie on either sides of snapshots. We introduce the concept of *reservation stations* for this purpose.

Reservation stations for tracking structural dependencies, as shown in Figure 7, are implemented as a table with one entry per execution unit. It stores three values per execution unit:
(i) the **instruction** using that execution unit;
(ii) **entry cycle**: cycle/time when the instruction was issued the particular execution unit;

Exec Unit	Instruction	Entry Cycle	Exit Cycle
integer add	a	103	104
integer mult	b	104	110
⋮	⋮	⋮	⋮
float add	m	123	129
float mult	x	136	150

Figure 7. Reservation Stations

(iii) **exit cycle**: cycle/time when the instruction exited from the execution unit.

7.2. Data Dependencies

Most modern pipeline use data-forwarding techniques (bypass) to reduce the wait times for instructions that are waiting on data (register values) to become ready. When data is produced, at the end of the execution stage of certain instructions, it can immediately be forwarded to instructions that are waiting on them. These data values are available to waiting instructions even *before* they are written into the register file. Such data forwarding techniques and their effects must be characterized correctly, if the CheckerMode architecture is to correctly capture the worst-case behavior of tasks.

From Figure 6, let us now assume that a_1 and a_2 have *only* a data dependency among them(and not structural dependency as explained in the previous section) such that a_2 must wait for a_1 to write a result into a register (say r_1) that is a source register for a_2. If another instruction (say a_3) on path "B" also writes to r_1, but earlier than when a_1 would have written to it. Since a_1 and a_3 resides on the opposite side of a join point when compared to a_2, the latter can gain access to the register value (r_1) earlier than it would have otherwise on the "A" path, and can execute earlier. Hence, the worst-case behavior of the program is not correctly preserved.

We use reservation stations for the register file to correctly track the data dependencies between instructions and the time(s) when data becomes available and when they can be used by dependent instructions. Each register now has an associated "reservation station" which stores the *latest* cycle when the register value was available (*i.e.* after the execution stage of the instruction that produced that value).

Note: The process of writing information into the reservation stations is *not* on the critical path. Hence, it does not affect the execution of instructions in the pipeline.

8. Snapshot Usage

A snapshot captured using the DR technique in the previous section consists of information about *(i)* instructions, *(ii)* their order of retirement, *(iii)* cycles between retirement of instructions and *(iv)* the last instruction that retired (q) before the snapshot instruction (r). In this section, we elaborate on how this information will be used.

When execution must be restarted from a snapshot, we must start fetching from instruction q because instructions that preceded it cannot directly affect the execution of r. While we can start fetching from q, we do not have information about the processor state at q. Hence, we must restore the last snapshot before q, which is S_n, as seen in Figure 2. To account for the worst case, we must actually start from the snapshot that precedes instruction p. Instruction p is the instruction that is at the theoretical bound for the capacity of the pipeline, max, relative to r. If there exists a snapshot between p and q, then we can reduce the pessimism by starting at the snapshot that *immediately* pre-

```
// DRM Algorithm to merge two snapshots

drm_merge( snapshot S_a, snapshot S_b ) {
  S_m ← NULL; // merged snapshot
  do{
      t_1 ← get next retire cycle in S_a ;
      I_1 ← get instructions retiring at t_1 in S_a ;
      t_2 ← get next retire cycle in S_b ;
      I_2 ← get instructions retiring at t_2 in S_b ;

      S_m ← { max( t1, t2 ), I_1 ∪ I_2 };
  } while( not_empty(S_a) and not_empty(S_b));

  S_m ← remaining retire times and instructions
        from non-empty snapshot;
  S_m ← older( S_a.q, S_b.q );
  S_m ← older( S_a.prev_snapshot, S_b.prev_snapshot );

  S_m ← merge_reservation_stations( S_a.RES, S_b.RES ) ;

  S_m ← merge_reg_file_state( S_a.RF, S_b.RF ) ;

  return S_m;
}
```

Figure 8. Snapshot Merge Algorithm (DRM)

cedes q since we know that instruction q defines the *realistic* capacity of the pipeline.

We can now restore information from this snapshot to bring the processor into a consistent state. We restart fetching from the instruction that immediately follows S_n and let execution proceed until the new/current snapshot, S_r (at instruction r), is reached. Once instruction q retires, we look up the information about subsequent instructions from S_r to see how much time elapses, if any, between their retirement. Each instruction starting from r can now retire only after the a number of cycles has elapsed, as determined by the information in S_n. Hence, instructions can retire at or after the number of cycles recorded *for it* in the snapshot, but *never before*. From Figure 4, we see that two instructions following q can retire at the same time, but no earlier than 1 cycle after q. The next instruction can only retire 2 cycles after the previous 2 instructions have retired, and so on.

A similar process is employed for the *issue* stage of the pipeline. We use the reservation stations to control what instructions are issued, when and into what execution unit. From Figure 7, we see that an instruction that wishes to use an execution unit (say the integer multiplier), cannot gain access to it before cycle 104, and once it has been given access, another instruction cannot obtain it until the previous instruction exits (which, in this case, will be cycle 110). This process ensures that structural dependencies between instructions on either side of the snapshot are still retained.

Similarly, instructions that depend on certain register values to be ready cannot proceed with their execution until the cycle stored in that register's reservation station comes

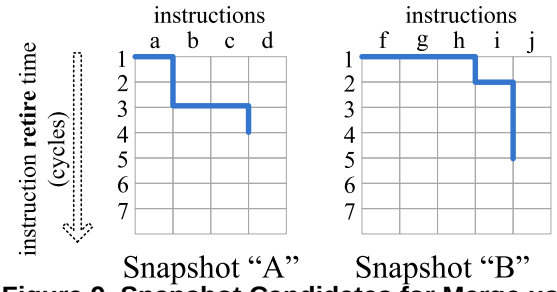

Figure 9. Snapshot Candidates for Merge using DRM Technique

to pass. This ensures that instructions that have data dependencies among them still retain the correct dependency information post-merge.

We see that the semantic meaning associated with snapshots, initially constrained to static aspects only, has been enhanced by dynamic information. This has an effect on the instruction flow through the pipeline. Hence, a snapshot is now defined as information that affects the flow of instructions through the pipeline when the "snapshot instruction" (r in the above case) is fetched. Snapshots are able to affect the pipeline behavior by allowing instructions captured in them to retire only with predetermined delays (*i.e.*, at or after the retire times captured in the snapshot) or gain access to execution units based on constraints enforced by the reservation stations, *etc.*. Modifications to the retire stage, mentioned in Section 3, as well as the issue stage (to look up the reservation stations) are useful in aiding the process of restoring snapshots by providing the ability to exercise fine control on when certain instructions (captured in the snapshot) are allowed to be issued or to retire.

9. Merging Pipeline Snapshots

When alternate paths meet, snapshots from both sides must be *merged* so that instructions that follow see a consistent state of the pipeline. Also, the merged state must inherit the worst-case behavior from either side. Another requirement is that pipeline effects resulting from anomalous behavior [12, 13] must still be retained post-merge. In this section, we present a merge technique that handles all of the above effects correctly. We shall refer to this merge technique as "*drain-retire merge*" (DRM). Section 10 will present a proof showing that timing anomalies are retained after applying DRM.

9.1. Merging Two Snapshots

Merging more than two snapshots is a simple extension of the same techniques. The algorithm to perform a merge of two snapshots (DRM) is illustrated in Figure 8. It proceeds by simultaneously retrieving snapshots from the top of each stack, extracting information and comparing it pairwise before composing a merged state and storing it in a new snapshot (S_m).

Remark: We pick the older "starting point" (q) from the two snapshots and also the older "last snapshot" ($S_{<q}$). This is to ensure that the state of the processor is correct when the merged snapshot state is restored within the processor.

To understand how this algorithm works, consider examples of snapshots shown in Figure 9. Snapshot "A" has three instructions (a, b, c) while snapshot "B" has four (f, g, h and i). Instructions d and j are not part of the snapshot. They are the first instructions that follow the snapshot. To perform the merge, we go through the following steps:

1. Start from the first (earliest) instruction in both snapshots.
2. Combine all instructions at the same cycle (level) into the new, merged snapshot. Hence, a, f, g and h will be combined so that they retire during the same cycle.
3. Compare instructions in both snapshots to find the snapshot with the longer number of cycles until the next retire occurs. From the figure, we see that b and c from "A" retire later than i from "B".
4. Set the retire time for all instructions on both paths to the longer delay. Hence, b, c *and* i will now retire at cycle 3.
5. Repeat for all remaining instructions/retire times in both snapshots.

Figure 10 shows the results of applying the DRM merge algorithm. After the merged snapshot is restored, we follow the instruction stream immediately after the join point. Also, for the sake of obtaining the WCET of the program, we must pick the longer path and its WCET for our analysis. If the path that provided snapshot "B" (P_B) is longer, then we use its instructions and WCET. The problem of finding the WCET for the entire program is then reduced to finding the longest path in short sections of code and using their WCETs for future calculations.

As explained in Section 8, restarting execution at a snapshot means restarting from an instruction that originates from before the particular snapshot (*e.g.*, instruction q in Figure 5). If P_B is the longer path, then q belongs to the mix of instructions that constitutes P_B. Even if P_B is very short and q happens to lie before the branch condition, we must pick the longer path to execute through the pipeline to reach the merged state, which will eventually pass through P_B in this case. Hence, at any point in time, only instructions from one path (P_A or P_B) will execute through the pipeline.

9.2. Incorrect Merge Technique

Figure 11 shows an incorrect way of performing the merge. In contrast to Figure 10, each instruction now retires at its original time. If we used the information from

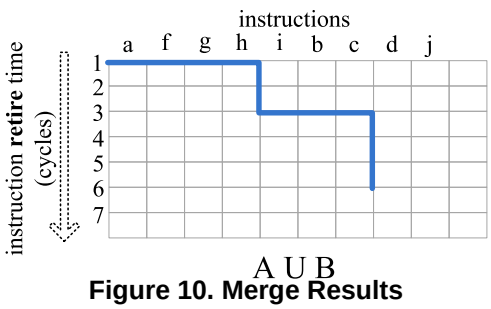

Figure 10. Merge Results

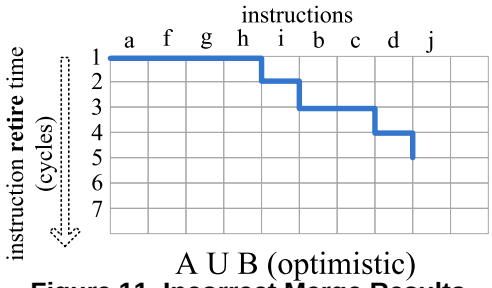

Figure 11. Incorrect Merge Results

Figure 11, we would always retire i at cycle 2 and not account for the fact that "A" could carry over some worse behavior where instructions b and c retire one cycle later.

While this alternate method may result in fewer cycles for the WCET, it does not safely capture effects due to the execution of alternate paths and the influence they may have on each other. The problem arises because we are searching for paths that show worst-case behavior *locally*. If effects from one part of the program affect the worst-case behavior of instructions that are at a large distance, then using this incorrect merge technique will result in wrong WCET estimates.

Consider the situation shown in Figure 12. Let paths P_A and P_B be the paths that produced snapshots "A" and "B", respectively. Let the result of merging them be as depicted in Figure 11. Let us further assume that P_B has the larger WCET. As explained before, only instructions from P_B will exist in the pipeline when this newly merged snapshot is encountered. Now let us assume that there exists a timing anomaly in this section of the program and the source of this anomaly is at point (I). Let us also assume that only certain instructions in path P_A (point (II) in the figure) depend directly on the instructions that form the anomaly. Instruction b in Figure 11 has its retire time increased due to this anomaly. Let other instructions following the merge (point (III) in the figure) depend indirectly (due to instructions at (II)) on the anomaly. Path P_B is not affected by the anomaly. *If we use the retire times shown in Figure 11, then the effects of the anomaly will not be felt post-merge, because we do not carry over the time dilation effects that resulted in an increase in (say) b's retire time.* This would have otherwise dilated the execution/retire times for instructions at point (III). Hence, by using the overly aggressive, incorrect merge we may not be handling worst-case pipeline effects (in particular timing anomalies) correctly. There exists a distinct possibility that future instructions (post-merge) will not execute based on the worst-case behavior. Such incorrect merge

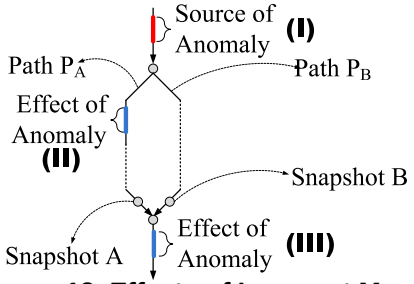

Figure 12. Effects of Incorrect Merge

techniques can result in an underestimation of the WCET estimates.

9.3. Merging Reservation Stations

The steps to perform a merge for reservation stations are shown in Figure 13. While merging reservation stations for the execution units, we pick the later of the two entry cycles as well as the later exit cycle. This ensures that the worst-case behavior of the program is carried forward post-merge.

```
merge_reservation_stations( S_a.RES, S_b.RES ) {
  for each ( execution_unit_entry E ){
      E_merged.res_station_entry_cycle =
              max(E_a.entry_cycle,E_b.entry_cycle);
      E_merged.res_station_exit_cycle =
              max(E_a.exit_cycle, E_b.exit_cycle);
  }

  for each ( register_entry R ){
      R_merged.register_cycle =
              max(R_a.cycle,R_b.cycle);      }

  return merged_res_station ;
}
```

Figure 13. Merging Reservation Stations

The merge for register reservation stations is similar in that a $\max$ of the reservation station entries from both paths is stored as the new value for that particular register. This ensures that instructions that execute post-merge cannot gain access to the register values until the cycle which is stored in the reservation station for that particular register. While the register values might be written earlier (from an alternate path), they cannot be used until the reservation station allows it.

9.4. Merge for More than Two Snapshots

The DRM algorithm can be extended to merge more than two snapshots. In such situations, we can call it recursively, as shown in Figure 14.

```
merge_n(S_1...S_n) {
      if( only two snapshots  S_x, S_y)
          return drm_merge( S_x, S_y ) ;
      return merge_n( merge_n(S_1...S_{n-1}), S_n ) ;
}
```

Figure 14. Merge for Multiple Snapshots

10. Proof of Correctness

The term "timing anomaly" refers to an anomaly in the execution of code in dynamically scheduled processors [12]. It was later generalized by others [3, 13, 25]. It denotes counter-intuitive results in timings, *e.g.*, a cache hit may result in longer execution times than a miss for a given path due to overlapped structural resource conflicts. These anomalous effects show up as pipeline effects, where execution times for instructions are dilated in ways that cannot

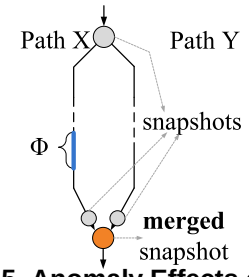

Figure 15. Anomaly Effects on Merge

be predicted easily. They prevent accurate modeling of out-of-order processors and thus prevent us from obtaining accurate estimates of worst-case execution times for such processors. Instead, we show how such effects can be safely bounded. Hence, any pipeline state merge algorithm must ensure that the effects in the pipeline due to such anomalies are retained, *i.e.*, the merge must not remove these anomalies from the pipeline and subsequent analysis.

Assumptions: We are only interested in pipeline effects in this work. Hence, any architectural/execution artifact that results in changes to the passage of instructions through the pipeline is considered. The causes could be internal (*e.g.*, data dependencies) or external (*e.g.*, cache hits/misses) to the pipeline. The causes for the effects could have occurred at a much earlier stage or just immediately before time dilation in the pipeline. Effects on other parts of the processor, including caches and branch predictors, are not yet considered here as they are subject to future work.

Theorem 1. *Correctness of Merging Two Snapshots: The DRM merge algorithm (Figure 8) retains all worst-case pipeline timing effects, including timing anomalies.*

Proof. (I) Consider the situation shown in Figure 15. It shows two alternate paths (X and Y with WCETs C_X and C_Y respectively), each of (possibly) different lengths. A snapshot is captured at the beginning (say S_{branch}) when the two paths diverge. Two snapshots are captured (say S_X and S_Y), one for each path, before the paths meet. These two snapshots are "merged" using the DRM algorithm to obtain the new, single snapshot (say S_m) that is used to initialize the state of the processor before execution proceeds. ϕ is the *potential* time dilation produced during the execution of path X due to pipeline effects (such as timing anomalies). Such dilation could lead to one of the following three cases related to the retire time of some instructions in X:

- **Case 1**: ϕ causes some instructions to retire *later*, *i.e.*, it increases the execution times for some (or all) instructions, thus resulting in an increase in C_X. These instructions also enter and leave their respective reservation stations later than they would have otherwise. They also produce results (and write them to register files) later.
- **Case 2**: ϕ causes some instructions to retire *earlier*, *i.e.*, it decreased the execution times for some (or all) instructions, thus resulting in a decrease in C_X. Hence, they are able to enter/exit reservation stations, as well

as produce results (to be written into registers) earlier than before.

- **Case 3**: ϕ does affect the retire times or reservation stateion usage for any instructions in the snapshot, *i.e.*, it neither increased nor decreased C_X.

(II) Consider the case of an arbitrary instruction k (part of path X) with its *original* reservation station times ([Entry, exit]) denoted as $[E, e]_k$, its retire time t_k^R which is part of snapshot S_X. Let RF_k be the time when the instruction writes its results (if any) into the register. Hence, RF_k is the time stored in the reservation station associated with the register that was written into by k. Let S_X also be affected by an anomaly. Hence, the time(s) of k and the WCET of X are now,

$$[E, e]'_k = [E \pm \phi, e \pm \phi]_k \quad (3)$$
$$t'_k = t_k^R \pm \phi \quad (4)$$
$$RF'_k = RF_k \pm \phi \quad (5)$$
$$C'_X = C_X \pm \phi \quad (6)$$

As part of the DRM process, k, its reservation station state and its retire time (t'_k) will be compared with instructions from the snapshot on the the alternate path (S_Y). Let $[E, e]_{\{i\}}$ and $RF_{\{i\}}$ be the state of the reservation stations and $t_{\{i\}}^R$ be the retire time that $[E, e]'_k$ and t'_k are being compared with (from the other snapshot), where $\{i\}$ represents the sequence of corresponding instructions from the other snapshot.

(III) **Case 1:** ***(a)*** ϕ increased the retire times for k. Hence,

$$[E, e]'_k = [E + \phi, e + \phi]_k \quad (7)$$
$$t'_k = t_k^R + \phi \quad (8)$$

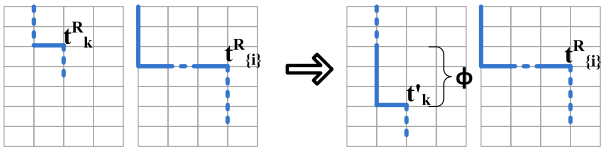

Figure 16. Case 1 (a) (i) t'_k is greater than $t_{\{i\}}^R$

(i) If $t'_k > t_{\{i\}}^R$ then the merged snapshot (S_m) will store t'_k as the retire cycle for all instructions $k \cup \{i\}$. Hence, we see (Figure 16) that the increase in time to retire for an arbitrary instruction k results in changes to the snapshot (instructions retire later), thus ensuring that the pipeline effect propagates beyond S_m.

Remark: These effects on S_m will materialize regardless of whether $t_k^R < t_{\{i\}}^R$ (seen in Figure 16) or $t_k^R > t_{\{i\}}^R$.

(ii) If $t'_k < t_{\{i\}}^R$ then the merged snapshot (S_m) will store $t_{\{i\}}^R$ as the retire cycle for all instructions $k \cup \{i\}$. We see here (Figure 17) that the increase in the retire time for k did not affect the snapshot. The retire time for k (t_k^R) would *never* have affected the merged snapshot because the larger

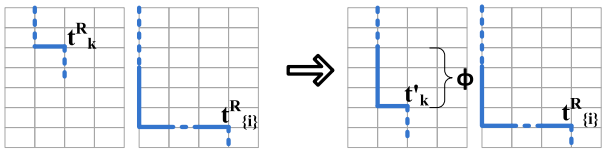

Figure 17. Case 1 (a) (ii) t'_k is less than $t_{\{i\}}^R$

$t_{\{i\}}^R$ value would have been picked anyways. This is due to the fact that we are trying to estimate the worst-case behavior of the program. We can also conclude that the pipeline effect would be contained within S_{branch} and S_m in this case because the retire time for the instructions affected are not part of the worst-case behavior of the path.

Remark: This situation can only occur if $t_k^R < t_{\{i\}}^R$ to begin with as shown in Figure 17. If t_k^R was larger, then it would default to case III (a) (i).

Case 1: ***(b)*** ϕ increased the WCET of X. Hence,

$$C'_X = C_X + \phi \quad (9)$$

(i) If $C'_X > C_Y$ (Figure 16), then the WCET for the entire construct will now be C'_X. Hence, the effects of ϕ will be included in the estimation of the total, increased WCET of the program. Again, this is regardless of whether $C_X > C_Y$ or $C_X < C_Y$.

(ii) If $C'_X < C_Y$ (Figure 17) then the WCET for the entire construct will now be C'_Y. This result means that the effects of ϕ would never have affected the WCET estimation of the program anyways as Y was always the longer path.

Remark: This result is only possible if $C_X < C_Y$ to start with, else we would default to case III (b) (i).

(IV) **Case 2:** ***(a)*** ϕ decreased the retire times for k. Hence,

$$t'_k = t_k^R - \phi \quad (10)$$

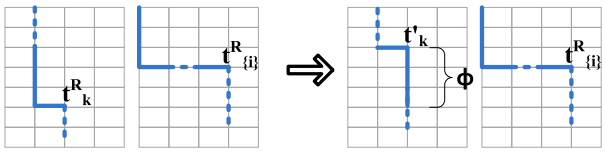

Figure 18. Case 2 (a) (i) t'_k is less than $t_{\{i\}}^R$

(i) If $t'_k < t_{\{i\}}^R$ then the merged snapshot (S_m) will store $t_{\{i\}}^R$ as the retire cycle for all instructions $k \cup \{i\}$. If $t_k^R < t_{\{i\}}^R$, then this change due to ϕ did not matter anyways, as k's retire time was not contributing to the worst-case state to be seen by future instructions. If, on the other hand, $t_k^R > t_{\{i\}}^R$ (as shown in Figure 18), then the effect of the anomaly is that it changed the worst-case behavior of instructions in path X. The significance of this effect is that instructions in the other path will contribute to the worst-case state that is carried forward beyond S_m, and the worst-case retire cycle is now $t_{\{i\}}^R$, which is less than the original t_k^R.

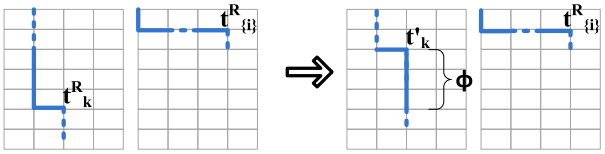

Figure 19. Case 2 (a) (ii) t'_k is greater than $t^R_{\{i\}}$

(ii) If $t'_k > t^R_{\{i\}}$, then the merged snapshot (S_m) will store t'_k as the retire cycle for all instructions $k \cup \{i\}$. Hence, we see that the decrease in retire time for k results in changes to the snapshot. Instructions previously retired at t^R_k now retire earlier (at t'_k) as seen in Figure 19. Thus the pipeline effect will propagate beyond S_m.

Remark: This condition holds only if $t^R_k > t^R_{\{i\}}$; otherwise, we would default to case IV (a) (i).

Case 2: ***(b)*** ϕ decreased the WCET of X. Hence,

$$C'_X = C_X - \phi \tag{11}$$

(i) If $C'_X < C_Y$ (Figure 18), then the WCET for the entire construct will now be C'_Y. If $C_X > C_Y$, then ϕ has already affected the WCET of the program. Where C_X would have been chosen originally for the WCET of the construct in Figure 15, C_Y is now chosen. Of course, if $C_X < C_Y$, then the anomaly limits its effects between S_{branch} and S_m. It does not affect the WCET for the two alternate paths because C_Y would have been chosen anyways.

(ii) If $C'_X > C_Y$ (Figure 19), then the WCET for the entire construct will now be C'_X. Hence, ϕ has resulted in a reduction of the WCET of the program from the original C_X.

Remark: This condition is true only if $C_X > C_Y$, else we revert to the situation in IV (b) (i).

(V) **Case 3:** ***(a)*** ϕ did not affect the retire times of any instructions in the snapshot. Hence,

$$t'_k = t^R_k \tag{12}$$

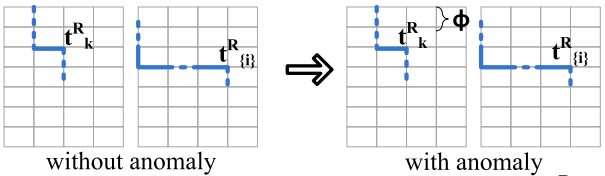

Figure 20. Case 3 (a) neither t'_k nor $t^R_{\{i\}}$ change

The effects of ϕ are completely encapsulated within the boundary between the two snapshots (*i.e.*, between S_{branch} and S_X). Hence, we need not consider the anomaly as it will not affect the execution of future instructions (beyond the merge point) because the effects of the anomaly on the pipeline have been dissipated/absorbed before the instructions in snapshot S_X reach the retire stage (Figure 20).

Case 3: ***(b)*** ϕ did not change the execution time of X (Figure 20). Hence,

$$C'_X = C_X \tag{13}$$

(VI) **Case 1:** ϕ increased the [Entry,exit] for k. Hence,

$$[E, e]'_k = [E + \phi, e + \phi]_k \tag{14}$$

(i) If $e'_k > e_{\{i\}}$, then the merged reservation station state will have an entry cycle of $\max(E'_k,\ E_{\{i\}})$ and an exit cycle of $\max(e'_k,\ e_{\{i\}}) = e'_k$. Hence, we see that instructions may gain access to the execution unit later than they would have (if $E'_k > E_{\{i\}}$). They are also not allowed to vacate the unit until later (e'_k). These effects are due to the increase in time by ϕ.

(ii) If $e'_k < e_{\{i\}}$, the merged reservation station state will have an entry cycle of $\max(E'_k,\ E_{\{i\}})$ and an exit cycle of $\max(e'_k,\ e_{\{i\}}) = e_{\{i\}}$. We see that the exit times for the merged state is not affected by this change. This is similar to the situation in II, Case (1)(a)(ii) where the effects of the anomaly would not have propagated since the reservation station state of the path comprising k does not reflect the worst-case behavior. Depending on whether E'_k or $E_{\{i\}}$ is greater, the instruction may or may not gain access to the reservation station earlier.

Case 2: ϕ decreased the [Entry,exit] for k. Hence,

$$[E, e]'_k = [E - \phi, e - \phi]_k \tag{15}$$

(i) If $e'_k < e_{\{i\}}$, the merged reservation station state will have an entry cycle of $\max(E'_k,\ E_{\{i\}})$ and an exit cycle of $\max(e'_k,\ e_{\{i\}}) = e_{\{i\}}$. If $e_k < e_{\{i\}}$, then the change did not matter since k's reservation station state was not contributing to the worst-case state of the merged snapshot. If, on the other hand, $e_k > e_{\{i\}}$, then the effect of the anomaly was to modify the worst-case behavior of instructions in the path. Instructions from the other path (with their corresponding state of reservation stations) will contribute to the worst-case behavior for the task. Hence, the effect of the anomaly is seen. Instructions that execute post-merge gain access to execution units earlier than they would have, thus reducing overall execution time and, hence, retaining the original effect of the anomaly.

Instructions that are a part of the snapshot also gain access to the reservation stations at a later time, depending on whether E'_k or $E_{\{i\}}$ is greater.

(ii) If $e'_k > e_{\{i\}}$, the effect of the anomaly was to reduce the time taken for instructions to be issued to execution units. The state of the merged reservation station will be, $[\max(E'_k,\ E_{\{i\}}),\ e'_k]$. Without reservation stations, instructions would have been able to exit from the execution units at e_k. Due to the presence of reservation stations, they now exit at time $e'_k < e_k$. Hence, the effect of the anomaly in reducing the execution time is carried forward beyond the merge.

(VII) **Case 1:** ϕ increased the RF entry for k. Hence,

$$RF'_k = RF_k + \phi \tag{16}$$

(i) If $RF'_k > RF_{\{i\}}$, then the merged register reservation station state will store the value, RF'_k. Hence, we see that instructions that depend on the register corresponsing to RF will gain access to the execution unit *later* than they would have due to the increase in time by ϕ.

(ii) If $RF'_k < RF_{\{i\}}$, the merged register reservation station state will will store the value $RF_{\{i\}}$. This is similar to the situation in II, Case (1)(a)(ii) and VI Case (1) (a) (ii), where the effects of the anomaly would not have propagated since the register reservation station state of the path comprising k does not reflect the true worst-case behavior.

Case 2: ϕ decreased the RF for k. Hence,

$$RF'_k = RF_k - \phi \tag{17}$$

(i) If $RF'_k < RF_{\{i\}}$, the merged register reservation station state will have a cycle of $RF_{\{i\}}$. If $RF_k < RF_{\{i\}}$, then the change did not matter since k's register reservation station state was not contributing to the worst-case state of the merged snapshot. If, on the other hand, $RF_k > RF_{\{i\}}$, then the effect of the anomaly was to modify the worst-case behavior of instructions in the path. Instructions from the other path (with their corresponding state of reservation stations) will contribute to the worst-case behavior for the task. Hence, the effect of the anomaly is seen. Instructions that execute post-merge are allowed to access the data written into the register file earlier than they would have, thus reducing overall execution time hence retaining the original effect of the anomaly.

(ii) If $RF'_k > RF_{\{i\}}$, the effect of the anomaly was to reduce the time taken for instructions to gain access to the data (they depend on) from the register file. The state of the merged register reservation station will be, RF'_k. Without register reservation stations, instructions would have been able to read the required data values at RF_k. Due to the presence of reservation stations, they now get the required inputs (from the register) at $RF'_k < RF_k$. Hence, the effect of the anomaly in reducing the execution time is carried forward beyond the merge.

Cases (I) – (VII) we proved that pipeline effects due to timing anomalies (or whatever other reasons) will be retained post-merge if the merge is based on the DRM algorithm. If the pipeline effects resulted in increases or decreases (execution time/retire cycles/*etc.*), then these effects are carried over if these effects changed the worst-case behavior of the path. Hence, this proof holds for merging any *two snapshots.* □

Theorem 2. *Correctness of Merging Multiple Snapshots: The algorithm in Figure 14 is correct with respect to preserving worst-case timing effects in the pipeline when merging multiple snapshots.*

Proof. The DRM algorithm is effectively applied recursively to perform merges on multiple snapshots. The "drm_merge" algorithm is called on *two* snapshots at a time to obtain a merged state, which is then merged with the next snapshot and so on. We have shown above that pipeline effects are not lost when merging two snapshots at a time. Since merging multiple snapshots occurs two at a time, we can infer that the pipeline effects will be retained across merging multiple snapshots if the said effects alter the worst-case behavior of the paths. Hence, the proof hold true for merging an arbitrary number of snapshots. □

11. Merging Register Files

To perform a merge on the register file state ("RF" from Figure 5) we use a simple technique on each register:

- If the register value is *unchanged* across the snapshots, then the merged state will retain that value in the register;
- If the register value is *different*, then set the merged value to a Not-A-Number (NaN) [16]. This is to handle values that are input-dependent which will not be known until run-time. This is safe due to the conservative semantics of any operation of NaN that, by definition, results in a conservative value (NaN unless trivial arithmetic rules apply, such as multiplication with zero) and in conservative temporal requirements (worst-case number of cycles for this operation under the given operands).

Performing the above checks/modifications on every register in the register file in both snapshots, we are able to easily construct a new "merged" state for the register file. *Note:* a merge on register files deals with the actual register values. This is different from merging reservation stations for register files (Section 9.3).

The ability to extract and/or write back register file state can be realized by simple modifications of existing microarchitecture features, *i.e.*, the Precise Event-Based Sampling (PEBS) with user-selected access to selected shadow buffers [27] present in the Intel X86 architecture. Our design makes buffers used in this and other architectural techniques uniformly available to the user.

12. Implementation

The *CheckerMode* infrastructure has been implemented on an enhanced *SimpleScalar* processor simulation framework [6]. It has the ability to model a variety of processor configurations (SMT, CMP, *etc.*). We had previously [16] enhanced the simulator by adding the ability to start/stop execution at given arbitrary program counter (PC) values as well as the ability to capture timing information for the given range of PCs. We have now further enhanced it to include the process of capturing the state of the processor during the *issue* stage (*i.e.*, using the concept of reservation stations introduced in this paper). We also have the ability to capture and merge snapshots and to reset the state of the the pipeline to a given snapshot as detailed in this paper.

13. Related Work

Accurate knowledge of worst-case execution times (WCETs) is critical for hard real-time systems. WCETs must be known or safely bounded *a priori* so that schedulability analysis can proceed. Methods to estimate WCETs range from dynamic analysis [5, 31, 33] to static analysis methods [4, 7, 8, 10, 17, 18, 20, 22, 28]. Recently, some hybrid methods [4, 9, 15] as well as hardware-related methods have been proposed [1, 2, 12]. Dynamic methods may produce unsafe results while static and hybrid methods tend to be extremely pessimistic and typically overestimate the WCETs so that the bound is not tight. Many methods are incapable of handling advanced micro-architectural features, such as OOO execution. Our work is able to fill this gap and contributes to high confidence in embedded systems design for time-critical software.

Our work is closest to Lundqvist *et. al.* [12, 13] who use symbolic execution with tightly integrated path and timing analysis to obtain WCET estimates for modern architectures. Their work is similar to ours in that they use the concept of "unknown" values to represent register values and addresses that are input dependent. They also capture processor state at branches and perform "merge" operations on previously captured state. However, their work differs from ours in significant ways. Their analysis is based on performing static timing analysis within an architectural simulator using *in-order* execution. They still require detailed modeling of the pipeline and other functional units within the processor. Their merge mechanism requires intricate knowledge of instruction flow through the pipeline – which instruction occupies/releases what resources, *etc*. Our approach avoids such costly and often impossible modeling of complicated pipeline structures using reservation stations. This technique avoids the detailed modeling of structural and data hazards, but is still able to *accurately account for their worst-case effects in the pipeline*. In fact, the inability to accurately model state-of-the-art processor pipelines is the core issue why the process of timing analysis for such processors is extremely difficult. We avoid larger overheads by introducing minor modifications to existing processor features that assist timing analysis. Another important problem with modeling is that processor vendors might be reluctant to share exact details about internal architectural details required for accurate conventional static WCET analysis. With our method, the vendors themselves can build the "checker mode" into their processors so that analysis can be performed without risk of divulging their intellectual property.

One other serious issue with analyzing dynamically scheduled processors was also pointed out by Lundqvist et. al. – that of *timing anomalies*. Such effects cause anomalous pipeline effects that complicate the task of modeling processors leading to large overestimations in safely bounding WCETs. We have shown in this and related work [16] that problems associated with timing anomalies can be handled in our framework because we do not intend to create models for processor behavior. Instead, we resort to the actual processor execution itself. This, coupled with the fact that our pipeline analysis is able to retain all pipeline effects (including timing anomalies) while retaining the ability to capture accurate worst-case execution times, means that we do not face the same problems with timing anomalies that other techniques have. Our analysis and merge techniques are able to deal with out-of-order pipelines at a finer granularity (instruction level) without the overheads of modeling or significant micro-architectural changes.

14. Future Work

We intend to focus on dynamic branch prediction. While we have been able to capture, merge and restore pipeline state for OOO processors, we intend to find good solutions to do the same for dynamic branch prediction schemes [11, 14]. We will also study how the snapshot capture/merge mechanisms discussed in this paper can be integrated with fixed-point analysis techniques for loops.

15. Conclusion

In this paper, we outlined a sophisticated pipeline analysis scheme that is able to estimate the worst-case behavior of out-of-order pipelines in a *safe* manner. We also show that we are able to correctly deal with timing anomalies. We are able to conduct our analysis in ways that are minimally invasive with respect to the processor. More specifically, we suggest minor changes to existing micro-architectural features that extends contemporary monitoring techniques already present in hardware. This work, when integrated with our CheckerMode infrastructure, utilizes interactions between hardware and software to make contemporary processors predictable and analyzable. Such processors may now be safely used in real-time systems, thus moving the state-of-the-art forward. We believe that this work will enhance the design choices that are available to designers of embedded and real-time systems, particularly on the high-end of computational requirements. To the best of our knowledge, the analysis methods presented in this paper are the first of their kind that deal with out-of-order processing and timing anomalies.

References

[1] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (VISA): Exceeding the complexity limit in safe real-time systems. In *International Symposium on Computer Architecture*, pages 250–261, June 2003.

[2] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Enforcing safety of real-time schedules on contemporary processors using a virtual simple architecture (visa). In *IEEE Real-Time Systems Symposium*, pages 114–125, Dec. 2004.

[3] C. Berg. Plru cache domino effects. In F. Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, number 06902 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. <http://drops.dagstuhl.de/opus/volltexte/2006/672> [date of citation: 2006-01-01].

[4] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2002.

[5] V. Braberman, M. Felder, and M. Marre. Testing timing behavior of real-time software. 1997.

[6] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR-96-1308, University of Wisconsin - Madison, CS Dept., July 1996.

[7] K. Chen, S. Malik, and D. I. August. Retargetable static timing analysis for embedded software. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, October 2001.

[8] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Dept. of Information Technology, Uppsala University, 2002.

[9] A. Hamann, M. Jersak, K. Richter, and R. Ernst. Design space exploration and system optimization with symta/s - symbolic timing analysis for systems. In *IEEE Real-Time Systems Symposium*, pages 469–478, Dec. 2004.

[10] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on advanced processor architectures. In *DATE*, pages 552–559, 2000.

[11] D. Jimenez. Reconsidering complex branch predictors. 2003.

[12] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3):183–208, Nov. 1999.

[13] T. Lunqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University, 2002.

[14] S. McFarling. Combining Branch Predictors. Technical Report TN-36, June 1993.

[15] S. Mohan and J. Helander. Temporal analysis for adapting concurrent applications to embedded systems. In *accepted at ECRTS*, 2008.

[16] S. Mohan and F. Mueller. Hybrid timing analysis of modern processor pipelines via hardware/software interactions. In *IEEE Real-Time Embedded Technology and Applications Symposium*, page (accepted), 2008.

[17] S. Mohan, F. Mueller, W. Hawkins, M. Root, C. Healy, and D. Whalley. Parascale: Expoliting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *IEEE Real-Time Systems Symposium*, pages 233–242, Dec. 2005.

[18] S. Mohan, F. Mueller, D. Whalley, and C. Healy. Timing analysis for sensor network nodes of the atmega processor family. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 405–414, Mar. 2005.

[19] T. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *MICRO-30*, pages 314–320, Dec. 1997.

[20] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[21] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA*, pages 206–218, 1997.

[22] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, Sept. 1989.

[23] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 71–80, Apr. 2006.

[24] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *IEEE Real-Time Systems Symposium*, pages 212–222, Dec. 2006.

[25] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Universitaet des Saarlandes, 2002.

[26] Smith, J. E. A study of branch prediction strategies. In *Proc. 8^{th} International Symposium on Computer Architecture*, pages 135–148, Minneapolis, 1981.

[27] B. Sprunt. Pentium 4 performance monitoring features. 2002.

[28] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*, June 2003.

[29] J. Turley. Embedded processors by the numbers, 1999.

[30] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2003.

[31] J. Wegener. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15:275–298(24), 1998.

[32] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.

[33] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, June 1997.

[34] R. White. *Bounding Worst-Case Data Cache Performance*. PhD thesis, Dept. of Computer Science, Florida State University, Apr. 1997.