Cache-Aware Timing Analysis of Streaming Applications

Samarjit Chakraborty[1] Tulika Mitra[1] Abhik Roychoudhury[1] Lothar Thiele[2]
[1]National University of Singapore
[2]Eidgenössische Technische Hochschule Zürich
{samarjit, tulika, abhik}@comp.nus.edu.sg, thiele@tik.ee.ethz.ch

Abstract

Of late, there has been a considerable interest in models, algorithms and methodologies specifically targeted towards designing hardware and software for streaming applications. Such applications process potentially infinite streams of audio/video data or network packets and are found in a wide range of devices, starting from mobile phones to set-top boxes. Given a streaming application and an architecture, the timing analysis problem is to determine the timing properties of the processed data stream, given the timing properties of the input stream. This problem arises while determining many common performance metrics related to streaming applications and the mapping of such applications onto hardware architectures. Such metrics include the maximum delay experienced by any data item of the stream and the maximum backlog or the buffer requirement to store the incoming stream. Most of the previous work related to estimating or optimizing these metrics take a high-level view of the architecture and neglect micro-architectural features such as caches. In this paper, we show that an accurate estimation of these metrics, however, heavily relies on an appropriate modeling of the processor micro-architecture. Towards this, we present a novel framework for cache-aware timing analysis of stream processing applications. Our framework accurately models the evolution of the instruction cache of the underlying processor as a stream is processed, and the fact that the execution time involved in processing any data item depends on all the previous data items occurring in the stream. The main contribution of our method lies in its ability to seamlessly integrate program analysis techniques for micro-architectural modeling with known analytical methods for analyzing streaming applications, which treat the arrival/service of event streams as mathematical functions. This combination is powerful as it allows to model the code/cache-behavior of the streaming application, as well as the manner in which it is triggered by event arrivals. We employ our analysis method to an MPEG-2 encoder application and our experiments indicate that detailed modeling of the cache behavior is efficient, scalable and leads to more accurate timing/buffer size estimates.

Keywords: Timing analysis, instruction cache, streaming applications.

1 Introduction

Stream processing applications are today widespread in several domains ranging from networked hand-held devices playing streaming audio and video, to mobile phone base stations and network routers implementing complex packet processing functionality at high line speeds. However, it is now increasingly being realized that conventional models, languages and design methodologies developed in the embedded systems domain are often not well-suited for implementing and analyzing these applications since they do not adequately exploit the notion of a "stream". To address this shortcoming, recently there has been a number of developments in the form of new "stream-centric" programming languages [40], compiler support [10], processor architectures [13, 34] and design methodologies [20, 23, 27, 43].

1.1 Problem Description

In this paper, we follow this line of development and address the following problem. We are given a block of code corresponding to an application that processes a stream of data items or events (the arrival of a data item may constitute an event and henceforth we will only refer to a stream of events). The events belonging to the stream are *typed* and the processing of events of different types requires the execution of different, but partially overlapping parts of the code. Given the arrival process (or timing properties) of an incoming stream, the problem is to determine the timing properties of the processed stream. Since the execution times of events depend on their *types*, incoming events might often have to wait as the processor might be busy processing earlier events. Hence, the timing properties of the processed stream might be very different from those of the incoming stream.

Such *timing properties* might be specified in the form of a period in case the incoming stream is periodic, or it might consist of a period and a jitter, or a minimum seperation time between the arrival of two consecutive events. A detailed discussion on other possible specifications of timing properties of event streams and their relationship with each other may be found in [30, 33]. In this paper we will resort to a very general specification of the timing properties of event streams, using the concept of *arrival curves*. We shall formally define arrival curves later in this paper; here it might suffice to say that most of the commonly studied models of event streams from the real-time systems literature (e.g. periodic, periodic with jitter, sporadic, etc.) can be shown to be special cases of arrival curves.

The timing analysis problem lies at the core of determining many common performance metrics related to streaming applications, the mapping of such applications onto hardware architectures, their code layout in the memory, etc. It is easy to see that by relating the timing properties of the processed stream with that of the incoming stream, it is possible to accurately estimate performance metrics such as the minimum buffer size required to store the incoming stream (which is equal to the maximum possible backlog of unprocessed events) and the maximum delay experienced by any event from its arrival till the time it is completely processed.

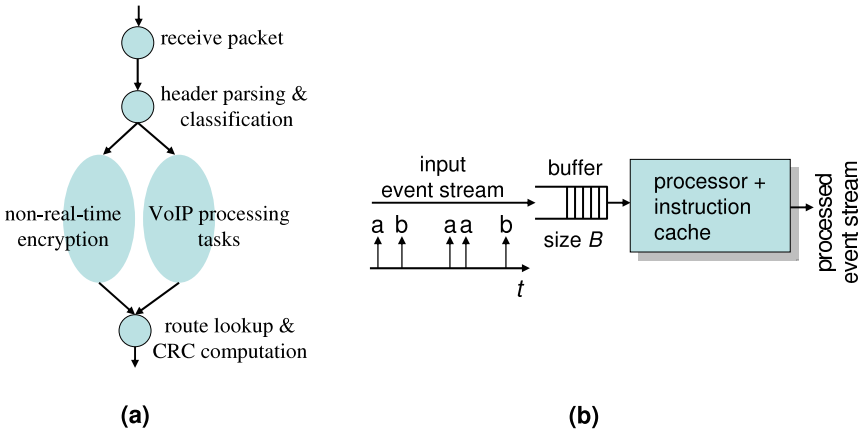


Figure 1: (a) Task graph corresponding to a simple software-based router which processes packets of two different types. (b) A processor with an instruction cache, on which the code (for example, that shown in (a)) processing an event stream is executed. Incoming events are stored in a buffer of size B, which is read by the processor.

Assuming that the worst case execution time (WCET) associated with each event type is given and is a constant, this timing analysis problem was addressed in [3], where the timing properties were specified using arrival curves. However, this problem becomes significantly more complicated if we take into account the fact that the WCET involved in processing events of even the *same type* might vary. This is because the sequence of previously processed events determine the *state* of the processor's micro-architecture (such as its cache), which affects the execution time involved in processing any subsequent event. Hence, the WCET associated with any event might vary considerably, depending not only on its type but also on the exact sequence of prior events. The main contribution of this paper over [3] is in modeling the effects of the instruction cache in solving the timing analysis problem. We show that this leads to considerably tighter bounds on the worst-case delay and buffer requirements. Our experimental results based on an MPEG-2 encoder application show that ignoring the effects of the instrucion cache leads to worst-case delay estimates that are more than 60% higher than those obtained by modeling the cache. The corresponding estimate on the buffer size requirement is more than 62% higher. For applications with higher cache reuse across different event types, or for different cache sizes and code layouts, the differences in these estimates can be even more significant. This clearly establishes the need for micro-architecture modelling in the timing analysis of streaming applications, even when such analysis is carried out at the system-level.

1.2 An Illustrative Example

To see where the above-mentioned performance estimation problem may be used, consider a streaming application processing a stream of events where each event is either real-time (*i.e.* with hard deadlines on processing time) or non real-time (with no constraints on processing time). One example of such a streaming application is a software based router

processing Voice-over-IP (VoIP) packets (with deadlines on processing time) and other packets. Moreover, some of the code processing the two types of packets may be common. The task graph for such a router appears in Figure 1(a) and the system model is shown in Figure 1(b). The arrival of any packet causes an interrupt, which is processed by the *receive packet* task. This is followed by packet header parsing and classification, after which the packet type is known and based on this type the code corresponding to either the right or the left hand side of the task graph is executed. The incoming stream of events get stored in a FIFO buffer of size B, which is read by the processor running the stream processing application, such as the software-based router shown in Figure 1(a).

We are given the arrival process or the timing properties of possible incoming streams to be processed. Since we are interested in the timing properties of a *class* of arrival patterns, rather than one concrete instance of a stream (or a trace), the specification provides *bounds* on the on the arrival process. For example, in the case of our router, the maximum *rate* at which packets arrive would typically be bounded. Such a bound might be specified in the form of the maximum possible bursts allowed over different time interval lengths, and a long-term arrival rate of the packets. In the communication networks domain, such a specification is based on the theory of *network calculus* [7, 14] and is called an *arrival curve*. We use these arrival curves to describe the input timing properties of our streaming application.

Given a setup such as the one described above, in this paper we solve the timing analysis problem by obtaining tight bounds on the WCET incurred in processing any event by the streaming application. Towards this, we take into account all possible sequences of events that might be processed prior to processing of any event in question, and how such possible sequences might modify the state of the instruction cache. As part of the problem specification, we are given a mapping of the different memory blocks corresponding to the code processing the events, onto a cache. We are also given a specification of the possible sequences of events that might arrive, for example, in the form of a finite state transition system. In other words, such a transition system describes the possible compositions of an event stream in terms of the different event types. Such a specification can be used to rule out certain sequences of arrivals, for example, that there can not be more than 10 consecutive VoIP packets. This enables us to rule out certain "worst-case" cache states and thereby bound the WCET in processing packets of a particular type.

Broadly speaking, our method proceeds as follows. First, we analyze the cache behavior of the code processing each event type. This is done by a least fixed-point computation on the control flow graph of the code processing each event type. The fixed point computation is required due to the (potential) presence of loops in the control flow graph. It yields the possible cache contents at each location of the control flow graph in an abstract fashion. We then use the results of our cache analysis to find the "minimum common code usage" across events. To explain this issue, consider the task graph in Figure 1(a). The code processing *receive packet* and *header parsing/classification* tasks are common to both event types. If an event is processed starting with an empty cache, there will be cold misses[1] in

[1]Misses due to first access of a memory block.

the instruction cache. However, if the code for any of these two tasks are not displaced by the subsequent tasks (*encryption, VoIP processing, route lookup/CRC*) then we can avoid some cache misses starting from the second event in an event stream.

The preceding setup presents a simple (even trivial) example of how cache analysis can be used to accurately estimate processing times of events in an event stream. In our method, we consider a transition system which captures all possible event arrival sequences; thus the edges of the transition system are annotated with event types. We then compose this transition system with the results of cache analysis of event types (which captures the possible cache states after each event). This composition yields a larger transition system which captures event arrival sequences as well as inter-dependencies across events due to the shared cache. A key issue here is to develop safe and efficient heuristics to limit the blow-up in size of the composed transition system. Finally, we analyze the composed transition system to find out maximum processing time of any allowed sequence of k consecutive events for any positive integer k. This function is composed with the arrival curve which captures the maximum number of event arrivals over a certain time interval. Thus by relating the worst-case processing time of k events with the time interval over which at most k events can arrive, we obtain the worst-case delay of any event, i.e., the maximum delay experienced by an event from its arrival to end of processing. The worst-case buffer requirement (the minimum buffer size to guarantee absence of overflow of the buffer containing waiting events) is also obtained in a similar fashion.

In summary, we present an accurate performance analysis method for typed event streams. Our analysis takes into account the possible event arrival sequences as well as the platform on which the code processing the events is executed. In particular, we show how the cache usage of streaming application can be taken into account to yield safe and precise delay/buffer estimates. Experimental results from the MPEG-2 encoder application validate our method by showing that it is important to consider the cache behavior of streaming applications for accurate performance analysis.

It may be noted that the main goal of this paper is to lay the foundations for integrating processor micro-architecture modeling into timing analysis of streaming applications. Towards this, we have only considered the effects of modeling the instruction cache, and show that this alone is a non-trivial problem. It would certainly be meaningful to extend this work to integrate the effects of other micro-architectural features such as data caches, pipelining, and speculative execution. In Section 8, we have outlined possible extensions of our current modeling and analysis techniques to integrate these features as well.

Organization: The rest of this paper is organized as follows. The next section summarizes related literature. In Section 3 we present an overview of our performance estimation framework along with an example. We then proceed to discuss the different components of our core analytical estimation framework: cache modeling (Section 4) and analysis of event streams using arrival curves (Section 5). Improvements to the core analytical framework appear in Section 6 and experimental results are discussed in Section 7. The paper concludes in Section 8 with discussion on possible future extensions of our technique.

2 Related Work

In this section we review literature on performance analysis of event streams, and cache behavior analysis in real-time systems.

2.1 Previous Work on Timing Analysis of Event streams

Our work builds upon the system-level timing analysis methods for event streams studied in [3, 4, 31, 32]. Although, these papers also deal with end-to-end delays experienced by event streams and the computation of maximum buffer fill levels, their focus is on multiprocessor architectures and the modeling of resource sharing. They do not consider an extended task model and do not focus on the distinction between different event types, as we do in this paper. More importantly, none of these papers consider the role of the processor micro-architecture on the execution time of the different events. Note that in many cases on-chip buffer/memory is available only at a premium because of its stringent area requirements. In such cases (e.g. in portable multimedia players) an accurate estimation of delay and buffer requirements is essential, and therefore calls for an appropriate modeling of the processor micro-architecture.

Some of the recent works in system-level performance analysis have analyzed typed event streams. The focus here is to avoid taking the WCET of each event (since it involves too much pessimism), and instead consider the possible event arrival sequences. In this respect, these works have a goal similar to our paper. However, the event stream specifications they consider are often coarser leading to less accurate analysis. In particular, Wandeler et. al. [44] consider type-rate curves; a type-rate curve is a function which given a constant k, returns the minimum (or maximum) number of occurrences of an event type among k consecutive events. In comparison, our typed event streams are described using an event transition system and an arrival curve. The event transition system captures the possible event arrival sequences more accurately than the type-rate curves and the arrival curve links up the event arrival sequences to the arrival times.

The work of [45] is closer to ours. This analysis method takes in an event automaton and an arrival curve as in our method. However, the processing unit is also modeled as an automaton which makes the analysis difficult to adopt in practice. Even though in principle we could consider the effect of micro-architectures by modeling the processing unit as a huge automation, such a modeling places an immense burden on the system designer. Instead, in this paper we demonstrate how state-of-the-art micro-architectural analysis methods (such as cache analysis) can be tightly integrated with event stream analysis without the designer having to explicitly model the effects of cache as an automaton.

2.2 Relationship to Previous Work on WCET Analysis

The work reported in this paper is related to the problem of statically analyzing the Worst-case Execution Time (WCET) of a program, which is an important problem in the domain of real-time and embedded software design. Note that WCET analysis techniques (see [24, 46]

for detailed surveys of this area) are conservative, that is, they compute an upper bound on the program's actual worst case execution time. Usually this involves a path analysis to find out infeasible paths in the program's control flow graph, and micro-architectural modeling. Both path analysis and micro-architectural modeling have been studied extensively [9, 17, 19, 26, 29, 36, 39] because of the inherent importance of deriving WCET estimates for schedulability analysis. However, we are not aware of any work specifically on the WCET analysis of streaming applications. In fact, most of the previous work on WCET analysis consider the uninterrupted execution of a program, which is similar to processing a *single* event in our setup. In this paper, we compute the WCET of a *stream* of events, where, to estimate the processing time of any event we consider the micro-architectural state resulting from the processing of previous events.

For multiple tasks executing on a processor (thereby sharing a cache), additional complications are introduced in the performance analysis. In particular, we need to measure the inter-task cache effects — the indirect effect of one task on another due to the shared cache.

2.3 Previous Work on Cache Management in Real-Time Systems

Several strategies have been proposed in the real-time systems community to ensure predictable execution of real-time tasks in the presence of caches. These works can be categorized as follows.

- Novel architectures to support predictable execution
- Compiler controlled caches and strategies for managing them
- Analysis techniques to bound inter-task cache effects

In the first category, the work of Lee et. al. [16] proposes a novel prefetching based memory architecture as an alternative to instruction caches. The prefetch scheme proceeds in two modes — real-time (where the prefetching tries to reduce WCET) and non real-time (where the prefetching tries to reduce average case execution time). However, adopting these solutions involve non-trivial changes to the processor running the real-time tasks.

In the second category, there exists a body of research work in the area of cache partitioning and cache locking. These works either try to partition the cache between the tasks (thereby eliminating any cache inteference) [35] or pre-load the cache with memory blocks thereafter locking its contents [28, 42]. The cache contents may be locked for the entire lifetime of execution or changed at specific execution points (for example pre-emption points). This line of work focuses on efficient algorithms to decide the memory blocks to be locked into the cache. Another proposed approach is to change the memory layout of the tasks so as to reduce inter-task cache effects [8].

The third category of work involves no changes to the compiler/architecture and is closest in flavor to our method. These efforts develop analysis methods to determine maximum *preemption costs* when multiple tasks share a processor. This is often referred to as the

cache related preemption delay (CRPD) [15, 26, 37, 41]. Some of these works have considered the effect of multiple pre-emptions [37] and integrated the CRPD analysis to develop an accurate polynomial scheduling method [38]. The aim here is to estimate the maximum number of additional cache misses incurred by a low-priority task when it resumes execution after getting preempted. Clearly, this requires an analysis of possible cache states in the low-priority task before and after its preemption. In this paper, our setup does *not* involve preemption of events. However, conceptually our analysis bears interesting similarities to CRPD analysis. In particular, to estimate the execution time and buffer fill levels of a stream of events (which is captured by our finite state transition system specification), we statically analyze the cache states before and after each event in the stream. This can be exploited to give tighter WCET bounds as compared to the situation where we estimate the WCET of any event type in isolation without considering the initial micro-architectural state(s).

We should emphasize at this point that our technique is a static analysis technique and is not a trace based simulation (*i.e.*, we always capture the various possible cache states as a stream of events is processed). Essentially we capture the minimum commonality in memory block usage between the processing code of different events. This commonality may be exploited across the processing of two events of the same event type or even across the processing of events of different event type (because some of the processing tasks may be common across event types). Thus, in a broad sense, what our performance analysis tries to capture (reduction in processing time of an event due to certain memory blocks being present in the cache from past events' execution) is the opposite of CRPD analysis (which tries to capture the increase of execution time of an event due to certain memory blocks being replaced from the cache by pre-empting events). Moreover, we have tightly integrated our cache modeling (which determines "minimum common cache usage" across events) with powerful state-of-the-art event stream analysis methods [3] to yield an accurate performance analysis technique for streaming applications.

3 Overview

In this section, we present a formal problem specification followed by an overview of the main steps involved in our cache-aware timing analysis method.

3.1 Problem Specification

Our problem specification consists of the following components.

1. A task graph, such as the one shown in Figure 1(a), which models the streaming application and the corresponding program or code. This application executes on a single processor.

2. The specification of the event stream to be processed by the application is composed of two parts. All events are *typed*. We denote the alphabet of event types using Σ.

- The first part of the stream specification is a transition system $\mathcal{T}$ which captures all possible *sequences of event types* that might occur in the stream. $\mathcal{T}$ can either be determined by analyzing the device or the system which generates the stream, or by analyzing a sufficiently large number of representative input streams.
- The second part of the stream specification is concerned with its timing properties. Towards this, we are given a function $\bar{\alpha} : \mathbb{R}^{\geq 0} \mapsto \mathbb{Z}^{\geq 0}$, which bounds the maximum number of events that can arrive within any time interval of a given length. We will refer to $\bar{\alpha}$ as an *arrival curve*. It may also be noted here that this specification is more general than the event models traditionally studied in the real-time systems literature, such as periodic, periodic with jitter or the sporadic event model [1, 2, 25] (see [3, 4] for further details).

3. For each event type in Σ, we are given the execution path through the task graph of the application. We are also given the worst case execution time for each event type assuming an empty cache state before the processing of the event starts.

4. Code layout of the application, i.e., the mapping of the different memory blocks of the application onto the cache blocks.

5. Cache configuration and cache miss penalty.

Example: Let us once again consider the task graph shown in Figure 1(a), implementing a simple software-based router, as our streaming application. It processes two different types of packets: VoIP packets, which have real-time constraints on their processing time, and packets which need to be encrypted without any constraints on their processing time. The processing of any VoIP packet follows the right hand side path in this task graph and the processing of all other packets follows the left hand side path in this graph. Figure 2 shows this task graph in an abstract fashion. The two paths in the task graph of Figure 2 correspond to the processing of non-real-time and real-time event types: let us call them a and b respectively. Thus, we have the following inputs to our timing analysis problem.

- The event type alphabet $\Sigma = \{a, b\}$.
- The transition system $\mathcal{T}$ capturing the possible arrival sequences of events appears in Figure 2. The alphabet of this transition system is Σ. Note that it does not capture all infinite strings over $\Sigma = \{a, b\}$. In fact, it captures the constraint that between two bursts of events of type b, at least two events of type a must arrive. The value T on each edge of the transition system shows the WCET for processing that event assuming an empty cache before the processing of the event starts.
- We need to specify the *arrival curve* $\bar{\alpha}$ for the event stream. In this simple example, we use the arrival curve $\bar{\alpha}$ to specify a bursty arrival process, with up to four events arriving within any time interval of length less than 150 time units. In other words, if

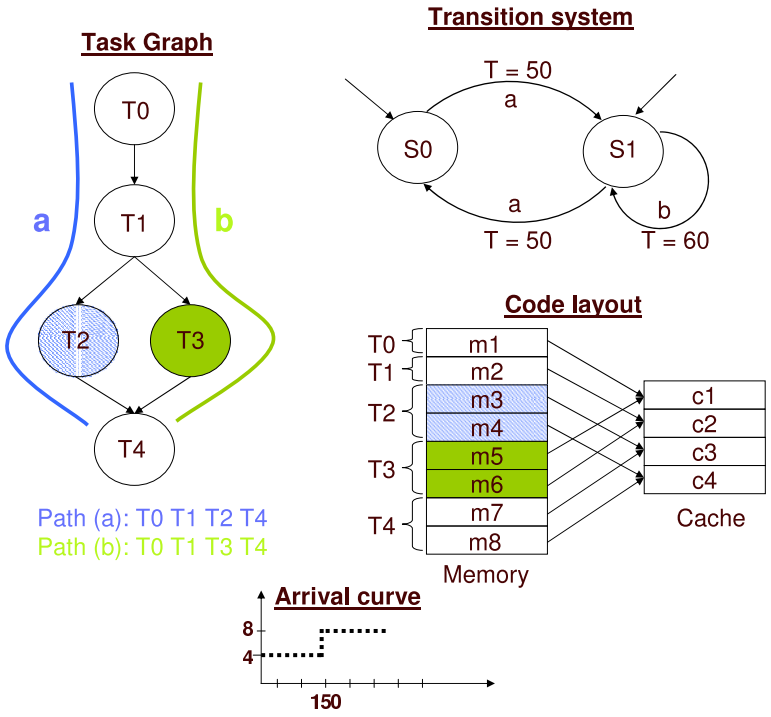


Figure 2: Input specification corresponding to our running example.

we consider any concrete (timed) trace of an arrival process, and slide a "window" of length less than 150 time units along this trace, then for any position of this window at most four events will be recorded inside the window. Similarly, if the window is of length 150, then for any position of this window at most eight events will be recorded inside the window.

- The execution path through the task graph corresponding to each event type is shown in Figure 2. The path for event a consists of tasks $\langle T0, T1, T2, T4\rangle$ and that of event b consists of tasks $\langle T0, T1, T3, T4\rangle$.

- Figure 2 also shows code layout for the task graph. The tasks are laid out in memory in the sequence $\langle T0, T1, T2, T3, T4\rangle$. Task T0 uses the memory block m1, task T1 uses the memory block m2 and so on. The code layout implicitly specifies the mapping of the memory blocks to the cache blocks. For example, given a direct-mapped cache consisting of 4 cache blocks as shown in the figure, it easy to see how the different memory blocks are mapped to the cache blocks.

Given the above, we would like to compute the maximum number of backlogged events (i.e., the maximum buffer size required) and the maximum delay experienced by any event. As discussed earlier, the main difficulty in this problem arises from the fact that the WCET

of any event depends on the state of the cache, and hence on all the events that arrive prior to this event. Of course, to estimate the cache behavior, we also need the code for the individual tasks $T0, T1, T2, T3, T4$. Instead of giving details of the code for each task, we just give their usage of memory blocks. Only this information is relevant for determining their cache states. Note that the memory block usage given below is consistent with Figure 2. The interesting case is for task $T4$, where the usage is memory block 7 or memory block 8; this can happen due to a conditional branch.

Task	Memory Block Usage
T0	Mem. Block 1
T1	Mem. Block 2
T2	Mem. Block 3 followed by Mem. Block 4
T3	Mem. Block 5 followed by Mem. Block 6
T4	Mem. Block 7 or Mem. Block 8

The cache configuration assumed is a direct mapped cache with four cache lines. Therefore, memory blocks 1 and 5 map to cache line 1, memory blocks 2 and 6 map to cache line 2, and so on. Using the memory block usage of the individual tasks we can find out the following:

- the possible cache states at the end of processing of event a or event b (in our cache modeling presented later, we will refer to these as Reaching Cache States of an event or RCS), and
- the possible first references to cache blocks during the execution of event a or event b (in our cache modeling presented later, we will refer to these as Live Cache States of an event or LCS).

Using the above summaries of cache behavior for each event, we link the evolution of cache states due to an event's processing with the arrival of events (which is specified by the arrival curve $\bar{\alpha}$).

Now, recall that our arrival curve allows for up to two bursts (of four events each) within a time interval of 150 time units. For estimating the maximum buffer fill level, it is important to check whether the processing of events arriving in a burst is completed by the time a second burst of events arrive. In this example, we show that such a check can be intricately linked to the effects of cache. With the given input specification, we used our timing analysis framework to observe that the delay/buffer size estimates are quite sensitive to the code layout, the cache miss penalty value and the cache configuration. For example, given the current layout and assuming a cache miss penalty of 5 time units, we get the following delay and buffer size estimates:

$$WCD = 226, \quad WCB = 6$$

In this case,there exist certain arrival patterns (such as $a(b)^{\omega}$) where all the events in the first burst can not be processed by the time the second burst arrives.

If we now increase the cache miss penalty to 15 time units, with everything else in the input specification remaining unchanged, the estimates get substantially altered:

$$WCD = 145, \quad WCB = 4$$

In this case, all the events in the first burst are guaranteed to be processed by the time the second burst arrives. Hence, the maximum buffer level is equal to the number of events in a single burst (which is four in this case). At first sight, the reduction in the WCD/WCB estimates due to an increase of the cache miss penalty may appear to be counter-intuitive. However, this means that there is substantial re-use of memory blocks across events in the event streams (allowed by the event transition system $\mathcal{T}$ in our example). Hence, the actual time required to execute individual events in an event stream is much lower than the WCET estimates of each event assuming an initial empty cache.

Furthermore, changes in code layout and cache configuration can also make non-trivial changes in estimates. By simply changing the code layout to $\langle T0, T1, T2, T4, T3\rangle$ (from $\langle T0, T1, T2, T3, T4\rangle$), we observed that the WCB estimates are less sensitive to changes in cache miss penalty in this case. These results confirm our main observation that micro-architectural modeling is important for accurate timing analysis of stream processing applications. It might be possible to reason about the effects we discussed above in the case of this simple example, without resorting to our proposed analytical framework. However, for realistic code and cache sizes our framework will help to uncover counter-intuitive effects of the code layout and the processor architecture.

3.2 Analysis Technique

Our cache-aware timing analysis technique consists of three main steps as shown in Figure 3. The first step is the cache modeling to find out the cache state at the end of a given event sequence. Given this information, we modify the transition system to capture the different execution time of the same event type corresponding to different cache states (arising due to the different event sequences that can lead to the same state in the transition system). Finally, we compute the timing properties of the processed event stream by taking into account the arrival curve and the modified transition systems. The next two sections describe these steps in more details.

4 Cache Modeling

In this section we show how to compute the WCET involved in processing a *single* event of any specified type, and the state of the cache after the processing of this event. In Section 5, we will then make use of this result to solve our timing analysis problem and compute the maximum backlog and delay experienced by a stream of events. The basic technique presented in Section 4.1 bears some similarity with the method used for computing cache related preemption delays in [26]. However, as already pointed out in Section 1, in this paper we do not consider task preemptions—we deal with a single stream, and events

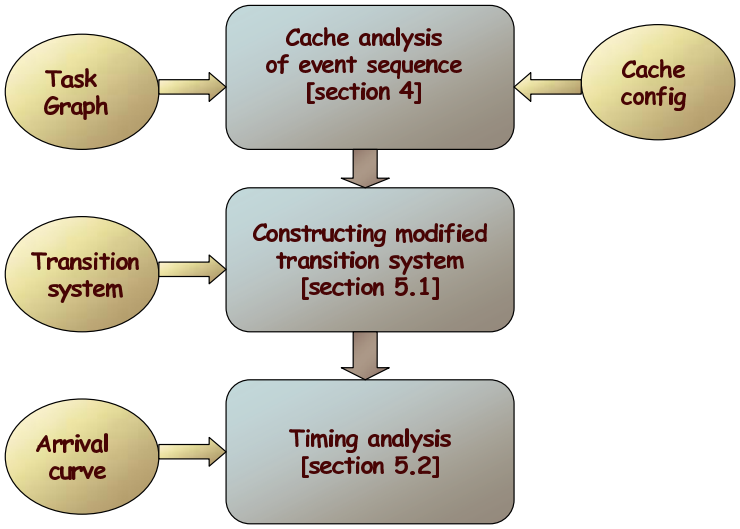


Figure 3: Overview of cache-aware timing analysis for streaming applications.

from this stream are processed to completion in a first-come-first-serve manner. This is elaborated in Sections 4.2 and 5.

4.1 Cache States

To describe our cache modeling, we first present the notion of *cache states*. By "cache state", we refer to the contents of all the cache blocks. For simplicity of exposition, in this paper we only consider direct-mapped caches. However, the techniques we present below can easily be generalized to set-associative caches; we discuss this issue later in this section. Let M denote the set of all memory blocks. For a direct mapped cache with n blocks, a *cache state* is a vector c of n elements $c[0], \ldots, c[n-1]$ where $c[i] = m$ if the cache block i holds the memory block m. If the ith cache block does not hold any memory block, we denote this as $c[i] = \perp$. Hence, a cache state is a vector of length n, where each element of the vector belongs to $M \cup \{\perp\}$. We assume that any operation $\odot$ over $M \cup \{\perp\}$ can be applied to the cache states, by applying this operation pointwise to its elements. For example, if $\odot$ is a binary operation over $M \cup \{\perp\}$ and c, c' are cache states then $c \odot c' = c''$ denotes $c''[i] = c[i] \odot c'[i]$ for all $0 \leq i < n$. To compute the WCET that might be incurred in processing an event, and the state of the cache after this event is processed, we will rely on the following two functions.

Definition 4.1 (Reaching Cache States) *The reaching cache states of an event σ, denoted as $RCS(\sigma)$, is the set of possible cache states when the end of the last basic block corresponding to any of the execution paths associated with the processing of σ is reached. We suppose that the cache is initially empty.*

Definition 4.2 (Live Cache States) *The live cache states of an event σ, denoted as $LCS(\sigma)$, is the possible first memory references to cache blocks via any execution path associated with the processing of σ.*

$RCS(\sigma)$ and $LCS(\sigma)$ can then be computed as follows. Let $\tau(\sigma)$ be the task graph associated with the processing of an event type σ, i.e. $\tau(\sigma)$ contains only the basic blocks and the control-flow relevant to σ. Therefore, the basic blocks in $\tau(\sigma)$ are a subset of all the basic blocks in our stream processing application which processes *all* event types. Further, some of the basic blocks in $\tau(\sigma)$ might also be in $\tau(\sigma')$, which is the task graph for another event type σ'. Examples of such common basic blocks are those in the nodes *receive packet*, *header parsing & classification* and *route lookup & CRC computation* in the task graph in Figure 1(a). Note that each of the nodes in this task graph might contain multiple basic blocks, conditional branches and also loops.

Now, let $\mathcal{B}_\sigma$ be the set of basic blocks appearing in $\tau(\sigma)$. Without loss of generality, we can assume that there is a unique start and end basic block in the control flow represented by $\tau(\sigma)$. With each $B \in \mathcal{B}_\sigma$ we associate the variables RCS_B^{IN} and RCS_B^{OUT} and initialize them as follows:

$$RCS_B^{IN} = \emptyset \quad \text{and} \quad RCS_B^{OUT} = gen_B$$

For any basic block B, $gen_B = [m_0, \ldots, m_{n-1}]$, where $m_i = m$ if m is the *last* memory block in B that maps to cache block i and $\perp$ if no memory block in B maps to cache block i. Hence, gen_B records all the memory blocks introduced into the cache due to the execution of B. The change in the cache state due to B is then captured by the equations:

$$RCS_B^{IN} = \bigcup_{p \in pred(B)} RCS_p^{OUT} \tag{1}$$

$$RCS_B^{OUT} = \{r \oplus gen_B \mid r \in RCS_B^{IN}\} \tag{2}$$

Here, $pred(B)$ is the set of all basic blocks in $\tau(\sigma)$ which are predecessors of B, and the operation $\oplus$ is defined over memory blocks as:

$$m \oplus m' \stackrel{def}{=} \begin{cases} m' & \text{if } m' \neq \perp \\ m & \text{otherwise} \end{cases}$$

If the control flow represented by $\tau(\sigma)$ is acyclic, then we can obtain $RCS(\sigma)$ directly from the above equations. In particular $RCS(\sigma) = RCS_{end(\sigma)}^{OUT}$ where $end(\sigma) \in \mathcal{B}_\sigma$ is the unique end basic block in the graph $\tau(\sigma)$. However, if the control flow in $\tau(\sigma)$ contains loops, the set of equations derived from Eqs. 1 and 2 will be recursive. Hence, we compute the values of RCS_B^{IN} and RCS_B^{OUT} as a least fixed point for each basic block B. After the fixed point is reached, we set $RCS(\sigma) = RCS_{end(\sigma)}^{OUT}$.

The computation of $LCS(\sigma)$ is similar to computing $RCS(\sigma)$. As before, we associate variables LCS_B^{IN} and LCS_B^{OUT} with each basic block B in $\tau(\sigma)$ and initialize them as follows:

$$LCS_B^{OUT} = \emptyset \quad \text{and} \quad LCS_B^{IN} = gen_B$$

Here, $gen_B = [m_0, \ldots, m_{n-1}]$ where $m_i = m$ if m is the *first* (instead of the last) memory block in B that maps to cache block i and $\perp$ if no memory block in B maps to cache block i. The rest of the procedure is the same as in the case of computing $RCS(\sigma)$, expect for the fact that Eqs. (1) and (2) are replaced by the equations:

$$LCS_B^{OUT} = \bigcup_{s \in succ(B)} LCS_s^{IN} \quad \text{and} \quad LCS_B^{IN} = \{l \oplus gen_B \mid l \in LCS_B^{OUT}\}$$

where $succ(B)$ is the set of all basic blocks in $\tau(\sigma)$ which are successors of B. Again, we solve these recursive equations by performing a least fixed-point computation. After a fixed point is reached, $LCS(\sigma)$ is set to $LCS_{start(\sigma)}^{IN}$ where $start(\sigma) \in \mathcal{B}_\sigma$ is the unique start basic block in the graph $\tau(\sigma)$.

$RCS(\sigma)$ is therefore the set of possible cache states after the processing of any event of type σ, and $LCS(\sigma)$ captures the possible usages of a cache state at the start of the processing of an event of type σ.

4.2 Computing the WCET of a Single Event

We first define two operations on cache states, namely merge and equality. The *merge* of two sets of cache states X and Y is defined as $X \oplus Y = \{x \oplus y \mid x \in X \wedge y \in Y\}$ where $x \oplus y$ is calculated over cache states by applying the operation $\oplus$ (defined earlier over memory blocks) to the individual elements of the cache states. Thus we have

$$x[i] \oplus y[i] \stackrel{def}{=} \begin{cases} y[i] & \text{if } y[i] \neq \perp \\ x[i] & \text{otherwise} \end{cases}$$

The *equality* of two sets of cache states X and Y is defined as $X \odot Y = \{x \odot y \mid x \in X \wedge y \in Y\}$ where,

$$x[i] \odot y[i] = \begin{cases} 1 & \text{if } x[i] = y[i] \\ 0 & \text{otherwise} \end{cases}$$

For a cache with n blocks, $X \odot Y$ is a set of boolean vectors of length n. Observe that for any two cache states $x \in RCS(\sigma)$ and $y \in LCS(\sigma')$, $x \odot y$ records the "useful" cache blocks for some execution path in the processing of σ', due to the prior processing of σ. Using this observation, we will now show how to obtain a more accurate estimate on the WCET involved in processing an event, compared to the case where the initial cache state before processing the event is considered to be empty.

Let us consider the processing of a sequence of consecutive events $\langle \sigma_1, \cdots, \sigma_N \rangle$. For simplicity, we use σ_i to refer to both an event and its type, and the actual meaning should be clear from the context. In the absence of cache state modeling, i.e. with a completely empty cache, let us assume that the WCET of σ_N is given by the function $WCET(\sigma_N, \perp)$, where $\perp$ denotes the set of cache states that contains only an empty cache. However, if the task graphs $\tau(\sigma_1), \ldots, \tau(\sigma_N)$ share some common memory blocks and the effects of the cache is taken into account, then our estimate of the WCET of σ_N can possibly be improved. More specifically, the WCET of σ_N will be reduced if during the processing of the events

$\langle \sigma_1, \cdots, \sigma_{N-1} \rangle$ some memory blocks are left in the cache, which are then referenced by $\tau(\sigma_N)$. We next show how to compute this reduced WCET using the merge and equality operators defined above.

If we start with a set of cache states cs, and process a sequence of events $\langle \sigma_1, \cdots, \sigma_{N-1} \rangle$, then the set of possible cache states after this sequence of events is processed can be given by the function

$$NSTATE(\langle \sigma_1, \cdots, \sigma_{N-1} \rangle, cs) = cs \oplus RCS(\sigma_1) \oplus \ldots \oplus RCS(\sigma_{N-1})$$

Note that the operator $\oplus$ is associative. Now, if the event σ_N is to be processed, then the set of possible *useful* cache blocks for σ_N is given by the function

$$useful(\sigma_N, NSTATE(\langle \sigma_1, \cdots, \sigma_{N-1} \rangle, cs)) = NSTATE(\langle \sigma_1, \cdots, \sigma_{N-1} \rangle, cs) \odot LCS(\sigma_N)$$

In other words, the function *useful* returns a set of boolean vectors. In any vector belonging to this set, a "1" in the position of any cache block indicates that the contents of this cache block might be used while processing σ_N, while a "0" indicates otherwise. Therefore, based on this set of boolean vectors, our revised estimation of the WCET of σ_N is given by $WCET(\sigma_N, cs')$, where the set of cache states is $cs' = NSTATE(\langle \sigma_1, \cdots, \sigma_{N-1} \rangle, cs)$. Clearly,

$$WCET(\sigma_N, cs') = WCET(\sigma_N, \perp) - penalty \cdot \min \{|v| \, : \, v \in useful(\sigma_N, cs')\} \tag{3}$$

where *penalty* is the cache miss penalty associated with any memory block and $|v|$ denotes the number of 1s in the boolean vector v, i.e. $|v| = \sum_{i=0}^{n-1} v[i]$. Note that $WCET(\sigma_N, cs') < WCET(\sigma_N, \perp)$ since $WCET(\sigma_N, \perp)$ denotes the execution time of σ_N assuming *all* first accesses to memory blocks result in cache misses. However, in defining $WCET(\sigma_N, cs')$ we take into account that at least min $\{|v| \, : \, v \in useful(\sigma_N, cs')$ of the first accesses result in cache hit.

Example: We now illustrate this cache re-use through a simple example. Consider two event types a and b. Given the LCS and RCS of a, b — we show how cache re-use can be drastically altered for different sequences of events. The LCS and RCS for event types a and b are given in Figure 4. The figure also shows the number of memory blocks that can be re-used for each possible 2-event sequences. For example, after execution of event a, the cache state consists of memory blocks m1, m2, m7 and m8. If this is followed by another event a, then the second event can reuse memory blocks m1 and m2 without incurring cache miss for these memory blocks. Surprisingly, the sequence $\langle a, b \rangle$ results in maximum reuse for this example, whereas the sequence $\langle b, a \rangle$ results in zero re-use.

5 Timing Analysis of Event Streams

In this section we will make use of our cache-based estimation of the WCET of a single event to solve our timing analysis problem. More specifically, we use this estimate to accurately compute the maximum delay and backlog experienced by a stream of events.

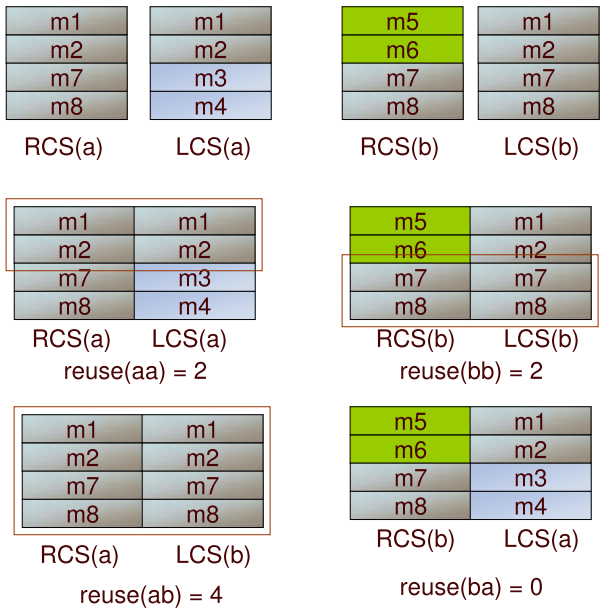


Figure 4: Effect of cache state on the number of cache misses for an event type.

5.1 Constructing the Modified Transition System

Recall from Section 3 that we are given a transition system $\mathcal{T} = (S, S_0, \Sigma, \Psi)$ which captures all possible sequences of event types that might occur in a stream. Using the cache modeling technique described in Section 4, and the transition system $\mathcal{T}$, we derive a transition system $\mathcal{T}' = (S', S'_0, D', \Psi')$ which captures *all* possible evolutions of the cache state, as a stream of events is processed. Each state $s' \in S'$ is a tuple (s, cs) where $s \in S$ and cs is a set of possible cache states at s. A transition from (s_1, cs_1) to (s_2, cs_2) belongs to Ψ' if and only if there exists a transition $s_1 \xrightarrow{\sigma} s_2$ in $\mathcal{T}$ and $cs_2 = NSTATE(\sigma, cs_1)$. The set of initial states S'_0 contains all tuples $(s, \perp)$ where $s \in S_0$. Finally, any transition ψ from (s_1, cs_1) to (s_2, cs_2) in Ψ', where $s_1 \xrightarrow{\sigma} s_2$, is annotated with $WCET(\sigma, cs_1)$. We denote this as $D'(\psi) = WCET(\sigma, cs_1)$. The construction of $\mathcal{T}'$ is formally specified by Algorithm 1.

Example: Figure 5 shows the transition system $\mathcal{T}'$. Each state in the transition system $\mathcal{T}'$ is now annotated with the corresponding cache state. The color coding shows how a state/transition in $\mathcal{T}$ maps to multiple states/transitions in $\mathcal{T}'$. For example, processing an event b takes 60 cycles in the original example. Now we notice that the preceding event can be of type a or b. If the preceding event is of type a, then we can save 4 cache misses (see Figure 4) and hence 20 cycles (assuming 5 cycle cache miss penalty). However, if the preceding event is of type b, then we can only save 2 cache misses, i.e., 10 cycles. Thus the state S1 in the original transition system generates two states in $\mathcal{T}'$ and also two transitions corresponding to the processing of event b.

Input: Transition system $\mathcal{T} = (S, S_0, \Sigma, \Psi)$ and the functions *NSTATE* and *WCET*;
Output: Transition system $\mathcal{T}' = (S', S'_0, D', \Psi')$;

```
Q ← S' ← S'_0 ← D' ← Ψ' ← ∅ ;
for all s ∈ S_0 do
    enqueue(Q, ⟨s, ⊥⟩) ;
    S' ← S' ∪ {⟨s, ⊥⟩} ;
    S'_0 ← S'_0 ∪ {⟨s, ⊥⟩} ;
end for
while Q ≠ ∅ do
    ⟨s, cs⟩ ⇐ dequeue(Q) ;
    for all transitions s --σ--> s' ∈ Ψ do
        cs' ← NSTATE(σ, cs) ;
        if ⟨s', cs'⟩ ∉ S' then
            enqueue(Q, ⟨s', cs'⟩) ;
            S' ← S' ∪ {⟨s', cs'⟩} ;
        end if
        Ψ' ← Ψ' ∪ {⟨s, cs⟩ --σ--> ⟨s', cs'⟩} ;
        D'(⟨s, cs⟩ --σ--> ⟨s', cs'⟩) ← WCET(σ, cs) ;
    end for
end while
```

Algorithm 1: Constructing the transition system $\mathcal{T}'$.

5.2 Timing Analysis

Recall from Section 3 that we are also given a function $\bar{\alpha}$, which bounds the maximum number of event arrivals over any time interval. More specifically, $\bar{\alpha}(\Delta)$ is the maximum number of events that can arrive over any time interval of length Δ. $\bar{\alpha}(\Delta)$ therefore specifies the timing properties of a *class* or *family* of arrival processes of event streams that are to be processed by the streaming application. To compute similar bounds on the timing properties of any processed stream and the maximum delay and backlog (which is a measure of the maximum buffer requirement) experienced by any input event stream, we need to compute the maximum processing requirement arising from the $\bar{\alpha}(\Delta)$ events. Towards this, let us define a function $\gamma(k)$ whose argument is an integer k and it returns the maximum processing time that can be demanded by *any* sequence of k consecutive events belonging to the stream.

We now show how to compute the function γ. Consider our transition system $\mathcal{T}'$, where each transition ψ from (s_1, cs_1) to (s_2, cs_2) represents the processing of an event of the type σ, where $s_1 \xrightarrow{\sigma} s_2 \in \mathcal{T}$. The annotation on each such transition, i.e. $D'(\psi)$, denotes the maximum processing time of the event σ, given that the cache state before the start of this processing is cs_1. Hence, $\gamma(k)$ is the weight of the maximum-weight path of length k in the transition system $\mathcal{T}'$. Given any k, $\gamma(k)$ can thus be easily computed by finding the single source longest paths (of length k) for all vertices in $\mathcal{T}'$; the single-source longest path for a given vertex is computed by standard dynamic programming methods. Algorithm 2 shows how to compute $\gamma(k)$ for all integers $1 \leq k \leq n$, where n is an input to this algorithm.

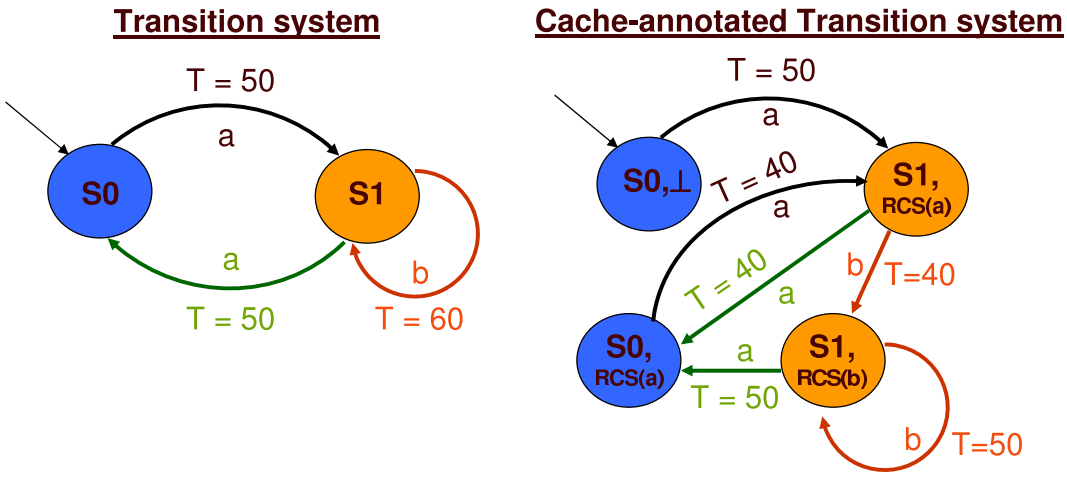


Figure 5: Constructing the transition system $\mathcal{T}'$.

Input: Transition system $\mathcal{T}' = (S', S'_0, D', \Psi')$, function $pred(s)$ which returns all predecessors of the state $s \in S'$, and an integer n;
Output: $\gamma(k)$ for all $1 \leq k \leq n$;
  $w_s(k) \leftarrow -\infty$ for all $s \in S'$, $1 \leq k \leq n$;
  $w_s(0) \leftarrow 0$ for all $s \in S'$;
  for $k = 1$ to n **do**
    for $\forall\ s \in S'$ **do**
      if $|pred(s)| > 0$ **then**
        $w_s(k) \leftarrow \max_{p \in pred(s)}\{w_p(k-1) + D'(p \rightarrow s)\}$
      end if
    end for
    $\gamma(k) \leftarrow \max_{s \in S'}\{w_s(k)\}$
  end for

Algorithm 2: Computing $\gamma(k)$.

It is easy to see that the function $\alpha(\Delta) = \gamma(\bar{\alpha}(\Delta))$ therefore represents the maximum processing requirement that can arise from the event stream within *any* time interval of length Δ, for $\forall\Delta \geq 0$. Using the results derived in [3], it is possible to show that worst case delay WCD (i.e. the maximum length of time between the arrival of any event and the time when it is completely processed) experienced by any event stream whose arrival process is bounded by $\bar{\alpha}(\Delta)$ is given by:

$$WCD = \sup_{\Delta \geq 0}\{\inf_{\tau \geq 0}\{\tau : \alpha(\Delta) \leq \Delta + \tau\}\}$$

Intuitively, WCD can be interpreted as the maximum horizontal distance between the curve $\alpha(\Delta)$ and the straight line representing the processor availability; see Figure 6. In this figure, the relevant point on $\alpha(\Delta)$ (that results in the *maximum* horizontal distance

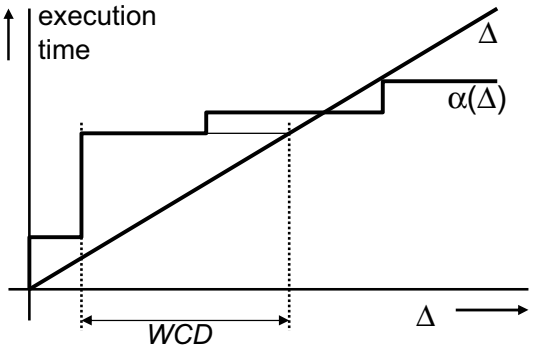


Figure 6: Computing the worst case delay *WCD* experienced by an event stream.

between this curve and the line denoting processor availability) denotes the time at which the event that suffers the maximum delay arrives. Similarly, the point on the line representing processor availability denotes the time at which sufficient (cummulative) processor capacity was available to meet the backlogged demand accumulated upto the point when the event that suffers the maximum delay arrived. Hence, the distance between these two points denotes the maximum delay.

To compute the maximum backlog, we first need to define a function γ^{-1}, which can be considered as the *pseudoinverse* of the function γ that we already defined above. We define, $\gamma^{-1}(\Delta) = \inf_{k \geq 0}\{k : \gamma(k) \geq \Delta\}$. Hence, $\gamma^{-1}(\Delta)$ returns the minimum number of events that can generate a processing requirement of Δ. In other words, *at least* $\gamma^{-1}(\Delta)$ events from the stream are guaranteed to be processed within a time interval of length Δ. Within this time interval, at most $\bar{\alpha}(\Delta)$ events might arrive. Hence, the backlog generated within this interval is $\bar{\alpha}(\Delta) - \gamma^{-1}(\Delta)$. Therefore, the maximum or worst case backlog *WCB* is given by:

$$WCB = \sup_{\Delta \geq 0}\{\bar{\alpha}(\Delta) - \gamma^{-1}(\Delta)\}$$

As in the case of computing WCD, intuitively, WCB can be interpreted as the maximum vertical distance between the curves $\bar{\alpha}(\Delta)$ and $\gamma^{-1}(\Delta)$ (see Figure 8).

To compute the timing properties of the processed stream, let us denote using $\bar{\alpha}'(\Delta)$ the maximum number of processed events that can possibly be seen at the output of the processor (see Figure 1(b)) within any time interval of length Δ. $\bar{\alpha}'(\Delta)$ is therefore exactly of the same form as $\bar{\alpha}(\Delta)$ which bounds an input stream. Again, using the results derived in [3], it may be shown that

$$\bar{\alpha}'(\Delta) = \sup_{\tau \geq 0}\{\bar{\alpha}(\Delta + \tau) - \gamma^{-1}(\tau)\}$$

The bounds on the timing properties of any processed stream and the maximum delay and backlog that we computed above, are more accurate compared to those computed in [3], where the effects of the processor's instruction cache was not taken into account. This

difference primarily stems from the use of the transition system $\mathcal{T}'$ in computing the function $\gamma(k)$. In contrast to this, the results in [3] rely on a significantly simpler approach of scaling the function $\bar{\alpha}(\Delta)$ by a constant representing the (same or constant) processing time per event, in order to obtain the function $\alpha(\Delta)$.

6 Improvements to the core framework

The running time of the algorithm presented so far would depend on the number of states in the transition system $\mathcal{T}$ and the number of cache states generated from our application and its code layout in the instruction cache. For many realistic problem instances, the number of such cache states might be very large, thereby our algorithm incurring a high running time. To get around this problem, there are three possible techniques that we can adopt (they are not mutually exclusive).

1. Partially constructing the transition system $\mathcal{T}'$.
2. Instead of computing $\gamma(k)$ for all values of k, exploit the fact that $\gamma(k)$ becomes periodic beyond a certain value of k.
3. Note that the computation of WCD, WCB, as well as $\bar{\alpha}'$ requires an iteration over all $\Delta \geq 0$. To avoid such an iteration over an unbounded range, we can approximate the functions $\bar{\alpha}(\Delta)$ and $\gamma(k)$ by a sequence of linear segments. Using such an approximation, we can then compute a Δ_{max} such that it would be sufficient to iterate only till this value for the computation of WCD, WCB and $\bar{\alpha}'$.

Note that while the first and the third techniques mentioned above will lead to a (safe) approximation of WCD, WCB and $\bar{\alpha}'$, the second technique will not lead to any loss of accuracy in our estimation of these quantities.

6.1 Partial Construction of $\mathcal{T}'$

Constructing the cache state annotated transition system $\mathcal{T}'$ is computationally expensive. This is because for each state s in the transition system $\mathcal{T}$ we need to compute the possible cache states with which s can be reached; each of these contribute to a state in $\mathcal{T}'$. Assuming a direct mapped cache with k cache lines and the program code of all event types spread over n contiguous memory blocks, the number of possible cache states is $\lceil (n/k) \rceil^k$. This leads to an obvious blowup in the number of states of $\mathcal{T}'$. To avoid this blowup, we can construct $\mathcal{T}'(U)$, an approximation of $\mathcal{T}'$; we assume that U is a pre-defined constant.

The basic idea for defining the approximation of $\mathcal{T}'$ is as follows. Clearly, a cache state is a function of the finite (but unbounded) execution history of events. We make the following observations about cache state evolutions.

- Given a bound U, the bounded execution history of the last U events provides a safe bound on the WCET of the current event, and

- Two event histories with common suffix may lead to the same cache state.

Our partial construction of $\mathcal{T}'$ is based on these two observations. In the transition system $\mathcal{T}'$ discussed earlier, each state of $\mathcal{T}'$ is of the form (s, cs). In the construction of $\mathcal{T}'(U)$, each state of this transition system is of the form (s, cs, seq) where s and cs are as defined in Section 4; seq is sequence of length at most U over the event alphabet Σ denoting the last U events (if less than U events have occurred, then seq contains fewer events). At first sight, our definition of the states of $\mathcal{T}'(U)$ seems to blowup the state space even further (as compared to the full construction of $\mathcal{T}'$). However, our construction of the transitions of $\mathcal{T}'(U)$ is such that the reachable state space of $\mathcal{T}'(U)$ is sparse.

We now describe the construction of $\mathcal{T}'(U)$. For this purpose, we use the algorithm for constructing $\mathcal{T}'$ (i.e., Algorithm 1) but with two important modifications. First of all, when we construct the destination states for a state (s, cs, seq) for event σ, the *NSTATE* function is applied for both the cache state and the sequence. Moreover, the *NSTATE* function for cache state is somewhat different from that employed in the construction of $\mathcal{T}'$. As the sequence associated with a state captures the last U events, we define the following. Note that $\circ$ denotes concatenation, and $seq = \langle \sigma_1, \sigma_2, \ldots, \sigma_U \rangle$.

$$NSTATE(\sigma, U, seq) = \begin{cases} seq \circ \sigma & \text{if } |seq| < U \\ \langle \sigma_2, \ldots, \sigma_U, \sigma \rangle & \text{if } |seq| = U \end{cases}$$

In other words, we ignore the events beyond the history of length U based on our first observation. Similarly, we need to ignore the effects on the cache states due to the events beyond the execution history of length U. So we define

$$NSTATE(\sigma, U, seq, cs) = \begin{cases} NSTATE(\sigma, cs) & \text{if } |seq| < U \\ NSTATE(\langle \sigma_2, \ldots, \sigma_U, \sigma \rangle, \perp) & \text{if } |seq| = U \end{cases}$$

Secondly, the check for whether a state (s', cs', seq') exists in S'' is done differently. The membership check performs state merging to exploit our second observation about cache state evolution. Two states (s', cs', seq') and (s', cs', seq'') are merged if seq' and seq'' share a common non-empty suffix. In this case, the execution history beyond the longest common suffix does not have any effect on the cache state and thus can be safely ignored. So the sequence for the merged state is simply the longest common suffix of seq' and seq''. *The modified algorithm for constructing the transition system $\mathcal{T}'(U)$ appears as Algorithm 3.*

As we are no longer maintaining exact cache states in $\mathcal{T}'(U)$, we need to show that a safe upper bound on WCET is obtained by analyzing $\mathcal{T}'(U)$ as opposed to $\mathcal{T}'$. If the WCET associated to an event σ at a certain cs does not decrease by removing the first event σ_1 from the event sequence that leads to cs, we can guarantee that analyzing the partially unrolled transition system $\mathcal{T}'(U)$ yields safe WCET bounds. Therefore, the following condition must be satisfied by the function $WCET(\sigma, cs)$.

$$\begin{array}{l} \text{WCET}(\sigma, cs) \leq \text{WCET}(\sigma, cs') \quad \text{for all } \sigma, \text{ where} \\ cs = \text{NSTATE}(\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle, \perp) \text{ and } cs' = \text{NSTATE}(\langle \sigma_2, \ldots, \sigma_n \rangle, \perp) \end{array}$$

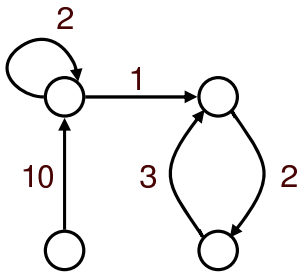
Input: Transition system $\mathcal{T} = (S, S_0, \Sigma, \Psi)$, the functions *NSTATE* and *WCET* and a positive integer U
Output: Transition system $\mathcal{T}'(U) = (S'', S_0'', D'', \Psi'')$;
$Q \leftarrow S'' \leftarrow S_0'' \leftarrow D'' \leftarrow \Psi'' \leftarrow \emptyset$;
for all s $\in S_0$ **do**
$enqueue(Q, \langle s, \perp, \epsilon \rangle)$; $S'' \leftarrow S'' \cup \{\langle s, \perp, \epsilon \rangle\}$;
$S_0'' \leftarrow S_0'' \cup \{\langle s, \perp, \epsilon \rangle\}$;
end for
while $Q \neq \emptyset$ **do**
$\langle s, cs, seq \rangle \Leftarrow dequeue(Q)$;
for all transitions $s \xrightarrow{\sigma} s' \in \Psi$ **do**
$cs' \leftarrow NSTATE(\sigma, U, seq, cs)$;
$seq' \leftarrow NSTATE(\sigma, U, seq)$;
if $\exists \langle s', cs', seq'' \rangle \in S''$ s.t. longest-common-suffix(seq',seq'') $\neq \epsilon$ **then**
$seq' \leftarrow$ longest-common-suffix(seq', seq'');
replace $\langle s', cs', seq'' \rangle$ with $\langle s', cs', seq' \rangle$ in S'' ;
else if $\langle s', cs', seq' \rangle \notin S''$ **then**
$enqueue(Q, \langle s', cs', seq' \rangle)$;
$S'' \leftarrow S'' \cup \{\langle s', cs', seq' \rangle\}$;
end if
$\Psi'' \leftarrow \Psi'' \cup \{\langle s, cs, seq \rangle \xrightarrow{\sigma} \langle s', cs', seq' \rangle\}$;
$D''(\langle s, cs, seq \rangle \xrightarrow{\sigma} \langle s', cs', seq' \rangle) \leftarrow WCET(\sigma, cs)$;
end for
end while

Algorithm 3: Constructing the transition system $\mathcal{T}'(U)$

That is, starting with an empty cache, executing σ after $\sigma_1, \sigma_2, \ldots, \sigma_n$ should not produce more cache misses than executing σ after $\sigma_2, \ldots, \sigma_n$. This is indeed the case for direct mapped as well as set-associative caches with common replacement policies such as LRU. To see that this is indeed the case, consider the cached execution of an event σ under two different histories $\sigma_1, \sigma_2, \ldots, \sigma_n$ and $\sigma_2, \ldots, \sigma_n$. What can be the effect of σ_1 on the execution of σ? The memory blocks of σ_1 which are replaced by $\sigma_2, \ldots, \sigma_n$, clearly have no effect on σ's execution. On the other hand, if some memory blocks of σ_1 do not get replaced by $\sigma_2, \ldots, \sigma_n$, these memory blocks of σ_1 can only reduce the cache misses in σ's execution.

6.2 Computing γ

Recall from Section 5 that $\gamma(k)$ is the weight of the maximum-weight path. Our computation of the maximum delay and backlog experienced by a stream requires the computation of $\gamma(k)$ for $k = 1, \ldots, n$, where the value of n would depend on the range of Δ over which we need to iterate. Here, we would like to point out that it is sufficient to compute $\gamma(k)$ for all $k \leq n_0$ (for some n_0). Typically, n_0 would be much smaller than the maximum value of k for which we will need to determine $\gamma(k)$ during our computation of *WCD* and *WCB*, and n_0 would only depend on the transition system $\mathcal{T}'$.



k	γ(k)
1	10
2	12
3	14
4	16
5	18
6	21
7	23
8	26
9	28

Figure 7: An example transition system $\mathcal{T}'$ (weighted graph) and the function $\gamma(k)$ corresponding to this transition system, for different values of k.

The above observation stems from the fact the the weight of the maximum-weight path of length k in a graph eventually becomes periodic with increasing k, beyond a certain value of k (see [6] and [12]). Let us denote this period as p, and the increment in the sum of the edge weights within this period as q. Given a graph, the values of p and q depend on the number of edges and the sum of the weights in the cycle with the maximum *mean* (i.e. the sum of the weights divided by the number of edges). Both, p and q can be efficiently determined (see [12]).

Therefore, $\gamma(k)$, for increasing values of k, is made up of a prelude of length n_0 followed by a periodic continuation. For any $k \geq n_0$, $\gamma(k)$ is given as:

$$\gamma(k) = \gamma((n_0 - p) + (k - n_0) \bmod p) + \lfloor \frac{k - n_0 + p}{p} \rfloor q \tag{4}$$

The value of n_0 can be determined by computing $\gamma(k)$ on $\mathcal{T}'$ for all $1 \leq k \leq n$ with a sufficiently large choice of n, and testing for periodicity. Towards this, we test if Eqn. 4 holds for the last p values of $\gamma(k)$ from $k = n$, with p and q determined from the cycle in $\mathcal{T}'$ with the maximum mean weight. For this test, we set $n_0 = n - p + 1$ and check if Eqn. 4 holds for all $n_0 \leq k \leq n_0 + p - 1$.

As a simple example, let us consider the weighted graph shown in Figure 7, corresponding to a transition system $\mathcal{T}'$. Figure 7 also lists $\gamma(k)$ for different values of k, corresponding to this graph. Clearly, the cycle in this graph with the maximum mean weight has a cycle length of 2. Hence, $p = 2$, and q corresponding to this cycle is $2 + 3 = 5$. From the list of different values of $\gamma(k)$, it may be seen that $n_0 = 6$. Hence, Eqn. 4 in the case of this graph may be formulated as: for all $k \geq 6$,

$$\gamma(k) = \gamma(4 + (k - 6) \bmod 2) + \lfloor \frac{e - 4}{2} \rfloor 5$$

Therefore, in the case of this graph, $\gamma(k)$ only needs to computed for $k < 6$. For any $k \geq 6$, $\gamma(k)$ can be computed in constant time.

6.3 Approximating $\bar{\alpha}$ and γ

Note that in general, $\bar{\alpha}$ and γ can be arbitrary functions. In this subsection we show that by approximating $\bar{\alpha}$ and γ using affine functions, it is possible to derive a $\Delta_{\max}$ such that it is sufficient to restrict our iteration of Δ only till this value, for the computation of the maximum delay and backlog experienced by a stream. In other words, the computation of *WCD* and *WCB* can now be given by:

$$\begin{aligned} WCD &= \sup_{0 \leq \Delta \leq \Delta_{\max}} \{\inf_{\tau \geq 0} \{\tau : \alpha(\Delta) \leq \Delta + \tau\}\} \\ WCB &= \sup_{0 \leq \Delta \leq \Delta_{\max}} \{\bar{\alpha}(\Delta) - \gamma^{-1}(\Delta)\} \end{aligned}$$

The approximation of any given $\bar{\alpha}$ and γ using affine functions involves the selection of constants $r_{\bar{\alpha}}$, $s_{\bar{\alpha}}$, r_γ and s_γ, such that the following two inequalities hold:

$$\begin{aligned} \bar{\alpha}(\Delta) &\leq r_{\bar{\alpha}} + \Delta \cdot s_{\bar{\alpha}},\ \forall \Delta \in \mathbb{R}^{\geq 0} \\ \gamma(k) &\leq r_\gamma + k \cdot s_\gamma,\ \forall k \in \mathbb{Z}^{\geq 0} \end{aligned}$$

Now recall from Section 5 the definitions of the functions α and γ^{-1}. Using our approximations of $\bar{\alpha}$ and γ, it is possible to derive affine bounds on α and γ^{-1} as well. These are given by:

$$\begin{aligned} \alpha(\Delta) &\leq r_\alpha + \Delta \cdot s_\alpha,\ \forall \Delta \in \mathbb{R}^{\geq 0} \\ \gamma^{-1}(\Delta) &\geq r_{\gamma^{-1}} + \Delta \cdot s_{\gamma^{-1}},\ \forall \Delta \in \mathbb{R}^{\geq 0} \end{aligned}$$

where,

$$r_\alpha = r_\gamma + r_{\bar{\alpha}} s_\gamma,\ s_\alpha = s_{\bar{\alpha}} s_\gamma,\ r_{\gamma^{-1}} = -\frac{r_\gamma}{s_\gamma} \text{ and } s_{\gamma^{-1}} = \frac{1}{s_\gamma}$$

From our computation of the maximum backlog, *WCB*, experienced by a stream, it is easy to see that $\Delta_{\max}$ can be the Δ-intercept of the intersection point of the affine bounds on α and γ^{-1} (see Figure 8). Such a $\Delta_{\max}$ is therefore given by:

$$\Delta_{\max} = \frac{r_\gamma + r_{\bar{\alpha}} s_\gamma}{1 - s_{\bar{\alpha}} s_\gamma}$$

The same value of $\Delta_{\max}$ can also be obtained from the computation of *WCD*, the maximum delay experienced by any event of the stream, by computing the intersection point of the affine bound on α with the straight line representing the processor availability.

Using the affine bounds on $\bar{\alpha}$ and γ, it is also possible to bound on *WCD* and *WCB* as follows:

$$WCD \leq r_\gamma + r_{\bar{\alpha}} s_\gamma\ ; \quad WCB \leq r_{\bar{\alpha}} + \max\{r_\gamma s_{\bar{\alpha}}, \frac{r_\gamma}{s_\gamma}\}$$

The function $\bar{\alpha}'$ can also be similarly bounded. Although these bounds are computationally simpler, in general they are not as tight as those derived in Section 5.

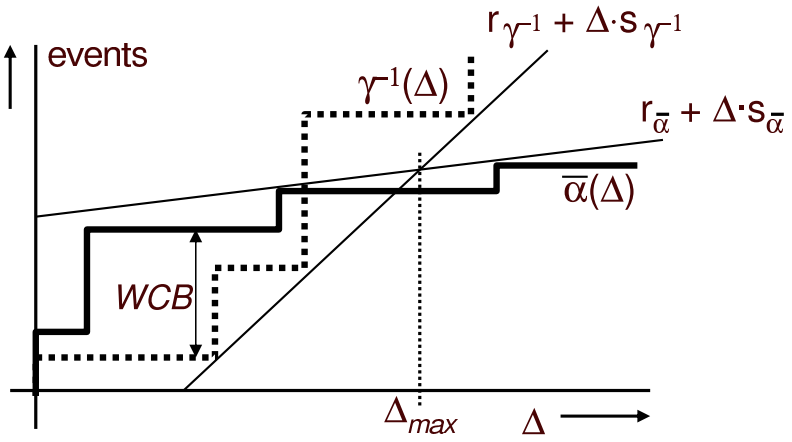


Figure 8: Computing $\Delta_{\max}$ from the affine bounds on γ^{-1} and $\bar{\alpha}$.

7 A Case Study

Our prototype implementation of the timing analysis framework consists of three parts, each of which correspond to the three steps shown in Figure 3. The detailed design flow of our framework is shown in Figure 9.

Given the streaming application and the program path in this application for each event type, we first identify the code for each event type. The code for each event type, the cache configuration and the code layout are provided to a *cache state analyzer* (implemented in C). This analyzer produces the Reaching Cache States (RCS) and the Live Cache States (LCS) for each event type (Definitions 4.1 and 4.2). We then provide this LCS/RCS information, as well as the transition system $\mathcal{T}$ to construct the cache-annotated transition system $\mathcal{T}'$, again implemented in C.

We have implemented our timing analysis framework (the results derived in Section 5) using the Mathematica toolkit [22]. The main motivation behind using Mathematica is that it supports symbolic computations, using which it is possible compute *WCD*, *WCB* and $\bar{\alpha}'$ (when α, $\bar{\alpha}$ and γ^{-1} are represented as a sequence of linear segments, not necessarily only affine) without resorting to "pointwise" computations. The input to our Mathematica program are the arrival curve $\bar{\alpha}(\Delta)$ and the cache-annotated transition system $\mathcal{T}'$. The outputs of this program are estimates of the worst case delay (*WCD*), worst case buffer fill level (*WCB*) and bounds on the timing properties of the processed streams $\bar{\alpha}'$.

We now present a realistic case study to illustrate how the estimated timing properties of an application are affected when the instruction cache is modeled using our proposed framework. This case study also serves to validate our framework and shows that our modeling of the cache behavior is efficient and scales to handle real-life setups.

Our application consists of an MPEG-2 encoder running on a device such as a Personal Digital Assistant (PDA) or a mobile phone, that has a small movie camera attached to it.

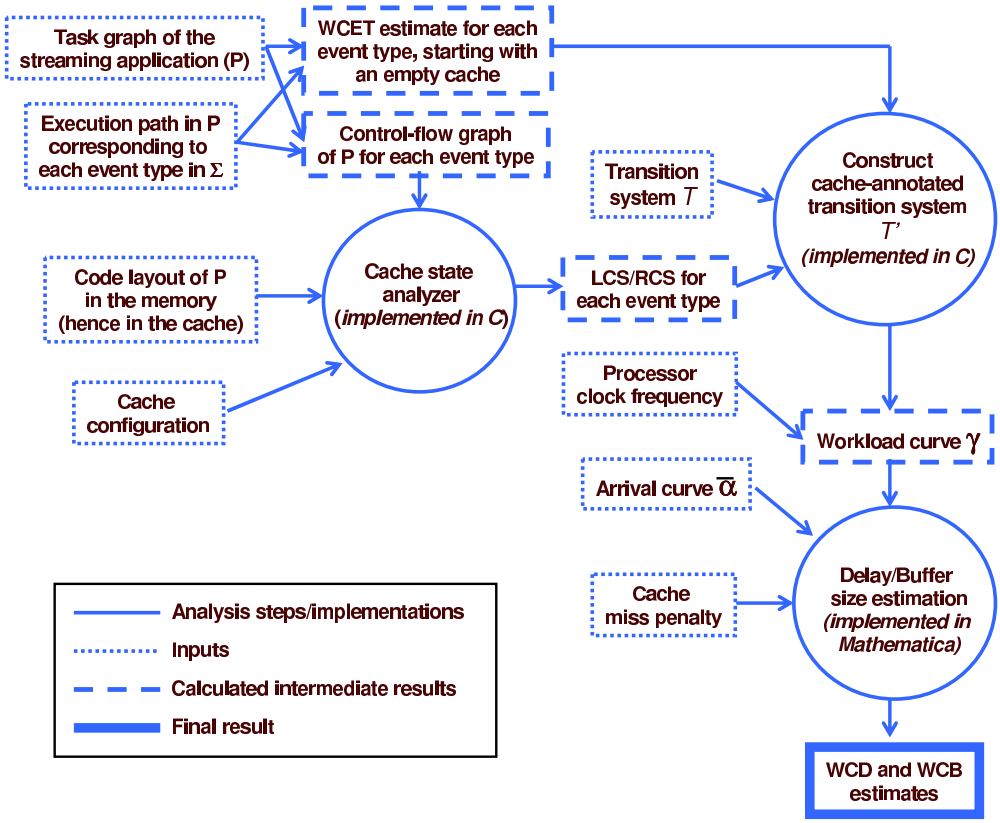


Figure 9: Design flow of our timing analysis framework.

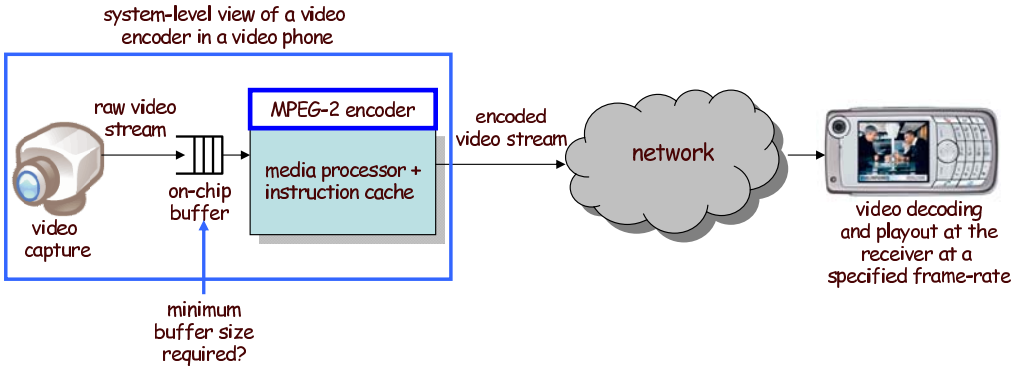


Figure 10: Application scenario: MPEG-2 encoder in a video phone.

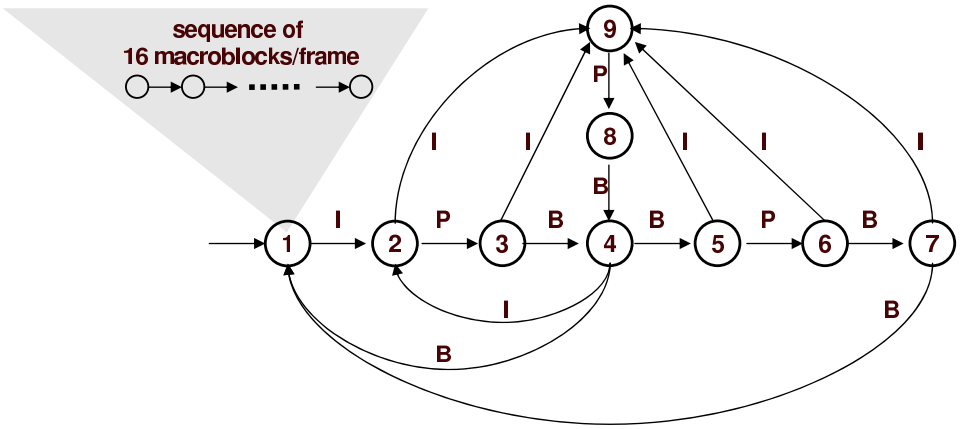


Figure 11: Transition system $\mathcal{T}$ specifying the possible frame patterns according to which a raw video stream is encoded.

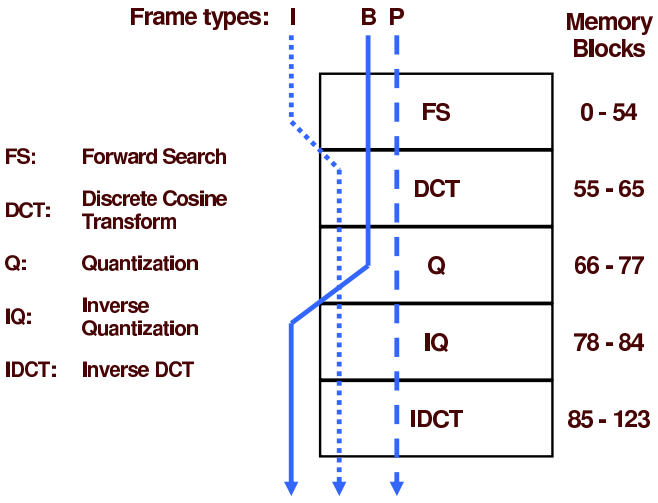


Figure 12: The MPEG-2 encoder's code layout in the memory and the sequence of tasks executed for each of the three different frame types, I, B and P.

Many of these devices today have general-purpose processors running a light-weight operating system and multiple applications. In our setup, the input to the encoder application is a constant bit-rate raw video stream and its output is a 64×64 pixel MPEG-2 encoded clip. We assume that such a clip would be played out at the rate of 30 frames per second, which in turn determines the sampling rate of the camera capturing the video. Our setup is shown in Figure 10. The raw bitstream is stored in a small on-chip buffer, which is read out by the processor running the encoder application. Since the computational workload involved in encoding each *macroblock* is dependent on the data being encoded, it is highly variable. Hence, the fill-level of the on-chip buffer varies over time and it is important to choose an appropriate buffer size at design time, especially since on-chip buffers are expensive and occupy a significant fraction of the chip area.

We modeled an encoder application with five *tasks*. These are *forward search* (FS), *discrete cosine transform* (DCT), *quantization* (Q), *inverse quantization* (IQ), and *inverse discrete cosine transform* (IDCT). The memory layout of these tasks appears in Figure 12. We consider a direct-mapped instruction cache with 64 cache lines and 64 bytes block size.

The incoming raw bitsream is encoded into a sequence of I, B and P frames, where possible patterns of I, B and P are determined by the transition system given in Figure 11. This transition system is determined by the implementation of the encoder application. It may be noted here that the MPEG-2 standard does not prescribe any particular encoder implementation. Given that the frame resolution in our case is 64×64 pixels, each frame is composed of 16 macroblocks, each of size 16×16 pixels. The encoding of macroblocks constituting different frame types requires a different sequence of tasks getting executed. For example, all macroblocks belonging to an I-frame requires the tasks DCT, Q, IQ and IDCT to be executed. This task set, along with the task sets corresponding to B and P frames are listed in the table below.

Frame Type	Task Set
I-Frame	DCT, Q, IQ, IDCT
P-Frame	FS, DCT, Q, IQ, IDCT
B-Frame	FS, DCT, Q

The worst-case execution times of the five different tasks (in terms of number of processor cycles), when processing macroblocks of different frame types are given in the table below. These numbers were obtained with an instruction cache miss penalty of 100 cycles.

	FS	DCT	Q	IQ	IDCT
I	0	23204	12624	5177	16061
P	285918	23307	15919	7258	15943
B	134601	23307	11656	0	0

As a sequence of macroblocks gets processed (or encoded), different tasks get executed following the pattern given by the transition system in Figure 11. Note that for any two

macroblocks belonging to different frame types, there is a significant overlap between the tasks that get executed.

The worst-case delay and buffer size with the processor frequency set to 105 MHz and an instruction cache penalty of 100 cycles is shown in the table below.

	Maximum delay experienced by any macroblock
With Cache Modeling	28 ms
Without Cache Modeling	44 ms
	Estimated minimum buffer size required
With Cache Modeling	13.30 macroblocks
Without Cache Modeling	20.68 macroblocks

From this table, it may be noted that modeling the effects of the instruction cache leads to tighter estimates of both – the on-chip buffer size and delay. The buffer size estimation reduces by 33% and the delay estimate reduces by 36%. Similar results were also obtained for other cache sizes and different code layouts. The results we reported above were obtained with the most "natural" code layout, given the sequence of tasks involved in processing each frame type. Such tighter estimates directly translate into better resource dimensioning and improved system design. As mentioned before, the crux of our approach is in accounting for the fact that there is significant overlap in the code involved in processing the different frame types in the event stream. We believe that this property can be exploited in a variety of streaming applications.

Finally, it may be noted that for this application (MPEG-2 encoder) and cache size, the cache annotated transition system had 339 states and took less than 3 secs to run using our C-based implementation. This indicates that our analysis indeed scales up to realistic applications. Given the relatively low running times in this case, there was no need to use the technique described in Section 6.1 to "approximate" the cache-annotated transition system. However, for significantly larger applications this might be necessary.

8 Concluding Remarks

In this paper we presented a framework for accurate timing analysis of streaming applications by taking into account the micro-architectural features of the processor running such an application (more specifically, the effects of the instruction cache). The framework we presented can be used to evaluate important performance metrics in the context of designing stream processing applications and architectures. These include buffer size and delay/response time analysis.

We note that there has been a lot of recent interest in accurately modeling the effects of correlations between different computation and communication tasks in the context of performance analysis of embedded systems (see [11] and the references therein). However, we are not aware of any work which models the effects of the correlation between different tasks/events on the execution times of these tasks/events. This paper addresses this issue:

it shows how to model the effects of execution context or history to capture the fact that the execution time of one occurrence of an event might be different from another occurrence of the same event. This is done by using a novel composition technique which integrates the specification of the event arrival process (or timing properties) with the specification of the execution time demand of the event stream. Our performance analysis framework does fine grained analysis of the execution time of each event instead of taking it as a constant. Our experimental results from the MPEG-2 encoder application show that our performance analysis framework can help in uncovering counter-intuitive effects of code layout and micro-architectural features on the performance of an embedded system running a streaming application.

We note that the current work is only the "tip of the iceberg" far as integrating micro-architectural modeling is concerned. We envision lot more work in this direction – integrating the modeling of set associative caches, complex replacement policies, data cache, branch prediction and pipelines into the timing analysis of streaming applications. We now briefly elaborate some of these avenues of future work.

Pipelines and timing anomaly: One important direction of future work is to consider the effects of other micro-architectural features (such as pipelines) on the timing analysis problem. This will require us to resolve several issues. First of all, if we consider more detailed micro-architectural features, then the number of possible micro-architectural states at the beginning of an event will certainly blow up. For this purpose, we plan to investigate the possibility of merging micro-architectural states in a manner similar to our partial construction of the cache annotated transition system $\mathcal{T}'$. A more technical issue also arises in this regard. In this paper, our instruction cache modeling finds the reaching cache states (RCS) for each event as a least fixed point. The RCS information is then used to find the possible cache states prior to any event σ in the transition system $\mathcal{T}'$; this involves another fixed point computation. Such a decomposition into two levels of fixed-point computation may not be possible if we consider other micro-architectural features (like pipelines). This is because of the well-known *timing anomaly* problem in out-of-order processor pipelines [18, 21]. As a result, we will need to: (a) replace the occurrence of events in the transition system $\mathcal{T}$ by their control flow graphs, and (b) perform a *single* fixed point computation on this graph to find all possible micro-architectural states. The time and space overheads of such an analysis will be investigated in future to evaluate its practicality.

Extension to set-associative caches: The cache modeling described in the paper is for direct-mapped caches. To extend the notion of cache states, live cache states (LCS) and reaching cache states (RCS) to set-associative caches, we need the following simple modifications. First of all, the definition of a cache state can be revised to a vector over $\{M \cup \{\perp\}\}^k$ for k-way set associative caches (where M is the set of memory blocks). Each iteration of the fixed point computation for LCS and RCS also requires a trivial change. In particular, the binary operation $\oplus$ over memory blocks needs to lifted to cache sets (of cardinality k). If m, m' are cache sets, then $m \oplus m'$ contains:

- all the non-empty cache lines of m', and
- $k - n$ cache lines of m (according to the replacement policy of the cache) where m' contains $n \leq k$ non-empty cache lines.

Clearly, if m and m' together contain more than k empty cache lines then $m \oplus m'$ also contains empty cache lines. Now which $k - n$ cache lines are chosen from m? That clearly depends on the replacement policy (such as LRU/FIFO). For the LRU policy, we will choose the last $k - n$ accessed cache lines of m; for the FIFO policy we will choose the last $k - n$ cache lines as per their relative entry times (into the cache).

Other specification formalisms: Finally, we note that there exist various other specification formalisms for expressing constraints on event arrivals and service availability for real-time systems. One such recent proposal is the Logic of Constraints [5] (LOC) which can describe performance as well as (certain kinds of) functional constraints. The authors of this paper have studied the expressivity of LOC with respect to well-established formalisms for studying functionality of event sequences such as Linear-time Temporal Logic (LTL). Instead of comparing LOC with LTL, we feel that LOC could be combined with LTL to form a powerful specification for streaming applications. Thus, LOC could play the role of our arrival curves whereas LTL properties could be used to describe the event arrival sequences captured by our event transition system. It would be interesting to study whether the performance analysis method described in this paper can directly handle such specifications as well.

Acknowledgements

Thanks are due to Unmesh Dutta Bordoloi and Cem Derdiyok for helping with some of the experiments. We would also like to thank the reviewers of our ECRTS 2007 submissions for several suggestions that have helped in improving this paper.

References

[1] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003.

[2] S. Baruah, D. Chen, S. Gorinsky, and A.K. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.

[3] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. 6th Design, Automation and Test in Europe (DATE)*, pages 190–195, Munich, Germany, March 2003.

[4] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister. Performance evaluation of network processor architectures: Combining simulation with analytical estimation. *Computer Networks*, 41(5), 2003.

[5] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Logic of constraints: A quantitative performance and functional constraint formalism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 23(8), 2004.

[6] G. Cohen, D. Dubois, J. P. Quadrat, and M. Viot. A linear-system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing. *IEEE Transactions on Automatic Control*, 30(3):210–220, March 1985.

[7] R. Cruz. A calculus for network delay, Parts 1 & 2. *IEEE Transactions on Information Theory*, 37(1), 1991.

[8] A. Datta, S. Choudhury, A. Basu, H. Tomiyama, and N. Dutt. Satisfying timing constraints of preemptive real-time tasks through task layout technique. In *IEEE International Conference on VLSI Design*, 2001.

[9] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, 2002.

[10] M. I. Gordon et al. A stream compiler for communication-exposed architectures. In *10th Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[11] M. Jersak, R. Henia, and R. Ernst. Context-aware performance analysis for efficient embedded system design. In *Proc. 7th Design, Automation and Test in Europe (DATE)*, 2004.

[12] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[13] B. Khailany et al. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[14] J.-Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. LNCS 2050, Springer, 2001.

[15] C-G. Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptiv e scheduling. *IEEE Transactions on Computers*, 47(6), 1998.

[16] M. Lee et al. A dual-mode instruction prefetch scheme for improved worst case and average case program execution times. In *IEEE International Real-Time Systems Symposium (RTSS)*, 1993.

[17] X. Li, T. Mitra, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *ACM Design Automation Conf. (DAC)*, 2003.

[18] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *IEEE Real-time Systems Symposium (RTSS)*, 2004.

[19] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3), 1999.

[20] P. Lieverse, T. Stefanov, P. van der Wolf, and E. F. Deprettere. System level design with Spade: an M-JPEG case study. In *ICCAD*, 2001.

[21] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[22] Mathematica 5, Wolfram Research. http://www.wolfram.com/products/mathematica/index.html.

[23] A. Maxiaguine, S. Künzli, S. Chakraborty, and L. Thiele. Rate analysis for streaming applications with on-chip buffer constraints. In *ASP-DAC*, 2004.

[24] T. Mitra and A. Roychoudhury. Worst-case execution time and energy analysis. In *Compiler Design Handbook (2nd Edition)*. CRC Press, 2007.

[25] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, 1997.

[26] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache related preemption delay. In *CODES+ISSS*, 2003.

[27] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and E.F. Deprettere. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, 2001.

[28] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2002.

[29] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-time Systems*, 1(2), 1989.

[30] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. IEEE Computer Society, 2002.

[31] K. Richter, M. Jersak, and R. Ernst. A formal approach to MpSoC performance verification. *IEEE Computer*, 36(4), 2003.

[32] K. Richter, R. Racu, and R. Ernst. Scheduling analysis integration for heterogeneous multiprocessor soc. In *IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[33] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proceedings of the Design Automation Conference (DAC)*. ACM, 2002.

[34] M. J. Rutten, J. T. J. van Eijndhoven, E. G. T. Jaspers, P. van der Wolf, O.P. Gangwal, and A. Timmer. A heterogeneous multiprocessor architecture for flexible media processing. *IEEE Design & Test of Computers*, 19(4):39–50, July-August 2002.

[35] J. E. Sasinowski and J. K. Strosnider. A dynamic programming algorithm for cache/memory paritioning for real-time systems. *IEEE Transactions on Computers*, 42(8), 1993.

[36] A. C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 1(2), 1989.

[37] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *ACM International Conference on Embedded Software (EMSOFT)*, 2004.

[38] J. Staschulat and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

[39] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analysis. *Journal of Real Time Systems*, May 2000.

[40] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th Conference on Compiler Construction*, LNCS 2304, pages 179–196, 2002.

[41] H. Tomiyama and N. Dutt. Program path analysis to bound cache-related preemption delay in preem ptive real-time systems. In *ACM Intl. Symp. on Hardware-Software Codesign (CODES)*, 2000.

[42] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *International Real-Time Systems Symposium (RTSS)*, 2003.

[43] V. D. Živković, P. van der Wolf, E. F. Deprettere, and E. A. de Kock. Design space exploration of streaming multiprocessor architectures. In *IEEE Workshop on Signal Processing Systems (SIPS)*, San Diego, California, 2002.

[44] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characterization of event streams in analysis of hard real-time applications. *Real-Time Systems*, 29(2), 2005.

[45] E. Wandeler and L. Thiele. Abstracting functionality for modular performance analysis of hard real-time systems. In *Asia and South Pacific Desing Automation Conference (ASP-DAC)*, 2005.

[46] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrm. The Determination of Worst-Case Execution Times – Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.