Symmetry Breaking for Multi-Criteria Mapping and Scheduling on Multicores

Pranav Tendulkar, Peter Poplavko*, and Oded Maler

Verimag (CNRS and University of Grenoble), France

Abstract. Multiprocessor mapping and scheduling is a long-old difficult problem. In this work we propose a new methodology to perform mapping and scheduling along with buffer memory optimization using an SMT solver. We target split-join graphs, a formalism inspired by synchronous data-flow (SDF) which provides a compact symbolic representation of data-parallelism. Unlike the traditional design flow for SDF which involves splitting of a big problem into smaller heuristic sub-problems, we deal with this problem as a whole and try to compute exact Pareto-optimal solutions for it. We introduce symmetry breaking constraints in order to reduce the run-times of the solver. We have tested our work on a number of SDF graphs and demonstrated the practicality of our method. We validate our models by running an image decoding application on the Tilera multicore platform.

Keywords: synchronous data-flow, multiprocessor, multicore, mapping, scheduling, SMT, SAT solver

1 Introduction

This work is motivated by a key important problem in contemporary computing: *how to exploit efficiently the resources provided by a multicore platform while executing application programs*. The problem has many variants depending on the intended use of the platform (general-purpose server or a dedicated accelerator), the specifics of the architecture (memory hierarchy, interconnect), the granularity of parallelism (instruction level, task level), the class of applications and the programming model. We focus on applications such as video, audio and other forms of signal processing which are naturally structured in a *data-flow* style as a network of interconnected software components (actors, filters, tasks). Such a description already exposes the precedence constraints among tasks and hence the task-level parallelism inherent in the application. More specifically we address applications written as *split-join graphs*, which can be viewed as a variant of the Synchronous Data-Flow (SDF) formalism [13,20], or an abstract semantic model of a subset of streaming languages such as StreaMIT [24]. Such formalisms, in addition to precedence constraints, also provide a compact *symbolic representation* of data-parallelism, namely, the presence of numerous tasks which have identical function and can be executed in parallel for different data. Once the split-join graph is annotated

* The research work of this author has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 288175 [Certainty].

with execution time figures and the data-parallel tasks have been explicitly expanded we obtain a task graph [3] whose *deployment* on the execution platform is the subject of optimization.

The deployment decisions that we consider and which may affect cost and performance are the following. First we can vary the number of processors used which gives a rough estimation of the cost of the platform (and its static power consumption). On a given configuration it remains to *map* tasks to processors, and to *schedule* the execution order on each processor. The performance measures to evaluate such a deployment are the total execution time (*latency*) and the size of the communication *buffers* which depend on the execution order. This is a multi-criteria (cost, latency and buffer size) optimization problem whose single-criterium version is already intractable. We take advantage of recent progress in SMT (SAT modulo theory) solvers [17,7] to provide a good approximation of the Pareto front of the problem. We encode the precedence and resource constraints of the problem in the theory of linear arithmetic and, following [15,14], we submit queries to the solver concerning the existence of solutions whose costs reside in various parts of the multi-dimensional cost space. Based on the answers to these queries we obtain a good approximation of the optimal trade-off between these criteria. The major computational obstacle is the intractability of the mapping and scheduling problems aggravated by the exponential blow-up while expanding the graph from symbolic to explicit form. We tackle this problem by introducing "symmetry breaking" constraints among identical processors and identical tasks. For the latter we prove a theorem concerning the optimality of schedules where instances of the same actor are executed according to a fixed *lexicographical* order.

The rest of the paper is organized as follows. In Section 2 we give some background on split-join graphs and their transformation into task graphs and prove a useful property of their optimal schedules. In Section 3 we write down in more detail the constraint-based formulation of deployment and present our multi-criteria cost-space exploration procedure. An experimental evaluation of our approach appears in Section 4, including a validation on the Tilera multicore platform. We conclude by discussing related and future work.

2 Split-Join Graphs

A parallelization factor is any number of the form α (split) or $1/\alpha$ (join) for $\alpha \in \mathbb{N}$. We use Σ^* to denote the set of sequences over a set Σ and use $\sqsubset$ for the prefix relation with $\xi \sqsubset \xi \cdot \xi'$, where $\xi \cdot \xi'$ denotes concatenation.

Definition 1 (Split-Join and Task Graphs). *A split-join graph is $S = (V, E, d, r)$ where (V, E) is a directed acyclic graph (DAG), that is, a set V of nodes, a set $E \subseteq V \times V$ of edges and no cyclic paths. The function $d : V \to \mathcal{R}_+$ defines the execution times of the nodes and $r : E \to \mathcal{Q}$ assigns a parallelization factor to every edge. An edge e is a* split, join *or* neutral *edge depending on whether $r(e) > 1$, < 1 or $= 1$. A split-join graph with $r(e) = 1$ for every e is called a task-graph and is denoted by $T = (U, \mathcal{E}, \delta)$, where the three elements in the tuple correspond to V, E and d.*

The decomposability of a task into parallelizable sub-tasks is expressed as a numerical label (parallelization factor) on a precedence edge leading to it. A label α on the edge from A to B means that every executed instance of task A spawns α instances of task B. Likewise, a $1/\alpha$ label on the edge from B to C means that all those instances of B should terminate and their outputs be joined before executing C (see Fig. 1). A task graph can thus be viewed as obtained from the split-join graph by making data parallelism *explicit* . To distinguish between these two types of graphs we call the nodes of the split-join graphs *actors* (task types) and those of the task graph *tasks*.

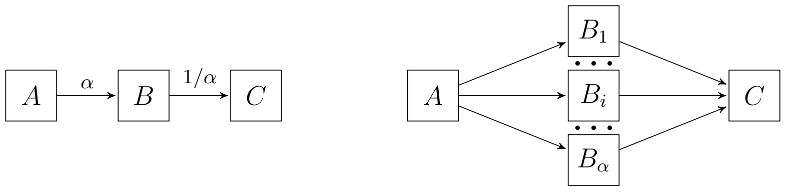


Fig. 1: A simple split-join graph and its expanded task graph. Actor B has α instances.

The DAG structure naturally induces a partial-order relation $\angle$ over the actors such that $v \angle v'$ if there is a path form v to v'. The set of *minimal* elements with respect to $\angle$ is $V_\bullet \subseteq V$ consisting of nodes with no incoming edges. Likewise, the *maximal* elements $V^\bullet$ are those without outgoing edges. An *initialized path* in a DAG is directed path $\pi = v_1 \cdot v_2 \cdots v_k$ starting from some $v_1 \in V_\bullet$. Such a path is *complete* if $v_k \in V^\bullet$. With any such path we associate the *multiplicity signature*

$$\xi(\pi) = (v_1, \alpha_1) \cdot (v_2, \alpha_2) \cdots (v_{k-1}, \alpha_{k-1})$$

where $\alpha_i = r((v_i, v_{i+1}))$. We will also abuse ξ to denote the projection of the signature on the multiplication factors, that is $\xi(\pi) = \alpha_1 \cdot \alpha_2 \cdots \alpha_{k-1}$.

To ensure that different instances of the same actor communicate with the matching instances of other actors and that such instances are joined together properly, we need an *indexing scheme* similar to indices of multi-dimensional arrays accessed inside nested loops. Because an actor may have several ancestral paths, we need to ensure that its indices via different paths agree. This will be guaranteed by a well-formedness condition that we impose on the multiplicity signatures along paths.

Definition 2 (Parenthesis Alphabet). *Let $\Sigma = \{1\} \cup \Sigma_{\{} \cup \Sigma_{\}}$ be any set of symbols consisting of a special symbol 1 and two finite sets $\Sigma_{\{}$ and $\Sigma_{\}}$ admitting a bijection which maps every $\alpha \in \Sigma_{\{}$ to $\alpha' \in \Sigma_{\}}$.*

Intuitively α and α' correspond to a matching pair consisting of a split α and its inverse join $1/\alpha$. These can be viewed also as a pair of (typed) left and right *parentheses*.

Definition 3 (Canonical Form). *The canonical form of a sequence ξ over a parentheses alphabet Σ is the sequence $\bar{\xi}$ obtained from ξ by erasing occurrences of the neutral element 1 as well as matching pairs of the form $\alpha \cdot \alpha'$.*

For example, the canonical form of $\xi = 5 \cdot 1 \cdot 3 \cdot 1 \cdot 1 \cdot 1/3 \cdot 1 \cdot 2$ is $\bar{\xi} = 5 \cdot 2$. Note that the (arithmetic) products of the factors of ξ and of $\bar{\xi}$ are equal and we denote this value by $c(\xi)$ and let $c(\epsilon) = 1$. A sequence ξ is *well-parenthesized* if $\bar{\xi} \in \Sigma_{(}^{*}$, namely its canonical form consists only of left parentheses. Note that this notion refers also to signature *prefixes* that can be *extended* to well-balanced sequences, namely, sequences with no violation of being well-parenthesized by a join not compatible with the *last* open split.

Definition 4 (Well Formedness). *A split-join graph is well formed if:*

1. *Any complete path π satisfies $c(\xi(\pi)) = 1$;*
2. *The signatures of all initialized paths are well parenthesized.*

The first condition ensures that the graph is meaningful (all splits are joined) and that the multiplicity signatures of any two paths leading to the same actor v satisfy $c(\xi) = c(\xi')$. We can thus associate unambiguously this number with the actor itself and denote it by $c(v)$. This *execution count* is the number of instances of actor v that should be executed.

The second condition forbids, for example, sequences of the form $2 \cdot 3 \cdot 1/2 \cdot 1/3$. It implies an additional property: every two initialized paths π and π' leading to the same actor satisfy $\bar{\xi}(\pi) = \bar{\xi}(\pi')$. Otherwise, if two paths would reach the same actor with different canonical signatures, there will be no way to close their parentheses by the same path suffix. Although split-join graphs *not* satisfying Condition 2 can make sense for certain computations, they require more complicated mappings between tasks and they will not be considered here, but see a brief discussion in Section 5. For well-formed graphs, a *unique canonical signature*, denoted by $\bar{\xi}(v)$, is associated with every actor.

Definition 5 (Indexing Alphabet and Order). *An actor v with $\bar{\xi}(v) = \alpha_1 \cdots \alpha_k$ defines an indexing alphabet A_v consisting of all k-digit sequences $h = a_1 \cdots a_k$ such that $0 \le a_i \le \alpha_i - 1$. This alphabet can be mapped into $\{0, \ldots, c(v) - 1\}$ via the following recursive rule:*

$$\mathcal{N}(\varepsilon) = 0 \quad \textit{and} \quad \mathcal{N}(h \cdot a_j) = \alpha_j \cdot \mathcal{N}(h) + a_j$$

We use $\ll_v$ to denote the lexicographic total order over A_v which coincides with the numerical order over $\mathcal{N}(A_v)$.

Every instance of actor v will be indexed by some $h \in A_v$ and will be denoted as v_h. We use notation h and A_v to refer both to strings and to their numerical interpretation via $\mathcal{N}$. In the latter case v_h will refer to the task in position h according to the lexicographic order $\ll_v$. See for example, tasks $B_0, B_1, \ldots$ in Figure 1.

Definition 6 (Derived Task Graph). *From a well-formed split-join graph $S = (V, E, d, r)$ we derive the task graph $T = (U, \mathcal{E}, \delta)$ as follows: $U = \{v_h | v \in V, h \in A_v\}$, $\mathcal{E} = \{(v_h, v'_{h'}) \mid (v, v') \in E, (h \sqsubseteq h' \vee h' \sqsubseteq h)\}$ and $\forall v, \forall h \in A_v,\ \delta(v_h) = d(v)$.*

Notation $h \sqsubseteq h'$ indicates that string h' is a prefix of h. To take an example, according to the definition, a split edge (v, v') is expanded to a set of edges $\{(v_h, v'_{h \cdot a}) \mid a = 0 \ldots \alpha - 1\}$, where $\alpha = r((v, v'))$. The tasks can be partitioned naturally according to their actors, letting $U = \bigcup_{v \in V} U_v$ and $U_v = \{v_h : h \in A_v\}$. A permutation $\omega : U \to U$ is *actor-preserving* if it can be written as $\omega = \bigcup_{v \in V} \omega_v$ and each ω_v is a permutation on U_v.

Definition 7 (Deployment). *A deployment for a task graph $T = (U, \mathcal{E}, \delta)$ on an execution platform with a finite set M of processors consists of a mapping function $\mu : U \to M$ and a scheduling function $s : U \to \mathcal{R}_+$ indicating the start time of each task. A deployment is called feasible if it satisfies precedence and mutual exclusion constraints, namely, for each pair of tasks we have:*

Precedence: $(u, u') \in \mathcal{E} \Rightarrow s(u') - s(u) \geq \delta(u)$

Mutual exclusion: $\mu(u) = \mu(u') \Rightarrow [(s(u') - s(u) \geq \delta(u)) \vee (s(u) - s(u') \geq \delta(u'))]$

Note that $\mu(u)$ and $s(u)$ are decision variables while $\delta(u)$ is a constant. The *latency* of the deployment is the termination time of the last task, $\max_{u \in U}(s(u) + \delta(u))$. The problem of optimal scheduling of a task-graph is already NP-hard due to the non-convex mutual exclusion constraints. This situation is aggravated by the fact that the task-graph will typically be exponential in the size of the split-join graph. On the other hand, it admits many tasks which are identical in their duration and isomorphic in their precedence constraints. In the sequel we exploit this symmetry by showing that all tasks that correspond to the same actor can be executed according to a *lexicographic order* without compromising latency.

Definition 8 (Ordering Scheme). *An ordering scheme for a task-graph $T = (U, \mathcal{E}, \delta)$ derived from a split-join graph $G = (V, E, r, d)$ is a relation $\prec = \bigcup_{v \in V} \prec_v$ where each $\prec_v$ is a total order relation on U_v.*

In the lexicographic ordering scheme $\ll$, the tasks $v_h \in U_v$ are ordered in the lexicographic order $\ll_v$ of their indices 'h'. We say that a schedule s is *compatible* with an ordering scheme $\prec$ if $v_h \prec v_{h'}$ implies $s(v_h) \leq s(v_{h'})$. We denote such an ordering scheme by $\prec^s$ and use notation $v[h]$ for the task occupying position h in $\prec^s_v$.

Lemma 1. *Let s be a feasible schedule and let v and v' be two actors such that $(v, v') \in E$. Then*

1. *If $r(v, v') = \alpha \geq 1$, then for every $h \in [0, c(v) - 1]$ and every $a \in [0, \alpha - 1]$ we have*
$$s(v'[\alpha h + a]) - s(v[h]) \geq d(v).$$
2. *If $r(v, v') = 1/\alpha$ then for every $h \in [0, c(v) - 1]$ and every $a \in [0, \alpha - 1]$ we have*
$$s(v'[h]) - s(v[\alpha h + a]) \geq d(v).$$

Proof. The precedence constraints for Case 1 are in fact $s(v'_{\alpha h + a}) - s(v_h) \geq d(v)$, and we have to prove that in this expression the lexicographic index v_h can be replaced by schedule-compatible index $v[h]$. Let $j = h\alpha + a$ and $j' = j + 1$. Since each instance of v is a predecessor of exactly α instances of v', the execution of $v'[j]$ must occur after the completion of *at least* $\lceil j'/\alpha \rceil$ instances of v. By construction, this is not earlier than the termination of the *first* $\lceil j'/\alpha \rceil$ instances of v to occur in schedule s. In our notation this can be written as:

$$s(v'[j]) \geq s(v[\lceil j'/\alpha \rceil - 1]) + d(v)$$

Substituting j and j' into the above formula we obtain our thesis. A similar argument holds for Case 2. $\square$

Theorem 1 (Lexicographic Ordering). *Every feasible schedule s can be transformed into a latency-equivalent schedule s' compatible with the lexicographic order $\ll$.*

Proof. Let ω_s be an actor-preserving permutation on U defined as $\omega_s(v_h) = v[h]$. In other words, ω_s maps the task in position h according to $\ll$ to the task occupying that position according to $\prec^s$. The new deployment is defined as

$$\mu'(v_h) = \mu(\omega_s(v_h)) \quad \text{and} \quad s'(v_h) = s(\omega_s(v_h)).$$

Permuting tasks of the same duration does not influence latency nor the satisfaction of resource constraints. All that remains to show is that s' satisfies precedence constraints. Each v_h is mapped into $v[h]$ and each of its v' sons (resp. parents) is mapped into $v'[\alpha h + a]$, $0 \leq a \leq \alpha - 1$. Hence a precedence constraint for s' of the form

$$s'(v_{h \cdot a}) - s'(v_h) \geq d(v)$$

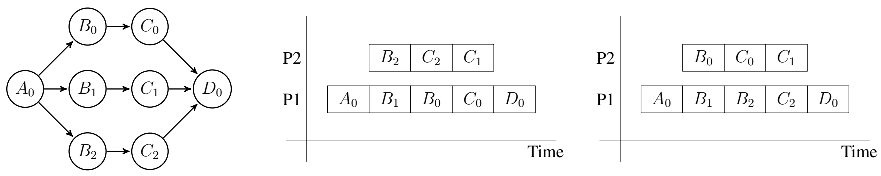
is equivalent to

$$s(v[\alpha h + a]) - s(v[h]) \geq d(v)$$

which holds by virtue of Lemma 1 and the feasibility of s. $\square$

For example, in Figure 2 we illustrate a task graph, a feasible schedule and the same schedule transformed into a lexicographic-compatible schedule by a permutation of the task indices.

The implication of this result is that we can introduce additional lexicographic constraints to the formulation of the scheduling problem without losing optimality and thus significantly reduce the search space, *i.e.,* we can do symmetry breaking.



(a) A task graph **(b)** A schedule **(c)** A lexicographic schedule

Fig. 2: Illustration of the lexicographic ordering theorem

3 Constraint-Based Feasible Cost-Space Exploration

In this section, to illustrate the effectiveness of the proposed symmetry breaking result, we encode the multicore deployment for split-join graphs as a quantifier-free SMT problem, defined by a set of constraints in the theory of linear arithmetics. Expressing scheduling problems using constraint solvers is fairly standard [1,2,27,15] and various formulations may differ in the assumptions they make about the application and the architecture and the aspects of the problem they choose to capture. For clarity and page limitation reasons, we present only the non-pipelined scheduling case.

To take advantage of symmetry breaking, we assume a multicore architecture where all cores are symmetric (homogeneous) both in terms of the computation times and the memory access times to the task communication data located in a shared memory. Fortunately, many advanced multicore architectures [16,25,11] either have a global symmetric shared memory for all processors or contain large groups of processors – so-called *clusters* – inside which this assumption holds. The access to the shared memory (including contentions and cache misses) is taken into account in the task execution times δ. In accordance with a common practice in SDF literature, we assume that a separate communication buffer is assigned to each edge (channel) (v, v') of the split-join graph so that tasks associated with the same actor read from and write to the same buffer.

To take buffer storage into account, we enrich the split-join graph model to become $G = (V, E, d, w, r)$ with $w(v, v')$ assigning to any edge in E the amount of data (in bytes) communicated from an instance of v to an instance of v' (this is called *token size* in the SDF literature). The corresponding task graph is $T = (U, \mathcal{E}, \delta, w^{\uparrow}, w^{\downarrow})$ where $w^{\uparrow}_{v,v'}$ is the amount of data *written* on the channel (v, v') by a task in U_v and $w^{\downarrow}_{v,v'}$ is the amount *read* by a task in $U_{v'}$. We assume that u allocates this memory space while starting and that u' releases it upon termination. The relation between w, $w^{\uparrow}$ and $w^{\downarrow}$ depends on the type of the edge: for a split edge $w^{\uparrow}_{v,v'} = \alpha w(v, v')$ and $w^{\downarrow}_{v,v'} = w(v, v')$ while for join edges we have $w^{\uparrow}_{v,v'} = w(v, v')$ and $w^{\downarrow}_{v,v'} = \alpha w(v, v')$.

In the following we write down the constraints that define a feasible schedule and its cost in terms of latency, number of processors and buffer size.

- **Completion time and precedence**: $e(u)$ is the time when task u terminates and a task cannot start before the termination of its predecessors.

$$\bigwedge_{u \in U} e(u) = s(u) + \delta(u) \ \wedge \bigwedge_{(u,u') \in \mathcal{E}} e(u) \leq s(u')$$

- **Mutual exclusion**: tasks running on same processor should not overlap in time.

$$\bigwedge_{u \neq u' \in U} (\mu(u) = \mu(u')) \Rightarrow (e(u) \leq s(u') \vee (e(u') \leq s(u))$$

- **Buffer**: these constraints compute the buffer size of every channel $(v, v') \in E$. They are based on the observation that buffer utilization is piecewise-constant over time, with jumps occurring upon initiation of writers and termination of readers. Hence the peak value of memory utilization can be found among one out of finitely-many starting points.
  The first constraint defines $W^{\uparrow}_{v,v'}(u, u_*)$, the contribution of writer $u \in U_v$ to the filling of buffer (v, v') observed at the start of a writer $u_* \in U_v$:

$$\bigwedge_{(v,v') \in E} \bigwedge_{u \in U_v} \bigwedge_{u_* \in U_v} \begin{array}{l} (s(u) > s(u_*)) \wedge (W^{\uparrow}_{v,v'}(u, u_*) = 0) \vee \\ (s(u) \leq s(u_*)) \wedge (W^{\uparrow}_{v,v'}(u, u_*) = w^{\uparrow}_{v,v'}) \end{array}$$

Likewise the value $W^{\downarrow}_{v,v'}(u', u_*)$ is the (negative) contribution of reader u' to buffer (v, v') observed at the start of writer u_*:

$$\bigwedge_{(v,v')\in E} \bigwedge_{u'\in U_{v'}} \bigwedge_{u_*\in U_v} \begin{array}{l} ((e(u') > s(u_*)) \wedge W^{\downarrow}_{v,v'}(u', u_*) = 0) \vee \\ (e(u') \le s(u_*)) \wedge (W^{\downarrow}_{v,v'}(u', u_*) = w^{\downarrow}_{v,v'}) \end{array}$$

The total amount of data in buffer (v, v') at the start of task $u_* \in U_v$, denoted by $R_{v,v'}(u_*)$, is the sum of contributions of all readers and writers already executed:

$$\bigwedge_{(v,v')\in E} \bigwedge_{u_*\in U_v} R_{v,v'}(u_*) = \sum_{u\in U_v} W^{\uparrow}_{v,v'}(u, u_*) - \sum_{u'\in U_{v'}} W^{\downarrow}_{v,v'}(u', u_*)$$

The buffer size for (v, v'), denoted by $B_{v,v'}$ is the maximum over all the start times of tasks in U_v:

$$\bigwedge_{(v,v')\in E} \bigwedge_{v_*\in U_v} R_{v,v'}(u_*) \le B_{v,v'}$$

– **Costs**: The following constraints define the cost vector associated with a given deployment, which is $C = (C_l, C_n, C_b)$, where the costs indicate, respectively, *latency* (termination of last task), number of *processors* used and total *buffer size*.

$$\bigwedge_{u\in U} e(u) \le C_l \;\wedge\; \bigwedge_{u\in U} \mu(u) \le C_n \;\wedge\; \sum_{(v,v')\in E} B_{v,v'} \le C_b$$

We refer to the totality of these constraints as $\varphi(\mu, s, C)$ which are satisfied by any feasible deployment (μ, s) whose cost is C.

– **Symmetry breaking:** We add two kinds of symmetry-breaking constraints, which do not change optimal costs. Firstly, we add the lexicographic task ordering constraints as implied from Theorem 1 – henceforth: *task symmetry*

$$\bigwedge_{v\in V} \bigwedge_{v_h, v_{h+1}\in U_v} s(v_h) \le s(v_{h+1})$$

where v_h denotes the instance of v at the h-th position in the order $\ll_v$.
Secondly we add fairly standard constraints to exploit *processor symmetry*: processor 1 runs task 1, processor 2 runs the lowest index task not running on processor 2, *etc.*. Therefore, let us number all tasks arbitrarily with a unique index: u^1, u^2, *etc.* The processor symmetry breaking is defined by the following constraint:

$$\mu(u^1) = 1 \;\wedge \bigwedge_{2\le i\le |U|} \mu(u^i) \le \max_{1\le j<i} \mu(u^j) + 1$$

More details on how all constraints were encoded in Z3 solver can be found in [22].

SAT and SMT solvers were designed for deciding satisfiability, not for optimization. However, such solvers can be used to find optimal costs by submitting queries concerning the existence of solutions with specific costs, which can be viewed as a binary search in the cost space with the solver acting as an oracle. We focus on *multi-criteria* optimization problems where we seek to find optimal trade-offs between latency C_l, number of processors C_n and buffer storage C_b. Such problems [8] do not admit a unique optimal solution but rather a set of *efficient* Pareto solutions [18] that cannot be improved in one cost dimension without being worsened in others. The set of such solutions, known as the *Pareto front*, represents the optimal trade-offs between the conflicting criteria. Following [14] we use queries to an SMT solver to find an approximation of the Pareto front. We summarize below the essence of the exploration methodology of [14], which can be viewed as a multi-dimensional generalization of binary search. Other approaches for multi-criteria optimization can be found in [8,28,6].

Let $Q(c)$ be a shorthand for the satisfiability query $\exists \mu \exists s$ s.t. $\varphi(\mu, s, c)$ which asks whether there is a feasible deployment whose cost vector is equal to c. If the solver answers affirmatively with some cost c we have a solution and may also conclude any cost in *forward cone* of c set $B^+(c) = \{c' \mid c' \geq c\}$ is feasible, which follows directly from the cost constraints. If the answer is negative we can conclude that any cost in the *backward cone* $B^-(c) = \{c' \mid c' \leq c\}$ is infeasible. After submitting any number of queries with different values of c we face a situation illustrated in Fig. 3. The sets $\overline{\mathcal{C}}_0$ and $\underline{\mathcal{C}}_1$ are, respectively, the maximal costs known to be infeasible (**unsat**) and minimal costs found (**sat**). Sets $\mathcal{C}_0$ and $\mathcal{C}_1$ are defined as the sets of all points known to be **unsat** and **sat**, they are equal to the forward/backward cone of the extremal points. The feasibility of costs which are outside $\mathcal{C}_0 \cup \mathcal{C}_1$ is unknown. The set $\underline{\mathcal{C}}_1$ constitutes an approximation of the Pareto front and its quality, defined as a kind of Hausdorff distance to the actual front, is bounded by its distance to the boundary of the backward cone of $\overline{\mathcal{C}}_0$.

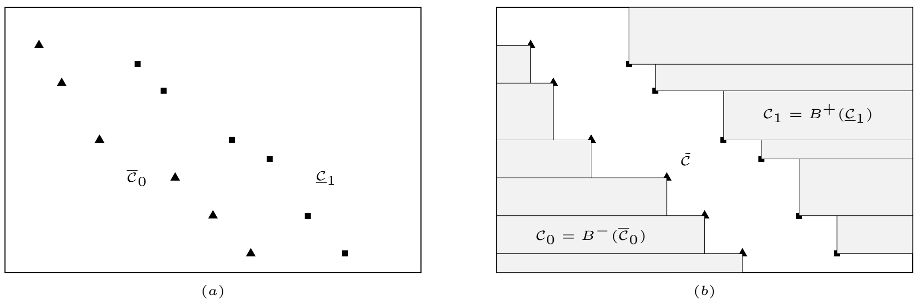

$\overline{\mathcal{C}}_0$
$\underline{\mathcal{C}}_1$
(a)
$\mathcal{C}_1 = B^+(\underline{\mathcal{C}}_1)$
$\tilde{\mathcal{C}}$
$\mathcal{C}_0 = B^-(\overline{\mathcal{C}}_0)$
(b)

Fig. 3: (a) Sets $\mathcal{C}_0$ (**unsat**) and $\mathcal{C}_1$ (**sat**) represented by their extremal points $\overline{\mathcal{C}}_0$ and $\underline{\mathcal{C}}_1$; (b) The state of our knowledge at this point as captured by $\mathcal{C}_0$ (infeasible costs) $\mathcal{C}_1$ (feasible costs) and $\tilde{\mathcal{C}}$ (unknown). The actual Pareto front is contained in the closure of $\tilde{\mathcal{C}}$.

Before we apply the exploration procedure we need to bound the cost space. For latency C_l, a lower bound is the size of the the longest path (in terms of δ) through the task graph. The upper bound is the total amount of work (sum of δ over all tasks). The bounds on buffers size are obtained by the two extreme scenarios. The lower bound is when each buffer is filled by the writer(s) to the minimal level required by the reader(s) to execute, that is, $B_{v,v'} = \alpha w(v, v')$ for an edge with multiplicity α or $1/\alpha$. The upper bound should cover the execution of all instances of v before any instance of v', $B_{v,v'} = w(v, v') \cdot \max(c(v), c(v'))$. The number of processors ranges trivially between 1 and the maximal number of processors on the platform. The width of the task-graph, when smaller than the number of processors, can serve as a tighter upper-bound as it limits the number of tasks that can execute in parallel.

Unlike the distance-oriented algorithm of [14], we use here a simpler exploration algorithm based on grid refinement. At every stage of the algorithm we refine the grid defined on the cost space and ask $Q(c)$-queries with c ranging over those newly-added grid points which are outside $\mathcal{C}_0 \cup \mathcal{C}_1$. Note that not all these new points will necessarily be queried because each query increases the size of $\mathcal{C}_0 \cup \mathcal{C}_1$ so as to include some of these points. The description so far was based on the assumption that all queries terminate. However it is well-known that as c gets closer to the boundary between **sat** and **unsat**, the computation time may grow prohibitively and the solver can get stuck. To tackle this problem we bound the time budget per query and when this bound is reached we abort the query and interpret the result as **unsat**. Choosing the appropriate value for this time-out bound is a matter of trial and error.

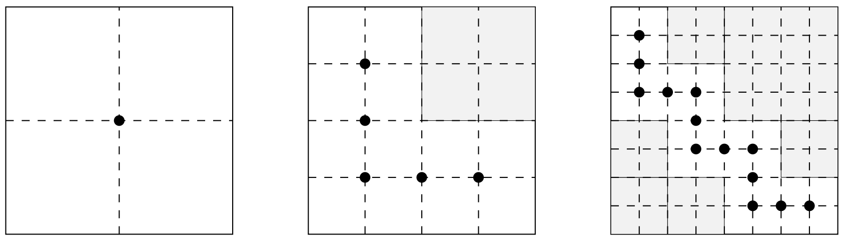
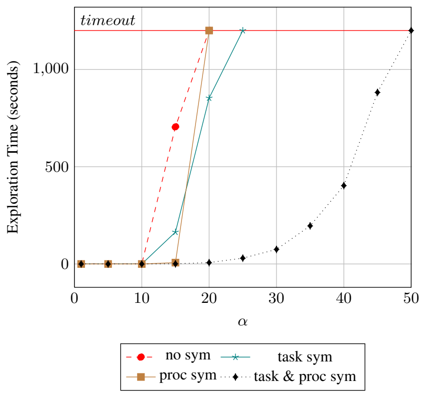


Fig. 4: Exploring the cost space via grid refinement. The dark points indicate the new candidates for exploration after each refinement.

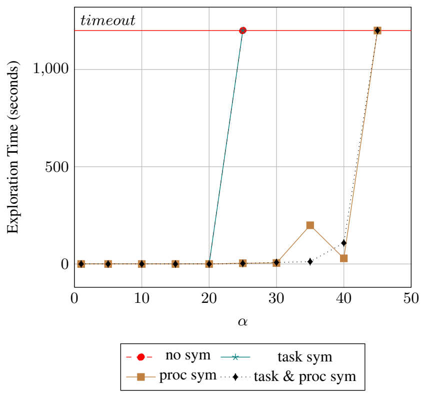
4 Experiments

In this section we investigate the performance of the cost-space exploration algorithm. First, we assess the contribution of the symmetry reduction constraints on the execution time and the quality of solutions for a synthetic example. Then we explore the cost space for a split-join graph derived from a real video application. These experiments use version 4.1 of the Z3 Solver [17] running on a Linux machine with *Intel Core i7* processor at 1.73 GHz with 4 GB of memory. Finally, we validate the model used to derive the solution by deploying a JPEG decoder on the Tilera platform [25] according to the derived schedule. The measured performance is very close to the predicted model.
Finding Optimal Latency: We use the split-join graph of Fig. 1 with various values of α to explore the effect of the symmetry reduction constraints on the performance of the solver. We start with a single cost version of the problem and perform binary search to find the minimum latency that can be achieved for a fixed number of processors. We solve the same problem using four variations of the constraints: without symmetry reduction, with processor symmetry, with task symmetry and with both. Figure 5 depicts the computation times for finding the optimal latency as a function of α on platforms

with 5 and 20 processors. We use time-out per query of 20 minutes, which is much larger than the one minute we typically use because we want to find the *exact* optimum in order to compare the effects of different symmetry constraints. Scheduling problems are known to be easy when the number of processors approaches the number of tasks. For the difficult case of 5 processors, task symmetry starts dominating beyond 10 tasks and the combination of both gives the best results. It increases the size of graphs whose optimal latency can be found (with no query executing more than 20 minutes) from $\alpha = 12$ to $\alpha = 48$. Not surprisingly, for 20 processors, the relative importance of processor symmetry grows. In Figure 5(b) we see no advantage from the task symmetry presumably because we could not try large values of α.



(a) 5-processor deployments

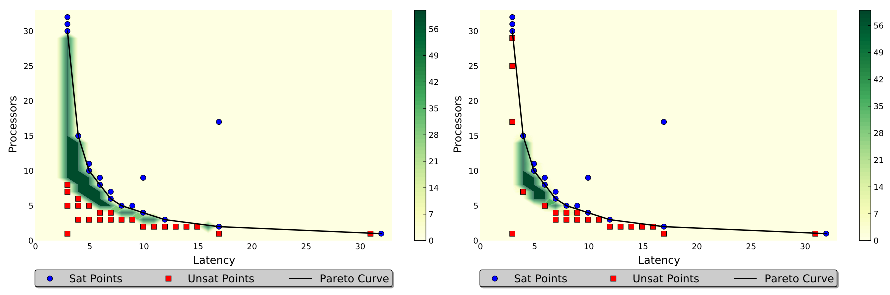


(b) 20-processor deployments

Fig. 5: Time to find optimal latency as a function of the number of tasks for 5 and 20 processors.

Processor-Latency Trade-offs: To demonstrate the effect of symmetry reductions on the Pareto front exploration we fix $\alpha = 30$ and seek trade-offs between latency and the number of processors. We use a time budget of one minute per query. Fig. 6 depicts the results obtained with and without symmetry breaking constraints. The square points show the **unsat** points whereas the circle are the **sat** points. The black curve is the approximation of the Pareto front, connecting all the minimal **sat** points. Points whose queries took long time to answer are surrounded by a dark halo whose intensity is proportional to the time (the darkest areas are around the **timeout** points). As one can see from the figure, symmetry constraints reduce significantly the number of time-outs with processor symmetry doing the job on the upper-left part of the curve while task symmetry is useful around the middle. The total time to find the minimal latency for each and every value of C_n is 42 minutes without symmetry, and 16 minutes with both types of symmetry constraints.

Video Decoder: Next we perform a 3-dimensional cost exploration for a model of a video decoder taken from [10] and described in more detail in [22]. The application admits 11 actors expanding to 122 tasks. Without any symmetry constraints the solver quickly times out for most queries of interest. Symmetry constraints do not completely eliminate time-outs but reduce them significantly and therefore the quality of the Pareto



(a) Without Symmetry Constraints **(b)** With Task and Proc Symmetry

Fig. 6: Pareto Exploration Result

front approximation is much better, as shown in Fig. 7. Note that for a sequential implementation ($C_n = 1$) the constraints improve the buffer size from 276 to 182 and for the most parallel deployment ($C_n = 122$) they reduce the latency from 10 K to 7 K and the buffer size from 333 to 229. The Pareto point $(14, 333, 62)$ found without symmetry reduction is strongly dominated by the point $(10, 229, 31)$ found with symmetry breaking. This solution improves the latency and buffer usage by roughly a third while using half of the processors. We believe it is a promising indication of the applicability of our approach and of the potential performance gains in treating the optimization problem globally.

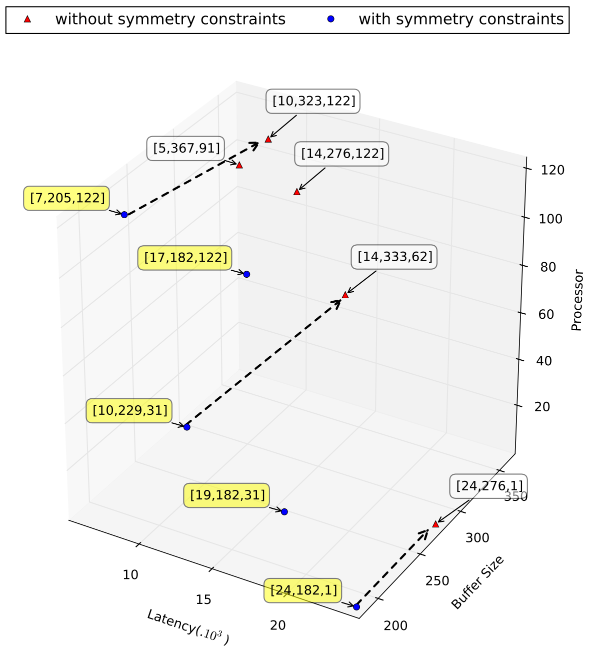


Fig. 7: Video Decoder Pareto Points

Jpeg Decoder: Finally we validate our model by deploying a JPEG decoder on the Tilera platform [25] which is a 64-core symmetric multicore platform running at 862.5 MHz. The theoretical scheduling problem that we solve is *deterministic* where task durations are assumed to be precisely known. The obtained schedule is time-triggered, given in terms of the *exact* start time function s. In reality, there are variations in execution times and in our implementation we use static order schedules, preserving only the *order* of task execution on each processor. This is a common way to generate schedules for task graphs and SDF, see for example [20]. When task durations agree with the nominal values used in the optimization problem, this scheduling policy coincides with s. Unlike the traditional work on dataflow mapping, we support mappings where the writers and readers of the same buffer storage can be spread over more than two different processors. Our experience confirms that this dynamic scheduling policy can be implemented with a reasonable amount of additional synchronization between the cores. Note also that when the schedule is compatible with lexicographical task order (justified by Theorem 1), the accesses to the channels automatically become FIFO and this facilitates the implementation of cyclic buffers.

The split-join graph of the decoder can be found in in [22]. It has three main actors: variable length decoding (vld), inverse quantization and inverse discrete cosine transform ($iq/idct$) combined and color conversion ($color$). To measure execution times we run the decoder several times on a single processor and measure the execution time of each actor. To mitigate cache effects, we consider the average execution time rather than worst case, which occurs only in the first execution due to cache misses. We use these average execution times in the model we submit to the solver. We then deploy the decoder on the platform and run it 100 times (again to dampen cache effects). The relation between the average latency (in μs) observed experimentally and the Pareto points computed by the SMT solver is depicted below and the deviation is typically smaller than 15%.

no. proc	1	3	4	6	8	12
predicted	506	314	278	261	243	226
measured	461	309	299	307	300	351

5 Discussion

The deployment of programs on parallel machines is a very old problem whose parameters change with the evolution of computer architecture. The problem exists in both software [12] and hardware [5] and in the latter it is viewed as an instance of *high-level synthesis* . Due to problem complexity the problem is often solved using heuristics such as list scheduling and/or decomposed into separate phases, for example, optimizing latency and buffers separately [21]. Recent advances in SAT and SMT solvers and other constraint propagation techniques suggest an opportunity to formulate and solve the problem in a monolithic way, avoiding the sub-optimality of decomposed solutions. For example, [15] exploits SMT solvers to combine multiple deployment sub-problems: the task-to-processor assignment, the ordering of tasks on each processor and the assignment of scalable voltage per processor. For SDF graphs, [2] and [27] combine multiple phases using a *constraint programming* (CP) engine. In the context of high-level synthesis, the tool FACTS (see [9] for references) uses branch-and-bound approach combined with constraint analysis, whereas [5] discusses various ILP formulations. In [26] a quantitative model checking engine is developed using a variant of timed automata for combined scheduling and buffer storage optimization of SDF graphs.

Various approaches to facilitate the task of the solver by additional symmetry breaking constraints have been tried, for example [19] for graph coloring or an automated method for discovering graph automorphism [4] which can lead to significant improvements [9]. However, our deployment problem does not require complex detection of isomorphic subgraphs. Instead we exploit the knowledge about the structure of the task graphs coming from the original split-join graph and not relying in any way on the graph automorphism. In fact, our approach leads to stronger symmetry reduction than could be obtained by exploiting the automorphism in the task graph as done in [9]. Theorem 1 provides the necessary compact symmetry breaking constraints that do the job. As for the restrictions that we imposed on the split-join graph compared to more general SDF graphs admitting non-divisible token production and consumption rate, let us first remark that Theorem 1 can be extended, somewhat less elegantly, to this more general case. Moreover, the extensive study of StreaMIT benchmarks found in [23] reports that most actors in most applications, fall into the category of well-formed split-join graphs that we treat.

The contribution of the paper can be summarized as follows. We provide a framework for multi-criteria optimization and cost-space exploration, not based on heuristic sub-optimal decomposition. Using symmetry reduction justified by Theorem 1, we could conduct a 3-dimensional cost-space exploration for a non-trivial problem with 122 tasks. The theorem itself generalizes the result of [9] which proves optimality of lexicographic order for one level of nesting. We prove the result for arbitrary nesting depth and give a simpler proof. In the future it might be interesting to apply this result in various alternative solution space exploration methods for the scheduling problems, *e.g.,* ILP, model checking or genetic programming.

In future, we plan to extend this work in several directions. First we will employ more refined models of data communication where different mappings imply different data transfer costs. Secondly we will consider *pipelined* executions as was done in [15,2,27,26], using *e.g.,* a finite unfolding. This will increase the number of tasks but will reduce the effect of precedence constraints. Thirdly we should adapt the methodology to a more significant variability in task duration and this will require an implementation of scheduling under uncertainty that can deviate from the task execution order provided by s. Finally we will seek ways for a more direct exploitation of the symbolic representation of data-parallel tasks and a tighter interaction between the cost exploration algorithm and the solver.

References

1. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-Based Scheduling. Kluwer international series in engineering and computer science, Kluwer (2001)
2. Bonfietti, A., Benini, L., Lombardi, M., Milano, M.: An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms. In: DATE. pp. 897–902. IEEE (2010)
3. Coffman, E.G.: Computer and job-shop scheduling theory. Wiley (1976)

4. Darga, P.T., Sakallah, K.A., Markov, I.L.: Faster symmetry discovery using sparsity of symmetries. pp. 149–154. DAC, ACM, New York, NY, USA (2008)
5. De Micheli, G.: Synthesis and optimization of digital circuits. Electrical and Computer Engineering Series, McGraw-Hill Higher Education (1994)
6. Deb, K.: Multi-Objective Optimization Using Evolutionary Algorithms. Wiley paperback series, Wiley (2009)
7. Dutertre, B., Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Computer Aided Verification, LNCS, vol. 4144, pp. 81–94. Springer (2006)
8. Ehrgott, M.: Multicriteria Optimization. Springer Berlin · Heidelberg (2005)
9. van Eijk, C.A.J., Jacobs, E.T.A.F., Mesman, B., Timmer, A.H.: Identification and exploitation of symmetries in DSP algorithms. DATE, IEEE, New York, NY, USA (1999)
10. Fradet, P., Girault, A., Poplavko, P.: SPDF: A schedulable parametric data-flow MoC. In: DATE. pp. 769–774. IEEE (2012)
11. Kalray: Kalray MPPA 256, `http://www.kalray.eu/`
12. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. 31(4), 406–471 (Dec 1999)
13. Lee, E., Messerschmitt, D.: Synchronous data flow. IEEE 75(9), 1235 – 1245 (1987)
14. Legriel, J., Guernic, C., Cotton, S., Maler, O.: Approximating the Pareto front of multi-criteria optimization problems. In: Esparza, J., Majumdar, R. (eds.) TACAS, LNCS, vol. 6015, pp. 69–83. Springer (2010)
15. Legriel, J., Maler, O.: Meeting deadlines cheaply. In: ECRTS. pp. 185–194. IEEE (2011)
16. Melpignano, D., Benini, L., Flamand, E., Jego, B., Lepley, T., Haugou, G., Clermidy, F., Dutoit, D.: Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. pp. 1137–1142. DAC, ACM, USA (2012)
17. Moura, L., Bjorner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C., Rehof, J. (eds.) TACAS, LNCS, vol. 4963, pp. 337–340. Springer (2008)
18. Pareto, V.: Manuel d'économie politique. Bull. Amer. Math. Soc. 18, 462–474 (1912)
19. Ramani, A., Aloul, F., Markov, I., Sakallah, K.: Breaking instance-independent symmetries in exact graph coloring. In: DATE. vol. 1, pp. 324–329 Vol.1. IEEE (2004)
20. Sriram, S., Bhattacharyya, S.: Embedded Multiprocessors: Scheduling and Synchronization, Second Edition. Signal Processing and Communications, Taylor & Francis (2009)
21. Stuijk, S., Geilen, M., Basten, T.: Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. pp. 899–904. DAC, ACM, USA (2006)
22. Tendulkar, P., Poplavko, P., Maler, O.: Symmetry breaking for multi-criteria mapping and scheduling on multicores. technical report TR-2013-3, Verimag (2013)
23. Thies, W., Amarasinghe, S.: An empirical characterization of stream programs and its implications for language and compiler design. pp. 365–376. PACT, ACM, USA (2010)
24. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Compiler Construction, LNCS, vol. 2304, pp. 179–196. Springer (2002)
25. Tilera, LTD: Tilera TILE64 processor, `http://www.tilera.com/`
26. Yang, Y., Geilen, M., Basten, T., Stuijk, S., Corporaal, H.: Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs. In: ESTImedia. pp. 96–105 (2009)
27. Zhu, J., Sander, I., Jantsch, A.: Buffer minimization of real-time streaming applications scheduling on hybrid CPU/FPGA architectures. pp. 1506–1511. DATE, IEEE, Belgium (2009)
28. Zitzler, E., Thiele, L.: Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. IEEE transactions on Evolutionary Computation 3(4), 257–271 (1999)