Timing Analysis of Concurrent Programs

Robert Mittermayr and Johann Blieberger

TU Vienna, Institute of Computer-Aided Automation
Treitlstr. 1–3, 1040 Vienna, Austria
{robert,blieb}@auto.tuwien.ac.at

Abstract

Worst-case execution time analysis of multi-threaded software is still a challenge. This comes mainly from the fact that the number of thread interleavings grows exponentially in the number of threads and that synchronization has to be taken into account. In particular, a suitable graph based model has been missing. The idea that thread interleavings can be studied with a matrix calculus is a novel approach in this research area. Our sparse matrix representations of the program are manipulated using Kronecker algebra. The resulting graph represents the multi-threaded program and plays a similar role for concurrent systems as control flow graphs do for sequential programs. Thus a suitable graph model for timing analysis of multi-threaded software has been set up. Due to synchronization it turns out that often only very small parts of the resulting graph are actually needed, whereas the rest is unreachable. A lazy implementation of the matrix operations ensures that the unreachable parts are never calculated. This speeds up processing significantly and shows that our approach is very promising.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.3.2 Program analysis

Keywords and phrases Worst-case execution time analysis (WCET), Concurrency, Thread Synchronization, Kronecker Algebra, Program Analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2012.59

1 Introduction

Since concurrent programs may contain blocking because of synchronization between threads, the terms execution time and worst-case execution time (WCET) do not apply directly to concurrent systems. Anyway, we stick to the term WCET for concurrent systems. The reader, however, has to be aware of the fact that the WCET includes blocking time.

With the advent of multi-core processors scientific and industrial interest focuses on analysis and verification of multi-threaded applications. The scientific challenge comes from the fact that the number of thread interleavings grows exponentially in a program's number of threads. All state-of-the-art methods suffer from this so-called *state explosion problem.*

The idea that thread interleavings of concurrent programs can be studied with a matrix calculus is novel in this research area. Our sparse matrix representations of the program are manipulated using a lazy implementation of Kronecker algebra. Similar to [3] we describe synchronization by Kronecker products and thread interleavings by Kronecker sums. The first goal is the generation of a data structure called *Concurrent Program Graph* (CPG) which describes all possible interleavings and incorporates synchronization while preserving completeness. CPGs play a similar role for concurrent systems as control flow graphs (CFGs) do for sequential programs.

In this paper CPGs are used to calculate the WCET of the underlying concurrent system.

In [12] it is shown that CPGs in general can be represented by sparse adjacency matrices. Thus the number of entries in the matrices is linear in their number of lines. In the worst-case the number of lines increases exponentially in the number of threads. The CPG, however, contains many nodes and edges unreachable from the entry node. If the program contains a lot of synchronization, only a very small part of the CPG is reachable. Our lazy implementation of the matrix operations computes only this part. This optimization speeds up processing significantly and shows that our approach is very promising.

The outline of our paper is as follows. In Section 2 Refined CFGs and Kronecker algebra are introduced. Our model of concurrency, its properties, and our lazy approach are presented in Section 3. Section 4 is devoted to WCET analysis of multi-threaded programs. In Section 5 we survey related work. Finally, we draw our conclusion in Section 6.

2 Preliminaries

In this paper we refer to both, a processor and a core, as a processor. Our computational model can be described as follows. We model concurrent programs by threads which use semaphores for synchronization. We assume that on each processor exactly one thread is running and each thread immediately executes its next statement if the thread is not blocked. Blocking may occur only in succession of semaphore calls.

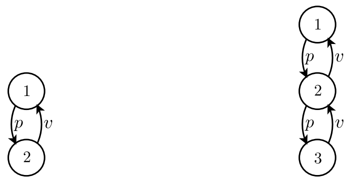
Threads and semaphores are represented by slightly adapted CFGs. *Edge Splitting* has to be applied to the edges containing semaphore calls. Each CFG is represented by an adjacency matrix. We assume that the edges of CFGs are labeled by elements of a semiring. Definitions and properties of the semiring can be found in [10, 12]. A prominent example for such semirings are regular expressions describing the behavior of finite state automata. Our semiring consists of a set of labels $\mathcal{L}$ which is defined by $\mathcal{L} = \mathcal{L}_{\mathrm{V}} \cup \mathcal{L}_{\mathrm{S}}$, where $\mathcal{L}_{\mathrm{V}}$ is the set of non-synchronization labels and $\mathcal{L}_{\mathrm{S}}$ is the set of labels representing semaphore calls ($\mathcal{L}_{\mathrm{V}}$ and $\mathcal{L}_{\mathrm{S}}$ are disjoint).

Usually two or more distinct thread CFGs refer to the same semaphore to model synchronization. The other labels are elements of $\mathcal{L}_{\mathrm{V}}$ and model ordinary program statements. The operations on the basic blocks are $\cdot, +$, and $*$ from a semiring (cf. [15]). Intuitively, $\cdot, +$, and $*$ model consecutive program parts, conditionals, and loops, respectively.

2.1 Refined Control Flow Graphs

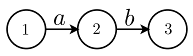
Usually CFG nodes represent basic blocks. Because our matrix calculus manipulates the edges we need to have basic blocks on the (incoming) edges. To keep things simple we refer to edges, their labels and the corresponding entries of the adjacency matrices synonymously. A basic block consists of multiple consecutive statements without jumps. For our purpose we need a finer granularity which we achieve by splitting edges. We apply it for semaphore calls (e.g. p_1 and v_1) and require that a semaphore call referred to as s_i has to be the only statement on the corresponding edge. Edge splitting maps a CFG edge e whose corresponding basic block contains k semaphore calls to a subgraph $\circ \xrightarrow{e_1} \circ \xrightarrow{s_1} \circ \xrightarrow{e_2} \circ \xrightarrow{s_2} \circ \cdots \circ \xrightarrow{e_k} \circ \xrightarrow{s_k} \circ \xrightarrow{e_{k+1}} \circ$, such that each s_i represents a single semaphore call, and e_i and e_{i+1} represent the consecutive parts before and after s_i, respectively ($1 \leq i \leq k$). Applying this procedure and edge splitting we result in a *Refined Control Flow Graph* (RCFG).

In Fig. 1a and 1b a binary and a counting semaphore are depicted. The latter allows two threads to enter at the same time. In a similar way it is possible to construct semaphores allowing n non-blocking p-calls ($n \in \mathbb{N}, n \geq 1$).

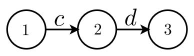


(a) Binary Semaphore **(b)** Counting Semaphore

Figure 1 Semaphores.



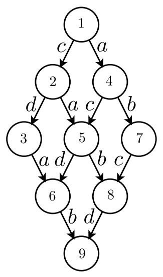
(a) C



(b) D

Interleavings
$a \cdot b \cdot c \cdot d$
$a \cdot c \cdot b \cdot d$
$a \cdot c \cdot d \cdot b$
$c \cdot a \cdot b \cdot d$
$c \cdot a \cdot d \cdot b$
$c \cdot d \cdot a \cdot b$

(c) Interleavings



(d) $C \oplus D$

Figure 2 A Simple Example.

2.2 Modeling Synchronization and Interleavings

Kronecker product and Kronecker sum form Kronecker algebra. In the following we define both operations. Proofs, additional properties, and examples can be found in [1, 5, 6, 12]. From now on we use matrices out of $\mathcal{M} = \{M = (m_{i,j}) \mid m_{i,j} \in \mathcal{L}\}$ only.

▶ **Definition 1** (Kronecker product)**.** Given a m-by-n matrix A and a p-by-q matrix B, their *Kronecker product* denoted by $A \otimes B$ is a mp-by-nq block matrix defined by

$$A \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}.$$

The Kronecker product is also being referred to as *Zehfuss product.* Kronecker product allows to model synchronization (cf. [3, 12, 13]).

▶ **Definition 2** (Kronecker sum)**.** Given a matrix A of order[1] m and matrix B of order n, their *Kronecker sum* denoted by $A \oplus B$ is a matrix of order mn defined by $A \oplus B = A \otimes I_n + I_m \otimes B$, where I_m and I_n denote identity matrices of order m and n, respectively.

Note that Kronecker sum calculates all possible interleavings (see e.g. [11] for a proof) even for general CFGs including conditionals and loops. The following example illustrates interleaving of threads and how Kronecker sum handles it.

▶ **Example 3.** Let the matrices $C = \begin{pmatrix} 0 & a & 0 \\ 0 & 0 & b \\ 0 & 0 & 0 \end{pmatrix}$ and $D = \begin{pmatrix} 0 & c & 0 \\ 0 & 0 & d \\ 0 & 0 & 0 \end{pmatrix}$. The CFGs of matrices C and D are shown in Fig. 2a and Fig. 2b, respectively. The regular expressions

[1] A k-by-k matrix is known as square matrix of order k.

associated with the CFGs are $a \cdot b$ and $c \cdot d$. All possible interleavings by executing C and D in an interleavings semantics are shown in Fig. 2c. In Fig. 2d the graph represented by the adjacency matrix $C \oplus D$ is depicted. It is easy to see that all possible interleavings are generated correctly. It is worth noting that $\oplus$ provides correct results even if the operands contain branches and loops.

3 Concurrent Program Graphs

Our system model consists of a finite number of threads and semaphores which are represented by RCFGs. The RCFGs are stored in form of adjacency matrices. The matrices have entries which are referred to as labels $l \in \mathcal{L}$ as defined in Sect. 2.

Formally, the system model consists of the tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where $\mathcal{T}$ is the set of RCFG adjacency matrices describing threads, $\mathcal{S}$ refers to the set of RCFG adjacency matrices describing semaphores, and the labels in $T \in \mathcal{T}$ and $S \in \mathcal{S}$ are elements of $\mathcal{L}$ and $\mathcal{L}_{\mathrm{S}}$, respectively. The matrices are manipulated by using Kronecker algebra.

A *Concurrent Program Graph* (CPG) is a graph $C = \langle V, E, n_e \rangle$ with a set of nodes V, a set of directed edges $E \subseteq V \times V$, and a so-called *entry* node $n_e \in V$. The sets V and E are constructed out of the elements of $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$. Details on how we generate the sets V and E follow below. Similar to RCFGs the edges of CPGs are labeled by $l \in \mathcal{L}$.

3.1 Generating a Concurrent Program's Matrix

Let $T^{(i)} \in \mathcal{T}$ and $S^{(i)} \in \mathcal{S}$ refer to the matrices representing thread i and semaphore i, respectively. According to Fig. 1a we have for binary semaphore i the adjacency matrix $S^{(i)} = \begin{pmatrix} 0 & p_i \\ v_i & 0 \end{pmatrix}$ of order two. We obtain the matrix T representing k interleaved threads and the matrix S representing r interleaved semaphores by

$$T = \bigoplus_{i=1}^{k} T^{(i)}, \text{ where } T^{(i)} \in \mathcal{T} \text{ and } S = \bigoplus_{i=1}^{r} S^{(i)}, \text{ where } S^{(i)} \in \mathcal{S}.$$

Note that the associativity properties (cf. [12]) of the operations $\otimes$ and $\oplus$ imply that the corresponding n-fold versions are well defined. In the following we define the Selective Kronecker product which we denote by $\oslash_L$. This operator synchronizes only labels identical in the two input matrices.

▶ **Definition 4** (Selective Kronecker product). Given two matrices A and B we call $A \oslash_L B$ their Selective Kronecker product. For all $l \in L \subseteq \mathcal{L}$ let $A \oslash_L B = (a_{i,j}) \oslash_L (b_{p,q}) = (c_{i \cdot p, j \cdot q})$, where

$$c_{i \cdot p, j \cdot q} = \begin{cases} l & \text{if } a_{i,j} = b_{p,q} = l,\ l \in L, \\ 0 & \text{otherwise.} \end{cases}$$

▶ **Definition 5** (Filtered Matrix). We call M_L a *Filtered Matrix* and define it as a matrix of order $o(M)$ containing entries $l \in L \subseteq \mathcal{L}$ of $M = (m_{i,j})$ and zeros elsewhere:

$$M_L = (m_{L;i,j}), \text{ where } m_{L;i,j} = \begin{cases} l & \text{if } m_{i,j} = l,\ l \in L, \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix representing program $\mathcal{P}$ is referred to as P. In [12] it is proved that P can be computed efficiently by

$$P = T \oslash_{\mathcal{L}_{\mathrm{S}}} S + T_{\mathcal{L}_{\mathrm{V}}} \otimes I_{o(S)}.$$

In addition, it is shown in [12] that the resulting CPG has at most n^k nodes and at most $2k\,n^k$ edges, if k is the number of threads and each thread has n nodes in its RCFG. Hence CPGs have a sparse adjacency matrix, i.e., $|E| = O\big(|V|\big)$. Thus memory saving data structures and efficient algorithms suggest themselves. In the worst-case, however, the number of CPG nodes increases exponentially in k.

3.2 Lazy Implementation of Kronecker Algebra

In general, a CPG contains unreachable parts if a concurrent program contains synchronization (cf. [12]). If a program contains a lot of synchronization, the reachable parts may be very small. This observation motivates the lazy implementation described in this subsection. In the following we denote the subgraph of a CPG, whose nodes are reachable from the entry node, by RCPG. An empirical analysis of our approach showed that the runtime complexity of generating a RCPG is linear in the number of RCPG nodes [12].

The reasons why parts of the CPG are unreachable can be summarized as follows: Kronecker product limits the number of possible paths such that the p- and v-operations are present in correct p-v-pairs in the RCPG. In contrast $T = \bigoplus_{i=1}^{k} T_i$ contains all possible paths even those containing semantically wrong uses of the semaphore operations.

Choosing a lazy implementation (cf. [9]) for the matrix operations ensures that, when extracting the reachable parts of the underlying graph, the overall effort is reduced to exactly these parts. By starting from the RCPG's entry node and calculating all reachable successor nodes our lazy implementation [12] exactly does this. Thus, for example, if the resulting RCPG's size is linear in terms of the involved threads, only linear effort will be necessary to generate the RCPG.

Our implementation distinguishes between two kind of matrices: Sparse matrices are used for representing threads and semaphores. Lazy matrices are employed for representing all the other matrices, i.e., those resulting from the operations of Kronecker algebra. Besides the employed operation, a lazy matrix simply keeps track of its operands. Whenever an entry of a lazy matrix is retrieved, depending on the operation recorded in the lazy matrix, entries of the operands are retrieved and the recorded operation is performed on these entries to calculate the result. In the course of this computation, even the successors of nodes are calculated lazily. Retrieving entries of operands is done recursively if the operands are again lazy matrices, or is done by retrieving the entries from the sparse matrices, where the actual data resides. The lazy implementation has proven to be very space and time efficient.

4 Worst-Case Execution Time Analysis on RCPGs

In order to calculate the WCET of a concurrent program we apply a dataflow based approach introduced in [2]. Dataflow equations are set up and solved according to [14]. Details can be found in [2, 14].

Each node of the RCPG is assigned a dataflow variable and a dataflow equation is set up based on the predecessors of the RCPG node. A dataflow variable is represented by a vector. Each component of the vector reflects a processor and is used to calculate the WCET of the corresponding thread. Recall that only a single thread is allocated to a processor.

Synchronizing nodes, introduced below, are nodes where blocking occurs. These nodes have an incoming edge labeled by a semaphore v-operation, an outgoing edge labeled by a p-operation of the same semaphore, and these edges are part of different threads. In this case the thread with the p-operation has to wait until the other thread's v-operation is finished.

Let the vector $\mathfrak{X} = (X_1, \ldots, X_\ell, \ldots, X_p)^\intercal$. We write $\mathfrak{X}^{(\ell)} = X_\ell$ to denote the ℓth component of vector $\mathfrak{X}$.

▶ **Definition 1.** Let $\mathfrak{X} = (X_1, \ldots, X_p)^\intercal$ and $\mathfrak{Y} = (Y_1, \ldots, Y_p)^\intercal$. Then we define

$$\max(\mathfrak{X}, \mathfrak{Y}) := (\max(X_1, Y_1), \ldots, \max(X_p, Y_p))^\intercal.$$

▶ **Definition 2.** A *synchronizing node* is a RCPG node s such that
- there exists an edge $e_{in} = (i, s)$ with label v_k and
- there exists an edge $e_{out} = (s, j)$ with label p_k,

where k denotes the same semaphore and e_{in} and e_{out} are mapped to different processors, i.e., $\mathfrak{P}(e_{in}) \neq \mathfrak{P}(e_{out})$.

▶ **Definition 3** (Setting up dataflow equations)**.** If n is a non-synchronizing node, then

$$\mathfrak{X}_n = \max_{k \in \mathrm{Pred}(n)} (\mathfrak{X}_k + \mathfrak{t}(k \to n)),$$

where the ℓth component of vector $\mathfrak{t}(k \to n)$ is the time assigned to edge $k \to n$ and edge $k \to n$ is mapped to processor ℓ. The other components of $\mathfrak{t}(k \to n)$ are zero. The set of predecessor nodes of node n is referred to as $\mathrm{Pred}(n)$.

Let s be a synchronizing node. In addition, let π_i and π_j be the processors which the edges $i \to s$ and $s \to j$ are mapped to, i.e, $\pi_i = \mathfrak{P}(i \to s)$ and $\pi_j = \mathfrak{P}(s \to j)$. Then for $\ell \neq \pi_j$

$$\mathfrak{X}_s^{(\ell)} = \max_{k \in \mathrm{Pred}(s)} \left(\mathfrak{X}_k^{(\ell)} + \mathfrak{t}(k \to s)^{(\ell)} \right)$$

and

$$\mathfrak{X}_s^{(\pi_j)} = \max \left(\mathfrak{X}_i^{(\pi_i)} + \mathfrak{t}(i \to s)^{(\pi_i)}, \max_{k : \mathfrak{P}(k \to s) = \pi_j} \left(\mathfrak{X}_k^{(\pi_j)} + \mathfrak{t}(k \to s)^{(\pi_j)} \right) \right)$$

where the first term considers the incoming v-edge and the second term takes into account all incoming edges of the blocking thread running on processor π_j.

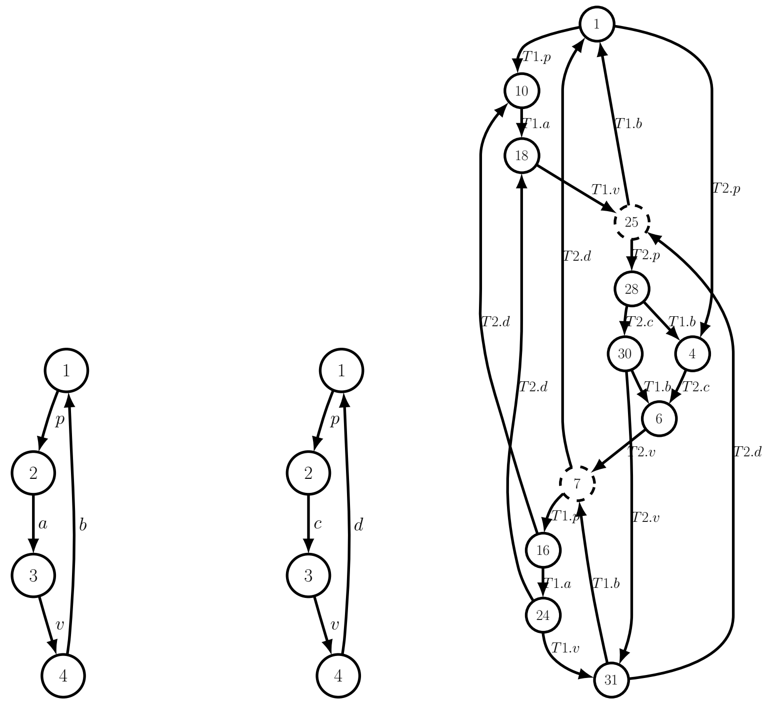
The system of dataflow equations can be solved by applying an algorithm presented in [14]. It relies on two operations: inserting one equation into another and solving recursions by so-called loop breaking. The order of these operations is completely determined by the DJ graph introduced in [14].

In contrast to [2] where CFGs are studied, RCPGs contain several copies of basic blocks in different places. Thus, during a loop breaking operation the number of loop iterations cannot be determined immediately. Instead we postpone the assigning of loop iterations and indicate this by "$*$" which denotes a number of loop iterations to be assigned later.

After the system of dataflow equations has been solved, we distribute the known number of loop iterations among all terms labeled by "$*$" such that the timing values achieve their maxima.

Example

We study an example consisting of two threads $T1$ and $T2$. Their RCFGs are depicted in Figures 3a and 3b, respectively. The behaviors of $T1$ and $T2$ are $(p \cdot a \cdot v \cdot b)^*$ and $(p \cdot c \cdot v \cdot d)^*$, respectively. The RCPG of the T1-T2-system together with a simple binary semaphore (depicted in Fig. 1a) is shown in Figure 3c. Note that the node numbers are generated by our implementation. Missing node numbers refer to unreachable nodes. The dashed nodes 7 and 25 are the only synchronizing nodes.



(a) RCFG of thread T1 **(b)** RCFG of thread T2 **(c)** RCPG of example program

Figure 3 RCFGs of threads T1 and T2 and the resulting RCPG.

According to Definition 3 the following equations are set up. For simplicity we do not distinguish between edge labels and the execution time of the corresponding basic blocks, i.e., the execution time of a basic block x is denoted by x.

$$\mathfrak{X}_1 = \max\left(\mathfrak{X}_7 + \begin{pmatrix}0\\d\end{pmatrix}, \mathfrak{X}_{25} + \begin{pmatrix}b\\0\end{pmatrix}\right), \qquad \mathfrak{X}_{10} = \max\left(\mathfrak{X}_1 + \begin{pmatrix}p\\0\end{pmatrix}, \mathfrak{X}_{16} + \begin{pmatrix}0\\d\end{pmatrix}\right)$$

$$\mathfrak{X}_{18} = \max\left(\mathfrak{X}_{10} + \begin{pmatrix}a\\0\end{pmatrix}, \mathfrak{X}_{24} + \begin{pmatrix}0\\d\end{pmatrix}\right), \qquad \mathfrak{X}_{25} = \begin{pmatrix}\mathfrak{X}_{18}^{(1)} + v\\ \max\left(\mathfrak{X}_{18}^{(1)} + v, \mathfrak{X}_{31}^{(2)} + d\right)\end{pmatrix}$$

$$\mathfrak{X}_{28} = \mathfrak{X}_{25} + \begin{pmatrix}0\\p\end{pmatrix}, \qquad \mathfrak{X}_{30} = \mathfrak{X}_{28} + \begin{pmatrix}0\\c\end{pmatrix}$$

$$\mathfrak{X}_4 = \max\left(\mathfrak{X}_1 + \begin{pmatrix}0\\p\end{pmatrix}, \mathfrak{X}_{28} + \begin{pmatrix}b\\0\end{pmatrix}\right), \qquad \mathfrak{X}_6 = \max\left(\mathfrak{X}_4 + \begin{pmatrix}0\\c\end{pmatrix}, \mathfrak{X}_{30} + \begin{pmatrix}b\\0\end{pmatrix}\right)$$

$$\mathfrak{X}_7 = \begin{pmatrix}\max\left(\mathfrak{X}_6^{(2)} + v, \mathfrak{X}_{31}^{(1)} + b\right)\\ \mathfrak{X}_6^{(2)} + v\end{pmatrix}, \qquad \mathfrak{X}_{16} = \mathfrak{X}_7 + \begin{pmatrix}p\\0\end{pmatrix}$$

$$\mathfrak{X}_{24} = \mathfrak{X}_{16} + \begin{pmatrix}a\\0\end{pmatrix}, \qquad \mathfrak{X}_{31} = \max\left(\mathfrak{X}_{24} + \begin{pmatrix}v\\0\end{pmatrix}, \mathfrak{X}_{30} + \begin{pmatrix}0\\v\end{pmatrix}\right)$$

We solve the above equations by using the DJ-graph [14] of our RCPG. For a concise presentation we use $\alpha = p + a + v$, $\gamma = p + c + v$, $T_1 = \alpha + b$, $T_2 = \gamma + d$, $M_2 = \alpha + \gamma + T_2^*$, and $M_1 = \max(T_1, M_2)$. We perform the following insertions and loop breaking operations:

$$24 \to 18 : \mathfrak{X}_{18} = \max\left(\mathfrak{X}_{10} + \begin{pmatrix}a\\0\end{pmatrix}, \mathfrak{X}_{16} + \begin{pmatrix}a\\d\end{pmatrix}\right)$$

$$
\begin{aligned}
24,30 \to 31 : \mathfrak{X}_{31} &= \max\left(\mathfrak{X}_{16} + \begin{pmatrix} a+v \\ 0 \end{pmatrix}, \mathfrak{X}_{28} + \begin{pmatrix} 0 \\ v+c \end{pmatrix}\right) \\
30 \to 6 : \mathfrak{X}_{6} &= \max\left(\mathfrak{X}_{4} + \begin{pmatrix} 0 \\ c \end{pmatrix}, \mathfrak{X}_{28} + \begin{pmatrix} b \\ c \end{pmatrix}\right) \\
16 \to 18 : \mathfrak{X}_{18} &= \max\left(\mathfrak{X}_{10} + \begin{pmatrix} a \\ 0 \end{pmatrix}, \mathfrak{X}_{7} + \begin{pmatrix} a+p \\ d \end{pmatrix}\right) \\
16 \to 31 : \mathfrak{X}_{31} &= \max\left(\mathfrak{X}_{7} + \begin{pmatrix} \alpha \\ 0 \end{pmatrix}, \mathfrak{X}_{28} + \begin{pmatrix} 0 \\ v+c \end{pmatrix}\right) \\
16 \to 10 : \mathfrak{X}_{10} &= \max\left(\mathfrak{X}_{1} + \begin{pmatrix} p \\ 0 \end{pmatrix}, \mathfrak{X}_{7} + \begin{pmatrix} p \\ d \end{pmatrix}\right) \\
28 \to 31 : \mathfrak{X}_{31} &= \max\left(\mathfrak{X}_{7} + \begin{pmatrix} \alpha \\ 0 \end{pmatrix}, \mathfrak{X}_{25} + \begin{pmatrix} 0 \\ \gamma \end{pmatrix}\right) \\
28 \to 6 : \mathfrak{X}_{6} &= \max\left(\mathfrak{X}_{4} + \begin{pmatrix} 0 \\ c \end{pmatrix}, \mathfrak{X}_{25} + \begin{pmatrix} b \\ c+p \end{pmatrix}\right) \\
28 \to 4 : \mathfrak{X}_{4} &= \max\left(\mathfrak{X}_{1} + \begin{pmatrix} 0 \\ p \end{pmatrix}, \mathfrak{X}_{25} + \begin{pmatrix} b \\ p \end{pmatrix}\right) \\
10 \to 18 : \mathfrak{X}_{18} &= \max\left(\mathfrak{X}_{1} + \begin{pmatrix} p+a \\ 0 \end{pmatrix}, \mathfrak{X}_{7} + \begin{pmatrix} p+a \\ 0 \end{pmatrix}\right) \\
18 \to 25 : \mathfrak{X}_{25} &= \begin{pmatrix} \max\left(\mathfrak{X}_1^{(1)}, \mathfrak{X}_7^{(1)}\right) + \alpha \\ \max\left(\max\left(\mathfrak{X}_1^{(1)}, \mathfrak{X}_7^{(1)}\right) + \alpha, \mathfrak{X}_{31}^{(2)} + d\right) \end{pmatrix} \\
25 \to 4 : \mathfrak{X}_{4} &= \max\left(\mathfrak{X}_{1} + \begin{pmatrix} 0 \\ p \end{pmatrix}, \begin{pmatrix} \max\left(\mathfrak{X}_1^{(1)}, \mathfrak{X}_7^{(1)}\right) + T_1 \\ \max\left(\max\left(\mathfrak{X}_1^{(1)}, \mathfrak{X}_7^{(1)}\right) + \alpha, \mathfrak{X}_{31}^{(2)} + d\right) + p \end{pmatrix}\right) \\
25 \to 31 : \mathfrak{X}_{31} &= \begin{pmatrix} \max\left(\mathfrak{X}_7^{(1)} + \alpha, \mathfrak{X}_1^{(1)} + \alpha\right) \\ \max\left(\mathfrak{X}_7^{(2)}, \mathfrak{X}_1^{(1)} + \alpha + \gamma, \mathfrak{X}_7^{(1)} + \alpha + \gamma, \mathfrak{X}_{31}^{(2)} + T_2\right) \end{pmatrix} \\
4,25 \to 6 : \mathfrak{X}_{6} &= \begin{pmatrix} \max\left(\mathfrak{X}_1^{(1)}, \mathfrak{X}_7^{(1)}\right) + T_1 \\ \max\left(\mathfrak{X}_1^{(2)} + p + c, \mathfrak{X}_1^{(1)} + \alpha + \gamma, \mathfrak{X}_7^{(1)} + \alpha + \gamma, X_{31}^{(2)} + d + p + c\right) \end{pmatrix} \\
6 \to 7 : \mathfrak{X}_{7} &= \begin{pmatrix} \max\left(\mathfrak{X}_1^{(2)} + \gamma, \mathfrak{X}_1^{(1)} + \alpha + \gamma, \mathfrak{X}_7^{(1)} + \alpha + \gamma, \mathfrak{X}_{31}^{(2)} + T_2, \mathfrak{X}_{31}^{(1)} + b\right) \\ \max\left(\mathfrak{X}_1^{(2)} + \gamma, \mathfrak{X}_1^{(1)} + \alpha + \gamma, \mathfrak{X}_7^{(1)} + \alpha + \gamma, \mathfrak{X}_{31}^{(2)} + T_2\right) \end{pmatrix} \\
31 \varnothing : \mathfrak{X}_{31} &= \begin{pmatrix} \max\left(\mathfrak{X}_7^{(1)} + \alpha, \mathfrak{X}_1^{(1)} + \alpha\right) \\ \max\left(\mathfrak{X}_7^{(2)}, \mathfrak{X}_1^{(1)} + \alpha + \gamma, \mathfrak{X}_7^{(1)} + \alpha + \gamma\right) + T_2^* \end{pmatrix} \\
31 \to 7 : \mathfrak{X}_{7} &= \begin{pmatrix} \max\left(\mathfrak{X}_1^{(2)} + \gamma, \mathfrak{X}_1^{(1)} + M_2, \mathfrak{X}_1^{(1)} + T_1, \mathfrak{X}_7^{(2)} + T_2^*, \mathfrak{X}_7^{(1)} + M_2, \mathfrak{X}_7^{(1)} + T_1\right) \\ \max\left(\mathfrak{X}_1^{(2)} + \gamma, \mathfrak{X}_1^{(1)} + M_2, \mathfrak{X}_7^{(2)} + T_2^*, \mathfrak{X}_7^{(1)} + M_2\right) \end{pmatrix} \\
7 \varnothing : \mathfrak{X}_{7} &= \begin{pmatrix} \max\left(\mathfrak{X}_1^{(2)} + \gamma, \mathfrak{X}_1^{(1)} + M_2, \mathfrak{X}_1^{(1)} + T_1\right) + M_1^* \\ \max\left(\mathfrak{X}_1^{(2)} + \gamma, \mathfrak{X}_1^{(1)} + M_2\right) + M_1^* + M_2 \end{pmatrix} \\
7 \to 31 : \mathfrak{X}_{31} &= \begin{pmatrix} \max\left(\mathfrak{X}_1^{(2)} + \gamma, \mathfrak{X}_1^{(1)} + M_2, \mathfrak{X}_1^{(1)} + T_1\right) + M_1^* + \alpha \\ \max\left(\mathfrak{X}_1^{(2)} + \gamma, \mathfrak{X}_1^{(1)} + M_2, \mathfrak{X}_1^{(1)} + T_1\right) + M_1^* + M_2 \end{pmatrix}
\end{aligned}
$$

$$7,31 \to 25 : \mathfrak{X}_{25} = \begin{pmatrix} \max\left(\mathfrak{X}_1^{(2)} + \gamma + M_1^*, \mathfrak{X}_1^{(1)} + M_2 + M_1^*, \mathfrak{X}_1^{(1)} + T_1 + M_1^*\right) + \alpha \\ \max\left(\mathfrak{X}_1^{(2)} + T_2, \mathfrak{X}_1^{(1)} + \alpha + T_2^*, \mathfrak{X}_1^{(1)} + T_1 + d\right) + M_1^* + M_2 \end{pmatrix}$$

$$7,25 \to 1 : \mathfrak{X}_1 = \begin{pmatrix} \max\left(\mathfrak{X}_1^{(2)} + \gamma + M_1^*, \mathfrak{X}_1^{(1)} + M_2 + M_1^*, \mathfrak{X}_1^{(1)} + T_1 + M_1^*\right) + T_1 \\ \max\left(\mathfrak{X}_1^{(2)} + T_2, \mathfrak{X}_1^{(1)} + \alpha + T_2^*, \mathfrak{X}_1^{(1)} + T_1 + d\right) + M_1^* + M_2 \end{pmatrix}$$

$$1 \not\circlearrowleft : \mathfrak{X}_1 = \begin{pmatrix} M_1^* + T_1^* \\ M_1^* + T_1^* + T_2^* + \alpha \end{pmatrix}$$

Finally we obtain the following formula for the WCET of our T1-T2-system

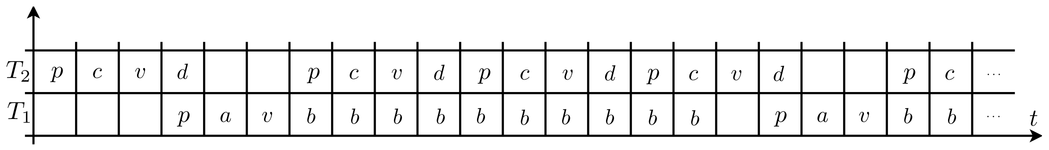
$$\text{WCET} = \max(\mathfrak{X}_1^{(1)}, \mathfrak{X}_1^{(2)}) = M_1^* + T_1^* + T_2^* + \alpha.$$

Assuming the execution times $p = v = a = c = d = 1$, $b = 10$, and the number of loop iterations of $T1$ and $T2$ to be r and s, respectively, we are left with distributing the number of loop iterations among the "$*$"-terms of our WCET formula. In this case we have $M_2(k) = \max(13, 6 + 4k)$ where the number 6 already includes one execution of a and c. Thus we write $M_2(k) = \max(13, 2 + 4k)$ and we have to choose k such that $M_2(k) = 2 + 4k$ and $M_2(i) = 13$ for $i < k$. This is a consequence of our system model presented in Section 2. We get $k = 3$, i.e., T_2 is executed three times while M_1 is executed once. Hence we obtain

$$\text{WCET} = \begin{cases} 14\left\lfloor \frac{s-1}{3} \right\rfloor + 13\left(r - \left\lfloor \frac{s-1}{3} \right\rfloor\right) + 3 & \text{if } r > \left\lfloor \frac{s-1}{3} \right\rfloor, \\ 14(r-1) + 4(s - 3(r-1)) + 3 & \text{if } r \le \left\lfloor \frac{s-1}{3} \right\rfloor. \end{cases}$$

A schedule of this case is shown in Fig. 4. It is easy to verify that the derived formula is a correct upper bound for the execution time of this $T1$-$T2$-system.

If we set $b = 1$ and $d = 10$, we get a similar result, but in this case M_2 is iterated once and M_1 three times where M_2 is started during the first iteration of M_1. During the second and third iteration of M_1, M_2 is still executing basic block d.



T_2
T_1
t
p c v d p c v d p c v d p c v d p c …
p a v b b b b b b b b b b p a v b b …

Figure 4 A Simple Schedule.

5 Related Work

Multiple data-flow-based WCET analysis frameworks are discussed in [2]. In this paper we adopt a dataflow approach and extend it in order to support concurrent programs.

Our Kronecker algebra based approach can be used for further analysis. In [12] we showed how to detect deadlocks.

In terms of how we generate a graph model for concurrent programs the closest work to ours was probably done by Buchholz and Kemper [3]. It differs from our work as follows. Our approach uses RCFGs and semaphores to model concurrent programs. Buchholz and Kemper worked on generating reachability sets in composed automata. In addition, we propose lazy calculation of matrix entries to optimize running time. Both approaches employ Kronecker algebra.

In [8] a method based on model checking of multi-core applications modeled as timed automata is investigated. The tool box UPPAAL is used and synchronization is modeled by using spinlock-like primitives.

Although not closely related we recognize the work done in the field of *stochastic automata networks* which is based on the work of Plateau [13] as related work. Basic operators are shared and some properties of Kronecker algebra were integrated into this paper.

6 Conclusion and Future Work

We established a framework for WCET analysis of concurrent systems based on Kronecker algebra. Thread synchronization is modeled by semaphores. Our graph representation of multi-threaded programs plays a similar role for concurrent systems as control flow graphs do for sequential programs. Thus a suitable graph model for timing analysis of multi-threaded software has been set up.

We consider optimizations like partial order reduction (cf. [4]) as future work. A standardized benchmark suite (similar to [7]) including concurrency would enable comparison of different approaches.

References

1 R. Bellman. *Introduction to Matrix Analysis.* Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 2nd edition, 1997.

2 J. Blieberger. Data-Flow Frameworks for Worst-Case Execution Time Analysis. *Real-Time Systems*, 22(3):183–227, 2002.

3 P. Buchholz and P. Kemper. Efficient Computation and Representation of Large Reachability Sets for Composed Automata. *Discrete Event Dyn. Systems*, 12(3):265–286, 2002.

4 E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

5 M. Davio. Kronecker Products and Shuffle Algebra. *IEEE Trans. Computers*, 30(2):116–125, 1981.

6 A. Graham. *Kronecker Products and Matrix Calculus with Applications.* Ellis Horwood Ltd., New York, 1981.

7 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proc. 10th International Workshop on Worst-Case Execution Time Analysis*, pages 136–146, 2010.

8 A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *Proc. 10th International Workshop on Worst-Case Execution Time Analysis*, pages 101–112, 2010.

9 P. Henderson and J. H. Morris, Jr. A Lazy Evaluator. In *3rd ACM Symposium on Principles of Programming Languages*, POPL '76, pages 95–103, January 1976.

10 W. Kuich and A. Salomaa. *Semirings, Automata, Languages.* Springer, 1986.

11 G. Küster. On the Hurwitz Product of Formal Power Series and Automata. *Theor. Comput. Sci.*, 83(2):261–273, 1991.

12 R. Mittermayr and J. Blieberger. Shared Memory Concurrent System Verification using Kronecker Algebra. Technical Report 183/1-155, Automation Systems Group, TU Vienna, http://arxiv.org/abs/1109.5522, Sept. 2011.

13 B. Plateau. On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms. In *ACM SIGMETRICS*, volume 13, pages 147–154, 1985.

14 V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. A New Framework for Elimination-Based Data Flow Analysis Using DJ Graphs. *ACM Trans. Program. Lang. Syst.*, 20(2):388–435, 1998.

15 R. E. Tarjan. A Unified Approach to Path Problems. *J. ACM*, 28(3):577–593, 1981.