Cache-Aware Timing Analysis of Streaming Applications

Samarjit Chakraborty[1] Tulika Mitra[1] Abhik Roychoudhury[1]
Lothar Thiele[2] Unmesh D. Bordoloi[1] Cem Derdiyok[3]
[1]National University of Singapore
[2]Eidgenössische Technische Hochschule Zürich
[3]Ecole Polytechnique Fédérale de Lausanne
{samarjit, tulika, abhik, unmeshdu}@comp.nus.edu.sg, thiele@tik.ee.ethz.ch, cem.derdiyok@epfl.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. Most of the previous work related to estimating or optimizing these timing properties take a high-level view of the architecture and neglect microarchitectural features such as caches. In this paper, we show that an accurate estimation of a streaming application's timing properties, however, heavily relies on an appropriate modeling of the processor microarchitecture, such as its instruction cache. Towards this, we present a novel framework for 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. We have implemented a prototype of this framework partly in C and partly in Mathematica and plan to integrate it into a design-space exploration tool for system-level design of hardware-software architectures for streaming applications.

1 Introduction

Today, stream processing applications are 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. Many of these domains have very stringent constraints pertaining to cost, performance and power consumption and have posed several challenges in terms of developing appropriate models, methodologies and design tools. Most applications in these domains are centered around the notion of a "stream", and it is now increasingly being realized that conventional models, languages and design methodologies developed in the embedded systems domain do not adequately exploit this notion. To address this shortcoming, recently there has been a number of developments in the form of new "stream-centric" programming languages and compiler support [6], processor architectures [17] and design methodologies.

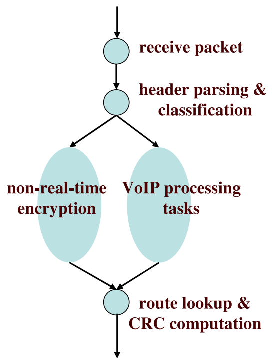

receive packet
header parsing & classification
non-real-time encryption
VoIP processing tasks
route lookup & CRC computation

Figure 1. Task graph corresponding to a simple software-based router which processes packets of two different types.

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 which 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. The task graph corresponding to such a code, implementing a simple software-based router is shown in Figure 1. 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 and which do not have 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. 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. Given the arrival process (timing properties) of a stream of incoming packets, we would like to determine the timing properties of the packet stream after it has been processed by this code. Assuming that the worst case execution time (WCET) associated with each packet type is given and is a constant, this timing analysis problem has been addressed in [3]. However, this problem becomes significantly more complicated if we take into account the fact that *the execution time involved in processing two packets of the same type also might vary*. This is because the sequence of previously processed packets determine the *state* of the processor's microarchitecture, which affects the execution time involved in processing any packet. Thus, the execution time associated with any packet might vary considerably, depending not only on the type of the packet but also on the sequence of packets processed prior to the packet in question.

Given a setup such as the one described above, in this paper we solve the above mentioned 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 at the input of the task graph shown in Figure 1. This problem specification will enable us to rule out certain "worst-case states" of the cache and thereby bound the WCET in processing any packet of a particular type.

The system model we consider in this paper is shown in Figure 2. 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. Apart from the code layout (mapping of the memory blocks onto the instruction cache) of the application, and a specification of the possible sequences of events, we are also 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 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* [5, 8]. We formally describe this specification in Section 2.

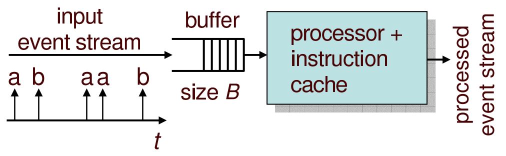


Figure 2. A processor with an instruction cache, on which the code (for example, that shown in Figure 1) processing an event stream is executed. Incoming events are stored in a buffer of size B, which is read by the processor.

By computing the timing properties of the processed event stream from those of the input stream, we show how to accurately compute important performance metrics pertaining to the application as well as the architecture. These include (i) the minimum buffer size B to guarantee that it does not overflow, and (ii) the maximum delay experienced by any event from its arrival, till the time it is completely processed.

Related work: The work reported in this paper is closely related to the problem of statically analyzing the WCET of a program, which is an important problem in the domain of real-time and embedded software design. Note that WCET analysis techniques 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 microarchitectural modeling. Both path analysis and micro-architectural modeling have been studied extensively [9, 10, 13, 14, 18, 19] 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 single event we consider the micro-architectural state resulting from the processing of all previous events.

As mentioned earlier, our work is also related to the system-level timing analysis problems studied in [3, 15, 16]. 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 microarchitecture on the execution time of the different events. In this context, the importance our work stems from the fact that in many cases on-chip buffer/memory is available only at a premium because of its high 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 microarchitecture as we attempt to do in this paper.

Organization of this paper: The rest of the paper is organized as follows. In the next section we formally state our problem and describe the underlying models used in this paper. This is followed by our cache modeling in Section 3. We then make use of this cache model to bound the WCET involved in processing a *single* event of any specified type. In Section 4 we then show how to use such WCET estimates for single events to perform a WCET analysis of a *stream* of events, i.e. solve our timing analysis problem. More specifically, we show how to compute the maximum buffer fill level and the maximum delay experienced by a stream of events. Some implementation issues to improve the running times of the algorithms presented in Sections 3 and 4 are then discussed in Section 5. A case study is presented in Section 6 to illustrate the utility of our proposed framework. Finally, in Section 7 we conclude by summarizing the implications of this work and then outlining some directions for future work.

2 Problem Formulation

Our problem specification has the following components:

1) A task graph, such as the one shown in Figure 1, which models the streaming application and the corresponding program or code. This application executes on a single processor and we are also given its code layout in the memory, i.e., the mapping of the different memory blocks of the application onto the instruction cache.

2) The specification of the event stream to be processed by the application is composed of two parts. As mentioned in Section 1, events are *typed*. Let us assume that these *types* are drawn from a finite set Σ. The first part of the stream specification is a transition system $\mathcal{T} = (S, S_0, \Sigma, \Psi)$ which captures all possible *sequences of event types* that might occur in the stream. Here, S is finite set of states, $S_0 \subseteq S$ is a set of initial states, and $\Psi \subseteq S \times \Sigma \times S$ is a set of transitions. Henceforth, we denote any transition $\langle s, \sigma, s' \rangle$ in Ψ by $s \overset{\sigma}{\to} s'$. Any sequence of events in the stream can only be generated as follows. The system starts in an initial state, and if $s \overset{\sigma}{\to} s'$ then the system can change its state from s to s' and generate an event of type σ. The transition system $\mathcal{T}$ can be used to model constraints on allowable sequences of events. $\mathcal{T}$ 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.

3) 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*. For a stream bounded by $\bar{\alpha}$, let $R(t)$ denote the number of events that arrived during the time interval $[0, t]$. Then, the inequality $R(t + \Delta) - R(t) \leq \bar{\alpha}(\Delta)$ holds for all $t \geq 0$ and $\Delta \geq 0$, for any concrete arrival process $R(t)$. As an example, $\bar{\alpha} = b + r\Delta$ specifies a stream with a *burst* size b (i.e. the number of events that can arrive at any instant in time) and a long-term arrival rate of r. As mentioned in Section 1, such an abstraction of the arrival process of streams (more specifically, packet flows) is common in the domain of communication networks [5, 8]. 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, 12], and can more accurately model streams exhibiting a high degree of burstiness (see [3]).

4) Lastly, for each event type in Σ we are given the execution path (or a set of paths) through the control-flow graph of the application. We are also given the worst case execution time for each of these execution paths, i.e. the execution time corresponding to the case where the cache is empty before the processing of any event starts. In other words, the first references to all memory blocks always result cache misses. Finally, we are also given the cache miss penalty, using which we can compute the processing time of an event in the case where some of the first memory references result in a cache hit.

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 in Section 1, 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. To compute the maximum delay and buffer size, the evolution of the cache state has to be linked to the arrival process of the events, which is bounded by $\bar{\alpha}$.

3 Cache Modeling

The basic technique used in this section bears some similarity with that used for computing cache related preemption delays in [13]. However, in this paper we do not consider task preemptions—we deal with a single stream, and events from this stream are processed to completion in a FCFS manner.

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. 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 1 (Reaching Cache States) *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 2 (Live Cache States) *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. 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 $\tau(\sigma)$. For any basic block $B \in \mathcal{B}_\sigma$, we can now compute two quantities RCS_B^{IN} and RCS_B^{OUT}. RCS_B^{IN} is the set of possible cache states when B is reached via any incoming program path and RCS_B^{OUT} is the set of possible cache states when B completes execution. These quantities are computed by propagation; thus RCS_B^{IN} will be computed using the RCS^{OUT} estimates of the basic blocks from where there is an incoming edge to block B (for more details the reader is referred to [13]). Since the control flow graph contains loops, the RCS computation will be iterative, where the RCS estimates for each basic block gets updated in every iteration. This is continued until a (least) fixed-point is reached. Convergence to a fixed point is guaranteed because the RCS estimates must monotonically increase for the fixed-point iterations to continue, and the total set of cache states is finite. After the fixed point is reached, we set $RCS(\sigma) = RCS_{end(\sigma)}^{OUT}$, where $end(\sigma)$ is the sink node in the control flow graph $\tau(\sigma)$.

The computation of $LCS(\sigma)$ is similar to computing $RCS(\sigma)$. We set $LCS(\sigma)$ to $LCS_{start(\sigma)}^{IN}$ where $start(\sigma) \in \mathcal{B}_\sigma$ is the start node 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.

3.1 Computing the WCET of a Single Event

We can now use the notion of cache states to compute the WCET of a single event σ (while considering the events executing prior to σ). We first define two operations on cache states, namely merge and equality.

We define the operation $\oplus$ is over memory blocks as:

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

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.

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

$$\begin{aligned} &NSTATE(\langle \sigma_1, ..., \sigma_{N-1} \rangle, cs) = \\ &\quad cs \oplus RCS(\sigma_1) \oplus ... \oplus RCS(\sigma_{N-1}) \end{aligned}$$

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

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

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 $WCET(\sigma_N, cs')$, defined as follows.

$$\begin{aligned} WCET(\sigma_N, cs') &= WCET(\sigma_N, \perp) - \\ &\qquad penalty \cdot \min \{|v| \,:\, v \in useful(\sigma_N, cs')\} \\ cs' &= NSTATE(\langle \sigma_1, \cdots, \sigma_{N-1} \rangle, cs) \end{aligned}$$

Here $penalty$ is the cache miss penalty associated with any memory block and $|v|$ denotes the number of 1s in the boolean vector v, i.e. $|v| = \sum_{i=0}^{n-1} v[i]$.

4 Timing Analysis of Event Streams

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

Recall from Section 2 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 3, 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 $D'(\psi) = WCET(\sigma, cs_1)$.

Again, recall from Section 2 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 next show how to obtain this 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 a parameter n, we can efficiently compute $\gamma(k)$ for all $1 \leq k \leq n$ by traversing the graph, and storing/updating the maximum weight path of length k ending at x for all nodes x in the graph. Details are omitted.

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.

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 3).

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 2) 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 $\alpha(\Delta)$.

5 Implementation Issues

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). (i) Partially constructing the transition system $\mathcal{T}'$. (ii) Instead of computing $\gamma(k)$ for all values of k, exploit the fact that $\gamma(k)$ becomes periodic beyond a certain value of k. (iii) 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.

5.1 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 may not be able to distinguish between different cache states, and

- A cache state can be reached with various event histories.

Our partial construction of $\mathcal{T}'$ is based on these two observations. In the full construction of $\mathcal{T}'$, 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 3; 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 $\mathcal{T}'(U)$ (refer Algorithm 1). For this purpose, we use the algorithm for constructing $\mathcal{T}'$ but with two important modifications. First of all, when we construct the destination states for a state (s, cs, seq) for event σ, apart from applying $NSTATE$ on cs, we also need to define $NSTATE$ on seq. Since the sequence associated with a state captures the last U events, we define the following. Note that $\circ$ denotes concatenation, and $seq = \langle \sigma_1, \sigma_2, \ldots, \sigma_U \rangle$.

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

Secondly, the check for whether a state (s', cs', seq') exists in S'' (see Algorithm 1) is done differently. The logical disjunction in this membership check performs two kinds of state merging. The two sources of state merging mentioned below exploit our two main observations about the cache state evolution.

- Two states (s', cs', seq') and (s', cs'', seq') are merged. This is the main source of size reduction in the construction of $\mathcal{T}'(U)$ since we are merging two states of $\mathcal{T}'$.
- Two states (s', cs', seq') and (s', cs', seq'') are merged. This ensures that the state space size of $\mathcal{T}'(U)$ is guaranteed to be bounded by the state space size of $\mathcal{T}'$.

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 $WCET(\sigma, cs)$.

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

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 why, 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.

5.2 Computing γ

Recall from Section 4 that $\gamma(k)$ is the weight of the maximum-weight path of length k in the transition system $\mathcal{T}' = (S', S'_0, D', \Psi')$. Our computation of the maximum delay and backlog experienced by a stream requires

Algorithm 1 Constructing the transition system $\mathcal{T}'(U)$

```
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
      $cs' \leftarrow NSTATE(\sigma, cs)$ ;
      $seq' \leftarrow NSTATE(\sigma, U, seq)$ ;
      if $(\langle s', cs', \_ \rangle \notin S'') \vee (\langle s', \_, seq' \rangle \notin S'')$ 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
```

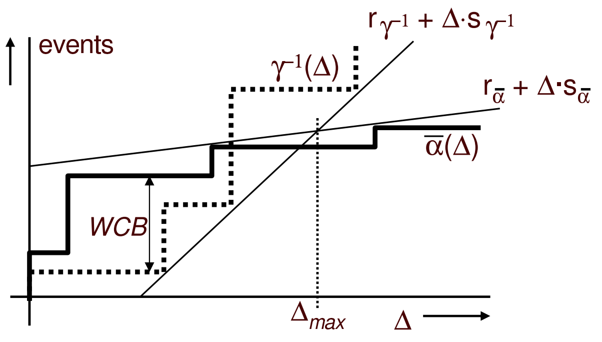
the computation of $\gamma(k)$ for $k = 1, \ldots, n$, where the value of n would depend on the range of Δ over which we need to iterate. Here, we would like to point out that it is sufficient to compute $\gamma(k)$ for $k = 1, \ldots, n$ for some $n = n_0$, and for all values of k larger than n_0 it would be possible to determine the value of $\gamma(k)$ without traversing the cache annotated transition system $\mathcal{T}'$. 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 $\mathcal{T}'$.

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 [4] and [7]). 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 [7]).

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:

$$\gamma(k) = \gamma((n_0 - p) + (k - n_0) \bmod p) + \lfloor \frac{k - n_0 + p}{p} \rfloor q \quad (1)$$

The value of n_0 can be determined by traversing $\mathcal{T}'$ and



events
$r_{\gamma^{1}} + \Delta \cdot s_{\gamma^{1}}$
$\gamma^{1}(\Delta)$
$r_{\bar{\alpha}} + \Delta \cdot s_{\bar{\alpha}}$
$\bar{\alpha}(\Delta)$
WCB
Δ_{max}
Δ

Figure 3. Computing $\Delta_{\max}$ from the affine bounds on γ^{-1} and $\bar{\alpha}$.

computing $\gamma(k)$ for all $1 \le k \le n$ where n is sufficiently large. We then test for periodicity. Towards this, we test if Eqn. 1 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. 1 holds for all $n_0 \le k \le n_0 + p - 1$.

5.3 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:

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

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:

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

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: $\alpha(\Delta) \le r_\alpha + \Delta \cdot s_\alpha,\ \forall \Delta \in \mathbb{R}^{\ge 0}$
$\gamma^{-1}(\Delta) \ge r_{\gamma^{-1}} + \Delta \cdot s_{\gamma^{-1}},\ \forall \Delta \in \mathbb{R}^{\ge 0}$ where,
$r_\alpha = r_\gamma + r_{\bar{\alpha}} s_\gamma,\ s_\alpha = s_{\bar{\alpha}} s_\gamma,\ r_{\gamma^{-1}} = -\dfrac{r_\gamma}{s_\gamma}$ and $s_{\gamma^{-1}} = \dfrac{1}{s_\gamma}$

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 3). Such a $\Delta_{\max}$ is therefore given by:

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

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

Table 1. Experimental results: Task WCET under different contexts.

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 WCD and WCB as follows:

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

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 4.

6 A Case Study

Our prototype implementation of the timing analysis framework consists of three parts. The first part consists of a *cache state analyzer*, which was implemented in C. It involves the LCS/RCS computation elaborated in Section 3. The second part is the construction of the cache annotation transition system. We use the improved method (Algorithm 1 in Section 5) which merges cache states to prevent blow-up of the transition system. The final step involves integrating the analysis of event streams (results derived in Section 4) with the cache annotated transition system to obtain tight delay/buffer size estimates. This part is implemented in Mathematica [11]. 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.

We now present a case study to illustrate how the estimated timing properties of a streaming 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. 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

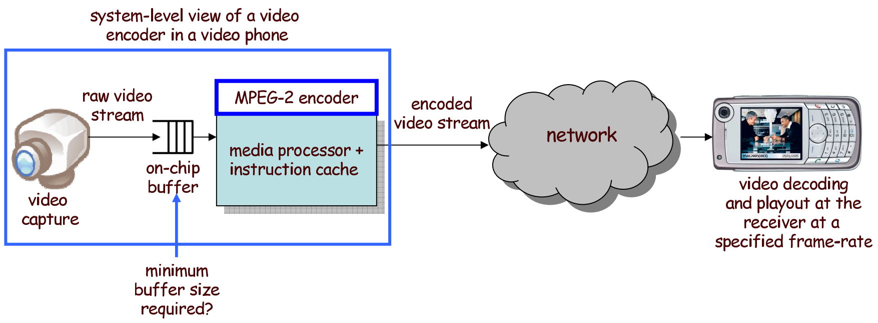

system-level view of a video encoder in a video phone
raw video stream
MPEG-2 encoder
media processor + instruction cache
on-chip buffer
video capture
minimum buffer size required?
encoded video stream
network
video decoding and playout at the receiver at a specified frame-rate

Figure 4. Application scenario: MPEG-2 encoder in a video phone.

per second, which in turn determines the sampling rate of the camera capturing the video. Our setup is shown in Figure 4. 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 consisting of five different *tasks*. These are *forward search* (FS), *discrete cosine transform* (DCT), *quantization* (Q), *inverse quantization* (IQ), and *inverse discrete cosine transform* (IDCT). The layout of these tasks in the memory is shown in Figure 5(a). We consider a direct-mapped instruction cache with 64 cache lines and 64 bytes block size. The incoming raw bitstream 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 5(b). This transition system is determined by the implementation of the encoder application. We note here that the MPEG-2 standard does not prescribe any particular encoder implementation. The transition system we derive here, and the patterns it attempts to compress, is taken from earlier works on timing analysis of event streams [20].

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 following table:

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 Table 1. These numbers were obtained with an instruction cache miss penalty of 100 cycles. As a sequence of macroblocks gets processed (or encoded), different tasks get executed following the pattern given by the transition system in Figure 5(b). Note that for any two macroblocks belonging to different frame types, there is a significant overlap between the tasks that get executed.

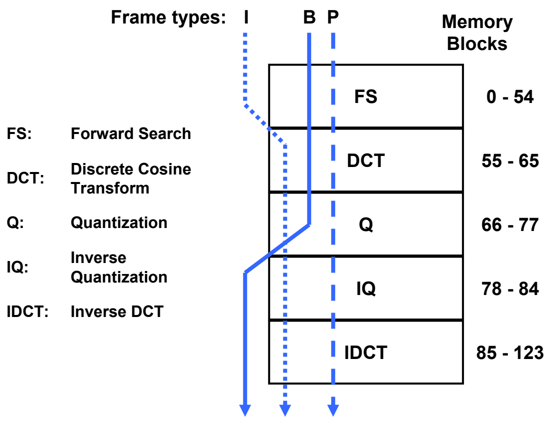
The results obtained from analyzing this setup using our proposed framework are shown in Table 2. These results were obtained with the processor frequency set to 105 MHz and an instruction cache penalty of 100 cycles. From this table, it may be noted that modeling the effects of the instruction cache leads to substantially tighter estimates of both – the on-chip buffer size and the maximum delay. The buffer size estimate reduces by 33% and the delay estimate reduces by 36%. 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 wide variety of streaming applications.

7 Concluding Remarks

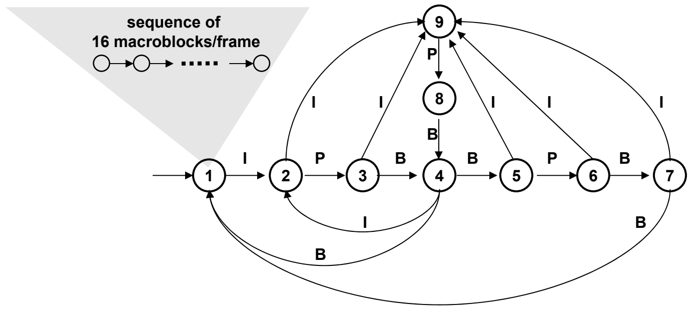
Currently, we are in the process of integrating this framework into a design space exploration tool and evaluating it with large applications and cache configurations. In contrast to the prototype implementation reported in this paper, we will replace the Mathematica code with an equivalent implementation in C and integrate it with our *cache state analyzer*.

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. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.



(a)



(b)

Figure 5. (a) The MPEG-2 encoder's code layout in the memory and the sequence of tasks executed for I,B,P — the three different frame types, and (b) Transition system $\mathcal{T}$ specifying the possible frame patterns according to which a raw video stream is encoded.

	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

Table 2. Experimental Results: Delay and Buffer Size estimation for MPEG encoder application.

[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] 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.

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

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

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

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

[9] X. Li, T. Mitra, and A. Roychoudhury. Modeling control speculation for timing analysis. *Real-time Systems*, 29(1), 2005.

[10] 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.

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

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

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

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

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

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

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

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

[19] 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.

[20] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characterization of event streams in analysis of hard real-time applications. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 450–461, 2004.