Executing synchronous data flow graphs on heterogeneous execution architectures using integer linear programming

Avinash Malik
School of computer science and statistics
Trinity College Dublin, Ireland

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

Abstract

This paper presents an integer linear programming (ILP) technique to partition and schedule *Synchronous Data Flow* (SDF) graphs onto heterogeneous execution architectures. Our ILP formulation gives a partition and schedule for SDF graphs, which provide the optimal execution time. We incorporate new methods of exploiting stateless data-parallelism on a heterogeneous architecture. We quantitatively show that our ILP formulation performs better compared to the current state of the art heuristic techniques available, distributing SDF graphs. In fact, our ILP formulation gives an execution boost ranging from 15% to 70% for heterogeneous architectures. Finally, in this paper we also explore a new optimization technique based on actor granularity, which further improves the overall throughput of the SDF graph.

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

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

The streaming model, where data continuously flows from the input to the output of the machine is increasingly becoming a common place programming approach. A plethora of real world applications, from complex real-time sensor networks to financial trading with very large amounts of data, adhere more closely to the streaming rather than the von Neumann model. Applications processing large quantities of data suffer from memory access bottlenecks, which stream computing can overcome by allowing sequential processes called actors to be dependent only upon their individual local copies of data, running in parallel, working together by passing data using *First In First Out* (FIFO) channels to achieve a given task. The streaming model exposes the concurrency in programs for exploitation.

The recent proliferation of multi-core and distributed systems, has lead to a resurgence in research in scheduling stream programs [3, 5, 11, 15, 18]. The current literature has two major drawbacks. First, most existing solutions do not take account of heterogeneous computer architectures, such as IBM's *Roadrunner* super-

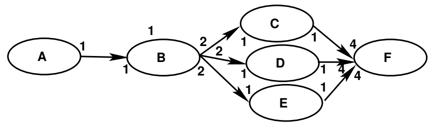


Figure 1. An example stream program

computer (which consists of a heterogeneous mix of conventional AMD multi-core processors and IBM's 9-core CELL processor [9]) and STMicro's P2012 [1] many core platform for embedded systems. Secondly, most existing approaches do not deal well with communication costs between processors, something which is very important for larger stream computers where communication costs may be significant. The result is that programmers who want to use the streaming model end up hand coding or manually partitioning the applications in order to exploit the parallelism of the underlying platform. The task of efficiently and automatically coordinating heterogeneous parallel processors is a major challenge.

Figure 1 shows a pedagogical example stream program. There are six actors, each connected to the other via FIFO channels. Actor `A` produces tokens that are consumed by actor `B`. Actor `B` duplicates these tokens and passes them onto actors `C`, `D`, and `E`. Finally, these actors after carrying out some transformations on the received tokens, send them onto the merge actor `F`, which joins the incoming tokens in a round robin manner. A single execution of the stream from `A` to `F` is called an iteration. The semantics of any stream application require that the state, number of tokens, of the buffers connecting the actors remain stable before and after an iteration, and hence, we term it a *stable state iteration*. The input/output channel ports for each actor are typed and annotated with the number of tokens produced (output rate of an actor) and consumed (input rate of an actor) for every invocation of that actor. If the number of tokens produced and consumed remain static throughout the life time of the application, we call such a stream application a *static stream application*, sometimes also termed a *synchronous data flow* (SDF) graph [12]. Finally, the schedule length, time required to complete a single stable state iteration, is termed *makespan*. Our overall goal in this paper is to automatically partition and statically schedule a SDF graph onto resource constrained heterogeneous execution architectures in order to obtain a minimal makespan. For well formed conditional constructs, implemented using `split` and `merge` nodes, we assume *dummy tokens* being sent across the unselected branches as described in [6].

A cornucopia of work exists [3, 5, 7, 11, 14, 18] that attempts to partition and statically schedule SDF graphs on multi-processor execution architectures. Almost all of the current state of the art [5, 7, 11, 14, 18] is dedicated to statically scheduling SDF graphs on homogeneous execution architectures, i.e., architectures where execution time of actors on different processors does not vary. Most of these attempts [5, 7, 11, 18] also do not consider time to communicate across different processors. A very recent attempt [3] made at incorporating a heterogeneous execution and communication architecture targets load balancing rather than a reduced makespan, which in turn also leads to the possibility of reduced throughput.

Another very important aspect that is unanswered in the current literature is the extraction of stateless data-parallelism from an SDF graph executing on a heterogeneous execution architecture. The current approaches [5, 7] incorrectly partition and replicate stateless actors without any concern for the varying execution times on different processors.

In this paper, we propose a new approach to partitioning and statically scheduling SDF graphs on resource constrained heterogeneous execution and communication architectures, which overcomes the aforementioned drawbacks. A more detailed comparison of our approach with the related work is provided later in Section 7. Our major contributions in this paper are:

1. To provide the very first *Integer Linear Programming* (ILP) formulation that:
    - Attempts to partition and statically schedule cyclic/acyclic SDF graphs onto resource constrained *heterogeneous* execution architectures to provide the *optimal* makespan and throughput.
        - Our approach finds a distribution and static schedule of SDF graphs on architectures where execution time of an actor varies across processors.
        - Our approach finds a distribution and static schedule of SDF graphs on architectures where communication time between actors allocated on different processors varies.
2. Our ILP formulation handles stateless data-parallelism so as to provide an optimal makespan.
3. We are the first to consider both multi-core and networked distributed systems and provide a *granularity* based optimization technique specific to networked systems.

The process allocation and scheduling problem tackled in this paper is NP-hard [5]. Nonetheless we believe it is worthwhile to develop techniques for computing the optimal makespan for heterogeneous architectures for two reasons:

- There is always a class of embedded applications where increased compilation time is worthwhile, when compared with increased developer time, or failing to meet performance or power requirements.
- There is currently a lack of good heuristic approaches to scheduling stream graphs for heterogeneous architectures and inter-processor communication costs. The optimal makespan obtained using our approach can be used as a benchmark against which good heuristic approaches targeting heterogeneous execution architectures can be developed and compared.

The rest of the paper is arranged as follows: Section 2 gives an intuitive description of the scheduling and partitioning of SDF graphs. In Section 3 we formalize the problem and describe our first, partitioning ILP formulation. We describe granularity based optimization in Section 4. Section 5 describes our compilation strategy and the resultant tool chain. Experimental results are presented in Section 6. Section 7 compares our work with the current state of the art. Finally, we conclude the paper in Section 8.

2. The problem explained – scheduling and partitioning SDF graphs

Before formalizing the problem statement, we present an intuitive description of the problem and related concepts.

2.1 Finding the granularity of actors

The SDF graph in Figure 1 needs to be scheduled for every stable state iteration. A number of algorithms have been proposed in the literature [3, 7, 13, 18], which schedule SDF graphs with static buffer sizes and static number of invocations for each actor. These algorithms use the notion of *balance equations* for allocating buffer sizes and scheduling a single stable state iteration. For example, in Figure 1 actor `B` requires one token at its input for invocation. Actor `A` on the other hand also produces only one token per invocation, and hence both are balanced. On the other hand, actors `C`, `D`, and `E`, only consume one token per invocation, but `B` produces two tokens. Thus, for every invocation of `B`, actors `C`, `D`, and `E`, need to be run twice. Same can be said for the edges connecting actors `C`, `D`, `E` and the join actor `F`.

$$\begin{array}{ll} A = B & 2B = C \\ C = 4F & 2B = D \\ D = 4F & 2B = E \\ E = 4F & \end{array} \tag{1}$$

The so called balance equations in Equation (1) relate the input/output rates of actors and determine the *minimal* number of invocation of any actor within the SDF graph for a single successful stable state iteration. The result of solving the balance equations is the vector $\mathbf{G} = \{2, 2, 4, 4, 4, 1\}$. Vector $\mathbf{G}$ is called the repetition vector and each element G_i gives the *natural granularity* of actor i in the SDF graph. All cycles in a SDF graph need delay tokens, sometimes also termed as initial tokens to be well formed. Any cyclic SDF graph without delay tokens cannot be scheduled and is considered incorrect by construction.

2.2 Partitioning SDF graphs

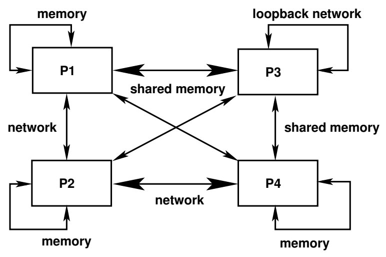


Figure 2. Example heterogeneous execution architecture and execution traces

Figure 2 shows an example heterogeneous execution architecture with four processors: `P1`, `P2`, `P3`, and `P4`. Processors `P1`, `P3`, `P3`, `P4`, `P2`, `P3`, and `P1`, `P4` are connected via shared memory. Processors `P2`, `P4` and `P2`, `P1` are connected via a network cable. All processors, except `P3`, use memory for storing data, `P3` uses loopback network and message passing.

Let actors `C` and `D` be stateless actors, i.e., actors where any invocation is independent of any other invocation of that actor. The example below shows a stateless actor.

```
while(counter<4){ i[c]  = j[c] + 1; c++;}
```

In the example loop above, the output `i` only depends upon the incoming input `j`. Thus, instead of carrying out the above computation four times in a sequence, we can parallelize the actor

(a) The optimal makespan obtained using our ILP formulation

	0	1	2	3	4	5	6	7	8	9
P1	A	A	B	B	C1	C2	C3	D3	D4	F
P2					D1	D1	D2	D2		
P3					C4	C4	C4			
P4					E	E	E	E		

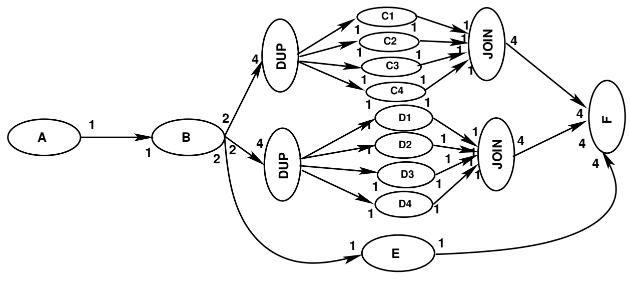
(b) The best makespan obtained using state of the art StreamIt heuristic [7]

	0	1	2	3	4	5	6	7	8	9	10
P1	A	A	B	B	CD1	CD1	CD4	CD4			F
P2					CD2	CD2	CD2	CD2			
P3					CD3	CD3	CD3	CD3	CD3	CD3	
P4					E	E	E	E			

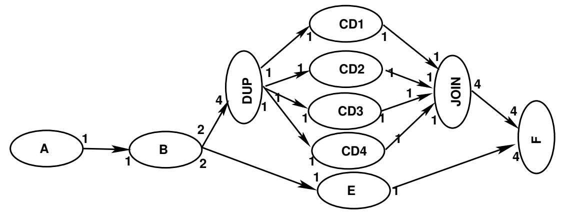
Table 1. Sample execution trace

into four different actors, each working on a separate part of the vectors `i` and `j`, respectively, once, and then joining them back together to decrease the overall makespan.

Figure 3(a), shows the loop level stateless parallelism being extracted from the SDF graph in Figure 1. The stateless actors are replicated processor (core) number of times. The granularity and input/output data-rates are adjusted accordingly [16]. Such loop level stateless parallelism is called naive stateless actor replication. Such naive stateless actor replication can cause communication bottlenecks and loss of task-parallelism (split/join nodes) in the SDF graph. In order to overcome this difficult, the StreamIt compiler introduces a judicious fusion-fission algorithm (a heuristic), which fisses stateless data-parallel actors without forgoing the task-parallelism inherent within a SDF graph.



(a) Extracting loop-level stateless data-parallelism



(b) StreamIt's approach to judicious fission

Figure 3. Modified SDF graph with stateless data-replication

The state of the art StreamIt heuristic [7], is a multi-step process, it first identifies the stateless actors (using simple static analysis). Next, it greedily fuses stateless actors together as much as possible. This fusion step might even fuse complete stateless split/join nodes and finally, it uses a simple heuristic (considering the fraction of work done by the stateless fused actor compared to the overall work done by the complete split/join) to fiss the stateless actor. The result of applying StreamIt heuristic (as is, i.e., without accounting for heterogeneity) to our SDF graph in Figure 1 is shown in Figure 3(b).

The execution architecture is heterogeneous. Let the execution time for the actors be given by the function ξ: $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}\} \rightarrow \{1, 1, \{1, 2, 3, 4\}, \{1, 2, 3, 4\}, 1\}$. From these sets we can see that actors `C` and `D` perform differently on the four available processors, while the others complete execution in the same time. We consider the communication time between processors to be 0 time units, purposely, since StreamIt heuristic is unable to handle communication times. We will revisit this state of the art heuristic and describe a new heuristic technique combining StreamIt and declustering [14] in Section 5.1, in order to handle stateless data-parallelism involving communication times.

Table 1, gives the optimal execution trace (Table 1(a)) obtained by our ILP formulation compared to the best possible execution trace that the StreamIt heuristic can produce. The schedule produced by our ILP formulation is 10% faster. More importantly, we have purposely chosen to allocate `CD1` and `CD4` on processor `P1` to get the best possible solution. According to the proposed heuristic in [7], the makespan of 17 time units, obtained by allocating `CD4` to on processor `P4` is as good as the execution trace in Table 1(b), because StreamIt heuristic assumes homogeneous computation times.

We have simplified the above example considerably by homogenizing the computation time for most of the actors and getting rid of the varying communication costs. Yet, the partitioning and scheduling solution obtained in Table 1(a) is non-obvious, due to the slight heterogeneity introduced in the execution architecture.

From Table 1, the following observations become essential:

- A partitioning and scheduling strategy, needs to consider communication *and* computation time for actors.
- A partitioning and scheduling strategy should be able to accommodate heterogeneity.
- A partitioning and scheduling strategy should be able to exploit stateless data-parallelism and task-parallelism considering both communication costs and heterogeneity.

Finally, we would like to reiterate that we purposely are trying to provide an optimal solution to this NP-hard problem. Consider that there are a myriad heuristics for partitioning and scheduling on homogeneous execution architecture [5, 7, 11, 14]. A heterogeneous execution architecture exponentially complicates the search space. There are a number of solutions, but the space for good solutions is much smaller. This in turn leads to complex heuristic approaches – how does one know that a heuristic targeting heterogeneous execution architecture is a *good* heuristic, if one does not know what is the optimal solution?

3. Partitioning and scheduling SDF graphs

3.1 Definitions

A SDF application is a graph $\mathcal{G}(V, E)$, where V are the vertices representing the actors and $E \subseteq V \times V$ are the edges representing the FIFO communication channels between actors. Let graph $\mathcal{A}(P, C)$ represent the heterogeneous execution architecture, where P represents the processors available for actor execution and $C \subseteq P \times P$ represents the communication link between processors. Finally, let Π represent the makespan of the schedule.

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

Proof. Let ω_i represent the number of bytes produced for each invocation of some actor $V_i \in V$. Thus, for a single stable state iteration of $\mathcal{G}$, the number of bytes produced by actor V_i is $\omega_i G_i$, recall that G_i is the natural granularity of actor V_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_i as:

$$\zeta_i = \frac{\omega_i G_i \mathcal{N}(\Pi)}{T} \ \forall i = \{1 \dots N\} \tag{2}$$

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:

$$\zeta = \sum_{i=1}^{N} \zeta_i \tag{3}$$

As we can see from Equation (2), the throughput and makespan are inversely proportional. Hence, minimizing makespan (Π) is equivalent to maximizing throughput (ζ).

$\square$

3.2 Integer linear programming formulation to partitioning SDF graphs for makespan

In this section we provide the ILP formulation that partitions the SDF graphs on heterogeneous resource constrained architectures to obtain the optimal makespan (Π).

$$\begin{array}{cc}
\text{objective} & \text{minimize}(\Pi) \\
\text{subject to} & \\
L1 & \forall (i,j) \in E \ \ a_j \geq a_i + \sum_{\forall x \in P} (b_{ix}.T_{ix}) + \sum_{g=1}^{M} \sum_{\forall (x,y) \in C} (b_{ixgy}.D_{xy}) \\
L2 & \forall k \in V_t \ \ \Pi \geq a_k + \sum_{\forall x \in P} (b_{kx}.T_{kx}) + \sum_{g=1}^{M} \sum_{\forall (x,y) \in C} (b_{kxgy}.D_{xy}) \\
L3 & \forall k \in V_s \ a_k \geq 0 \\
P1 & \forall x \in P \ \ \sum_{\forall i \in V} b_{ix} = 1 \\
P2 & \forall i \in V, \ \forall j \in V, \ \forall x \in P \ \ s_{ij} = max(|b_{ix} - b_{jx}|) \\
P3 & \forall (i,j) \in E, \forall (x,y) \in C \ \ b_{ixjy} \geq b_{ix} + b_{jy} - 1 \\
S1 & \forall i \in V \ \forall j \in V \ r_{ij} + r_{ji} = 1 - s_{ij} \\
S2 & \forall i \in V \ \forall j \in V \ a_j \geq a_i + \sum_{\forall x \in P} (b_{ix}.T_{ix}) + \sum_{g=1}^{M} \sum_{\forall (x,y) \in C} (b_{ixgy}.D_{xy}) - M_{\infty}(1 - r_{ij}) \\
 & \forall a_i \geq 0, \forall i \in V, \forall x \in P, \ T_{ix} = E_{ix} * G_i, E_{ix} \in \mathbb{N}^* \\
 & where E_{ix} \text{ is i's execution time for a single invocation}
\end{array} \tag{4}$$

Equation (4) gives the overall ILP formulation used to partition the SDF graph on a resource constrained heterogeneous execution architecture. We will go through each constraint describing them in detail.

3.3 Constraints

Resource constraints Each actor is allocated to exactly one processor. Note that for the purposes of our formulation we consider a processor to be any computing device that can execute an actor. For example, a "processor" may be a single core of multi-core processor. Or it may be a part (or whole) of an FPGA that can be programmed to perform the work of some actor. Or it may even be a GPU which can be devoted to executing an actor (with the actor implemented as a GPU kernel with many small threads of control internally, but treated as a single execution from the point of view of the formulation). The important thing is that an actor can be allocated exclusively to the "processor", and that no other actor executes on the same "processor" simultaneously. For purposes of illustration, it is probably easiest to think of a "processor" as being a single core of a multi-core processor.

One limitation of our approach is that it does it does not model simultaneous multi-threading (SMT) well. SMT, which allows multiple threads of execution to share the resources of a single hardware core, can be modelled badly in our formulation. Each SMT hardware thread can be treated as a separate "processor" in the same way as operating systems such as Linux do. However, there is usually significant competition for resources among the hardware threads in an SMT core. Our experience has been that the execution times of actors varies significantly depending on what is running on the other hardware threads of an SMT core. For this reason, we do not exploit the SMT capability of the machines we use in our experiments, and instead leave the more accurate modelling of SMT cores to future work.

We define a $0-1$ integer variable b_{ix} to represent the allocation of actor $i \in V$ on processor $x \in P$. Thus, the constraint $\forall x \in P \sum_{\forall i \in V} b_{ix} = 1$, $P1$ in Equation (4), guarantees that each actor is allocated to exactly one processor.

Actors allocated to the same processor need to be identified in order that only a single actor is executing on a given processor at any point in time. We identify actors allocated on the same processor with a $0-1$ integer variable $s_{ij}, \forall i \in V, \ \forall j \in V$. Thus, the constraint $P2$ in Equation (4), $\forall x \in P, s_{ij} = max(|b_{ix} - b_{jx}|)$, gives a value of 0 or 1, for actors i and j being allocated to same or different processors, respectively.

Finally, since in a heterogeneous architecture the communication cost between actors allocated to different processors varies, we need to identify, which communication link will be used. We define yet another $0-1$ variable called b_{ixjy}, for actors i and j, allocated to processors x and y, respectively. Note that x and y might be the same processor. Thus, the constraint $P3$ in Equation (4), $b_{ixjy} \geq b_{ix} + b_{jy} - 1$ gives a value of 1 or 0 if the communication link $(x,y) \in C$ is utilized or not, respectively.

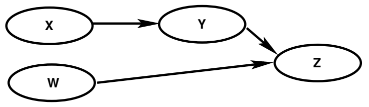


Figure 4. Example SDF graph

Scheduling constraints Consider the very simple SDF graph in Figure 4, allocated to a two processor system. Suppose that allocating actors `X` and `W` on one processor and `Y` and `Z` on another processor gives the smallest makespan. In such a scenario, the execution order of actors `X` and `W` on a single processor needs to be decided since that affects the overall makespan. We define this ordering as a scheduling constraint, $S1$ in Equation (4), for all actors on any given processor. Two $0-1$ integer variables r_{ij} and r_{ji} define the execution order of actors i and j, respectively. In the constraint, $r_{ij} + r_{ji} = 1 - s_{ij}$ the ILP solver is allowed to set either r_{ij} or r_{ji} to 1 in order to find the best schedule order. Notice that this is the only constraint where variable s_{ij} is useful, one can get rid of the s_{ij} variable and use b_{ix} instead, but we keep the s_{ij} variable for ease and clarity of understanding. Also, dependencies in the SDF graph are considered when ordering actors on a single processor. Hence, while the ILP solver is allowed to order actors `X` and `W` in any order it sees fit when both are allocated to the same processor, actors `X` and `Y` are ordered one after the other provided they are scheduled on the same processor.

Every actor in the SDF graph has a start or activation time. We identify the activation time for an actor by the integer variable $a_i, \forall i \in V$. The actors in the graph can be divided into two distinct sets; source actors, identified by the set V_s and terminal actors identified by the set V_t. All actors in the source set are considered to have an activation time of 0 (e.g., $a_x = a_w = 0$ in Figure 4). On the other hand, the activation time for actors in the terminal set is affected by the schedule of the graph.

Since the execution of all actors on a single processor is serialized, consequently, the activation time for actors is delayed by this serialization. Consider the aforementioned scenario of actors `X` and `W` being scheduled on the same processor. Suppose, variable r_{xw} is set 1, then the activation time for actor `W` is delayed by the computation and communication time of actor `X`. We represent this delay in activation time by scheduling constraint $S2$. Thus, the activation time for actor `W` according to constraint $S2$ is given by:

$$\begin{gathered} a_w \geq a_x + b_{xp1}.T_{xp1} + b_{xp2}.T_{xp2} + b_{xp1yp1}.D_{p1p2}+ \\ b_{xp1yp2}.D_{p1p2} + b_{xp2yp1}.D_{p1p2} + b_{xp2yp2}.D_{p1p2}- \\ M_{\infty}.(1 - r_{xw}) \end{gathered}$$

Where, $a_x = 0$ is the activation time for actor `X`, T_{xp1} and T_{xp2} are the computation time for actor `X` on processors `P1` and `P2`, respectively. The $0-1$ integer variables b_{xp1}, b_{xp2}, b_{xp1yp1}, b_{xp1yp2}, b_{xp2yp1}, b_{xp2yp2}, represent the allocation of actor `X` on processor `P1` or `P2` and the use of communication link between processors `P1` and `P2`, respectively, when communicating between actors `X` and `Y`. The integer variable D_{p1p2} gives the varying communication time when differing communication links are used. Finally, M_{∞} represents a fairly large integer number. Notice that a_w is only affected if r_{xw} is one, else it remains unaffected. Supposing that actors `X` and `W` are both allocated on processor `P1` and the computation and communication time for actor `X` is 1, we can solve the above equation to get:

$$\begin{gathered} a_w \geq 0 + 1.1 + 0 + 0 + 1.1 + 0 + 0 - 0 \\ \Rightarrow a_w \geq 2 \end{gathered}$$

Dependency constraints Except for the scheduling and resource constraints, one also needs to consider the dependency constraints in the SDF graphs. As stated previously all actors have an activation time: $a_i, \forall i \in V$, and there are two sets of actors, source actors V_s and terminal actors V_t. We consider the activation time for all actors in the source set to be zero, denoted by the constraint $L3$ in Equation (4). The makespan (Π) is determined by the addition of the activation time of all actors in the terminal set and their individual computation and communication times. This constraint is defined by $L2$ in Equation (4).

In our ILP formulation we need to consider the delay in activation time for some actor $i \prec j$ (a_j) due to an edge ($(i, j) \in E$). We do not implement a software pipelined schedule. Software pipelining can be easily incorporated in our framework by ignoring all edges, except for the strongly connected components or cycles in the SDF graph. Instead, we concentrate on exploiting stateless data and task parallelism in this paper. The delay in the activation time for any given actor due to an edge dependency is shown by constraint $L1$ in Equation (4). The activation time for some actor j is delayed by the computation and communication time of actor i, $i \prec j$ and edge $(i, j) \in E$. Constraints $L1$, $L2$, and $S2$ are similar, because they all calculate delays in activation time for actors.

Optimal exploitation of stateless data and task parallelism Let us revisit our original example in Figure 1 and its stateless data-parallel actor replication shown in Figure 3(a). Our ILP formulation does not need any changes to accommodate optimal stateless data and task parallelism. Constraints, $L1$ and $S2$ together guarantee that the optimal solution is found. For example, constraint $L1$ leads to the placement of actors `C1` and `C4` on processors `P1` and `P3`, respectively, while constraint $S2$ leads to the delay in the activation times for actors `C2` and `C3` when placed on processor `P1`. Same can be said for stateless actor `D`. Thus, the resultant execution trace in Table 1(a).

If we consider the stateless actor `C`, then three copies of `C`: `C1`, `C2`, and `C3` are run on processor `P1` and the fourth copy is run on processor `P3`. This partitioning can also be restated as: actor `C` is split into two copies with the first copy running on processor `P1` with a granularity of 3 and the second one running on `P3` with a granularity of one. Similarly, `D` is split into two copies, allocated onto processors `P1` and `P2` and each copy runs with a granularity of 2. This observation inherently depends upon the fact that every copy of a stateless actor is equivalent. By restating the stateless actor replication we get the final graph, which is implemented by our compiler as in Figure 5.

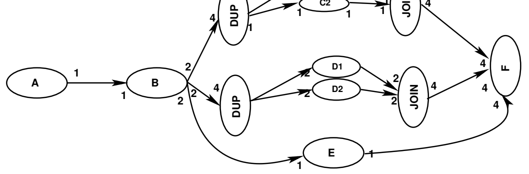


Figure 5. Optimal stateless data and task parallelism exploited by our ILP formulation

4. Granularity based optimization

Our optimal data and task parallel exploitation approach does not coarse the granularity of actors (by greedy stateless actor data fusion) explicitly like in StreamIt. Instead we explore a new optimization technique: *granularity based optimization*. The basic premise of this technique is to increase the granularity of actors in the SDF graph in integer multiples provided the throughput increases. This technique is specifically targeted at network systems, because of the nature of communication flow.

Granularity based optimization technique and exploiting stateless data-parallelism are complimentary optimization techniques as we will show in this section. Granularity based optimization techniques should always be applied after exploiting stateless data-parallelism, especially if exploiting stateless data-parallelism gives worse results than the original graph.

Consider the SDF graph in Figure 6(a). Consider that actor `A`'s computation time is one time unit and communication time is three time units. Similarly, assume that computation time for `B` is one time unit and communication time is five time units. Without loss of generality assume that these runtimes are valid for all processors and communication links in Figure 2.

If actor `A` is a stateless actor, one can replicate the four invocations of actor `A` into four independent actors as shown in Figure 6(b). We term this restructuring of the SDF graph as homogenization. In a homogenized SDF graph, all actors have a natural

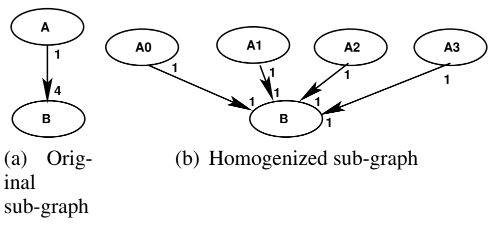


Figure 6. Sub-graph from Figure 1 and its homogenized version

granularity of 1. The execution trace of the two SDF graphs in Figure 6(a) and 6(b) on the four processor system in Figure 2 is shown in Table 2.

Table 2(a) gives the makespan the SDF graph in Figure 6(a), while Table 2(b) gives the execution trace for the homogenized version in Figure 6(b). As we can see from the two tables, the non-homogenized version has a makespan of 13 time units, while the homogenized version has a makespan of 19 time units. The four separate copies of actor A execute for one time unit on each of the processors and then communicate with the join actor B. All four copies cannot communicate at once, because processors (in this case P1) are single threaded entities. Even with increased utilization of resources (4 processors in Table 2(b)) the homogenized SDF graph has a longer makespan. In fact, the exploitation of data-parallelism has lead to a increase in makespan by $\approx 31\%$. The main reason for this disparity is the communication costs. In the non-homogenized version, the communication between A and B takes place only once, at the end of the four individual invocations of actor A, whereas in the homogenized version, data is sent after every invocation of the replicated actor.

If we assume that the communication costs remain constant for a range of bytes being transferred across processors, then we can increase granularity, which leads to increased throughput and reduced makespan. When the granularity is increased and if the communication costs remain constant, more work is done by the actors, because increasing granularity increases the number of actor invocations, which in turn leads to production of more tokens in less amount of time. For the example in Figure 6, increasing granularity by a multiple of 10, leads to a makespan difference of 21 time units, between the makespan of the homogenized and non-homogenized SDF graphs in the favor of the homogenized SDF graph, i.e., an improvement of $\approx 67\%$. Increasing granularity 20 times, makes the difference larger still, at 51 times units in favor of the homogenized SDF graph. Thus, one can search for an optimal granularity of the SDF graph for a given execution architecture.

Our granularity based optimization technique assumes a constant communication time for a range of bytes $[\omega_{lb}, \omega_{ub}]$, i.e., the communication is modeled as a step function. Communication links in networked systems exhibit stepwise increase in communication costs. Network protocols such as TCP/IP send packets with minimum and maximum bounds. We utilize these characteristics of networked systems by increasing the granularity of the computation actors, while maintaining the granularity of communication as 1 to increase the overall graph throughput.

This optimization technique *may* be applicable to an architecture of *only* a single chip multi-core system depending upon the model of communication employed. If the communication model assumes data transfer by copy, then the communication costs might grow linearly with increasing number of bytes. A detailed analysis of the target architecture is required to determine the communication costs in a data transfer by copy model, which we do not tackle in this paper. If a shared memory-based communication protocol is assumed, the main communication cost is often the cost of synchronizing access to shared buffers. In this case, the communication cost may not vary significantly with the amount of data sent, and can be treated as a constant. In this paper we assume a shared-memory based communication model for the system on chip architecture and hence, run *only* multi-core systems at the highest possible granularity, bounded by only the bandwidth of the communication fabric. In distributed memory heterogeneous architectures (the main focus of this paper) the network communication costs far outweigh the system on chip communication costs and the granularity is bounded by these costs.

This granularity based optimization technique applied to our ILP framework leads to two important results.

1. *Our approach gives a value of the granularity, $G_i, \forall i \in V$, which provides the optimal makespan*. We iteratively increase the granularity of computation actors, by an integer multiple and input it into the ILP solver until we obtain a maximum fixed point. Since the granularity of all actors in a SDF graph is related and needs to be increased by the exact same multiple, the final result is guaranteed to be the optimal solution. This approach allows us to optimize the granularity generally.
2. *Searching for an optimal makespan by increasing granularity is linear in time*. The increase in granularity is carried out in integer multiples greater than or equal to 1. Increasing the granularity of a single computation actor by some multiple requires an increase in the granularity of all other computation actors with the exact same multiple, because of the balance equations and semantics of the SDF graph.

5. Generating ILP formulations from StreamIt

We have used the StreamIt [17] language as a representative from amongst a plethora of other stream languages [2, 4, 15] due to ease of use and familiarity, but our approach is language agnostic. Our compilation procedure is shown in Figure 7.

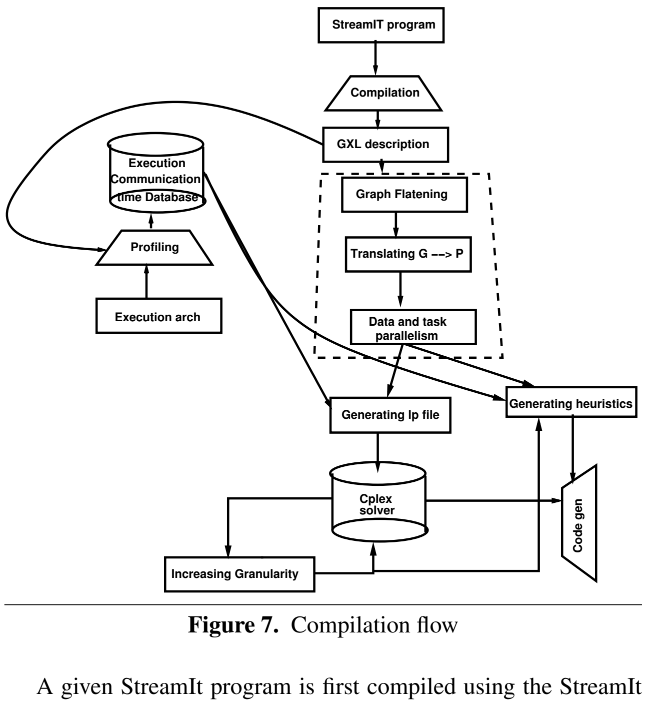


Figure 7. Compilation flow

A given StreamIt program is first compiled using the StreamIt compiler. We use a modified StreamIt compiler, from the StreamIt subversion repository. Our modifications disable task and data parallel heuristics. Instead, we apply a modified set of heuristics,

(a) Execution trace 1

P1	A	A	A	A	A	A	A						
P2								B	B	B	B	B	B

(b) Execution trace 2

P1	A0	A0	A0	A0										B	B	B	B	B	B
P2	A1				A1	A1	A1												
P3	A2							A2	A2	A2									
P4	A3										A3	A3	A3						

Table 2. Execution traces for Figure 6 on the execution architecture in Figure 2

which will be shown in this section. The StreamIt compiler produces a "dot" file. We translate the "dot" file into an XML based representation called "GXL" (*Graph eXchange Language*) [10]. All our tools work on this intermediate format and hence, any language that can be translated into this format can use our tool for partitioning and scheduling. Every actor in the compiled StreamIt program is separately executed and profiled on the target execution architecture, using *oprofile*. The communication time between processors is profiled by sending varying sized data packets. A database is built using these profiled times.

The GXL file is used as an input into our tool chain, shown by the dotted trapezoid in Figure 7. The SDF graph is first translated into a precedence graph, which makes the communication actors explicit. Figure 8, shows the precedence graph generated from the SDF graph in Figure 1.

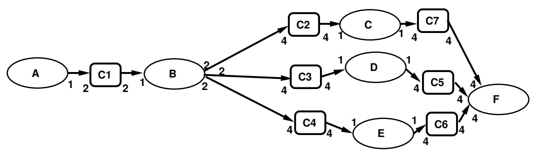


Figure 8. Precedence graph for the SDF graph in Figure 1

Communication costs play an important role in the makespan minimization problem. For the sake of uniformity we translate the FIFO channels into communication actors. The translation of $\mathcal{G}$ into a precedence graph results in a new graph $\mathcal{P}$ where the FIFO channels are made explicit. As we can see from Figure 8, the communication actors have their input and output data rates calculated by looking up the natural granularity of their respective source actors. From Section 2.1, we know that the natural granularity for the computation actors in $\mathcal{P}$ is given by $\{2, 2, 4, 4, 4, 1\}$. Hence, for the communication actor C1, its input and output rates are 2×1, because its source execution actor, A, has a natural granularity of 2 and an output rate of 1 token per invocation, same for the others.

Intuitively, the communication actor's input and output rates represent the size of the buffers used for sending data between actors for a single stable state iteration. Every communication actor is invoked only once for execution. During this invocation it passes all the data from its source to its target.

5.1 Heuristic algorithms

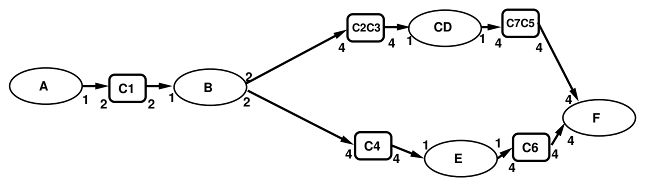
We compare our ILP formulation with heuristic techniques to gauge its effectiveness. As there are no readily available heuristic techniques that can be used for comparison, we need to use our own. We used modified versions of declustering [14] and critical path scheduling techniques for comparison. Both these techniques needed to be modified in order to account for heterogeneity of the execution architecture and judicious data/task parallelism. In order to account for heterogeneity, we consider the average computation/communication time for any given actor on the execution architecture. For example, from Section 2.2, we know that actor C' computation time is given by the set $\{1, 2, 3, 4\}$, for the execution architecture of Figure 2. Thus, for heuristics we consider the average, 5 time units, as the computation time for actor C. Introducing judicious data/task parallelism is a bit more involved. We introduce a modified version of the StreamIt judicious data/task parallelism heuristic as described in [7] in declustering and critical path scheduling. This heuristic is applied before applying the declustering and critical path scheduling algorithms. The modified StreamIt heuristic is shown in Algorithm 1.

ALGORITHM 1: Judicious exploitation of data and task parallelism – accommodating heterogeneous communication and computation

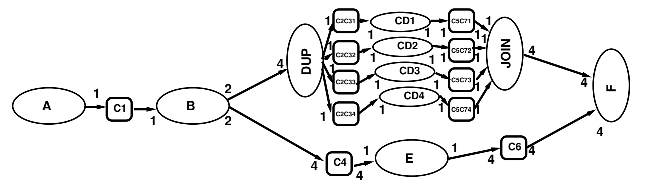
```
Input: Precedence graph $\mathcal{P}$, Number of processors $N$
Output: Modified precedence graph $\mathcal{P}$
▷ Greedily fuse stateless computation and communication actors in $\mathcal{P}$
set $S$ is all stateless actors in $\mathcal{P}$
▷ Estimate work done by F as a fraction of the complete work done
by all branches of split/join node
$fraction = 1.0$
for all $F \in S$ do
    Stream $parent \leftarrow getParent(F)$
    Stream $child \leftarrow F$
    while $parent \neq \emptyset$ do
        if $parent$ is split-join then
            $totalwork \leftarrow$
            $\Sigma_{c \in children(parent)}$AVGWORKPERFILTER$(c)$
            $mywork \leftarrow$ AVGWORKPERFILTER$(child)$
            $fraction \leftarrow fraction * mywork/totalwork$
        end
        $child \leftarrow parent$
        $parent \leftarrow getParent(parent)$
    end
end
▷ Fiss $F$ according to fraction
Fiss $F$ into CEIL$(fraction * N)$ filters
```

Figure 9(a), gives the result of greedy stateless actor fusion when applied to the precedence graph of the running example in Figure 8. The communication actors connected to stateless computation actors (C and D) are fused separately. It is essential that these actors be fused separately, in order to exploit communication links later on during declustering and critical path scheduling phases. The result of this greedy fusion is removal of task parallelism, which will be later reinstated as data parallelism.

The judicious fission algorithm is employed on this resulting precedence graph, the result of which is shown in Figure 9(b). We know that the communication time for actor C5C7 is 0 time units, from Section 2. The computation time for actor CD is 10 time units, which is the sum of the average computation time for actors C and D, 5 time units each, respectively. The total time for the split/join node is 11 time units. Thus, $fraction = 10/11$ and the total number of copies for actor CD is CEIL$(10/11 *$



(a) Modified precedence graph after applying greedy stateless actor fusion on the prcedence graph in Figure 8



(b) Modified precedence graph after applying judicious fission from Algorithm 1

Figure 9. Exploiting data and task parallelism judiciously

$4) = 4$. Hence, we split CD into 4 copies, which may execute on the 4 available processors, depending upon the declustering and critical path scheduling phases. Once this judicious data and task parallelism algorithm is applied the resultant precedence graph is then ready to be input into the declustering and critical path scheduling heuristics.

Our implementation of the declustering and CP scheduling algorithms further modifies the original ones. These modifications are essential to account for heterogeneity of the execution architecture. Without these modifications these heuristic perform abysmally. In our implementation, we assume the average computation and communication cost for all processors and communication fabrics during the clustering phase. During the final list scheduling and load balancing phases the real costs are considered. CP-scheduling approach permutes through all possible processor allocations to find the minimum possible makespan, declustering on the other hand fixes the processor allocation during the list scheduling phase. Thus, it is possible for the declustering algorithm to make poor allocation decisions (see [14] for more details). Finally, CP clustering does not exploit speedups in the sequential sub-graphs of the SDF graph, i.e., complete clusters are allocated to a single processor. Declustering approach only carries out partial load balancing (dependent upon edge nodes, see [14]). Like in the clustering phase, the original declustering does not consider varying execution costs of actors on different processors, we modified the load balancing algorithm to account for this heterogeneity.

6. Experimental results

We have carried out a number of experiments to gauge the effectiveness of our approach. Our experimental setup is shown in Figure 10. The setup is a heterogeneous execution architecture consisting of: 2.6GHz Intel Core2Duo processors with 4GB of random access memory (RAM), 2.6GHz newer Intel Core i7 processors with 8GB of RAM and finally, two different types of NVIDIA GPUs, the newer GTX 480 card with 700MHz graphics clock speed and 1.5GB of graphics memory and an older Geforce 320M clocked at 450MHz and with 256MB shared memory.

The speedup in makespan for a number of benchmark examples from StreamIt and three of our own; proportional-integral-differential (PID) controller, simple meeting scheduler (SMS), and the human tracking system (SS), which include complex cycles, is shown in Figure 11. The number of nodes in Figure 11 indicates the size of the benchmarks. All the numbers in Figure 11 are for natural granularity only, i.e., we haven't applied any granularity based optimizations. The increase in throughput due to granularity based optimizations is shown in Figure 11(c). Finally, the runtime, to find the optimal makespan solutions at natural granularity, using the cplex solver is shown in Table 3. The heuristic solutions are magnitudes faster compared to the cplex solver. This is to be expected, because the allocation problem is a NP-hard problem further exacerbated by the heterogeneity of the execution architecture.

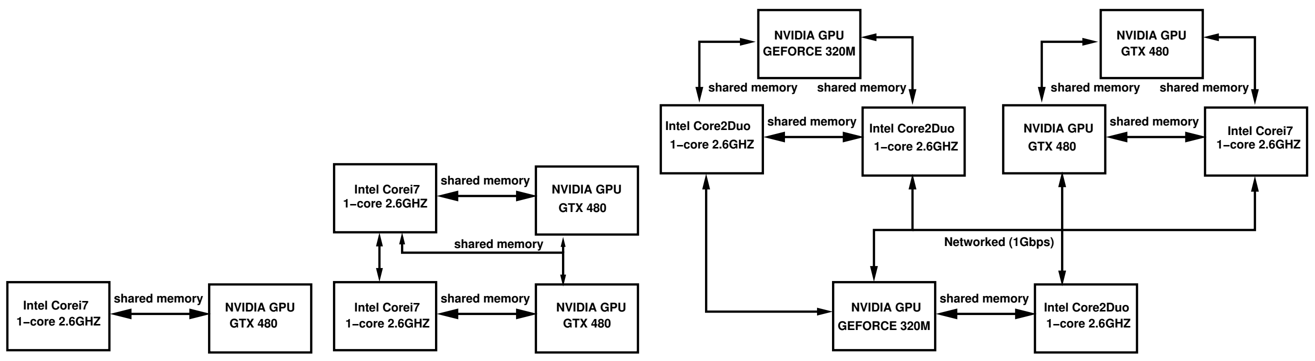
Benchmarks	**MILP**	**Declustering**	**CP**
FFT	1856	4	**3**
Audiobeam	22	9	**7**
Bitonicsort	26793	6	**3**
TDE	26984	**10**	**10**
Vocoder	38190	11	**8**
DES	5718	17	**12**
Mpeg3decoder	36	6	**3**
Radar	1837	8	**2**
SMS	2367	7	**4**
PID	1592	**2**	**2**
SS	89278	9	**6**

Table 3. MILP Vs model-checking solve times (sec)

Even with our modifications the heuristic solutions perform poorly compared to the optimal solution, with the difference ranging from 8.6% to 29% for the declustering algorithm and 10.8% to 26% for the CP algorithm. This poor performance of heuristics can be attributed to the fact that the heterogeneity is not considered *appropriately* in the judicious data and task parallel algorithm from StreamIt or the declustering and critical path scheduling algorithms. Even our modifications to StreamIt and declustering/CP cannot overcome the inherent deficiency in these algorithms. For example, the StreamIt heuristic with our modification creates 4 copies of the fused FFT/DFT actor *and* allocates them onto the 4 processors, thereby resulting in a makespan of 14 time units, that is a difference of $\approx$ 29% from the optimal (Table 1). In our experiments we have found that using the maximum or the minimum amongst all the computation/communication times for any given actor gives worse performance compared to using the average. So, what values should one use: the mean? How to compensate for outliers in the computation/communication times? All these questions are unanswered when considering applying the current heuristic techniques to heterogeneous architectures. These quantitative results reinforce our belief that our work is essential in further progressing research on compile time distribution and scheduling.

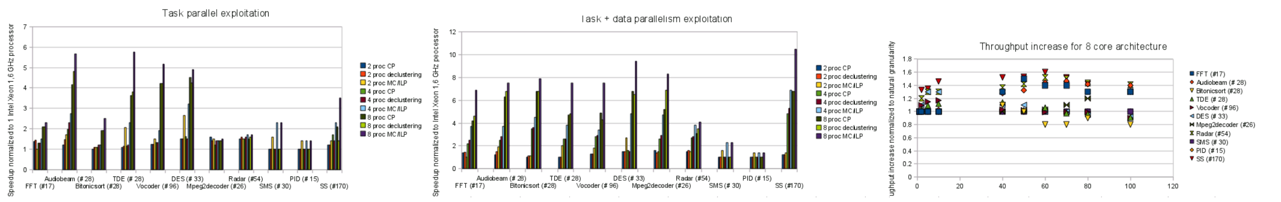
7. Related work

A large body of work exists for scheduling SDF graphs on varying architectures with varying goals. Farhad et.al. [5] propose a heuristic algorithm for partitioning a SDF graph onto multi-core homogeneous platforms in order to reduce the input token arrival rate for actors. Their heuristic technique does not consider a heterogeneous execution platform where actors might take different amount of time to execute on different platforms and also the time for communication between different processors in not accounted for. Also, their approach is compared with a bin-packing style ILP formulation, which does not model cyclic SDF graphs correctly, giving a under approximation of the makespan. Gordon et.al. [7] exploit parallelism for the RAW architecture [19] without considering the communication cost or the varying execution costs prevalent in a heterogeneous environment. Similar attempts have been made by Udupa et.al. [18] and Kudlur et.al. [11] with their individual ILP formulations targeting reduced makespan on GPUs and multi-core platforms, respectively, without consideration for communication



(a) 2 processor system (b) 4 processor system (c) 8 processor system

Figure 10. Experimental setup



(a) Makespan speedup, task parallel exploitation only (b) Makespan speedup, task and data parallel exploitation (c) Throughput increase due to granularity optimizations on 8 core architecture

Figure 11. Experimental benchmarks

costs and heterogeneity. We are able to handle complex cycles, encapsulating branches, they are unable to do so. The same authors have also previously attempted an ILP formulation for scheduling SDF graphs on general purpose processors [8]. [8] and [18] are related and inherit the same deficiency of ignoring complex cyclic SDF graphs. Another drawback of [8], is that it models resource unconstrained architectures, which is generally not the case. Finally, [18] reports sub-optimal makespan and buffer allocation, because their ILP formulation is not an optimization problem, rather it is a constraint problem.

The closest attempt to including heterogeneity is presented in [3], where stream programs are mapped to a heterogeneous execution platform and then load-balanced using heuristic techniques. Carpenter et.al. [3] also consider communication costs when applying their heuristic techniques. Yet, [3] provides an unsatisfactory result, because; (1) their formulation does not consider reducing the makespan of the SDF graph, rather they target load balancing actors on the architecture to equally utilize processor resources and (2) they consider convex graphs without cycles. Lastly, this approach is based on Kernighan's heuristic graph algorithm, and hence, proivdes a sub-optimal solution.

One related approach that targets reducing the makespan, while considering communication costs and targeting global reduction in makespan is presented in [14]. Sih et.al. [14] use a heuristic technique called "declustering". The declustering algorithm takes a SDF graph as input, carries out critical path based clustering algorithm to partition the graph into basic clusters most viable for partitioning. Next, it combines these clusters together to form a binary tree with leaves as the basic clusters. Finally, looking at the topology of the execution architecture the binary tree is declustered by allocating the clusters in the descending order, from most recently clustered to the basic ones, in the process allocating and list-scheduling clusters onto processors in order to obtain the smallest possible makespan. This approach again suffers from drawbacks such as lack of assumptions about differing communication and execution time for actors, and a non-optimal partition.

8. Conclusions and Future Work

In this paper we have described an *Integer Linear Programming* (ILP) approach to distribution and scheduling of *Synchronous Data Flow* (SDF) graphs. Our ILP formulation is able to accommodate task and data-parallelism in an optimal manner. Moreover, we have shown a granularity based optimization technique that compliments the task/data parallel exploitation and is well suited for networked systems. We have compared our ILP approach with modified declustering and critical-path scheduling heuristics. We have introduced modifications to these heuristics to account for judicious task/data parallelism. Moreover, we have also introduced new ways to account for heterogeneity of execution framework. Even with these introduced changes the heuristic solutions perform poorly compared to the ILP formulation, which gives the optimal solution. This work shows the need to research for better heuristics that would give good solutions compared to the optimal. Our ILP formulation can be used as a benchmark to develop these new heuristic solutions.

References

[1] L. Benini. P2012: A many-core platform for 10Gops/mm2 multimedia computing. In *Symposium on Rapid System Prototyping*, 2010.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23:777–786, August 2004. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1015706.1015800. URL `http://doi.acm.org/10.1145/1015706.1015800`.

[3] P. M. Carpenter, A. Ramirez, and E. Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *Proceedings of the 2009 international conference on Compilers, archi-*

tecture, and synthesis for embedded systems, CASES '09, pages 57–66, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-626-7. doi: http://doi.acm.org/10.1145/1629395.1629406. URL `http://doi.acm.org/10.1145/1629395.1629406`.

[4] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. Spidle: a DSL approach to specifying streaming applications. In *GPCE '03: Second International Conference on Generative programming and component engineering*, pages 1–17, New York, NY, USA, 2003.

[5] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Orchestration by approximation: Mapping Stream Programs onto Multicore Architectures. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 357–368, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. doi: http://doi.acm.org/10.1145/1950365.1950406. URL `http://doi.acm.org/10.1145/1950365.1950406`.

[6] G. R. Gao, Y.-B. Wong, and Q. Ning. A timed Petri-net model for fine-grain loop scheduling. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '91, pages 395–415. IBM Press, 1991. URL `http://portal.acm.org/citation.cfm?id=962111.962140`.

[7] 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`.

[8] R. Govindarajan, G. R. Gao, and P. Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing*, 31, 2002.

[9] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA'05: Proceedings of the 2005 International Symposium on High-Performance Computer Architecture*, pages 258–262. IEEE Computer Society, 2005.

[10] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 162–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0881-2. URL `http://portal.acm.org/citation.cfm?id=832307.837099`.

[11] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 114–124, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375596. URL `http://doi.acm.org/10.1145/1375581.1375596`.

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

[13] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36:24–35, 1987.

[14] G. C. Sih and E. A. Lee. Declustering: A new multiprocessor scheduling technique. *IEEE Trans. Parallel Distrib. Syst.*, 4:625–637, June 1993. ISSN 1045-9219. doi: 10.1109/71.242160. URL `http://portal.acm.org/citation.cfm?id=628910.629186`.

[15] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throughput stream programming in Java. *SIGPLAN Not.*, 42:211–228, October 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1297105.1297043. URL `http://doi.acm.org/10.1145/1297105.1297043`.

[16] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization.* Marcel Dekker, Inc., New York, NY, USA, 1st edition, 2000. ISBN 0824793188.

[17] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC '02: 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002.

[18] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software Pipelined Execution of Stream Programs on GPUs. In *CGO*, pages 200–209, 2009.

[19] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal. Baring it all to software: The raw machine. Technical report, MIT, Cambridge, MA, USA, 1997.