Theorems for model-checking

Avinash Malik
IBM Research
Ireland

David Gregg
School of computer science and statistics
Trinity College Dublin, Ireland

Abstract

In this paper we describe the theorems and other related material when performing model-checking. These theorems form the basis for using model-checking as a tool for code optimization and distribution of stream graphs onto heterogeneous multi-processor architectures.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Encoding the distribution and scheduling problem into the Uppaal model-checker

We have used the Uppaal model-checker [1] as a representative example amongst all the model checkers supporting *Computational Tree Logic* (CTL) properties. Any other model checker with support for verifying CTL properties would suffice. Each model checker implements exploration techniques in different ways. Also, the input language of each model checker differs. Moreover, the quantitative performance (computation time and memory consumption) of the model checking process itself would vary depending upon the model checker used. Uppaal has been shown to provide promising results for timing analysis by Roop et al. and Behrmann et al. 2005, 2009, and hence we use Uppaal. Comparing different model checkers for partitioning and scheduling would make a good case study, but is outside the scope of this paper.

1.1 Formalizing the problem statement

A *Synchronous Data-Flow* (SDF) application is a graph $\mathcal{G}(V_g, E_g)$, where V_g are the vertices representing the filters and $E_g \subseteq V_g \times V_g$ are the edges representing the FIFO communication channels between filters. An example SDF graph for a smart surveillance system is shown in Figure 1b.

Let graph $\mathcal{A}(P, C)$ represent the heterogeneous execution architecture, where P represents the processors available for filter execution and $C \subseteq P \times P$ represents the communication link between processors. Thus, we define the makespan (schedule length) for a single stable state iteration of $\mathcal{G}$ as in $\phi(\mathcal{G}, \mathcal{A})$. Function $\phi(\mathcal{G}, \mathcal{A})$, from here on referred to as just $\mathcal{M}$ for sake of brevity, denotes the makespan of $\mathcal{G}$ on $\mathcal{A}$

Our objective is to find an allocation for every vertex $V_{g_i} \in V_g$, where $i \in \{1..N\}$ on some processor $P_j \in P$, where $j \in \{1..|P|\}$ to minimize $\mathcal{M}$.

Definition 1. *Let ω_{g_i} represent the number of bytes produced for every invocation of some filter $V_{g_i} \in V_g$. Thus, for a single stable state iteration of $\mathcal{G}$, the number of bytes produced by filter V_{g_i} is $\omega_{g_i} q_{g_i}$. Recall that q_{g_i} is the natural granularity of filter V_{g_i}. Let T represent some absolute time elapsed from the start of execution of $\mathcal{G}$ on $\mathcal{A}$. We define the throughput of V_{g_i} as:*

$$\zeta_{g_i} = \frac{\omega_{g_i} q_{g_i} \mathcal{N}(\Pi)}{T} \ \forall i \in \{1...N\} \tag{1}$$

where $\mathcal{N}(\Pi) = T/\Pi$ gives the number of stable state iterations of $\mathcal{G}$ in T. Finally, we define the throughput of the graph $\mathcal{G}$ as: $\zeta = \sum_{i=1}^{N} \zeta_{g_i}$

Lemma 1. *The allocation solution that minimizes makespan also provides the highest throughput.*

Proof. As we can see from Equation (3), the throughput and makespan are inversely proportional. Hence, minimizing makespan ($\mathcal{M}$) is equivalent to maximizing throughput (ζ). □

1.2 Modeling communication and computation

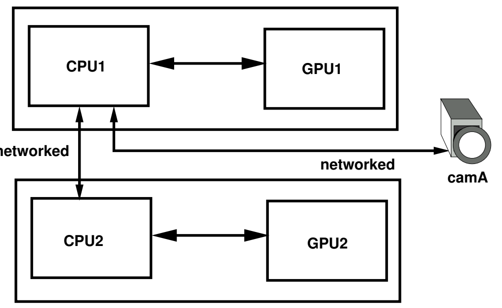
The very first transformation that is carried out is the translation of the communication channels $E_g \in \mathcal{G}$ into filters. Communication costs play an important role in the makespan minimization problem. For the sake of uniformity we translate the FIFO channels into filters. The translation of $\mathcal{G}$ into a precedence graph results in a new graph $\mathcal{P}$ where the FIFO channels are made explicit.

Figure 2 gives the precedence graph $\mathcal{P}$ translated from the SDF graph $\mathcal{G}$ in Figure 1b. $\mathcal{P}(V, E)$ is a directed graph, where $E \subseteq V \times V$ are the precedence relations, $V_A \subseteq V$ are the computation filters and $V_C \subseteq V$ are the communication filters.

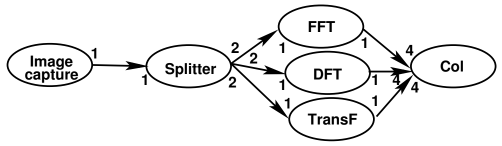
As we can see, all FIFO edges are converted into filters. The communication filters have their input and output data rates calculated by looking up the natural granularity of their respective source filters. We know that the natural granularity for the computation filters in $\mathcal{P}$ is given by $\{2, 2, 4, 4, 4, 1\}$. Hence, for the communication filter `C1`, its input and output rates are 2×2, because its source computation filter, `Image capture`, has a natural granularity of 2 and an output rate of 1 token per invocation, same for the others.

Lemma 2. *The introduction of communication filters representing the FIFO channels in $\mathcal{P}$ does not change the schedule and buffer sizes calculated for the computation filters in $\mathcal{P}$.*

Proof. The proof is by substitution on a closed form of the precedence graph. See [4] for details. □



(a) A smart embedded networked tracking system



(b) The abstract stream model representing the tracking algorithm

Figure 1: A motivating example

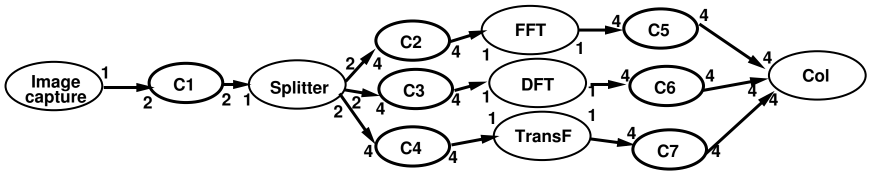
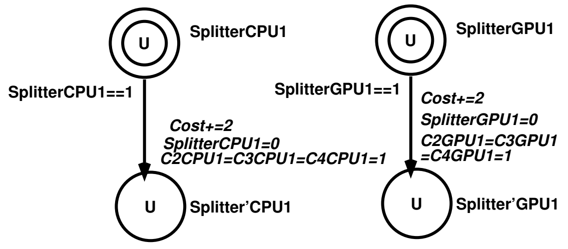
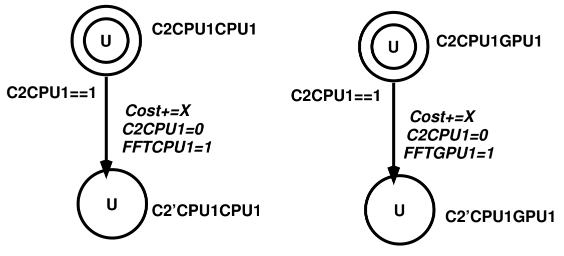


Figure 2: Precedence graph the SDF graph in Figure 1b



(a) `Splitter` filter allocation



(b) `C2` filter allocation

Figure 3: Uppaal representation of computation and communication allocation

1.3 Encoding the filter allocation and distribution problem into Uppaal automaton

Modeling computation filter allocation

Every computation filter in the set V_A can be allocated to some processor $P \in \mathcal{A}$. Every such allocation is represented by an Uppaal automaton. Figure 3a shows the Uppaal automata representing allocation of filter `Splitter` on two of the four available processors in Figure 1a.

Every location in the automata is marked with a `U`, representing urgency, i.e., any transition if enabled needs to be taken. The location is named by joining the name of the filter and the processor and is represented by the set: $\{\texttt{Splitter}, \texttt{CPU1}\}$ for the first automaton. The transition is guarded by the condition `SplitterCPU1==1`. Upon transitioning, a global variable `Cost` is incremented by the computation time of `Splitter` on `CPU1`, which in this case is 1*2, where 1 is the computation cost and 2 is the natural granularity of `Splitter`. Finally, the actions also disable the guard condition and set the next (communication in this case) filter guards high for further transitioning.

Modeling communication filter allocation

Every communication filter in set V_C can be allocated to a communication link $C = (l, m)|l \in P,\ m \in P$. Figure 3b, shows the `C2` filter allocation on the communication link joining processors `CPU1` to itself and to `GPU1`, respectively. The name of the locations is represented by the set: $\{\texttt{C2}, \texttt{CPU1}, \texttt{CPU1}\}$ for the first automaton. The guard is `C2CPU1` and the actions disable the guard and set the next filter guard high. The next filter is `FFT` in this case. Finally, the global variable `Cost` is updated by the communication costs on these links.

Such basic automata are produced for all possible filter allocations. These automata only represent sequential execution of filters; parallelism is not described *yet* and will be described in the next section. For example, when run through Uppaal, with the instructions to find the path with the least `Cost` in the state space from the starting state to the terminal state, it might choose the first automaton in Figure 3a, which increments the cost by 2, this transition in turn would enable `C2`. This time around the second automaton might be chosen, which again increments the cost by `X` time units. Since `Cost` always increments sequentially, no parallelism can possibly be described by these automata.

1.4 Modeling task and data-parallelism

Modeling task-parallelism

Task-parallelism is explicitly denoted in the SDF graph and consequently in the precedence graph by split and join nodes. The three filters `FFT`, `DFT`, and `TransF` denote task-parallel filters that can possibly run in parallel provided there are no resource constraints. Consider the basic automata representing `FFT` and `DFT` allocation on processor `CPU1` and `GPU1`. We know that their location

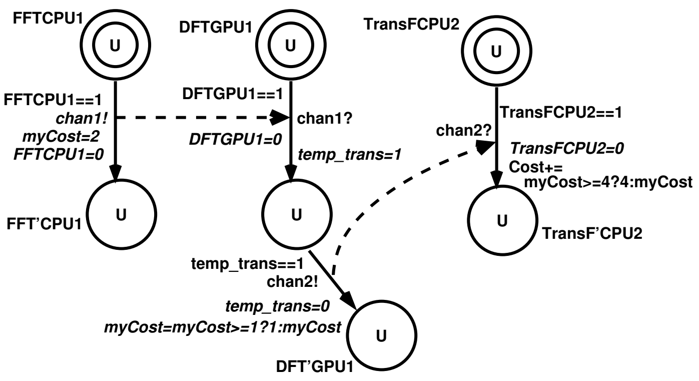


Figure 4: A network of Uppaal automata representing parallel filter execution

names can be identified by the sets: $\{\texttt{FFT},\texttt{CPU1}\}$, $\{\texttt{FFT},\texttt{GPU1}\}$, $\{\texttt{DFT},\texttt{CPU1}\}$, and $\{\texttt{DFT},\texttt{GPU1}\}$, respectively. We build network of basic automata connected via rendezvous channels provided the intersection of the set of location names results in a $\emptyset$ set. Thus, $\{\texttt{FFT},\texttt{CPU1}\} \cap \{\texttt{DFT},\texttt{CPU1}\} = \texttt{CPU1}$, means that these two automata represent execution of two different filters on the same processor (CPU1) and hence, cannot be run in parallel, whereas, $\{\texttt{FFT},\texttt{CPU1}\} \cap \{\texttt{DFT},\texttt{GPU1}\} = \emptyset$, represents two automata that can be run in parallel.

Figure 4, shows an example network combining basic automata representing allocation of FFT, DFT, and TransF, on CPU1, GPU1, and CPU2, respectively. The first automaton (FFTCPU1) rendezvous with the first transition of the second automaton (DFTGPU1) via channel chan1. This rendezvous forces the two transitions to take place together, in the process transferring the execution cost of FFT on CPU1 (myCost=2, say). Upon completion of this rendezvous, the actions set the guard for the second transition (temp_trans) high. This allows the second transition of the second automaton to rendezvous with the third automaton via channel chan2. The maximum of the received value, 2, and the execution cost of DFT on GPU1 is transferred to the third automaton. The final automaton in turn increments the global Cost variable by the maximum of the received value (max(2,1)) and its own execution cost (4). Thus, the execution cost of the three automata running in parallel is the maximum of the three execution costs.

We generate more such networks exhibiting other possible combinations that might run in parallel. For example, a network combining just two of the three automata in Figure 4. In such a case, the overall execution cost would be calculated as the maximum of the two automata in parallel and then incremented by the third filter running in sequence after the parallel execution.

Modeling data-parallelism Exploitation of task-parallelism in the precedence graph of Figure 2 is not enough. As we can see from Figure 4, we are only ever able to utilize 3 of the 4 available processors. Replication of stateless filters to utilize idle processor resources is a well known technique amongst the compiler optimization community. A naive way to replicate a stateless filter is to replicate the filter P times. This makes sure that all processors are utilized.

Figure 5a shows replication of the stateless FFT and DFT filters four times, one for each processor. As we can see, this technique allows utilization of all 4 processors (unlike *just* task-parallelism), but leads to communication overheads and may result in more filter copies than the number of available processors. For example, when running the 4 FFT copies no other filters can be run. Thus, it is essential to judiciously replicate stateless filters in order to obtain good throughput.

Our model-checking approach provides an optimal solution to this judicious stateless data-replication problem. Our approach is a multi-step process: first off, we naively replicate all the stateless filters, as shown in Figure 5a. Next, we build the basic automata modeling the execution of these filters on the processor set $\mathcal{P} \in \mathcal{A}$, as shown in Figure 5b. Finally, we build extra automata modeling fusion of these stateless filters for each processor as shown in Figure 5c.

In Figures 5b and 5c, we haven't shown all the Uppaal automata that are generated due to lack of space (the ... in Figure 5, show the other automata that would be generated). The important point to note is that the algorithm (Algorithm 1) modeling fused stateless filter execution is exhaustive. For example, we build automata modeling execution of filters FFT1 and FFT2, together, then FFT1, FFT2 and FFT3 and so on and so forth for all filters for each processor. A total of $\sum_{i=1}^{N} i$ extra automata are generated, where N is the number of stateless filter copies ($N = 4$ in this case). These automata are combined with the rest of the automata and passed through Uppaal to find the path with the least Cost in the state space.

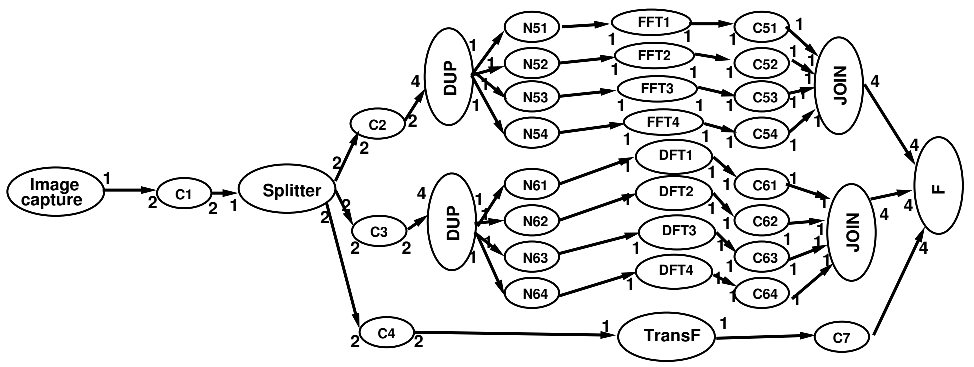
ALGORITHM 1: Building fused replicated filter states

```
Input: Precedence graph $\mathcal{P}$, Execution architecture graph $\mathcal{A}$
Output: A set $S$ of fused filter states
set $S = \emptyset$; int i = 0;
set $SP$ = the set of all state-less split joins in $\mathcal{P}$;
for each $SP(i)$ do
    set $B$ = branches in $SP(i)$;
    if arity of $B \geq 2$ then
        for each $b_j$ of $B$ do
            for $P \subseteq A$ do
                int index = j;
                state $s$ = new state($V \in b_{index}, V \in b_{index++} | b_{index} \in B, b_{index++} \in B$);
                $S \cup \{s\}$;
                while index < arity of $B$ do
                    int count=0;
                    state $s$ = new state($V \in b_{count} | \forall count \in \{0, ..index\}, b_{count} \in B$);
                    $S \cup \{s\}$;index++;
                end
            end
            j=j+1;
        end
    end
    i = i +1;
end
```

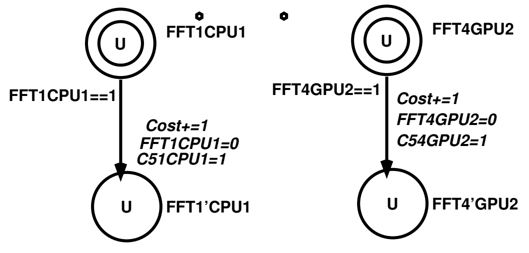
A keen reader might have noticed that the increments in Cost is different for automata in Figure 5b and 5c. Consider the first automata in Figure 5c, suppose that we fuse FFT1 and FFT2, while leaving the other stateless filter copies untouched. This would be equivalent to saying that instead of making 4 copies of the stateless FFT filter we have made 3 copies, where the first one has a granularity of 2, while the others only have a granularity of 1. Thus, the fused filter automata represent different granularities and number of copies of the stateless filters.

State sharing

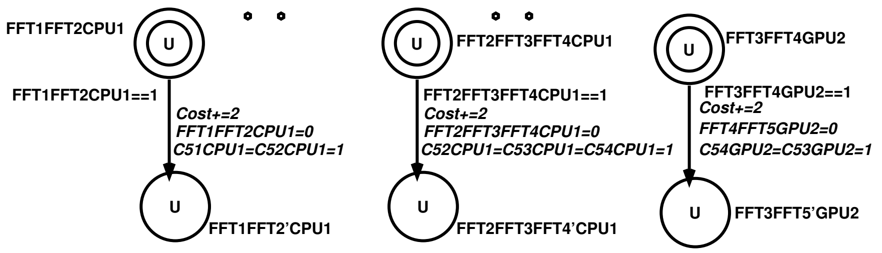
State sharing is an optimization technique, which essentially removes duplicate copies of shared data. State sharing can be achieved by fusing two or more filters within a single execution thread. Such fusion results in pointer based communication between different filters rather than copying data from one filter to the other, thereby reducing the communication overhead. We fuse all filters allocated to the same processor into a single kernel thread in order to avoid communication overheads.



(a) Naive replication of stateless FFT and DFT filters



(b) Basic automata modeling stateless FFT filter execution



(c) Extra automata representing fused stateless filters

Figure 5: Optimal exploitation of data-parallelism

S2. Theorems and proofs

S2.1 Definition and Semantics of precedence graph $\mathcal{P}$

Definition 2. *A transformation $\tau : \mathcal{G} \rightarrow \mathcal{P}$, translates a SDF graph $\mathcal{G}$ into a precedence graph $\mathcal{P}$. A precedence graph $\mathcal{P}(V, E)$, where $V_A \subseteq V$ represent the computation filters and $V_C \subseteq V$ represent the communication filters. The edges $E \subseteq V \times V$ represent the precedence relations (dependencies) between the vertices. The input and output data rates of the communication filters (V_C) is equal to output rate * natural granularity of the source computation filter in set V_A* where $V_A \prec V_C$. *$\prec$ is the precedence operator lifted to sets V_A and V_C, where every communication filter has a unique immediate predecessor computation filter, i.e., $V_{A_i} \prec V_{C_j}$ and $V_{A_i} \in V_A, V_{C_j} \in V_C, \forall i \in \{1..N\}, \forall j \in \{1...M\}$. Finally, $V_A = V_g$.*

Lemma 3. $\mathbf{q}_A = \mathbf{q}_g$. *Where* $\mathbf{q}_A$ and $\mathbf{q}_g$ *give the repetition vectors for the computation filters for sets $V_A \in \mathcal{P}$* and $V_g \in \mathcal{G}$, *respectively.*

Proof. The above is obviously true for a two filter graph $\mathcal{G}$ transformed into $\mathcal{P}$. Consider pairwise execution of filters for every edge $E \in \mathcal{P}$. Thus, every filter $V_{A_i} \in V_A$ executes at the natural granularity specified by $\mathbf{q}_g$. From the semantics of data-flow graphs, for some filter $V_{A_i} \in V_A$ and $V_{C_j} \in V_C$, we have the balance equation:

$$q_i \times output - data - rate = q_j * input - data - rate$$

Thus, from the above equation and definition of $\mathcal{P}$ it follows that every communication filter has a natural granularity of 1. By substituting this result in the balance equation for the complete graph $\mathcal{P}$ we can obtain $\mathbf{q}_A \subseteq \mathbf{q}_P$ for a consistent single stable state schedule of $\mathcal{P}$ and show that $\mathbf{q}_A = \mathbf{q}_g$. Note the $\mathbf{q}_P$ gives the repetition vector for the graph $\mathcal{P}$. □

S2.2 Formulating makespan Π

In this section we formulate the definition of makespan and prove our claim about granularity.

Definition 3. *Automaton is a tuple $Q = (S, s0, g, L, A, T, \xi)$, where S are the states, $s0 \subseteq S$ is the starting state, g represents the guards on the transition, L represents the set of names representing filters and resources, A represents the actions during transition, and T represents the transition function, and ξ associates names to states.*

Definition 4. *Allocation of an execution filter $V_{A_i} \in V_A$,* forall some $i \in \{1..N\}$ *on a processor $P_k \in P$,* for some $k \in \{1..H\}$ *is defined* *as:*

$$\begin{gathered}
alloc(V_{A_i}, P_k) = Q, \text{where :} \\
S \text{ is the set of states} \\
L = \{\mathcal{L}(V_{A_i}) \cup \mathcal{L}(P_k)\} \\
\xi : S \to NAME, \text{ satisfying } \xi(S) \subseteq L \\
s0 \subseteq S \\
T = (s, g/A, s') \\
g : T \to BOOLEXP, \text{ satisfying} \\
g(T) = \bigwedge_{t=1}^{E}(l_t), l_t \in \{\mathcal{L}(V_{A_i}) \times \mathcal{L}(P_k)\} \to \{0, 1\} \\
A : T \to EXP, \text{ satisfying} \\
A(T) = \{l_p = 1 | l_p \in \{\mathcal{L}(V_r) \times \mathcal{L}(P_k), \forall p \in \{1..M\}, \\
V_{A_i} \prec V_r, \forall r \in \{1..N\}\} \\
\cup\{l_t = 0 | l_t \in \{\mathcal{L}(V_{A_i}) \times \mathcal{L}(P_k)\}, \forall t \in \{1..E\} \\
\cup\{\mathcal{COST}(Q) = \mathcal{COST}(V_{A_i}, P_k) * q_{A_i}\}
\end{gathered}$$

Every allocation of an computation filter to some processor in the architecture results in an automaton Q. Where the transition guard is the conjunction of cross product of names of Boolean literals, representing that the filter is ready for execution and the processor is free for allocation, respectively. $\mathcal{L}$ is the function that produces a set of fresh names representing the Boolean literals. For a filter (e.g., V_{A_i}) the arity of this set is equal to the arity of set $F \subseteq E, E \in \mathcal{P}$, where $F \subseteq V_o \times V_{A_i}$, where $\forall o \in \{1..R\}$, where $V_o \prec V_{A_i}$. Informally, the arity is equal to the the number of immediate predecessors of V_{A_i}. For an empty set, possible in the case of the source filter in the precedence graph, $\mathcal{L}$ randomly assigns a single name representing the start of execution. The arity of the set $\mathcal{L}(P_k)$ is always 1, which is a unique name representing the availability of processor resource P_k. Upon successful evaluation of the guard condition, the transition is taken, and the action consists of updating the Boolean literals for all immediate successor filters, setting the current filter execution Boolean literal to 0 and, finally, setting the execution cost of this transition. $\mathcal{COST}(V_{A_i}, P_k)$ and q_{A_i} represent the cost of single invocation of filter V_{A_i} on processor P_k and the natural granularity of this filter, respectively.

Definition 5. *Allocation of a communication filter* $V_{C_j} \in V_C$, for some $j \in \{1..M\}$ *on a communication link* $C_m^l \in C$, *for some* $(l, m) \in \{1..H\}$ *is defined as:*

$$\begin{gathered}
alloc(V_{C_j}, C_m^l) = Q, \text{where} \\
S \text{ is the set of states} \\
L = \{\mathcal{L}(V_{C_j}) \cup \mathcal{L}(P_l) \cup \mathcal{L}(P_m)\} \\
\xi : S \to NAME, \text{ satisfying } \xi(S) \subseteq L \\
g : T \to BOOLEXP, \text{ satisfying} \\
g(T) = \bigwedge_{t=1}^{E}(l_t), l_t \in \{\mathcal{L}(V_{C_j}) \times \mathcal{L}(P_l)\} \to \{0, 1\} \\
A : T \to EXP, \text{ satisfying} \\
A(T) = \{l_p = 1, l_p \in \{\mathcal{L}(V_k) \times \mathcal{L}(P_m)\} | \forall p \in \{1..M\}, \\
V_{C_j} \prec V_k, \forall k \in \{1..N\}\} \\
\cup\{l_t = 0, l_t \in \{\mathcal{L}(V_{C_j}) \times \mathcal{L}(P_l)\} | \forall t \in \{1..E\} \\
\cup\{\mathcal{COST}(Q) = \mathcal{COST}(V_{C_j}, C_m^l) * q_{C_j}\}
\end{gathered}$$

The definition of allocation of communication filter on a communication link connecting two processors is similar to that of computation filter. Except, $\mathcal{L}(V_{C_j}), \mathcal{L}(P_l)$ and $\mathcal{L}(P_m)$, represent the readiness of execution of the communication filter, and the availability of source and target processors of the communication link C_m^l, respectively.

Definition 6. *Let Q_1 and Q_2 be two automata and Π be the makespan. The resultant transition system, of the asynchronous composition of Q_1 and Q_2 with mutually exclusive access to resource, which gives the resultant $\mathcal{M}$ is defined as:*

$$Q_1//Q_2 = \{S_1 \odot S_2, (s0_1, s0_2), g_1 \cup g_2, A_1 \cup A_2, T\}$$

$$T = \begin{cases}
\{(s_1, s_2), g_1/A_1 \cup \{\Pi+ = \mathcal{COST}(Q_1)\}, (s_1', s_2)\} \\
\cup\{(s_1, s_2), g_2/A_2 \cup \{\Pi+ = \mathcal{COST}(Q_2)\}, (s_1, s_2')\} \\
\cup\{(s_1, s_2), (g_1 \wedge g_2), (A_1, A_2) \cup \\
\{\Pi+ = max(\mathcal{COST}(Q_1), \mathcal{COST}(Q_2))\}, (s_1', s_2') \\
| \{(L_1) \cap (L_2) = \emptyset\}\}
\end{cases}$$

Definition 6 is the classical asynchronous product with the restriction of mutually exclusive access to shared resources (e.g., processor and communication links), shown by the last transition condition. Now we can finally formulate the definition of Π.

Definition 7. *Given a SDF graph $\mathcal{G}$, an architecture description $\mathcal{A}$ and the literal M from Equation (6). We can define the makespan function ϕ as:*

$$alloc(V_{A_i}, P_k)//alloc(V_{C_j}, C_m^l) : \Pi \to R^+ \quad (2)$$

where,

$$\begin{gathered}
V_{A_i} \in V_A, V_{C_j} \in V_C, V_A \in \mathcal{P}, V_C \in \mathcal{P} \text{ and } \tau : \mathcal{G} \to \mathcal{P} \\
\forall i \in \{1..N\} \text{ and } \forall j \in \{1..O\} \\
\text{and } R^+ \in \mathbb{R}^+
\end{gathered}$$

Thus, function $//$, is the asynchronous composition (as defined in Equation (2) *all* computation and communication filters in $\mathcal{P}$ allocated to different processors and communication links in $\mathcal{A}$, which transforms updates $\mathcal{M}$ into the final makespan, a real number, by applying max-plus algebraic identities. τ is the transformation function that transforms $\mathcal{G}$ into $\mathcal{P}$ from Definition 2.

S2.3 Granularity based optimizations

First we prove that, for a two filter graph $\mathcal{P}$, with a single computation filter $V_{A_i} \in V_A, i = \{1\}$ and a single communication filter $V_{C_j} \in V_C, j = \{1\}$, provided the communication cost between processors remains constant for a range of bytes sent across these nodes, increasing granularity, by positive integer multiple G, of the computation filter (V_{A_i}) increases the graph throughput (ζ_{A_i}), as long as the number of bytes transferred remains within this range.

Lemma 4. *$\zeta_{A_{iG}} \geq \zeta_{A_i}$ for $q_{A_{iG}} = q_{A_i} * G$ and $\mathcal{COST}(V_{C_j}) = K$ for $I \leq \omega_{A_i} * q_{A_{iG}} \leq T$ for some $V_{A_i} \prec V_{C_j}$, where K, I, T are constants in set $\mathbb{R}^+$.*

Proof.

$$\zeta_{g_i} = \frac{\omega_{g_i} q_{g_i} \mathcal{N}(\Pi)}{T} \; \forall i \in \{1...N\} \quad (3)$$

From Equation (3) and Lemma 3 we have:

$$\zeta_{A_i} = \omega_{A_i} * q_{A_i}/\Pi$$

from Equation (2) we get:

$$\zeta_{A_i} = \omega_{A_i} * q_{A_i}/(alloc(V_{A_i}, P_k)//alloc(V_{C_j}, C_m^l) : \Pi \to R^+)$$

for a single computation filter V_{A_i} and communication filter V_{C_j} and from Definitions 4 and 5 we get:

$$\zeta_{A_i} = \omega_{A_i} * q_{A_i}/(\mathcal{COST}(V_{A_i}, P_k) * q_{A_i} + \mathcal{COST}(V_{C_j}, C_m^l) * q_{C_j})$$

for some $G \in \mathbb{N}^*$ we have:

$$\zeta_{A_{iG}} = \omega_{A_i} * q_{A_i} * G/(\mathcal{COST}(V_{A_i}, P_k) * q_{A_i} * G + \mathcal{COST}(V_{C_j}, C_m^l) * q_{C_j})$$

substituting the constant K in the equation $\zeta_{A_{iG}}/\zeta_{A_i}$ gives:

$$\zeta_{A_{iG}}/\zeta_{A_i} = \frac{(\mathcal{COST}(V_{A_i}, P_k) * q_{A_i} + K * q_{C_j}) * G}{\mathcal{COST}(V_{A_i}, P_k) * q_{A_i} * G + K * q_{C_j}}$$

finally, **because** $\mathcal{COST}(V_{A_i}, P_k) \in \mathbb{R}^+$, $G \in \mathbb{N}^*$, $K \in \mathbb{R}^+$, and from Lemma 3 $q_{C_j} = 1$, we can say that:

$$\zeta_{A_{iG}}/\zeta_{A_i} \geq 1$$

□

Before proceeding to extend the above proof for the complete SDF graph. We will prove some preliminary results on max-plus algebraical identities.

Lemma 5. *Let $a \oplus b = max(a, b), a \in \mathbb{R}^+, b \in \mathbb{R}^+$. Given some multiplicand $G \in \mathbb{N}^*$ we prove that $G(a \oplus b) = Ga \oplus Gb$.*

Proof.

$$\begin{gathered} let\ a \oplus b = a \\ thus,\ G(a \oplus b) = Ga\ and\ Ga \oplus Gb = Ga \\ hence, G(a \oplus b) = (Ga \oplus Gb) \end{gathered} \tag{4}$$

□

Lemma 6. *Let $a \otimes b = a + b, a \in \mathbb{R}^+, b \in \mathbb{R}^+$. Given some multiplicand $G \in \mathbb{N}^*$ we prove that $G(a \otimes b) = Ga + Gb$.*

Proof.

$$\begin{gathered} G(a \otimes b) = Ga + Gb \\ Ga \otimes Gb = Ga + Gb \\ hence, G(a \otimes b) = Ga \otimes Gb \end{gathered} \tag{5}$$

□

Now we extend the Lemma 4 for a two filter graph $\mathcal{P}$ to a general R filter graph $\mathcal{G}$ in Theorem 1.

Theorem 1. *Let $\{K_1 \ldots, K_N\}$ represent the communication costs for the set $V_C \in \mathcal{P}$ and $\{\omega_{A_1} * q_{A_{1G}} \ldots, \omega_{A_i} * q_{A_{iG}}\}$ represent the number of bytes produced by computation filters in the set $V_A \in \mathcal{P}$ and $V_A \prec V_C$. Then $\zeta_G \geq \zeta$ for some SDF graph $\mathcal{G}$, provided $G \in \mathbb{N}^*$.*

Proof. From Lemma 4 it follows:

$$\begin{gathered} \zeta_{A_{iG}} / \zeta_{A_i} = \frac{(alloc(V_{A_i}, P_k) // alloc(V_{C_j}, C^l_m) : \Pi \rightarrow R^+) * G}{alloc_G(V_{A_i}, P_k) // alloc_G(V_{C_j}, C^l_m) : \Pi \rightarrow R^+} \\ V_{A_i} \in V_A, V_{C_j} \in V_C, V_A \in \mathcal{P} \\ \forall i \in \{1..N\}, \forall j \in \{1..M\} \\ P_k \in P, P \in \mathcal{A}, C^l_m \in C, C \in \mathcal{A} \\ \forall k \in \{1..H\}, \forall (l, m) \in \{1..H\}. \end{gathered} \tag{6}$$

From Equations (6) and (2) we know that the $alloc$ function transforms a literal $\mathcal{M}$ into a real number using max-plus algebraical identities. From Equations (4) and (5) we can trivially prove:

$$G((a \oplus b) \otimes c) = (Ga \oplus Gb) \otimes Gc \tag{7}$$

Let $alloc_G(V_{C_j}, C^l_m)$ be defined by Definition 5, where $q_{C_{jG}} = q_{C_j} * G$. From Equations (6) and (7) it follows:

$$\zeta_{A_{iG}} / \zeta_{A_i} = \frac{(alloc(V_{A_i}, P_k) // alloc(V_{C_j}, C^l_m) : \Pi \rightarrow R^+) * G}{(alloc(V_{A_i}, P_k) // alloc(V_{C_j}, C^l_m) : \Pi \rightarrow R^+) * G} \tag{8}$$

Thus, $\zeta_{A_{iG}} = \zeta_{A_i}$. But,

$$\begin{gathered} alloc_G(V_{C_j}, C^l_m) : \Pi \rightarrow R^+ = \\ K_j \leq \mathcal{COST}(V_{C_j}, C^l_m) * q_{C_j} * G \\ for\ G \in \mathbb{N}^*, R^+ \in \mathbb{R}^+, K_j \in \mathbb{R}^+, and\ q_{C_j} = 1 \end{gathered}$$

hence,

$$\zeta_{A_{iG}} \geq \zeta_{A_i}$$

Finally, from Lemma 3 and $\zeta = \sum_{i=1}^{N} \zeta_{g_i}$, the theorem follows. □

S2.4 Compilation flow for CP (critical-path)/Declustering heuristics and including the modified StreamIt judicious data-replication heuristic – continued from experimental section

Figure 6 gives the optimization flow that we have implemented in our heuristic techniques. We introduce a modified version of the StreamIt judicious data/task parallelism heuristic as described by Gordon et al. 2006 in declustering and critical path scheduling. This heuristic is applied before applying the declustering and critical path scheduling algorithms.

References

[1] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise, and W. Yi. Modeling and verification of parallel processes. chapter UPPAAL: now, next, and future, pages 99–124. Springer-Verlag New York, Inc., New York, NY, USA, 2001. ISBN 3-540-42787-2. URL `http://portal.acm.org/citation.cfm?id=766794.766799`.

[2] G. Behrmann, M. Hendriks, and A. Mader. A.: Production scheduling by reachability analysis - a case study. In *In: Workshop on Parallel and Distributed Real-Time Systems (WPDRTS), published by IEEE Computer*. Society Press, 2005.

[3] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGOPS Oper. Syst. Rev.*, 40:151–162, October 2006. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1168917.1168877. URL `http://doi.acm.org/10.1145/1168917.1168877`.

[4] A. Malik and D. Gregg. Theorems for model-checking. Technical report, Department of Computer Science and Statistics, Trinity College Dublin, 2012.

[5] P. S. Roop, S. Andalam, R. Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis of synchronous C programs. Technical Report 0912, Christian-Albrechts-Universitat Kiel, Department of Computer Science, May 2009.

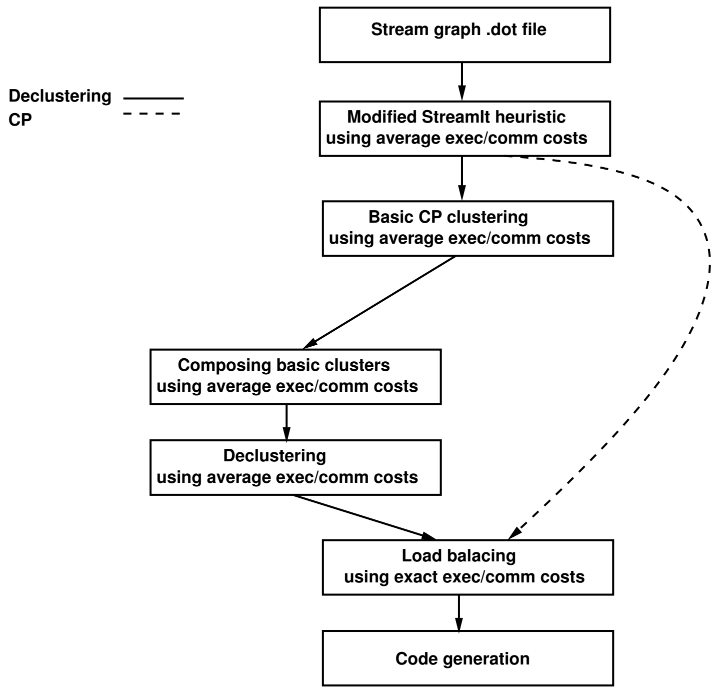

Declustering
CP
Stream graph .dot file
Modified StreamIt heuristic using average exec/comm costs
Basic CP clustering using average exec/comm costs
Composing basic clusters using average exec/comm costs
Declustering using average exec/comm costs
Load balacing using exact exec/comm costs
Code generation

Figure 6: Flow of heuristic optimization techniques