Reduction techniques for synchronous dataflow graphs.

Reduction Techniques for Synchronous Dataflow Graphs

Marc Geilen
Eindhoven University of Technology
Department of Electrical Engineering
Den Dolech 2, 5600 MB, Eindhoven, The Netherlands
m.c.w.geilen@tue.nl

ABSTRACT

The Synchronous Dataflow (SDF) model of computation is popular for modelling the timing behaviour of real-time embedded hardware and software systems and applications. It is an essential ingredient of several automated design-flows and design-space exploration tools. The model can be analysed for throughput and latency properties. Although the SDF model is fairly simple, the analysis algorithms are often of high complexity and the models that need to be analysed may be fairly large. This paper introduces two graph transformations for reducing large SDF graphs into simpler, smaller ones that can be analysed more efficiently and give a conservative and often tight estimation of the timing of the original model and hence of the hard real-time system. We can make SDF based methods more efficient and prove that analyses that were done manually in an ad-hoc fashion in the past, can be done automatically and with guaranteed correctness. Additionally we introduce a novel conversion from SDF to Homogeneous SDF, a step applied in many analysis methods for SDF, which yields an up to 250X improvement on the number of actors, thus mitigating the problems with the size explosion observed in the traditional conversion.

Category 1.1 System specification, modeling, simulation, verification, and performance analysis

General Terms Algorithms, Performance, Languages, Theory

Keywords Synchronous Dataflow Graphs, model-based design, reduction techniques

1. INTRODUCTION

For many modern embedded applications, real-time constraints are important, while at the same time, the amount of resources that are spent on an application need to be minimised. Worst-case execution time models help to analyse the timing behaviour of an application, mapped on a specific platform with given resources and can serve as the basis for an automated design-flow. A popular modelling formalism for such timing analysis are Synchronous Dataflow (SDF) graphs [11]. SDF graphs consist of actors that periodically execute a function (sometimes extended with information capturing their execution time), while communicating tokens of information between each other. These actors can be used to model the algorithms, networks, arbiters, memories and other components of a system under study in a conservative fashion [3, 16, 13, 4, 15, 2]. SDF graphs with timing annotations moreover allow for off-line analysis or conservative estimation of their timing behaviour. In this way it can be verified at design time whether a particular design will be able to meet its deadlines at run-time. [1, 13, 15].

Although the SDF model is fairly simple, some of the actual analysis algorithms are of high complexity. Some require the transformation of the SDF graph into an equivalent homogeneous SDF graph (explained later). The size of the graph may however expand exponentially with this conversion [11, 15]. In combination with that, some of the modelling techniques that allow embedded systems to be analysed with SDF graphs, lead to graphs with large numbers of actors. However, these graphs often have a regular or almost regular structure. These observations have lead us to investigate firstly the possibilities of transforming such graphs into smaller and hence easier to analyse, equivalent or almost equivalent models that will still allow us to conclude that a system will meet its real-time deadlines and secondly to improve on the traditional algorithm to convert SDF graphs to homogeneous SDF graphs.

The former problem is illustrated by the (homogeneous) SDF graph depicted in Figure 1(a), which has a typical structure of e.g., the prefetching of data from a remote memory for some block based image processing application. It is clear that all actors Ai serve a similar role, as well as all actors Bi and their dependencies are fairly regular, except for the beginning and the end of for instance a video frame. Such behaviour is typical for processing of a finite amount of data such as an audio or video frame consisting of smaller blocks, where only start or end or borders have slightly different dependencies. The size of this type of graph is determined by the number of blocks in a frame and can be large. We argue that the behaviour of the graph can be modelled by a simpler graph like the one in Figure 1(b). How we arrive at this graph is explained in detail in Section 4.1. We show in this paper how this graph can be automatically derived and proved to be a conservative estimate of the worst-case execution time of the original graph.

The rest of the paper is organised as follows. Section 2 discusses related work. Section 3 provides preliminary definitions w.r.t. SDF graphs. The abstraction method is introduced in Section 4. In Section 5 we argue that the method is sound and gives a conservative estimate of the throughput of the original model. Section 6 presents a novel transformation from SDF to homogeneous SDF. Experimental results are described in Section 7 and Section 8 concludes.

2. RELATED WORK

SDF is a model of computation that is popular to study scheduling and performance of embedded systems and software. The model is introduced by Lee [11] and further studied in several other pub-

DAC'09, July 26-31, 2009, San Francisco, California, USA

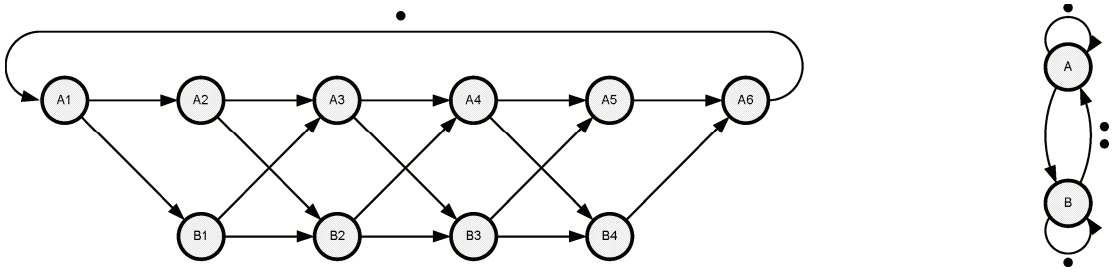

Figure 1: Example regular (a) and abstracted (b) SDF graph

lications, among which the following [4, 15, 1]. [4, 15] study the use of SDF for software synthesis and scheduling. [1] describes a more general mathematical framework called max-plus algebra in which SDF graphs can be described.

A recent trend is to not only describe the embedded software and software synthesis in terms of SDF graphs, but also the underlying hardware platform and multiprocessor systems-on-a-chip, in order to be able to derive guaranteed bounds on execution times and throughput for real-time applications [15, 3, 2, 13, 16].

SDF analysis has been used for throughput analysis [11, 15, 8], latency analysis [15, 9] or buffer sizing by heuristics [19] or exact analysis [18]. Complexity and efficiency of various approaches of throughput analysis based on translation from SDF to HSDF have been studied in [5]. Many of the analysis problems are coming within reach of exact analysis algorithms, although they may still fail on tough cases where heuristics are required.

To the best of our knowledge, the only conversion algorithm to convert SDF to HSDF is the traditional one given in [11, 15]. [10] considers the equivalence between the SDF graph and its corresponding HSDF in more detail. The algorithm in this paper is the first to provide an improved conversion algorithm, although the equivalence with the original graph is less tight.

3. PRELIMINARIES

We start with some definitions concerning SDF graphs, that will allow us to develop the abstraction and reduction mechanisms of this paper. Figure 2 shows an example of an SDF graph. Circles denote *actors*, which can repeatedly *fire*. This models for instance an amount of data processing. A firing consumes and produces at constant rates, *tokens* which are communicated over the edges (denoted by arrows) of the graph, which act as unbounded FIFO channels. Tokens represent data- or other causal dependencies between actor firings. Tokens are indicated by dots. The amount of tokens consumed and produced per firing is constant. Formally, an SDF graph is defined as follows.

DEFINITION 1. (SDF GRAPH) *An SDF graph (SDFG) is a pair (A, D) consisting of a set A of actors and a set $D \subseteq A \times A \times \mathbb{N}^3$ of dependency edges. A dependency (a, b, p, c, d) represents a dependency of actor b on actor a with production rate p of a and a consumption rate c of b and an initial delay of d tokens.*

Because of the constant consumption and production rates, the execution of an SDFG has a periodic behaviour. A graph is *consistent* [11], if there exists fixed numbers of firings for each of the actors such that execution the corresponding numbers of firings brings the graph back into its original state, i.e. with the same distribution of tokens. A vector denoting this number for each actor is called the *repetition vector*. The total number of firings in such an iteration depends on the rates of the edges and can therefore be exponential in terms of the number of actors of the graph. Because many algorithms need to deal with iterations of the graph, complexity of the analysis can sometimes be high.

A special kind of SDF graphs are *homogeneous SDFGs* (HSDFGs) where all production and consumption rates equal 1. In case of an HSDFG, the length of an iteration is equal to the number of actors of the graph. In literature, there is a classical conversion algorithm to go from an SDFG to an equivalent HSDFG [11, 15]. However, this conversion will always increase the number of actors to exactly the length of an iteration of the graph which, as mentioned before, can be exponential in terms of the size of the graph. In Section 6 we introduce a novel transformation, which yields a graph that is at most quadratic in the number of initial tokens of the graph. Next, we add timing information to SDF graphs. The execution time of an SDF graph (A, D) is a mapping $T : A \to \mathbb{N}$, that assigns to every actor $a \in A$ the time it takes to execute the actor once, the time that elapses between consumption of the input tokens and production of the output tokens.

DEFINITION 2. (TIMED SDF GRAPH) *A timed SDF graph is a triple (A, D, T) consisting of an SDF graph (A, D) and a corresponding execution time T.*

For lack of space we cannot formalise the semantics of timed SDF graphs in this paper. We assume standard *self-timed execution* [1, 4]. The performance of an SDF graph is characterised by the number of times actors can fire per unit of time, its *throughput* [11, 8].

To study graph reductions later on we make the following observation to compare two graphs. If a timed SDFG contains at least all the actors and all the dependency edges of another graph (and possibly more), its execution times are at least as long and it has not more initial tokens, then the throughput of the former graph is a lower bound for the throughput of the latter.

PROPOSITION 1. *If (A, D, T) and (B, E, U) are two timed SDF graphs with $A \subseteq B$, the execution times T and U are such that $T(a) \leq U(a)$ for all $a \in A$ and for every $(a, b, p, c, d) \in D$, there is some $(a, b, p, c, d') \in E$ such that $d' \leq d$, then the throughput of (A, D, T) is at least as high as the throughput of (B, E, U).*

The intuitive reason for this is that the actors are slower and there are more and stricter dependencies, which will slow the graph down. Proofs in the paper are omitted for space reasons, see [6].

4. ABSTRACTION

Important aspects of SDF graphs, such as their throughput or buffer sizes can be analysed in principle. However, some analyses have a high complexity [12, 15, 18]. In this section, we present an abstraction method to transform an SDF graph into a smaller SDF graph from which a conservative and often reasonably tight estimate of the behaviour of the original graph can be obtained.

4.1 Example

We return to the example of Figure 1(a). It shows an almost regular SDF graph. This type of graph may be obtained from modelling a memory pre-fetching mechanism [16]. We can make an abstract version of it by taking all the Ai actors together into one actor A and similarly put the Bi actors together in one actor B. This results in the abstract graph of Figure 1(b). Firings of the abstract actor model subsequently firings of $A1$, $A2$, ..., $A6$ and then back to $A1$. Note that because actor firing of the same actor can be concurrent in SDF, this does not automatically mean in general that the new graph is much slower than the original. With this connection between the original graph and the abstract graph in mind, we translate the dependency edges of the original graph into dependency edges of the abstract graph. The cycle of edges Ai to $A(i+1)$ and back to $A1$, can be captured by the self edge on A, forcing them to be sequential. Similar for the edges between Bi, although here the original graph does not have the edge back from $B4$ to $B1$. All edges Ai to Bi are captured by the singe edge A to B. The edges back from Bi to $A(i+2)$ result in the edge from B to A. It gets two initial tokens, because the edges connect B's with A's from a different step in the cycle of 6.

Assume that the execution times of the actors are as follows: $A1$, $A2$: 2 time units, $A3$ and $A4$: 5 time units and $A5$ and $A6$: 3 time units. $B1$ up to $B4$ all consume 4 time units. To be conservative, the abstract actor A then takes 5 time units and B takes 4, being the maximum of the firings they represent. It is straightforward to check that a single execution of the graph of Figure 1(a) takes 23 time units. The throughput is $\tau(a) = \frac{1}{23}$ for every actor $a \in A$. In general, for a graph with n copies of the Ai actor, the throughput is: $\frac{1}{5n-7}$. The throughput of the abstract graph is $\frac{1}{5}$ and hence it estimates the throughput of the original graph with n copies as $\frac{1}{5n}$. The estimate is conservative ($\frac{1}{5n-7} \geq \frac{1}{5n}$) and as n gets larger, the relative error made by the abstract graph decreases and it provides a better approximation of the throughput of the large graph.

4.2 The abstraction method

We give the precise definition of the transformation, illustrated by the example in Figure 2(a). The abstraction groups the Ai actors together and the Bi actors together. The result of the abstraction is then the graph depicted in Figure 2(b). The executions of the A actor model subsequent executions of $A1$, $A2$, $A3$, $A1$, etcetera. The executions of abstract actor B models executions of $B1$, $B2$ and some 'dummy' actor to account for the fact that the numbers of actors that have been abstracted are not equal.

The principle behind the construction is as follows. A group of actors *with identical firing rates* will be ordered and represented by a single new actor. If the groups of actors that are combined are not of equal size, then the abstract actor may have a number of 'dummy' firings that do not correspond to firings of the original actors. The firing time of the new, abstract actor is adapted to the slowest of the actors that it represents. We first define an abstraction. It tells us which actors are going to be grouped together and in what order.

DEFINITION 3. (ABSTRACTION) *An abstraction (α, I) of a consistent SDF graph (A, D) consists of a function $\alpha : A \to B$ that maps actors $a \in A$ to abstract actors $\alpha(a) \in B$. The function $I : A \to \mathbb{N}$ assigns to every actor $a \in A$ an index number. α and I are further such that:*

- *if $\alpha(a_1) = \alpha(a_2)$ then $I(a_1) \neq I(a_2)$ and $\gamma(a_1) = \gamma(a_2)$ where γ is the repetition vector of the graph.*
- *for every edge $(a, b, p, c, d) \in D$, $I(a) \leq I(b)$ or $d > 0$.*

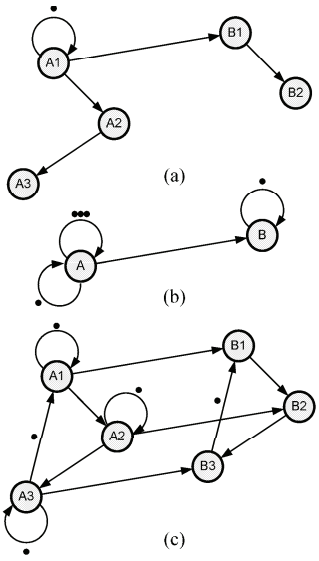

Figure 2: Abstracting and unfolding a graph

Given an SDF graph and an abstraction, we can determine the resulting smaller SDF graph. We consider here for clarity the case that the original graph is homogeneous, but the method can be extended to non-homogeneous graphs as well.

DEFINITION 4. (ABSTRACT GRAPH) *The abstract timed graph of (A, D, T) according to the abstraction (α, I) is the graph $(A, D, T)^{\alpha,I} = (A', D', T')$, where (with $N = \max_{a \in A} I(a)$)*

- $A' = \{\alpha(a) \mid a \in A\}$
- $D' = \{(\alpha(a_1), \alpha(a_2), p, c, I(a_2) - I(a_1) + Nd) \mid (a_1, a_2, p, c, d) \in D\}$
- $T'(b) = \max_{a \in A, \alpha(a)=b} T(a)$.

The new set of actors consists of all abstractions $\alpha(a)$ for the original actors $a \in A$. An edge is introduced for every edge in the original graph and the number of initial tokens is adapted to the number of initial tokens in the original graph (which must be multiplied by N, because firings of the new actor representing firings of the original actor occur only once every N firings) and the relative index ($I(a_2) - I(a_1)$) of the actors. Finally, the execution time is set to the maximum of the execution times of the actors mapped to it. Note that according to the definition, every dependency edge in the original graph results in an edge in the abstract graph. For a regular graph, suitable for the abstraction, many edges of the original graph are mapped onto the same edge of the new, reduced graph. Still, this might lead to having several edges between two actors in the abstract graph. Such a set of edges can always be pruned to only the one with the smallest number of initial tokens. E.g., in Figure 2, the abstract graph, the self-edge on actor A with three initial tokens is redundant because there is another one with only one token and it could have been removed. For the unfolding this would mean that the self-edges on the A actors disappear, without changing its throughput.

5. CONSERVATIVITY

In this section, we give a brief sketch of the proof that every behaviour of the original graph is mimicked by the new graph in a conservative fashion and hence that the method is sound to verify

that real-time constraints are met. This is shown by the following steps. First we define an unfolding of the abstract graph, splitting every abstract actor back into N individual actors. Figure 2(c) shows this for the example. It gives a graph that has more actors and more dependencies and larger execution times than the original graph (compare Figure 2(a) and Figure 2(c)) and according to Proposition 1, its throughput is a lower bound on the throughput of the original graph and all individual firing times are conservative approximations of the original firing times. Note that how tight the conservative bounds are depends strongly on how well-suited the original graph is for abstraction and how well-chosen the abstraction functions are. We define an unfolding of an SDF graph, which will be used to unfold the abstract graph to be able to directly compare it to the original graph using Proposition 1. Formally, the unfolding procedure is defined as follows.

DEFINITION 5. *Let (A, D, T) be a timed SDF graph. The N-fold unfolding unf$(A, D, T, N) = (B, E, U)$ where*

- $B = \{a_i \mid a \in A, 0 \leq i < N\}$.
- *for every edge $(a, b, p, c, d) \in D$, there are precisely N edges in E. For every $0 \leq i < N$, let $j = (i + d) \mod N$ and let $d' = d \text{ div } N + t$, where $t = 1$ if $j < i$ and $t = 0$ otherwise, then $(a_i, b_j, p, c, d') \in E$.*
- *$U(a_i) = T(a)$ for every $a \in A$ and $0 \leq i < N$.*

The timed SDF graph and its unfolding mimic each other exactly. If actor a can execute for the i'th time in the original graph, then actor $a_{i \mod N}$ can execute for the i div n 'th time in the unfolded graph. They have the same throughput up to the factor N.

PROPOSITION 2. *Let (A, D, T) be a timed SDF graph and $(B, E, U) =$ unf(A, D, T, N) its N-fold unfolding. Let τ be the throughput of (A, D, T) then τ', with $\tau'(a_i) = \tau(a)/N$ is the throughput of (B, E, U).*

For the remainder of this section, assume that we have a timed SDF graph (A, D, T), an abstraction (α, I) for this graph and the abstract graph (A', D', T'). We further compute the N-fold unfolding of the abstract graph as (B, E, U), where $N = \max_{a \in A} I(a)$. We investigate the relationship between the graphs (A, D, T) and (B, E, U). With every actor $a \in A$ we associate an image actor in B as follows. $\sigma(a) = a'_i$ if $\alpha(a) = a'$ and $I(a) = i$. Note that such $\sigma(a)$ exists for every $a \in A$. One can show that the actor $\sigma(a)$ conservatively mimics a. Firstly, it actor has at least the same amount of execution time.

PROPOSITION 3. *If $a'_i = \sigma(a)$ then $T(a) \leq U(a'_i)$.*

For every edge, there is a corresponding one with at most the same amount of tokens.

PROPOSITION 4. *Let $(a, b, p, c, d) \in D$, then there is some $(a'_i, b'_j, p, c, d') \in E$ such that $a'_i = \sigma(a)$, $b'_j = \sigma(b)$ and $d' \leq d$.*

As such, σ represents a one-to-one mapping from the original SDF graph to the unfolding of the abstract graph. This shows that the throughput of the original graph is at least as high as the throughput of the unfolding.

THEOREM 1. *Let (A, D, T) be a timed SDF graph with throughput τ and (α, I) an abstraction. Then the throughput of (A, D, T) can be conservatively estimated from $(A, D, T)^{\alpha, I}$.*
$\tau(a) \geq \tau^u(\sigma(a)) = \tau(\alpha(a))/N$.

The throughput of the abstracted graph is equivalent to its unfolding by Proposition 2, and the unfolding conservatively models the original SDF, because it follows from Propositions 3 and 4 that the conditions of Proposition 1 are met.

Algorithm 1 Convert SDFG to HSDFG

```
1: ConvertToHSDF(G = (A, D, T))
2: V ← {(t_k, i_k) | t_k ∈ InitialTokens(D)}
3: γ̄ ← RepetitionVector((A, D))
4: σ ← SequentialSchedule((A, D), γ̄)
5: for j = 1 to Length(σ) do
6:    Actor a ← σ[j] /* a the j'th actor in the schedule */
7:    fire a consuming tokens W ⊆ V
8:    V ← V\W
9:    produce output tokens with time-stamp
          ḡ_p ← max{ḡ(t) | t ∈ W} + T(a)
10:   add new tokens to V with time stamp ḡ_[
11: end for
12: create graph according to coefficients of vectors ḡ_k = [g_{j,k}]
    and structure of Figure 4
```

6. A NOVEL HSDF CONVERSION

In this section we introduce a novel transformation from an SDFG to an equivalent HSDFG, which is in most cases considerably smaller than the HSDFG obtained from the classical transformation of [11, 15]. However, a note is in order to explain what we mean by 'equivalent'. In the traditional transformation, every actor is duplicated a number of times, according the number of firings of the actor in a single iteration. The firing times of the actors correspond one-to-one to the firing times of the corresponding firing in an iteration. Such a direct correspondence is not maintained by our new transformation. We seek to obtain a graph which has the same throughput and latency as the original graph. When specific actor firings within an iteration are of particular interest, for instance, a dedicated 'output actor' of the system, then it is straightforward to include this information in the constructed graph.

In order to derive an equivalent HSDFG, we consider a symbolic execution of the original graph. Recent developments in throughput calculation of SDFGs [8] have shown that the most efficient method for throughput calculation is to do an execution of the graph until a recurrent state is found and the behaviour becomes periodic, then the throughput is obtained from examining the throughput achieved in the periodic part of the behaviour. Although the number of states that need to be explored in the behaviour is in the same order as the number of actors in the equivalent HSDFG according to the traditional transformation, the main strength of the algorithm comes from the fact that hardly any of these states need to be stored in memory and used later on.

In the same spirit, we use state-space exploration to determine a reduced HSDFG. Instead however of doing an explicit exploration, we apply the method *symbolically*. If a firing actor consumes n tokens from various channels and we know that these tokens have become available at times t_1, t_2, ..., t_n respectively, then we know that the actor starts its firing at: $t = \max_i t_i$ If the duration of the firing of that actor is E, then the actor ends it firing at: $t' = E + \max_i t_i = \max_i t_i + E$. This is also the time when all tokens produced by that firing become available to actors that depend on such tokens. We will show that if we continue execution like this, the time at which the token becomes available can always be expressed using an equation of the form:

$$t = \max_i (t_i + g_i)$$

using suitable constants g_i. Assuming the original t_i are fixed, we can characterise such a *symbolic time stamp* as a vector $\overline{g} = [g_i]$.

We illustrate this with the simple graph of Figure 3. An iteration consist of three firings, two of the left and one of the right

actor. In the initial state (a), the symbolic time stamps of the initial tokens in the graph are simply t_1, t_2, t_3 and t_4. The left actor fires consuming tokens labelled t_1 and t_2. Hence the firing takes place at time $\max(t_1, t_2)$ and ends at $\max(t_1 + 3, t_2 + 3)$, which is the symbolic time stamp of the newly produced tokens. Towards state (c), the left actor fires another time, consuming tokens t_3 and $\max(t_1 + 3, t_2 + 3)$, so it starts at $\max(t_1 + 3, t_2 + 3, t_3)$ and ends at $\max(t_1 + 6, t_2 + 6, t_3 + 3)$. Finally, the right actor fires, bringing the graph back in the original state, but with symbolic tokens representing the impact of a single iteration.

In general, if an actor fires consuming tokens with a start time equation characterised by $\bar{g}_n = [g_{n,i}]$ then its starting time $t = \max_j(\max_i(t_i + g_{j,i})) = \max_i(t_i + (\max_j g_{j,i}))$ In other words, the starting time can be characterised by the vector $\overline{g} = \max_j \overline{g}_j := [\max_j g_{i,j}]$. The actor ending time, which is then equal to the time stamps of the new tokens is equal to: $\max_j \overline{g}_j + E$. This is again of the same form, it shows that we can characterise the time of *any* token in the graph symbolically by an expression of this form.

We mimic the state-space exploration algorithm using symbolic time stamps instead of concrete time stamps. After a single iteration of the graph (executing every actor a number of times in accordance with the repetition vector, all tokens are back at their original places and labelled with a time stamp which expresses an equation relating their production time to the production times of the original, initial tokens.

Now it is straightforward to construct an HSDFG which mimics exactly this behaviour. If $t'_k = \max_j g_{j,k} + t_j$, then t'_k needs to respect certain minimum distances ($g_{j,k}$) to the previous token times t_j. This can be enforced by placing a single actor between them. Because these distances are enforced pair-wise, additional multiplexing and de-multiplexing of tokens is required. This ultimately leads to a homogeneous graph of the form depicted in Figure 4. In the middle there is a matrix of actors for the coefficients $g_{j,k}$. Not all tokens depend on each other. If there is no dependence, the actor need not be present, as illustrated by the gray actors in Figure 4. In practice, this matrix is often quite sparse. To the left and at the bottom we use actors with execution time 0, which act as multiplexors and de-multiplexors to the tokens on the channels between them. These actors only need to be present if there is actually more than one actor that needs the token or multiple actors from which the tokens need to synchronise.

The algorithm for calculating the graph is straightforward and illustrated in Algorithm 1. (The algorithm is derived from an algorithm to convert an SDFG into a MaxPlus matrix [8, 7].) Firstly, the repetition vector of the graph is computed and an arbitrary sequential schedule for one iteration of the graph, using well-known methods [11, 15]. The initial tokens in the graph are initialised with the corresponding vector representations. t_1 is represented by $\bar{i}_1 := [0; \ -\infty; \ -\infty; \ldots]$, t_2 by $\bar{i}_2 := [-\infty; \ 0; \ -\infty; \ \ldots]$ and so on. $-\infty$ is used here to denote when there is no dependency on the corresponding token. ($-\infty$ is the neutral element of the $\max$ operator and zero element of addition in Max-Plus algebra which formalises these types of equations [1].) Next, the schedule is executed and with every actor firing, the symbolic input tokens are read, the symbolic starting time and ending time of the actor are computed and new tokens are produced carrying this symbolic ending time of the actor. In this way, the iteration is completed and returns the graph to its original token distribution. By reading the vector $[g_{j,k}]$ corresponding to token k, we get the coefficients for the execution times in the k-th column of the matrix of Figure 4. If the entry in the vector is $-\infty$, no actor is created. Finally, the (de-)multiplexing actors are created when necessary and all edges are created including the edges with initial tokens between the (de-)multiplexing actors.

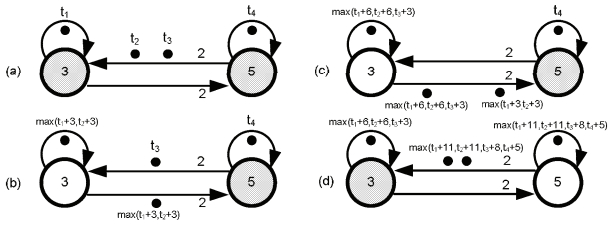

Figure 3: Example symbolic execution

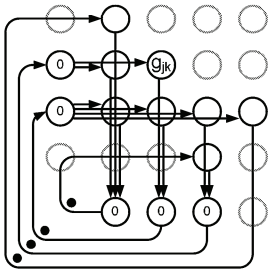

Figure 4: Structure of HSDF conversion

The Max-Plus matrix is a square matrix and the number of rows and columns corresponds to the number (N) of initial tokens in the graph. From Figure 4 we see that the resulting graph has at most $N(N + 2)$ actors, $N(2N + 1)$ edges and N initial tokens.

7. EXPERIMENTS

The method described in this paper was inspired by practical work in which SDF models were used for worst case timing analysis of multiprocessor system-on-chip realisations of multimedia applications [16, 13]. Here, informal arguments were used to defend that the very large SDF models that resulted from the analysis methods could be replaced by simpler models, which could be analysed by hand. The results of this paper allow one to ensure oneself that this method is sound. [16] investigates the embedding of large data structures in the memory of a multiprocessor system with network-on-chip. The algorithm mapped onto the platform is a full-search block matching algorithm for detecting motion vectors in video sequences, typically used in video encoding algorithms such as H.263 and MPEG-2. For efficient operation of the algorithm, data is to be pre-fetched from a frame-memory residing in a different tile on the SoC. The data has to be transported over the network-on-chip. The result of the analysis is essentially the model depicted in Figure 5, where the CA actors represent the communication assist components on either side of the network. The A actors take care of the pre-fetch requests and the actual computations. In total, 1584 of such computations (all taking the same amount of time) have to be performed within the duration of a single video frame. By the obvious abstraction, the graph is reduced to the model on the right of Figure 5, which in this case, has exactly the same throughput as the original graph. Note that in the abstract graph of 5 redundant edges have been pruned for readability. We do not show any experimental results for the abstraction technique, because they are not intended to be used on arbitrary graphs, but rather specific graphs which show a specific regular structure. Results for arbitrary graphs would not be good and regular graphs can

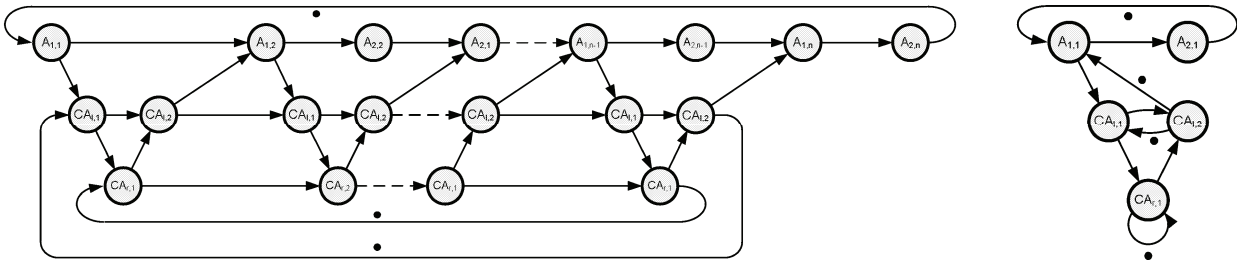

Figure 5: Remote memory access model and abstraction

Table 1: HSDF Transformations Compared

test case	Traditional conversion number of actors	new conversion number of actors	ratio
1. h.263 decoder	1190	10	119
2. h.263 encoder	201	11	18.3
3. modem	48	210	0.23
4. mp3 dec. block par.	911	8	114
5. mp3 dec. granule par.	27	8	3.38
6. mp3 playback	10601	38	279
7. sample rate conv.	612	31	19.7
8. satellite	4515	217	20.8

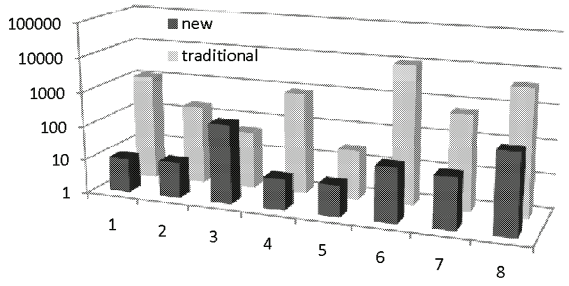

Figure 6: Experimental results

be constructed for which the abstraction returns small graphs with a perfectly accurate prediction of performance.

The novel HSDF conversion algorithm has been implemented as an extension to the SDF^3 tool set available from [17]. For a benchmark of available SDFGs of various applications [14] we compare the sizes of the HSDFGs in terms of their numbers of actors. The run-time of the algorithms is a few milliseconds. The results in Table 1 and visually presented in Figure 6, show that in all but one case, the new conversion algorithm yields much smaller graphs, The differences in size vary from a small factor up to a factor of 279 times fewer actors. Only for the case of the modem graph [11], the result is actually larger. This is a graph which is itself 'almost HSDF' with only few rates different from 1 and with a large number of initial tokens. Because the size of the traditional HSDF is exactly predictable and a bound on the size of the new method can be estimated from the number of initial tokens, it is possible to assess beforehand when this might occur.

8. CONCLUSIONS

We have presented two SDF graph reduction techniques. A transformation that allows simplification of SDF models to allow for a more efficient analysis with guaranteed conservative results. Experiments show that the results can be used in practice assure that transformations that are often used manually in practice, are indeed correct. Secondly a novel conversion algorithm has been introduced to transform SDF graphs into HSDF graphs.

9. REFERENCES

[1] F. Baccelli, G. Cohen, G. Olsder, and J.P.Quadrat. *Synchronization and Linearity*. John Wiley & Sons, 1992.

[2] N. Bambha, V. Kianzad, M. Khandelia, and S. Bhattacharya. Intermediate representations for design automation of multiprocessor DSP systems. *Design Automation for Embedded Systems*, 7:307–323, 2002.

[3] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. V. Meerbergen. Predictable multiprocessor system design. In *SCOPES 2004, 8th Int. Workshop on Software and Compilers for Embedded Systems*, 2004.

[4] S. Bhattacharyya, P. Murthy, and E. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[5] A. Dasdan, S. S. Irani, and R. K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 37–42, New York, NY, USA, 1999. ACM.

[6] M. Geilen. Reduction techniques for synchronous dataflow graphs. Technical Report ESR-2009-01, Electronic Systems Group, Dept. of Electrical Engineering, Eindhoven University of Technology, 2009.

[7] M. Geilen. Synchronous data flow scenarios. *Transactions on Embedded Computing Systems, Special issue on Model-driven Embedded-system Design, to be published*, 2009.

[8] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput analysis of synchronous data flow graphs. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 27–30. IEEE Computer Society Press, Los Alamitos, CA, USA, 2006, 2006.

[9] A. Ghamarian, S. Stuijk, T. Basten, M. Geilen, and B. Theelen. Latency minimization for synchronous data flow graphs. *Digital Systems Design, Euromicro Symposium on*, pages 189–196, 2007.

[10] R. Govindarajan and G. R. Gao. Rate-optimal schedule for multi-rate dsp computations. *J. VLSI Signal Process. Syst.*, 9(3):211–232, 1995.

[11] E. Lee and D. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, 75(9):1235–1245, Sept. 1987.

[12] P. K. Murthy. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow*. PhD thesis, University of California, Berkeley, December 1996.

[13] P. Poplavko, T. Basten, and J. van Meerbergen. Execution-time prediction for dynamic streaming applications with task-level parallelism. In *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 228–235, Washington, DC, USA, 2007. IEEE Computer Society.

[14] SDF^3. http://www.es.ele.tue.nl/sdf3.

[15] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., New York, NY, USA, 2000.

[16] S. Stuijk, T. Basten, B. Mesman, and M. Geilen. Predictable embedding of large data structures in multiprocessor networks-on-chip. In *Proc. DSD, 8th Euromicro Conference, DSD'05*, pages 388–395. IEEE Computer Society Press, 2005.

[17] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006.

[18] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. Comput.*, 57(10):1331–1345, 2008.

[19] M. Wiggers, M. Bekooij, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *DAC*, pages 658–663, 2007.