Orchestrating Stream Graphs Using Model Checking

AVINASH MALIK, IBM Research Ireland
DAVID GREGG, Lero, Trinity College Dublin

In this article we use model checking to statically distribute and schedule *Synchronous DataFlow* (SDF) graphs on *heterogeneous execution architectures*. We show that model checking is capable of providing an optimal solution and it arrives at these solutions faster (in terms of algorithm runtime) than equivalent ILP formulations. Furthermore, we also show how different types of optimizations such as task parallelism, data parallelism, and state sharing can be included within our framework. Finally, comparison of our approach with the current state-of-the-art heuristic techniques show the pitfalls of these techniques and gives a glimpse of how these heuristic techniques can be improved.

Categories and Subject Descriptors: D.3.4 [Processors]: Compilers

General Terms: Languages, Performance

Additional Key Words and Phrases: Streaming, dataflow, parallelization, compiler

ACM Reference Format:

Malik, A. and Gregg, D. 2013. Orchestrating stream graphs using model checking. *ACM Trans. Architect. Code Optim.* 10, 3, Article 19 (September 2013), 25 pages.

DOI: <http://dx.doi.org/10.1145/2512435>

1. MOTIVATION

With the advent of multicore processors, there is an increasing need for programming models that simplify the task of parallel programming. Synchronous DataFlow (SDF) provides a simple parallel programming model that is particularly suited to applications that operate on streams of data. Several important classes of application are well suited to this streaming model. These include applications that process streams of sensor data, signal processing applications, financial applications that operate on streams of market data, and communications applications that operate on streams of data packets.

In the SDF model, the programmer describes the application as a network of *filters* with data *tokens* flowing between filters through *channels*, forming a *stream graph*. In the stream graph, all filters can execute in parallel, with data dependencies enforced by the flow of tokens between filters along channels. Thus, there is a simple model of parallelism, where data dependencies are explicit in the stream graph. Each filter in the graph accepts one or more tokens on each invocation of the filter and produces one or more tokens. The only mechanism for communication between parallel filters is through channels. Thus the programmer describes a set of filters using a sequential programming language which can be executed in parallel within the stream graph.

Perhaps the most important task of a compiler for an SDF graph is to schedule the filters in the stream graph to parallel execution units on the target machine. The compiler

Authors' addresses: A. Malik and D. Gregg (corresponding author), School of Computer Science and Statistic, Trinity College Dublin, Ireland; email: david.gregg@cs.tcd.ie.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/09-ART19 \$15.00

DOI: <http://dx.doi.org/10.1145/2512435>

is fully aware of the available parallelism between filters, because data dependencies are explicit in the dataflow graph. But the problem of mapping the filters to execution units remains. Where the hardware processing elements are homogeneous and one does not have to consider communication costs between processing elements, the problem is NP-complete [Sarkar 1989], but some good heuristics exist [Coffman et al. 1978].

However, modern computer systems increasingly contain heterogeneous execution units, such as combinations of CPUs and GPUs. In the presence of heterogeneous execution units the problem of mapping the stream graph to the execution units becomes far more difficult, not least because the execution time of one invocation of a filter is likely to vary significantly depending on the type of hardware unit to which it is allocated. There is a lack of good algorithms for mapping stream graphs to heterogeneous architectures.

A related problem arises with the cost of communicating between different processing elements. On a simple shared-cache multicore processor it may be an acceptable simplification to ignore communication costs. But on a system combining CPUs, GPUs, or other types of heterogeneous units, communication costs between different types of units are likely to be greater, and the communication costs themselves are likely to be heterogeneous between different types of units. A good algorithm will take account of these communication costs in creating the schedule for the stream graph.

A final problem that must be solved to create a good mapping of stream graph to hardware relates to data parallelism. Although the semantics of SDF graphs is that there is no parallelism within filters, in many cases filters operate on long sequences of tokens, where there are no data dependencies between the processing of each token. These are known as *stateless* filters. In this case, the compiler may be able to divide the data parallel work of the filter over several execution units so that the tokens are processed in parallel. Some SDF systems replicate stateless filters as a heuristic prepass before the graph is mapped to hardware [Gordon et al. 2006; Carpenter et al. 2009]. However, replication of stateless filters has a large effect on the level of available parallelism in the graph, and better decisions can be made if we consider data parallelism and scheduling of filters to the parallel hardware within a single framework.

In this article we present an algorithm for orchestrating stream graphs on heterogeneous execution architectures. Our algorithm models heterogeneous execution units and heterogeneous communication costs. It also solves the problem of replicating stateless filters to exploit data parallelism within the same framework as scheduling the stream graph. Our algorithm solves all these problems by expressing stream graph scheduling as a model-checking problem. We demonstrate that model checking provides a novel and powerful framework for solving parallelism orchestration problems, and that the flexibility offered by the approach allows us to solve difficult scheduling problems such as heterogeneity and data parallelism within a single framework. We find that model checking performs well in practice, and compares well with more traditional approaches to these types of problems, such as integer linear programming.

The rest of the article is arranged as follows. In Section 2 we describe the streaming model and provide a motivating example to describe the problem in more detail. Section 3 gives an overview of related work. Section 4 explains the state space (model-checker) modeling of the distribution and scheduling problem. Experimental results comparing the model-checking approach with *Mixed Integer Linear Programming* (MILP) and heuristic solutions is provided in Section 5. Finally, the discussion and conclusions follow (Section 6).

2. BACKGROUND

Consider the smart embedded and networked human tracking system shown in Figure 1 (a). The execution architecture consists of two different processing components: *Graphic Processing Units* (GPUs) and *Central Processing Units* (CPUs). The

Fig. 1. A motivating example.

images flow in from the one *Pan/Tilt/Zoom* (PTZ) camera (camA), and these images are captured over the network and different tracking algorithms are applied to the incoming images. Finally, the cameras are repositioned depending upon the movement of the object in the field of view. This tracking system adheres closely to the streaming paradigm, where data flows from one hardware *and* software processing unit to the other as continuous image streams.

2.1. The Streaming Model

Figure 1(b) shows an abstract stream representation of the complete tracking system. We have used an abstract representation for ease of understanding and for brevity. The software process (Image capture), called an actor or filter, continuously captures incoming images from the cameras. Next, the Splitter splits copies of images for parallel processing. The split images are sent to different algorithms working in parallel (DFT, FFT, and a special transform, TransF, algorithm). DFT and FFT are both stateless filters, that is, filters where every invocation is independent of any other previous invocation. TransF is a statefull filter. The final result of these parallel processes then gets collated (Col) to make decisions.

Edges between filters are *First-In First-Out* (FIFO) channels. These edges are typed and annotated with the number of tokens produced and consumed on each invocation of that filter. Image capture produces and consumes 1 token on each invocation. Splitter, on the other hand, consumes 1 token but produces 2 tokens on each of its branches for every invocation. The three parallel filters connected to Splitter consume only a single token for every invocation. Thus, for an iteration of this stream graph to be successful, it is necessary that the three parallel algorithms are run twice, consuming both tokens produced by the Splitter. Similarly we need a *balance* of tokens being consumed and produced on the edges connecting the parallel filters with Col. The notion of balance equations [Lee and Messerschmitt 1987] helps one decide the size of FIFO and schedule for the stream graph. Since these rates remain constant throughout the lifetime of the application we call it a *Synchronous DataFlow* (SDF) graph. The number of invocations required for each filter in the SDF graph satisfying the balance equations is called the natural granularity of the filters. For Figure 1(b), the natural granularity is given by the set $\mathbf{q} = \{2, 2, 4, 4, 4, 1\}$. The time taken to complete a single iteration of the SDF graph is called the *makespan*.

2.2. Problem

Let us now assume that the computation time for filters in the graph, for the architecture in Figure 1(a), is given by Table I. For simplicity, we assume that the

Table I. Computation Cost of Filters in Figure 1(b)
on the Execution Architecture in Figure 1(a)
in Units of Time

Filters	CPU1	GPU1	CPU2	GPU2
Image Capture	1	1	1	1
Splitter	1	1	1	1
FFT	1	2	3	4
DFT	1	2	3	4
TransF	1	1	1	1
Col	1	1	1	1

Table II. The Distribution and Schedule Giving the Optimal Throughput

	0	1	2	3	4	5	6	7	8	9	10
GPU1					FFT	FFT	FFT		DFT	DFT	Col
GPU2					DFT	DFT	DFT	DFT			
CPU1					FFT	FFT	FFT				
CPU2	Image capture	Image capture	Splitter	Splitter	TransF	TransF	TransF	TransF			

communication costs are 0 time units, just for this simple example. Given the heterogeneous nature of the execution hardware, what is the distribution and schedule for the filters in Figure 1(b), giving the maximal throughput?

The very nonobvious solution is shown in Table II. As we can see from Table II, even for very simple SDF graphs, heterogeneity of the execution architecture makes compile-time distribution and scheduling complex. In this article our main contribution is a model-checking-based approach to providing an optimal solution to the distribution and scheduling problem on a heterogeneous execution architecture. In particular, we answer the following questions: (1) What processor is best suited to execute a given filter? (2) What communication links should be exploited? (3) Is load-balancing necessary? (4) Can we determine a compile-time schedule for filters so that runtime overheads can be avoided? (5) How do we include popular optimization techniques such as data parallelism *and* task parallelism in the heterogeneous execution framework?

2.3. Why Model Checking?

We argue both quantitatively and qualitatively that model checking is not only a comparable, but a better solution compared to other standard approaches such as *Integer Linear Programming / Mixed Integer Linear Programming* (ILP/MILP), and heuristics which are currently used for partitioning and scheduling stream graphs. In particular, we show that: (1) Our model-checking approach is faster (in term of algorithm runtime) than equivalent MILP formulations. (2) Our model-checking approach finds better schedules than the currently used heuristics. (3) Our model-checking approach can easily be extended to multicriteria scheduling. For example, we can accommodate criteria such as power consumption and throughput trade-offs without changing the model, an advantage unfulfilled by any ILP/MILP or heuristic solutions.

Before proceeding with the rest of the article we give an informal and brief introduction to the concepts involved in model checking and the Uppaal [Amnell et al. 2001] model checker specifically.

2.4. Brief Background on Model Checking

Model checking is the process of *formally* verifying if some properties are satisfied on a given state machine. This state machine is the model or representation of a software

Fig. 2. Example model-checking path formulae. $A\Box\psi$ describes safety properties. $E\Diamond\psi$ describes the reachability property. $\varphi \leadsto \psi$ describes liveness properties.

or hardware program. The properties like the model itself are described in a formal language. There are a number of different formal languages that can be used for describing these properties; herein we concentrate on the *Computational Tree Logic* (CTL) [Burch et al. 1990; Clarke et al. 2000; Gupta et al. 1996] formulae. CTL formulae can be divided into two main categories: formulae verifying state properties and those verifying path properties.

- State formula.* A state formula is a property that can be verified for any state of the model without looking at the paths in the state machine. For example, $(i==7)$ is a state property that guarantees that the value of variable i is 7 in any given state of the model.
- Path formula.* A path formula, on the other hand, is more complex and can describe reachability, liveness, safety properties, etc. Figure 2 shows some of these essential property descriptions in an informal manner. A safety property described as $A\Box\psi$ states that a given formula ψ should be satisfied on all paths starting from the initial state to the final state. In Figure 2, the bold paths show that the property ψ holds. A reachability property is described as $E\Diamond\psi$, which states (unlike the safety property) that there exists at least one path from the starting state to the final state where a formula ψ holds. Finally, liveness property $\varphi \leadsto \psi$ states that given φ holds in some state, there exists at least one path from this state to the final state where ψ holds. In traditional model checking, a safety property is used to verify that nothing bad ever happens, a reachability property is used for sanity checking, and liveness properties are usually used to make sure that an output event is generated for a given input event (e.g., communication protocols).

2.5. Brief Background on the Uppaal Model Checker

In this article we use the Uppaal model checker [Amnell et al. 2001] to formulate and carry out our partitioning and scheduling algorithm. In this section we very briefly describe the important components in the Uppaal model-checking engine. For a detailed report the reader is referred to Amnell et al. [2001].

Uppaal is a toolbox for verification of real-time systems. Uppaal models networks on timed automata and verifies *Timed-CTL* properties on these models.

We use a lamp on/off system to describe the concepts in the Uppaal model checker. This example is obtained from Behrmann et al. [2004]. Figure 3 shows an example lamp control system. There are two state machines in this system. The one on the left models the lamp controller, while the one on the right models the user. The lamp can be turned on from the initial off state (shown with the double circle) if the user presses the on button. Moreover, the lamp can be turned into a bright state if the on button is pressed twice within the first 5 seconds. If the second press is after 5 seconds of the first press, then the lamp switches off.

Fig. 3. An example lamp system in Uppaal.

Each state machine in the Uppaal model is driven by one or more synchronous *dense clocks*, C , for example, y is a clock in Figure 3. Time (modeled by the dense clock C) passes continuously in any given state. Valuations of clocks are in the real positive domain, that is, $u : C \rightarrow \mathbb{R}^+$. A state transition in any given model is guarded by a combination of boolean equations of integer variables or a set of conjunctions over simple conditions on clocks. Guards always evaluate to a boolean condition. On a true evaluation of a guard, an action might be performed, for example, $y := 0$ is the resetting of the clock y back to zero action on the transition from *off* to *low* locations. Channel synchronization is another form of action that allows one or more concurrently executing models to synchronize. In Uppaal, channel synchronization is carried out using either the CSP [Hoare 1978]-style synchronous blocking communication between a single sender (*press!*) and a single receiver (*press?*) or using nonblocking sender broadcasting to multiple receivers.

Other than the aforementioned features Uppaal also supports the concept of *urgent* and *committed* locations. In an urgent location time cannot proceed further, while in a committed location other than the stopping of the real-time clock one always needs to take an outgoing transition. Uppaal supports a number of different types such as structures and arrays over clocks and integer values just like in “C” to make programming the system easier. Finally, operators $[]$, $\langle \rangle$, and $\rightarrow$ are used to describe the safety, reachability, and liveness properties, respectively. For example, $E \langle \rangle t < 5$ and $A [] m == 5$, states that there needs to be at least one path in the model where the value of some variable t never exceeds 5, while in all the paths in the model the value of some variable m is always 5, respectively. Simple conjunctions of such CTL properties and boolean integer expressions are allowed in Uppaal.

3. RELATED WORK

A large body of work exists for scheduling SDF graphs on varying architectures with varying goals. Farhad et al. [2011] propose a heuristic algorithm for partitioning an SDF graph onto multicore homogeneous platforms in order to reduce the input token arrival rate for filters. Their heuristic technique does not consider a heterogeneous execution platform. Gordon et al. [2006] exploit parallelism for the RAW architecture [Waingold et al. 1997] without considering the communication cost or the varying computation costs prevalent in a heterogeneous environment. Similar attempts have been made by Udupa et al. [2009] and Kudlur and Mahlke [2008] with their individual ILP formulations targeting reduced makespan on GPUs and multicore platforms, respectively, without consideration for communication costs and heterogeneity. The same authors have also previously attempted an ILP formulation for scheduling SDF graphs on general-purpose processors [Govindarajan et al. 2002]. Govindaranjan et al. [2002] and Udupa et al. [2009] have the same deficiency of ignoring complex cyclic SDF graphs. Another drawback of Govindarajan et al. [2002] is that it models resource-unconstrained architectures, which is generally not the case for embedded systems.

Finally, Udupa et al. [2009] report suboptimal makespan and buffer allocation, because their ILP formulation is not an optimization problem, but rather a constraint problem.

The closest attempt at including heterogeneity is presented in Carpenter et al. [2009], where stream programs are mapped to a heterogeneous execution platform and then load-balanced using heuristic techniques. Carpenter et al. [2009] also consider communication costs when applying their heuristic techniques. Yet, Carpenter et al. [2009] provide an unsatisfactory result, because: (1) their formulation does not consider reducing the makespan of the SDF graph, but rather they target load-balancing filters on the architecture to equally utilize processor resources; load-balancing filters across processors is *not* equivalent to reducing the makespan. For example, the smallest makespan might be allocating all the filters on a single processor, while load-balancing, on the other hand, forcefully allocates filters across processors, especially if there are free processors available. (2) They consider convex graphs without cycles. Lastly, this approach is based on Kernighan's heuristic graph algorithm, and hence provides a suboptimal solution.

One related approach that targets reducing the makespan while considering communication costs and targeting global reduction in makespan is presented by Sih and Lee [1993]. Sih and Lee [1993] use a heuristic technique called "declustering". The declustering algorithm takes an SDF graph as input, and carries out a 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 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 drawbacks such as not considering a heterogeneous architecture with differing communication and computation costs for filters and a nonoptimal schedule.

Overall, ours is the very first attempt to provide a distribution and schedule that results in an optimal makespan for heterogeneous architectures with heterogeneous computation and communication costs. Optimal schedules for this NP-hard problem [Farhad et al. 2011] are important; for example, consider that there are a myriad of heuristics for partitioning and scheduling SDF graphs on a homogeneous execution architecture [Farhad et al. 2011; Gordon et al. 2006; Kudlur and Mahlke 2008; Sih and Lee 1993]. Finding a schedule for a heterogeneous execution architecture is NP-hard, because of the exponentially large search space. This in turn leads to complex heuristic approaches, thus, how does one know that a heuristic targeting a heterogeneous execution architecture is a *good* heuristic, if one does not know what is the optimal solution?

In this article we perform an extensive quantitative evaluation, comparing the state-of-the-art heuristic solutions [Carpenter et al. 2009; Sih and Lee 1993; Udupa et al. 2009] for stream graphs and a number of solutions for scheduling task graphs [Kohler 1975; Adam et al. 1974] with our model-checking approach. Where appropriate we even modify the existing algorithms, in order to make them suitable for heterogeneous architectures and to perform a fair comparison. Our results, in Section 5, clearly show that extensive research is required to accommodate heterogeneity and the current heuristics perform poorly.

4. ENCODING THE DISTRIBUTION AND SCHEDULING PROBLEM INTO THE UPPAAL MODEL CHECKER

We have used the Uppaal model checker [Amnell et al. 2001] 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,

Fig. 4. Precedence graph for Figure 1(b).

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. [2009] and Behrmann et al. [2005], 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 article.

4.1. Formalizing the Problem Statement

An 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. 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 Π 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 Π .

Definition 4.1. 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

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

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

PROOF. As we can see from Eq. (1), the throughput and makespan are inversely proportional. Hence, minimizing makespan (Π) is equivalent to maximizing throughput (ζ). $\square$

4.2. 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 4 gives the precedence graph $\mathcal{P}$ translated from the SDF graph $\mathcal{G}$ in Figure 1(b). $\mathcal{P}(V, E)$ 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.

Fig. 5. Uppaal representation of computation and communication allocation.

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. From Section 2, 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, and same for the others.

LEMMA 4.3. *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 Malik and Gregg [2012b] for details. $\square$

4.3. 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 5(a) shows the Uppaal automata representing allocation of filter Splitter on two of the four available processors in Figure 1(a).

Every location in the automata is marked with a U representing urgency, that is, 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: $\{\text{Splitter}, \text{CPU1}\}$ for the first automaton. The transition is guarded by the condition $\text{SplitterCPU1}==1$. Upon transition, 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, from Section 2.2. 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 5(b), shows the

Fig. 6. A network of Uppaal automata representing parallel filter execution.

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: {C2, CPU1, 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 5(a), 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.

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

Figure 6 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 ($myCost=2$ from Section 2.2). 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

(a) naive replication of stateless FFT and DFT filters

(b) basic automata modeling stateless FFT filter execution

(c) extra automata representing fused stateless filters

Fig. 7. Optimal exploitation of data parallelism.

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 6. 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 4 is not enough. As we can see from Figure 6, 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 7(a) 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 multistep process: first off, we naively replicate all the stateless filters, as shown in Figure 7(a). 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 7(b). Finally, we build extra automata modeling fusion of these stateless filters for each processor as shown in Figure 7(c).

In Figures 7(b) and 7(c), we haven't shown all the Uppaal automata that are generated due to lack of space (the ... in Figure 7 shows 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 $P \subseteq \mathcal{A}$ **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 are different for automata in Figure 7(b) and 7(c). Consider the first automata in Figure 7(c), 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.

Fig. 8. Exploiting granularity-based optimization.

Table III. Execution Traces for Figure 8 on the Execution Architecture in Figure 1(a)

(a) Execution trace 1													
P1	A	A	A	A	C	C	C						
P2								B	B	B	B	B	B

(b) Execution trace 2													
P1	A0	C0	C0	C0									B
P2	A1				C1	C1	C1						B
P3	A2							C2	C2	C2			B
P4	A3										C3	C3	B

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.

4.5. Granularity-Based Optimization

Granularity-based optimization is a novel contribution of this article. Granularity-based optimization compliments judicious data and task parallel exploitation. Consider the precedence graph in Figure 8(a), where A is a stateless filter and B is a stateful filter. Assume that A's computation time is 1 time unit for all processors on the execution architecture in Figure 1(a). Also, let the communication time on the fabric be 3 time units for any given communication link in Figure 1(a). Similarly assume that B's computation and communication time is 1 time unit and 5 time units, respectively.

The optimal task and data parallel algorithm, described in Section 4.4, applied to the precedence graph in Figure 8(a) results in the graph in Figure 8(b). The execution trace for Figure 8(a) and 8(b) is shown in Table III. As we can see, the original graph gives a lower makespan at 13 time units, while exploiting stateless data parallelism results in *worse* performance at 19 time units. That's a performance degradation of $\approx 31\%$. The four separate copies of filter A execute for one time unit on each of the processors and then communicate with the join filter B. All four copies cannot communicate at once, because there is only a single communication channel on the receiving end (filter B). Even with increased utilization of resources (4 processors in Table III) the precedence graph in Figure 8(b) has a longer makespan.

The main reason for this disparity is the communication costs. In the original version, the communication between A and B takes place only once, at the end of the four

Fig. 9. State transitions in Uppaal.

individual invocations of filter A, whereas in the data parallel version, data is sent after every invocation of the replicated filter.

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 (amortization of communication costs). When the granularity is increased and if the communication costs remain constant more work is done by the filters, because increasing granularity increases the number of filter invocations, which in turn leads to production of more tokens in less amount of time. For the example in Figure 8(b), increasing granularity by a multiple of 10 leads to a makespan difference of 21 time units between the makespan of the data parallel and the original precedence graphs in the favor of the data parallel precedence graph, that is, an improvement of $\approx 67\%$. Increasing granularity 20 times makes the difference larger still, at 51 time units in favor of the data parallel precedence graph. Thus, one can search for an optimal granularity of the stream graph for a given execution architecture.

Our granularity-based optimization technique assumes a constant communication time for a range of bytes $[\omega_{lb}, \omega_{ub}]$, that is, the communication cost 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 (maximum transmission unit). We utilize these characteristics of networked systems by increasing the granularity of the computation filters, while maintaining the granularity of communication as 1 to increase the overall graph throughput. Similar communication amortization is applicable to *Network-on-Chip* (NoC) architectures. We formally prove that increasing granularity can improve throughput in Malik and Gregg [2012b].

For multicore systems with shared on-chip caches we do not apply the granularity optimization and the filters are simply run at their natural granularity. The main reason is that we communicate using shared memory buffers in on-chip caches so the communication cost is low.

4.6. State Space and Reachability Property

In this section we use a simple example to show our technique of using the Uppaal model checker to find the optimal makespan. Assume that only the basic automata modeling sequential execution (from Figure 5) of Splitter and its communication channels C2, C3, and C4 are built. Similarly, also assume that *only* task parallelism is exploited for filters FFT, DFT, and TransF, without consideration for data parallelism, as shown in Figure 6. Now, if these automata are input into Uppaal and a *reachability* property $E \langle \rangle (\text{TerminalState} \text{ and } \text{Cost} < \infty)$ is asked to be verified, then the result is the possible state space transitions shown in Figure 9.

Uppaal carries out state transitions and outputs the result that the property is satisfied. The trace generated from Uppaal (Figure 9) gives the allocation within the states and the schedule via the transitions. For example, at start, *Splitter* can possibly be allocated to any of the 4 processors, while Uppaal in Figure 9 chooses CPU1 (shown by the first transition). During this transition *Cost* is updated by 2 (from Figure 5(a)). The communication transitions happen next, one after the other, each of them updating *Cost* by 0, because we have modeled a 0 communication cost for the very simple example. Finally, all three parallel filters *FFT*, *DFT*, and *TransF* make a transition together (from Figure 6), shown by the filled-in ellipse. This parallel transition increments *Cost* by 4. The result is reaching the terminal state with the value of global variable *Cost* being 6, which satisfies the reachability property.

If we now reiterate through this model-checking process with the reachability property modified as: $E \langle \rangle (\text{TerminalState and Cost} < 6)$, which reads: find at least one path from the starting state to the terminal state where the *Cost* never exceeds 6 units, then Uppaal carries out an exhaustive state space exploration and outputs that the property is not satisfied, that is, the minimum possible execution *Cost* is 6 time units. Such an iterative approach can be used to find the minimum makespan, since *Cost* is the makespan. For the very first iteration, ∞ can be modeled with an arbitrarily chosen large number.

Instead of an optimization problem one can change this into a worst-case execution-time analysis problem, using a safety property such as: $A[] (\text{TerminalState and Cost} < \text{num})$, which reads: make sure that all paths from the starting state to the terminal state have *Cost* less than *num*. More interestingly, a multicriteria schedule and partition can be easily accommodated in this framework. For example, if we want to trade off between power consumption, denoted by *Power*, and execution cost, then we can change the reachability property to: $E \langle \rangle (\text{TerminalState and Power} < \text{num1 and Cost} < \text{num2})$. Multicriteria scheduling requires little effort to introduce in the model itself. We do not address multicriteria scheduling in this article.

5. EXPERIMENTAL RESULTS

We have compared our model-checking approach with MILP and heuristic solutions as described in Malik and Gregg [2012a]. The flow of our compiler is shown in Figure 10. We use *oprofile* for CPUs and the event system for GPUs to profile the execution cost of filters (for a single invocation) on different processors and communication latency between processors by executing individual filters in the graph separately and sending/receiving varying-size data packets between processors, respectively. We have used the CPLEX solver for solving our MILP formulations. Our solvers (CPLEX) and Uppaal were run on a Core2Duo 2.4 GHz processor with 2GB of RAM.

Our experimental platform is shown in Figure 11. 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 Figures 12 through 17 and 18. All the numbers in Figure 12 through 17 are for natural granularity only, that is, we haven't applied any granularity-based optimizations.

As stated previously, there is no current published work on ILP other than our own Malik and Gregg [2012a] that can be used for direct comparison. Since both MILP and model-checking formulations give optimal solutions to an NP-hard problem in exponentially long time, we also compared these solutions against modified declustering and critical-path scheduling techniques. These techniques were chosen because they account for computation and communication time of filters on the execution architectures, although only targeting homogeneous platforms. Furthermore, to include stateless data parallel filters these techniques were modified with the *StreamIt*

Fig. 10. Compilation and execution flow.

judicious data parallelism heuristic as described in Gordon et al. [2006]. The StreamIt judicious stateless filter replication algorithm is applied to the SDF graph, before passing it on to these scheduling heuristics. A brief description of the declustering algorithm has already been provided in Section 3. The critical-path algorithm is used to find the most expensive path in terms of computation and communication times in the SDF graph using *back flow* [Kohler 1975]. The StreamIt heuristic is a very simple algorithm which first of all greedily composes stateless filters together and then splits these stateless nodes on processors (depending upon the fraction of work done compared to the overall work in a split/join node). Both these heuristic algorithms are targeted at homogeneous execution architectures. We modify these algorithms to allocate and schedule filters greedily during the list-scheduling phase [Adam et al. 1974], considering the heterogeneity of the architecture. The implemented declustering and critical-path scheduling algorithms are shown in Algorithms 2, 3, and 4.

The declustering algorithm (Figures 13 and 16), even with our modifications, performs poorly compared to the proposed model-checking solution. In the worst case declustering speedup is 62.5% slower compared to the optimal model-checking solution. We think that the main reason for this discrepancy is the fact that a complete cluster is assigned to a processor. In a heterogeneous architecture it is essential that basic clusters be formed not *just* depending upon the communication costs, but also accounting for their heterogeneity. We advocate that basic clusters be formed with filters, which exhibit equivalent average computation times.

The critical-path (Figures 12 and 15) scheduling technique performs worse than the declustering technique in general, except in some cases, because of the unrestricted partitioning and allocation of filters onto processors. The main disadvantage of both these approaches is the fact that both are based on list-scheduling techniques, which does not allow delaying filter execution, that is, all filters ready to be executed at any given level need to do so, which in turn reduces makespan.

ALGORITHM 2: The stages in the Declustering algorithm

Input: Graph $\mathcal{G}$ architecture $\mathcal{A}$
Output: $\mathcal{F} : \mathcal{G} \rightarrow \mathcal{A}$
 //First apply StreamIt data-parallel heuristic;
 StreamIt_data_parallel_heuristic()
 //Make clusters, making sure that IPC costs are smaller than computation costs ;
 Make_basic_clusters();
 //Make a binary tree with basic clusters as the leaves;
 Make_hierarchical_binary_tree();
 //Decluster btree using multi-processor list scheduling;
for each $P \in \mathcal{A}$ **do**
 for each basic_cluster $\in$ btree **do**
 allocate_basic_cluster(basic_cluster, P);
 Multi_processor_list_scheduling();
 keep_schedule_if_smallest_makespan();
 end
end
 //Load balance the filters, looking at just the edge filters;
 Load_balance_allocated_filters()

ALGORITHM 3: StreamIt_data_parallel_heuristic

Input: F : the filter to fission; N : the total number of cores
Output: $\mathcal{F} : F \rightarrow N$
 $fraction = 1.0$;
 Stream $parent \leftarrow GET_PARENT(F)$;
 Stream $child \leftarrow F$;
while $parent \neq null$ **do**
 if parentissplitjoin **then**
 $totalwork \leftarrow \sum_{c \in CHILDREN(parent)} AVGWPFILTER(c)$;
 $mywork \leftarrow AVGWPFILTER(child)$;
 $fraction \leftarrow fraction \times mywork/totalwork$;
 end
 $child \leftarrow parent$;
 $parent \leftarrow GET_PARENT(parent)$;
end
 Fission F into $CEIL(fraction * N)$ filters ;

Another state-of-the-art heuristic algorithm that we compare against is that of Carpenter et al. [2009]. Carpenter et al. propose an algorithm for allocation and scheduling of SDF graphs onto heterogeneous execution architectures, wherein, like us, they consider heterogeneous computation and communication costs. The heuristic is quite involved, so herein we give a general overview of the approach. Carpenter's heuristic is a restricted form of bin-packing problem. Note that bin packing considers the SDF graph to be a *Directed Acyclic Graph* (DAG). In their algorithm all strongly connected (cyclic) graphs are collapsed into a single filter. Moreover, their heuristic consists of two possible solutions depending upon convexity and connectedness constraints of the partition.

—A connected partition loosely put is a loosely connected subgraph. Their algorithm in the strongest case requires that every processor be allocated a connected subgraph $S_C \subseteq \mathcal{G}$.

Fig. 11. Heterogeneous experimentation architectures.

ALGORITHM 4: Multi_processor_list_scheduling using HLFET

```

read_filter_list ← Get_ready_filters ();
while ready_filter_list ≠ ∅ do
    avail_proc_list ← Get_avail_processors ();
    Put_read_filters_in_HLFET ();
    for each filter ∈ read_filter_list do
        if filter ≠ assigned_proc then
            | Alloc_proc_with_min_cost (ready_filter_list, avail_proc_list);
        end
    end
    min_cost ← Get_min_cost ();
    makespan ← makespan + min_cost;
    Remove_done_filters_free_procs ();
    read_filter_list ← Get_ready_filters ();
end

```

—The convexity constraint requires that filters be allocated onto processors in such a way that there are no cyclic dependencies between processors.

Both the preceding constraints are motivated by the observation that processors can proceed without stalling for communication. These constraints lead to two possible scheduling solutions.

- (1) Strictly connect partitions, where all allocations on any given processor form a loosely connected subgraph.
- (2) Loosely connect a partition, where connected constraints are relaxed.

Carpenter et al.'s heuristic is meant to be used in an iterative compiler. The stages in the iterative compiler are shown in Algorithm 5.

ALGORITHM 5: Carpenter's high-level iterative compilation approach

```

while true do
    Blocking_and_splitting ();
    //Carpenter et al.'s partitioning heuristic;
    Partitioning ();
    Pipelining ();
end

```

ALGORITHM 6: Carpenter et al.'s partitioning heuristic

```

Get_basic_connected_sets ();
// Initial partition is obtained by dividing the target architecture into equally intensive
compute parts and then using branch and bound
Get_initial_partition ();
//The refinement phase is split into 4 different parts
//A greedy algorithm merges low cost tasks and frees processor resources
Merge_tasks ();
//Next, bottle-necked processors are load-balanced by moving filters from such
processors onto those that are less loaded or free
Move_bottle_necks ();
//In the next step new tasks are created and connected and convexity constraints are
relaxed, i.e., filters are moved from one connected set to the next
Create_new_tasks ();
//Finally, a greedy algorithm improves the allocation of tasks to processors
Relocate_tasks ();

```

Carpenter's main partitioning algorithm is involved and hence we give a high-level description only in Algorithm 6. Algorithm 6 uses the average computation and communication values for filters when carrying out the partitioning.

Figures 14 and 17 give the difference in makespan of Carpenter et al. versus our model-checking formulation. As we can see from Figures 14 and 17, our approach gives a smaller makespan value compared to that of Carpenter et al. The difference is large, especially in the case of *cyclic* examples (SS, PID, SMS) where our algorithm is able to find a partition and schedule which gives $\approx 16\times$ better makespan value. This large difference for the cyclic SS (Security Surveillance) example between Carpenter et al.'s algorithm and our model-checking solution comes from the fact that in SS there is a cycle encompassing the complete graph. Carpenter et al.'s algorithm reduces the whole graph into a single node, which in turn leads to loss of both task and data parallelism. Our approach is the very first one to handle complex cyclic graphs correctly. Such complex cyclic graphs are commonplace in advanced control systems such as SS.

Another drawback with Algorithm 6 is the fact that Carpenter et al. are trying to maintain a connectedness and convexity constraint, whereas our model-checking formulation results in an unrestricted partition. Furthermore, the `create_new_task` procedure and `move_bottle_neck` steps are guided by user-chosen heuristic values. For our experiments, we bound these procedures by $\approx 3\%$ performance improvement

Fig. 12. Critical-path scheduling vs MC/ILP: only exploiting task parallelism. This figure shows the speedup obtained without exploiting data parallelism, that is, we do not replicate stateless filters.

Fig. 13. Declustering vs MC/ILP: only exploiting task parallelism. This figure shows the speedup obtained without exploiting data parallelism, that is, we do not replicate stateless filters.

and ≈ 1000 iterations, respectively. If we set the performance improvement percentage and number of moves to ∞ , then the complexity of Carpenter's algorithm increases exponentially, since the algorithm reduces to a bin-packing problem.

Figure 18 shows the effect of increasing the granularity of our benchmark stream graphs. Rather than computing the optimal granularity for each, we instead show the effect of several granularity multiples. We see that increasing the granularity of the stream graphs can result in significant speedups of up to 1.6 in our experiments. This is the result of amortizing communication costs over larger amounts of data being sent each time. However, one needs to be vigilant when performing these optimizations. We do not always see an improvement in the throughput either, since increasing granularity might suddenly increase the cost of communication significantly if the

Fig. 14. Carpenter et al. [2009] vs MC/ILP: only exploiting task parallelism. This figure shows the speedup obtained without exploiting data parallelism, that is, we do not replicate stateless filters.

Fig. 15. Critical-path scheduling vs MC/ILP: exploiting both task and data parallelism.

threshold of the step in the networked step function is suddenly crossed by the increased granularity.

Finally, the runtime to find the optimal makespan solutions at natural granularity for the model checker and the CPLEX solver is shown in Table IV. CPLEX was run in multithreaded and nonmultithreaded mode, while Uppaal only supports single-threaded execution. As we can see from Table IV, for comparable (single-threaded) CPLEX and Uppaal solutions, Uppaal results in $\approx 44\%$ better runtime. Compared to multithreaded CPLEX performance Uppaal is better still at $\approx 9\%$. The reason for this difference is multifold.

—*Variable reduction.* Variable reduction involves explicitly setting variables to a known value, when they are not used in order to reduce the state space explosion problem. Notice that throughout our formulation a number of variables (e.g., `SplitterCPU1`) are left in a persistent known state thereby making the rest of the state transitions independent of these variables, which makes the states bisimilar and hence helps in pruning the state space.

Fig. 16. Declustering vs MC/ILP: exploiting both task and data parallelism.

Fig. 17. Carpenter et al. [2009] vs MC/ILP: exploiting both task and data parallelism.

Fig. 18. Throughput increase due to granularity-based optimizations on the 8-core architecture.

Table IV. MILP vs MC vs Heuristic Solve Times (sec)

Benchmarks	#	MILP	MILP-P	MC	Decustering	CP	Carpenter
FFT	17	2942	1856	742	4	3	3
Audiobeam	28	31	22	31	9	7	9
Bitonicsort	28	53586	26793	21434	6	3	4
TDE	28	43968	26984	25203	10	10	13
Vocoder	90	50380	38190	26733	12	18	12
DES	33	9748	5718	1716	17	12	8
Mp3decoder	26	36	36	36	6	3	2
Radar	54	3531	1837	1672	14	9	12
SMS	30	5245	2367	3078	2	1	1
PID	15	2944	1592	1751	2	2	2
SS	170	158556	89278	102370	2	2	1

is the number of filters in the benchmark example. MILP-P is the parallelized version of the CPLEX solver. MC and CP indicate the model-checking and the critical-path methods, respectively.

—*Atomicity*. Atomicity is the property of reducing state space by reducing the number of interleavings in the state space explosion problem. The number of interleavings can be reduced by marking states *urgent* or *committed*, which reduces the number of possible state transitions as explained in Section 2.5.

6. DISCUSSION

As we can see from Figures 12 through 17 and 18, model checking and MILP formulations give equivalent solutions. This is not surprising since they both model the same behavior. The heuristic solution performs poorly compared to the optimal solutions. In the worst case, heuristics give a schedule with a makespan $\approx 16\times$ slower than the optimal. 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. As for Carpenter et al. [2009], the main drawback is the convexity and nonconsideration of cyclic graphs. Even our modifications [Malik and Gregg 2012b] 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 filter *and* allocates them onto the 4 processors, thereby resulting in a makespan of 14 time units that is a slowdown of $\approx 29\%$ from the optimal (Table II). In our experiments we have found that using the maximum or the minimum amongst all the computation/communication times for any given filter gives worse performance compared to using the average. These quantitative results reinforce our belief that our work is essential in further progressing research on compile-time distribution and scheduling.

When CPLEX is run in single-threaded mode, model checking requires $\approx 44\%$ less execution time than MILP. When run in multithreaded mode using 2 cores, the runtime for model checking and MILP solvers are in the same ballpark (MC performs on average $\approx 9\%$ faster than MILP). This is an encouraging result, because this means we can use model checking as a basis for applying heuristics. Finally, as a measure of scalability we were able to find an optimal solution for our largest SS example on a 16-core (12 CPU, 4 GPU same processor cores as Figure 11) architecture in 233200 seconds, while CPLEX ran out of memory and could not complete.

7. CONCLUSION

We have described a novel methodology for automatically partitioning Synchronous DataFlow (SDF) graphs onto a heterogeneous execution architecture using model

checking. Our approach encompasses both heterogeneous processing elements and heterogeneous communication costs between processors. We also model data parallelism, where the work of stateless filters may be replicated across multiple processors. We model the execution of the SDF graph on the heterogeneous architecture using computational tree logic and the optimal makespan problem is encoded as a reachability property and verified using model checking.

Our experiments show that there is real value in finding an optimal solution to the difficult problem of orchestrating stream graphs on heterogeneous architectures. We found that heuristic solutions were almost always lower quality than an optimal one, and occasionally they were much worse—up to 16 times worse in one experiment.

When compared with more traditional approaches to finding an optimal schedule, such as mixed ILP, our algorithm finds an equivalent schedule. However, our model-checking approach requires around 44% less execution time to find the optimal solution. This result shows that our novel approach to orchestrating stream graphs using model checking is comparable to traditional approaches. Furthermore, in our experiments our approach results in significantly lower execution times to find the schedule. This promising result suggests that model checking may be a suitable framework for solving a variety of multiprocessor orchestration problems.

REFERENCES

- ADAM, T. L., CHANDY, K. M., AND DICKSON, J. R. 1974. A comparison of list schedules for parallel processing systems. *Comm. ACM* 17, 12, 685–690.
- AMNELL, T., BEHRMANN, G., BENGTTSSON, J., DARGENIO, P. R., DAVID, A., FEHNKER, A., HUNE, T., JEANNET, B., LARSEN, K. G., MOLLER, M. O., PETTERSSON, P., WEISE, C., AND YI, W. 2001. *Modeling and Verification of Parallel Processes*. Springer, 99–124.
- BEHRMANN, G., DAVID, A., AND LARSEN, K. G. 2004. A tutorial on uppaal. In *International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, M. Bernardo and F. Corradini, Eds. Lecture Notes in Computer Science, vol. 3185. Springer, 200–237.
- BEHRMANN, G., HENDRIKS, M., AND MADER, A. 2005. Production scheduling by reachability analysis - A case study. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'05)*. IEEE Computer Society Press.
- BURCH, J., CLARKE, E., McMILLAN, K., DILL, D., AND HWANG, L. 1990. Symbolic model checking: 1020 states and beyond. In *Proceedings of the Symposium on Logic in Computer Science (LICS'90)*. 428–439.
- CARPENTER, P. M., RAMIREZ, A., AND AYGADE, E. 2009. Mapping stream programs onto heterogeneous multiprocessor systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*. ACM Press, New York, 57–66.
- CLARKE, E. M., GRUMBERG, O., AND PELED, D. 2000. *Model Checking*. MIT Press.
- COFFMAN, JR., E., GAREY, M., AND JOHNSON, D. 1978. An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.* 7, 1, 1–17.
- FARHAD, S. M., KO, Y., BURGSTALLER, B., AND SCHOLZ, B. 2011. Orchestration by approximation: Mapping stream programs onto multicore architectures. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM Press, New York, 357–368.
- GORDON, M. I., THIES, W., AND AMARASINGHE, S. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGOPS Oper. Syst. Rev.* 40, 151–162.
- GOVINDARAJAN, R., GAO, G. R., AND DESAI, P. 2002. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *J. VLSI Signal Process. Syst.* 31, 3, 207–229.
- GUPTA, V., JAGADESAN, R., AND SARASWAT, V. 1996. Hybrid cc, hybrid automata, and program verification. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control*, R. Alur, T. A. Henzinger, and E. D. Sontag, Eds. Lecture Notes in Computer Science, vol. 1066. Springer, 52–63.
- HOARE, C. 1978. Communicating sequential processes. *Comm. ACM* 21, 8, 666–677.
- KOHLER, W. 1975. A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. *IEEE Trans. Comput.* 24, 12, 1235–1238.
- KUDLUR, M. AND MAHLKE, S. 2008. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM Press, New York, 114–124.

- LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.* 36, 24–35.
- MALIK, A. AND GREGG, D. 2012a. Executing synchronous data flow graphs on heterogeneous execution architectures using integer linear programming. Tech. rep. TCD-CS-2012-03, School of Computer Science and Statistics, Trinity College Dublin.
- MALIK, A. AND GREGG, D. 2012b. Theorems for model-checking. Tech. rep. TCD-CS-2012-17, School of Computer Science and Statistics, Trinity College Dublin.
- ROOP, P. S., ANDALAM, S., HANXLEDEN, R., YUAN, S., AND TRAUlsen, C. 2009. Tight wcr analysis of synchronous c programs. Tech. rep. 0912, Christian-Albrechts-Universitat Kiel, Department of Computer Science. May.
- SARKAR, V. 1989. Partitioning and scheduling parallel algorithms for execution on multiprocessors. Ph.D. thesis, Stanford University. <http://www.cs.rice.edu/~vs3/PDF/Sarkar89.pdf>.
- SIH, G. C. AND LEE, E. A. 1993. Declustering: A new multiprocessor scheduling technique. *IEEE Trans. Parallel Distrib. Syst.* 4, 625–637.
- UDUPA, A., GOVINDARAJAN, R., AND THAZHUTHAVEETIL, M. J. 2009. Software pipelined execution of stream programs on gpus. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'09)*. 200–209.
- WAINGOLD, E., TAYLOR, M., SARKAR, V., LEE, V., LEE, W., KIM, J., FRANK, M., FINCH, P., DEVABHAKTUMI, S., BARUA, R., BABB, J., AMARSINGHE, S., AND AGARWAL, A. 1997. Baring it all to software: The raw machine. Tech. rep., MIT. <http://groups.csail.mit.edu/cag/raw/documents/Waingold-LCS-TR-1997.pdf>.

Received March 2012; revised November 2012; accepted January 2013