A General Constraint-centric Scheduling Framework for Spatial Architectures

Tony Nowatzki† Michael Sartin-Tarm† Lorenzo De Carli† Karthikeyan Sankaralingam†
Cristian Estan* Behnam Robatmili‡[1]

(tjn@cs.wisc.edu, msartintarm@wisc.edu, lorenzo@cs.wisc.edu, karu@cs.wisc.edu, cristian@estan.org, behnamr@qti.qualcomm.com)

†University of Wisconsin-Madison *Broadcom ‡ Qualcomm Research Silicon Valley

Abstract

Specialized execution using spatial architectures provides energy efficient computation, but requires effective algorithms for spatially scheduling the computation. Generally, this has been solved with architecture-specific heuristics, an approach which suffers from poor compiler/architect productivity, lack of insight on optimality, and inhibits migration of techniques between architectures.

Our goal is to develop a scheduling framework usable for all spatial architectures. To this end, we expresses spatial scheduling as a constraint satisfaction problem using Integer Linear Programming (ILP). We observe that architecture primitives and scheduler responsibilities can be related through five abstractions: placement of computation, routing of data, managing event timing, managing resource utilization, and forming the optimization objectives. We encode these responsibilities as 20 general ILP constraints, which are used to create schedulers for the disparate TRIPS, DySER, and PLUG architectures. Our results show that a general declarative approach using ILP is implementable, practical, and typically matches or outperforms specialized schedulers.

Categories and Subject Descriptors D.3.4 [*Processors*]: optimization, retargetable compilers; G.1.6 [*Optimization*]: Integer programming, Linear programming

Keywords Spatial Architectures; Spatial Architecture Scheduling; Integer Linear Programming

1. Introduction

Hardware specialization has emerged as an important way to sustain microprocessor performance improvements to address transistor energy efficiency challenges and general purpose processing's inefficiencies [6, 8, 19, 28]. The fundamental insight of many specialization techniques is to "map" large regions of computation to the hardware, breaking away from instruction-by-instruction pipelined execution and instead adopting a *spatial architecture* paradigm. Pioneering examples include RAW [50], Wavescalar [46] and TRIPS [9], motivated primarily by performance, and recent energy-focused proposals include Tartan [39], CCA [10], PLUG [13, 35], FlexCore [47], SoftHV [15], MESCAL [31], SPL [51], C-Cores [48], DySER [25, 26], BERET [27], and NPU [20]. A fundamental problem in all spatial architectures is the scheduling of computation to the hardware resources. Specifically, five intuitive abstractions in terms of graph-matching describe the scheduling problem: i) *placement of computation* on the hardware substrate, ii) *routing of data* on the substrate to reflect and carry out the computation semantics - including interconnection network assignment, network contention, and network path assignment, iii) *managing the timing of events* in the hardware, iv) *managing utilization* to orchestrate concurrent usage of hardware resources, and v) forming the *optimization objectives* to meet the architectural performance goals.

Thus far, these abstractions have not been modeled directly, and the typically NP-complete (depending on the hardware architecture) spatial architecture scheduling problem is side-stepped. Instead, the focus of architecture-specific schedulers has typically been on developing polynomial-time algorithms that approximate the optimal solution using knowledge about the architecture. Chronologically, this body of work includes the BUG scheduler for VLIW proposed in 1985 [17], UAS scheduler for clustered VLIW [41], synchronous data-flow graph scheduling [7], RAW scheduler [36], CARS VLIW code-generation and scheduler [33], TRIPS scheduler [12, 40], Wavescalar scheduler [37], and CCA scheduler proposed in 2008 [43]. While heuristic-based approaches are popular and effective, they have three problems: i) *poor compiler developer/architect productivity* since new algorithms, heuristics, and implementations are required for each architecture, ii) *lack of insight* on optimality of solution, and iii) *sandboxing of heuristics* to specific architectures — understanding and using techniques developed for one spatial architecture in another is very hard.

Considering these problems, others have looked at exact mathematical and constraint-theory based formulations of the scheduling problem. Table 1 classifies these prior efforts, which are based on integer linear programming (ILP) or Satisfiability Modulo Theory (SMT). They lack in some prominent ways - which perhaps explains why the heuristic-based approaches continue to be preferred. In particular, Feautrier [22] is the most related - but it lacks three of five abstractions required, and the static-placement/static-issue of VLIW restrict its applicability to the general problem. These techniques and their associated problems serve as the goal and inspiration for our work, which is to develop a declarative, constraint-theory based universal spatial architecture scheduler.

By unifying multiple facets of the related work above, specifically the past experience of architecture-specific spatial schedulers, the principal of attaining architectural generality, and the mathematical power of integer linear programming, we seek to create a solution which allows high developer productivity, provides provable properties on results, and enables true architectural generality. Achieving architectural generality through the five scheduling abstractions mentioned above is the key novelty of our work.

Implementation: In this paper, we use Integer Linear Programming (ILP) because it allows a high degree of constraint expressability, can provide strong bounds on the solution's optimality, and has fast commercial solvers like CPLEX, GUROBI, and XPRESS.

[1] Majority of work completed while author was a PhD student at UT-Austin

PLDI'13, June 16–19, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2014-6/13/06…$15.00

Year	Technique	Comments or differences to our approach
1950	ILP machine sched. [49]	M-Job-DAG to N-resource scheduling. No job communication modeling, or network contention modeling. (missing ii,iv)
1992	ILP for VLIW[22]	Modulo scheduling. Cannot model an interconnection network, spatial resources, or network contention. (missing i,ii,iv)
1997	Inst scheduling [16]	Single-Processor Modulo Scheduling. (missing i,ii,iv)
2001	Process scheduling [18]	M-Job-DAG to N-resource scheduling using dynamic programming. Has no network routing or contention modeling, fixed job delays, and no flexible objective. (missing ii,iv,v)
2002	ILP for RAW [3]	M-Job-DAG to N-resource scheduling. Not generalizable as it does not model network routing or contention, just fixed network delays. (missing ii,iv)
2007	Multiproc Sched. [34,45]	M-Job-DAG to N-resource scheduling - No path assignment or contention modeling, just fixed delays. (missing ii,iv)
2008	SMT for PLA [21]	Strict communication and computation requirements: no network contention or path assignment modeling (missing ii,iv).

Table 1. Related work – Legend: i) computation placement ii) data routing iii) event timing iv) utilization v) optimization objective

Specifically, we use the GAMS modeling language. We show that a total of 20 constraints specify the problem. We implement these constraints and report on results for three architectures picked to stress our ILP scheduler in various ways. To stress the performance deliverable by our general ILP approach, we consider TRIPS because it is a mature architecture with sophisticated specialized schedulers resulting from multi-year efforts [1, 9, 12, 40]. To represent the emerging class of energy-centric specialized spatial architectures, we consider DySER [26]. Finally, to demonstrate the generality of our technique, we consider PLUG [13, 35], which uses a radically different organization. Respectively, only 3, 1, and 10 additional (and quite straightforward) constraints are required to handle architecture-specific details. We show that standard ILP solvers allows succinct implementation (our code is less than 50 lines in GAMS), provide solutions in tractable run-times, and the mappings produced are either competitive with or significantly better than those of specialized schedulers. The general and declarative approach allows schedulers to be *specified, implemented*, *evaluated* rapidly. Our implementation is provided as an open-source download, allowing the community to build upon our work.

Paper Organization: The next section presents background on the three architectures and an ILP primer. Section 3 presents an overview of our approach, Section 4 presents the detailed ILP formulation, Section 5 discusses architecture-specific modeling constraints. Section 6 presents evaluation and Section 7 concludes. Related work was covered in the introduction.

2. Spatial Architectures and ILP Primer

2.1 Spatial Architectures

We use the term *spatial architecture* to refer to an architecture in which some subset of the hardware resources, namely functional units, interconnection network, or storage, are exposed to the compiler, whose job, as part of the scheduling phase, is to map computation and communication primitives in the instruction set architecture to these hardware resources. VLIW architectures, dataflow machines likes TRIPS and Wavescalar, tiled architectures like RAW and PLUG, and accelerators like CCA, SoftHV, and DySER all fit this definition. We now briefly describe the three spatial architectures we consider in detail, and a short primer on ILP. A detailed diagram of all three architectures is in Figure 8 (page 7).

The **TRIPS architecture** is organized into 16 tiles, with each tile containing 64 slots, with these slots grouped into sets of eight. The slots from one group are available for mapping one block of code, with different groups used for concurrently executing blocks. The tiles are interconnected using a 2-D mesh network, which implements dimension-ordered routing and provides support for flow-control and network contention. The scheduler must perform computation mapping: it takes a block of instructions (which can be no more than 128 instructions long) and assigns each instruction to one of the 16 tiles and within them, to one of the 8 slots.

The **DySER architecture** consists of *functional units* (FUs) and *switches*, and is integrated into the execution stage of a conventional processor. Each FU is connected to four neighboring switches from where it gets input values and injects outputs. The switches allow datapaths to be dynamically specialized. Using a compiler, applications are profiled to extract the most commonly executed regions, called path-trees, which are then mapped to the DySER array. The role of the scheduler is to map nodes in the path-trees to tiles in the DySER array and to determine switch configurations to assign a path for the data-flow graph edges. There is no hardware support for contention, and some mappings may result in unroutable paths. Hence, the scheduler must ensure the mappings are correct, have low latencies and have high throughput.

The **PLUG architecture** is designed to work as an accelerator for data-structure lookups in network processing. Each PLUG tile consists of a set of SRAM banks, a set of no-buffering routers, and an array of statically scheduled in-order cores. The only memory access allowed by a core is to its local SRAM, which makes all delays statically determinable. Applications are expressed as dataflow graphs with code-snippets (the PLUG literature refers to them as code-blocks) and memory associated with each node of the graph. Execution of programs is data-flow driven by messages sent from tile to tile - the ISA provides a *send* instruction. The scheduler must perform computation mapping and network mapping (dataflow edges $\rightarrow$ networks). It must ensure there is no contention for any network link, which it can do by scheduling when *send* instructions execute in a code-snippet or adjusting the mapping of graph nodes to tiles. It must also handle flow-control.

In all three architectures, multiple instances of a block, region, or dataflow graph are executing concurrently on the same hardware, resulting in additional contention and flow-control.

2.2 ILP Primer

Integer Linear Programs (ILP) are algebraic models of systems used for optimization [52]. They are composed of three parts: 1) decision variables describing the possible outcomes, 2) linear equations on these variables describing the set of valid solutions, 3) an objective function which orders solutions by desirability. A short tutorial on ILP modeling and solving techniques is here:
`http://wpweb2.tepper.cmu.edu/fmargot/introILP.html`.

2.3 Non goals

Graph abstractions and ILP (Integer Linear Programming) techniques are common in architecture and programming languages, and are used for a variety of applications unrelated to spatial scheduling. *Our goal is not to unify this wide and diverse domain.* In the following we discuss a few examples of "non-related work" and "non-goals", highlighting the difference from our techniques.

Notably, uses of ILP in register allocation, code-generation, and optimization ordering for conventional architectures [32, 42] are unrelated to the primitives of spatial architecture scheduling. Affine loop analysis and resulting instruction scheduling/code-generation for superscalar processors is a popular use of mathematical models [2, 5, 44], and since it falls within the data-dependence analysis role of the compiler, not its scheduler, is a non-goal for us. In general, modeling loops or any form of back-edges is meaningless for

#	Architecture feature	Scheduler responsibility	TRIPS	DySER	PLUG
1	Compute HW organization	Placement of computation	Homogeneous compute units	Heterogeneous compute units	Homogeneous compute units
2	Network HW organization	Routing of data	2-D grid, dimension-order routing	2-D grid, unconstrained routing	2-D multi-network grid, dimension-order routing
3	HW timing and synchronization	Manage timing of events	Data-flow execution and dynamic network arbitration	Data-flow execution and conflict-free network assignment + flow control	Hybrid data-flow and in-order execution with static compute and network timing
4	Concurrent HW usage **within a block**	Manage utilization	8-slots per compute-unit, reg-tile, data-tile	No concurrent usage; dedicated compute units, switches, links	32 slots per compute-unit; bundled links and multicast communication
	Concurrent HW usage **across blocks**		Concurrent execution of different blocks	Concurrent usage across blocks with pipelined execution	Pipelined execution across different tiles
5	Performance Goal **architecturally mandated**	Naturally enforced by ILP constraints	Any assignment legal	Throughput	Throughput
	Performance Goal **high efficiency**	ILP objective formulation	Throughput and Latency	Latency & Latency Mismatch	Latency

Table 2. Relationship between architectural primitives and scheduler responsibilities.

our work because of the nature of our scheduler's role (scheduling happens at a finer granularity than loops, so we always deal with Directed Acyclic Graphs by design).

On the architecture side, ILP and similar optimization theories have been used for hardware synthesis and VLSI CAD problems [4, 11, 31]. These techniques focus on taking a fixed computational kernel and generating a specialized hardware implementation. This is generally accomplished by extending/customizing a well-defined hardware pipeline structure. Therefore, even if in principle they share some responsibilities with our scheduler, the concrete approach they take differs significantly and cannot be applied to our case. For example, in [4] contention and timing issues are avoided by design, by provisioning enough functional units to meet the target latency, and by statically adjusting the system to timing constraints. In [31], ILP is used as a sub-step to synthesize pipelined hardware implementations of loops. Their formulation is specific to the problem and system, and does not apply to the more general scheduling problem. Finally, performance modeling frameworks, such as [38], are also orthogonal to our work. We emphasize that we have cited only a small subset of representative literature in the interest of space.

3. Overview

We present below the main insights of our approach in using constraint-solving for specifying the scheduling problem for spatial architectures. We distill the formulation into five responsibilities, each corresponding to one architectural primitive of the hardware. For a more general discussion of limitations and concerns related to our approach, see the comments in section 7 (Conclusions).

The scheduler for a spatial architecture works at the granularity of "blocks" of code, which could be basic-blocks, hyper-blocks, code-regions, or other more sophisticated partitions of program code. These blocks, which we represent as directed acyclic graphs (DAGs) consist of computation instructions, control-flow instructions, and memory access instructions that must be mapped to the hardware. We formulate the scheduling problem as spatially mapping a typed computation DAG G to a hardware graph H under certain constraints as shown by Figure 1 on page 4. For ease of explanation, we describe G as comprised of vertices and edges, while H is comprised of nodes, routers and links(formal definitions and details follow in Section 4).

To design and implement a general scheduler applicable to many spatial architectures, we observe that five fundamental architectural primitives, each with a corresponding scheduler responsibility, capture the problem as outlined in Table 2 (columns 2 and 3). Implementing these responsibilities mathematically is a matter of constraint and objective formulas involving integer variables, which form an ILP model, covered in depth in Section 4. Below we describe the insight connecting the primitives and responsibilities and highlight the mathematical approach. Table 2 summarizes this correspondence (in columns 2 and 3), and describes these primitives for three different architectures.

Computation HW organization → Placement of computation: The spatial organization of the computational resources, which could have a homogeneous or heterogeneous mix of computational units, requires the scheduler to provide an assignment of individual operations to hardware locations. As part of this responsibility, vertices in G are mapped to nodes in the H graph.

Network HW organization → Routing of data: The capabilities and organization of the network dictate how the scheduler must handle the mapping of communication between operations to the hardware substrate, i.e. the scheduler must create a mapping from edges in G to the links represented in H. As shown in the 2nd row of Table 2, the network organization consists of the spatial layout of the network, the number of networks, and the network routing algorithm. The flow of data required by the computation block and the placement of operations defines the required communication. Depending on the architecture, the scheduler may have to select a network for each message, or even select the exact path it takes.

Hardware timing/synchronization → Manage timing of events: The scheduler must take into consideration the timing properties of computation and network together with architectural restrictions, as shown in the 3rd row of table 2. In some architectures, the scheduler cannot determine the exact timing of events because it is affected by dynamic factors (e.g. memory latency through the caching hierarchy). For all architectures, the scheduler must have at least a partial view of timing of individual operations and individual messages to be able to minimize the latency of the computation block. In some architectures, the scheduler must exert extensive fine-grained control over timing to achieve static synchronization of certain events.

Concurrent hardware resource usage → Managing Utilization: Central to the difficulties of the scheduling problem is the concurrent usage of hardware resources by multiple vertices/edges in G of one node/link in H. We formalize this concurrent usage with a notion of *utilization*, which represents the amount of work a single hardware resource performs. Such concurrent usage (and hence > 1 *utilization*) can occur *within* a DAG and across concurrently executing DAGs. Overall, the scheduler must be aware of resource

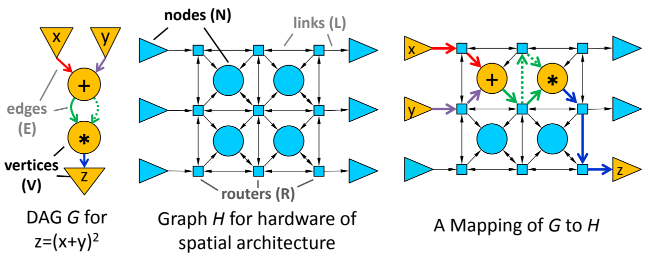


Figure 1. Example of computation G mapped to hardware H.

limits in H and which resources can be shared as shown in Table 2 row 4. For example, in TRIPS, within a single DAG, 8 instruction-slots share a single ALU (node in H), and across concurrent DAGs, 64 slots share a single ALU in TRIPS. In both cases, this node-sharing leads to contention on the links as well.

Performance goal $\rightarrow$ Formulate ILP objective: The performance goals of an architecture generally fall into two categories: those which are enforced by certain architectural limitations or abilities, and those which can be influenced by the schedule. For instance, both PLUG and DySER are throughput engines that try to perform one computation per cycle, and *any legal* schedule will naturally enforce this behavior. For this type of performance goal, the scheduler relies on the ILP constraints already present in the model. On the other hand, the scheduler generally has control over multiple quantities which can improve the performance. This often means deciding between the conflicting goals of minimizing the latency of individual blocks and managing the utilization among the available hardware resources to avoid creating bottlenecks, which it manages by prioritizing optimization quantities.

4. General ILP framework

This section presents our general ILP formulation in detail. Our formal notation closely follows our ILP formulation in GAMS instead of the more conventional notation often used for graphs in literature. We represent the computation graph as a set of vertices V, and a set of edges E. The computation DAG, represented by the adjacency matrix $G(V \cup E, V \cup E)$, explicitly represents edges as the connections between vertices. For example, for some $v \in V$ and $e \in E$, $G(v, e) = 1$ means that edge e is an output edge from vertex v. Likewise, $G(e, v) = 1$ signifies that e is an input to vertex v. For convenience, lowercase letters represents elements of the corresponding uppercase letters' set.

We similarly represent the hardware graph as a set of hardware computational resource nodes N, a set of routers R which serve as intermediate points in the routing network, and a set of L unidirectional links which connect the routers and resource nodes. The graph which describes the network organization is given by the adjacency matrix $H(N \cup R \cup L, N \cup R \cup L)$. To clarify, for some $l \in L$ and $n \in N$, if the parameter $H(l, n)$ was 0, link l would not be an input of node n. Hardware graphs are allowed to take any shape, and typically do contain cycles. Terms vertex/edge refer to members in G, and node/link to members in H.

Some of the vertices and nodes represent not only computation, but also inputs and outputs. To accommodate this, vertices and nodes are "typed" by the operations they can perform, which also enables the support of general heterogeneity in the architecture. For the treatment here, we abstract the details of the "types" into a compatibility matrix $C(V, N)$, indicating whether a particular vertex is compatible with a particular node. When equations depend on specific types of vertices, we will refer this set as V_{type}.

Figure 1 shows an example G graph, representing the computation $z = (x + y)^2$, and an H graph corresponding to a simplified version of the DySER architecture. Here, triangles represent input/output nodes and vertices, and circles represent computation nodes and vertices. Squares represent elements of R, which are routers composing the communication network. Elements of E are shown as unidirectional arrows in the computation DAG, and elements of L as bidirectional arrows in H representing two unidirectional links in either direction.

Inputs: Computation DAG Description (G)	
V	Set of computation vertices.
E	Set of Edges representing data flow of vertices
$G(V \cup E, V \cup E)$	The computation DAG
$\Delta(E)$	Delay between vertex activation and edge activation.
$\Delta(V)$	Duration of vertex.
$\Gamma(E)$ (PLUG)	Delay between vertex activation and edge reception.
B_e	Set of bundles which can be overlapped in network.
B_v (PLUG only)	Set of mutually exclusive vertex bundles.
$B(E \cup V, B_e \cup B_v)$	Parameter for edge/vertex bundle membership.
P (TRIPS only)	Set of control flow paths the computation can take
$A_v(P, V)$, $A_e(P, E)$ (TRIPS)	Activation matrices defining which vertices and edges get activated by given path

Inputs: Hardware Graph Description (H)	
N	Set of hardware resource Nodes.
R	Routers which form the network
L	Set of unidirectional point-to-point hardware Links
$H(N \cup R \cup L, N \cup R \cup L)$	Directed graph describing the Hardware
$I(L, L)$	Link pairs incompatible with Dim. Order Routing.

Inputs: Relationship between G/H	
$C(V, N)$	Vertex-Node Compatibility Matrix
MAX_N, MAX_L	Maximum degree of mapping for nodes and links.

Variables: Final Outputs	
$M_{vn}(V, N)$	Mapping of computation vertices to hardware nodes.
$M_{el}(E, L)$	Mapping of edges to paths of hardware links
$M_{bl}(B_e, L)$	Mapping of edge bundles to links
$M_{bn}(B_v, N)$ (PLUG only)	Mapping of vertex bundles to nodes
$\delta(E)$ (PLUG)	Padding cycles before message sent.
$\gamma(E)$ (PLUG)	Padding cycles before message received.

Variables: Intermediates	
$O(L)$	The order a link is traversed in.
$U(L \cup N)$	Utilization of links and nodes.
$U_p(P)$ (TRIPS)	Max Utilization for each path P.
$T(V)$	Time when a vertex is activated
$X(E)$	Extra cycles message is buffered.
$\lambda(b, e)$ (PLUG)	Cycle when e is activated for bundle b
LAT	Total latency for scheduled computation
SVC	Service interval for computation.
MIS	Largest Latency Mismatch.

Table 3. Summary of formal notation used.

The scheduler's job is to use the description of the typed computation DAG and hardware graph to find a mapping from computation vertices to computation resource nodes and determine the hardware paths along which individual edges flow. Figure 1 also shows a correct mapping of the computation graph to the hardware graph. This mapping is defined by a series of constraints and variables described in the remainder of this Section, and these variables and scheduler inputs are summarized in Table 3.

We now describe the ILP constraints which pertain to each scheduler responsibility, then show a diagram capturing this responsibility pictorially for our running example in Figure 1.

Responsibility 1: Placement of computation.

The first responsibility of the scheduler is to map vertices from the computation DAG to nodes from the hardware graph. Formally, the scheduler must compute a mapping from V to N, which we represent with the matrix of binary variables $M_{vn}(V, N)$. If $M_{vn}(v, n) = 1$, then vertex v is mapped to node n, while

$M_{vn}(v,n) = 0$ means that v is not mapped to n. Each vertex $v \in V$ must be mapped to exactly one compatible hardware node $n \in N$ in accordance with $C(v,n)$. The mapping for incompatible nodes must also be disallowed. This gives us:

$$\forall v \;\; \Sigma_{n|C(v,n)=1} M_{vn}(v,n) = 1 \tag{1}$$

$$\forall v, n | C(v,n) = 0, \;\; M_{vn}(v,n) = 0 \tag{2}$$

An example mapping with corresponding assignments to M_{vn} is shown in Figure 2.

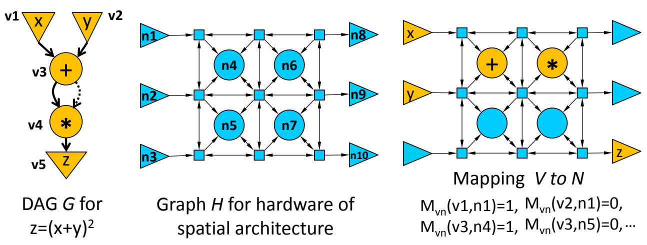

v1
v2
v3
v4
v5
DAG G for z=(x+y)²
n1
n2
n3
n4
n5
n6
n7
n8
n9
n10
Graph H for hardware of spatial architecture
Mapping V to N
M_vn(v1,n1)=1, M_vn(v2,n1)=0,
M_vn(v3,n4)=1, M_vn(v3,n5)=0,...

Figure 2. Placement of computation

Responsibility 2: Routing of data

The second responsibility of the scheduler is to map the required flow of data to the communication paths in the hardware. We use a matrix of binary variables $M_{el}(E, L)$ to encode the mapping of edges to links. Each edge e must be mapped to a sequence of one or more links l. This sequence must start from and end at the correct hardware nodes. We constrain the mappings such that if a vertex v is mapped to a node n, every edge e leaving from v must be mapped to one link leaving from n. Similarly, every edge arriving to v must be mapped to a link arriving to n.

$$\forall v, e, n | G(v,e), \Sigma_{l|H(n,l)}, M_{el}(e,l) = M_{vn}(v,n) \tag{3}$$

$$\forall v, e, n | G(e,v), \Sigma_{l|H(l,n)}, M_{el}(e,l) = M_{vn}(v,n) \tag{4}$$

In addition, the scheduler must ensure that each edge is mapped to a *contiguous* path of links. We achieve this by enforcing that for each router, either we have no incoming or outgoing links mapped to a given edge, or we have exactly one incoming and exactly one outgoing link mapped to the edge.

$$\forall e \in E, r \in R \quad \Sigma_{l|H(l,r)}, M_{el}(e,l) = \Sigma_{l|H(r,l)} M_{el}(e,l) \tag{5}$$

$$\forall e \in E, r \in R \quad\quad \Sigma_{l|H(l,r)}, M_{el}(e,l) \leq 1 \tag{6}$$

Figure 3 shows these constraints applied to the example.

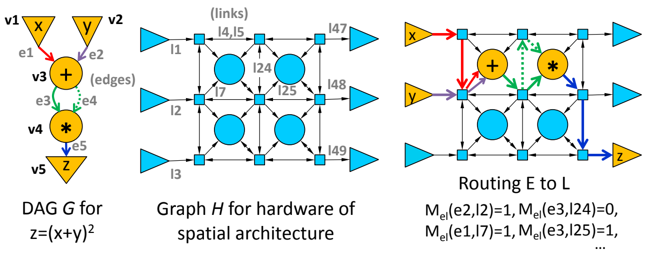

v1
v2
e1
e2
v3
(edges)
e3
e4
v4
e5
v5
DAG G for z=(x+y)²
(links)
l4,l5
l47
l1
l24
l2
l7
l25
l48
l49
l3
Graph H for hardware of spatial architecture
Routing E to L
M_el(e2,l2)=1, M_el(e3,l24)=0,
M_el(e1,l7)=1, M_el(e3,l25)=1,
...

Figure 3. Routing of data.

Some architectures require dimension order routing: a message propagating along the X direction may continue on a link along the Y direction, but a message propagating along the Y direction cannot continue on a link along the X direction. To enforce this restriction, we expand the description of the hardware with $I(L,L)$, the set of link pairs that cannot be mapped to the same edge (i.e. an edge cannot be assigned to a path containing any link pair in this set).

$$\forall l, l' | I(l,l'), e \in E, \;\; M_{el}(e,l) + M_{el}(e,l') \leq 1 \tag{7}$$

Responsibility 3: Manage timing of events

We capture the timing through a set of variables $T(V)$ which represents the time at which a vertex $v \in V$ starts executing. For each edge connecting the vertices v_{src} and v_{dst}, we compute the $T(v_{dst})$ based on $T(v_{src})$. This time is affected by three components. First, we must take into account the $\Delta(E)$, which is the number of clock cycles between the start time of the vertex and when the data is ready. Next is the total routing delay, which is the sum of the number of mapped links between v_{src} and v_{dest}. Since the data carried by all input edges for a vertex might not all arrive at the same time, the variable $X(E)$ describes this mismatch.

$$\forall v_{src}, e, v_{dest} | G(v_{src}, e) \& G(e, v_{dest}),$$

$$T(v_{src}) + \Delta(e) + \Sigma_{l \in L} M_{el}(e,l) + X(e) = T(v_{dest}) \tag{8}$$

The equation above cannot fully capture dynamic events like cache misses. Rather than consider all possibilities, the scheduler simply assumes best-case values for unknown latencies (alternatively, these could be attained through profiling or similar means). Note that this is an issue for specialized schedulers as well.

With the constraints thus far, it is possible for the scheduler to overestimate edge latency because the link mapping allows fictitious cycles. As shown by the cycle in the bottom-left quadrant of Figure 4, the links in this cycle falsely contribute to the time between input "x" and vertex "+". This does not violate constraint 5 because each router involved contains the correct number of incoming / outgoing links.

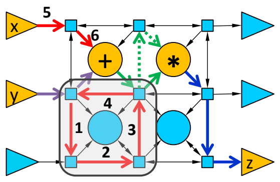


Figure 4. Fictitious cycles.

In many architectures, routing constraints (see constraint 7) make such loops impossible, but when this is not the case we eliminate cycles through a new constraint. We add a new set of variables $O(L)$, indicating the partial order in which links activated. If an edge is mapped to two connected links, this constraint enforces that the second link must be of later order.

$$\forall l, l', e \in E | H(l,l'), \; O(l) + M_{el}(e,l) + M_{el}(e,l') - 1 \leq O(l') \tag{9}$$

Figure 5 shows the intermediate variable assignments that the constraints for timing provide.

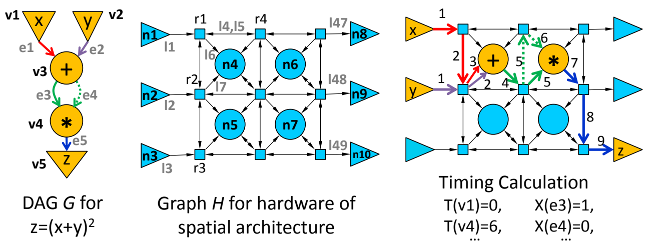


Figure 5. Timing of computation and communication.

Responsibility 4: Managing Utilization

The utilization of a hardware resource is simply the number of cycles for which it can not accept a new unit of work (computation or communication) because it is handling work corresponding to another computation. We first discuss the modeling of link utilization $U(L)$, then discuss node utilization $U(N)$.

$$\forall l \in L, \quad U(l) = \Sigma_{e \in E} M_{el}(e,l) \tag{10}$$

The equation above models a link's utilization as the sum of its mapped edges and is effective when each edge takes up a resource. On the other hand, some architectures allow for edges to be overlapped, as in the case of multicast, or if it is known that sets of messages are mutually exclusive (will never activate at the same time). This requires us to extend our notion of utilization with the concept of *edge-bundles*, which represent edges that can be mapped to the same link at no cost. The set B_e denotes edge-bundles, and $B(E, B_e)$ defines its relationship to edges. The following three constraints ensure the correct correspondence between the mapping of edges to links and bundles to links, and compute the link's utilization based on the edge-bundles.

$$\forall e, b_e | B(e, b_e), l \in L, \qquad M_{bl}(b_e, l) \geq M_{el}(e, l) \tag{11}$$

$$\forall b_e \in B_e, l \in L, \quad \Sigma_{e \in B(e,b_e)} M_{el}(e, l) \geq M_{bl}(b_e, l) \tag{12}$$

$$\forall l \in L, \qquad U(l) = \Sigma_{b_e \in B} M_{bl}(b, l) \tag{13}$$

To compute the vertices' utilization, we must additionally consider the amount of time that a vertex fully occupies a node. This time, $\Delta(V)$, is always 1 when the architecture is fully pipelined, but increases when the lack of pipelining limits the use of a node n in subsequent cycles. To compute utilization, we simply sum $\Delta(V)$ over vertices mapped to a node:

$$\forall n \in N \;\; U(n) = \Sigma_{v \in V} \Delta(v) M_{vn}(v, l) \tag{14}$$

For many spatial architectures we use utilization-limiting constraints such as those below. One application of these constraints are hardware limitations in the number of registers available, instruction slots, etc. Also, they ensure lack of contention with operations or messages from within the same block or other blocks executing concurrently.

$$\forall l \in L, \quad U(l) \leq MAX_L \tag{15}$$

$$\forall n \in N, \quad U(n) \leq MAX_N \tag{16}$$

As shown in the running DySER example below in Figure 6, we limit the utilization of each link $U(l)$ to $MAX_L = 1$. This ensures that only a single message per block traverses the link, allowing the DySER's arbitration-free routers to operate correctly.

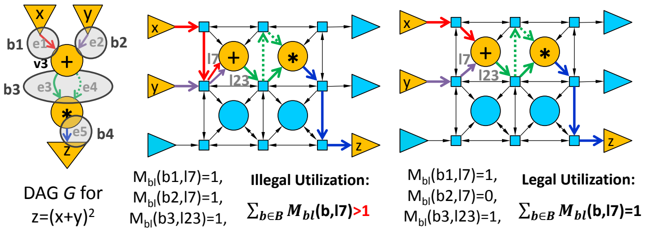


Figure 6. Utilization Management.

Responsibility 5: Optimizing performance

The constraints governing the previous sections model the quantities which capture only *individual* components for correctness and performance. However, the final responsibility of the scheduler is to manage the overall correctness while providing performance in the context of the overall system. In practice, this means that the scheduler must balance notions of latency and throughput. Having multiple conflicting targets requires strategic resolution, since there is not necessarily a single solution which optimizes both. The strategy we take is to supply to the scheduler a set of variables to optimize for with their associated priority.

To calculate the critical path latency, we first initialize the input vertices to zero (or some known value) then find the maximum latency of an output vertex LAT. This represents the scheduler's estimate of how long the block would take to complete.

$$\forall v \in V_{in}, \qquad T(v) = 0 \tag{17}$$

$$\forall v \in V_{out}, \quad T(v) \leq LAT \tag{18}$$

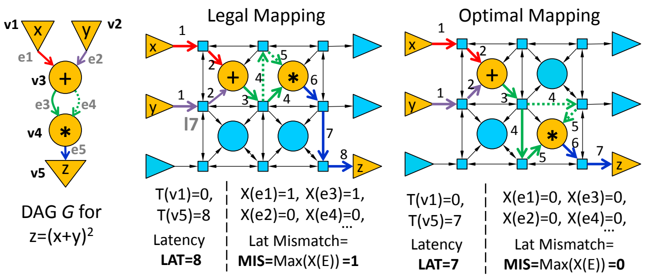
To model the throughput aspects, we utilize the concept of the service interval SVC, which is defined as the minimum number of cycles between successive invocations when no data dependencies between invocations exists. We compute SVC by finding the maximum utilization on any resource.

$$\forall n \in N, \quad U(n) \leq SVC \tag{19}$$

$$\forall l \in L, \quad U(l) \leq SVC \tag{20}$$

For fully pipelined architectures, SVC is naturally forced to 1, so it is not an optimization target. Other notions of throughput are possible, as in the case of DySER, where minimizing the latency mismatch MIS is the throughput objective (see Section 5.2).

For our running example, the final solution is shown in Figure 7, where the critical path latency LAT and the latency mismatch MIS (mentioned above), are both optimized by the scheduler.



v1
x
y
v2
e1
e2
v3
e3
e4
v4
e5
v5
z
DAG *G* for z=(x+y)²
Legal Mapping
l7
T(v1)=0, T(v5)=8
Latency LAT=8
X(e1)=1, X(e3)=1, X(e2)=0, X(e4)=0, …
Lat Mismatch= MIS=Max(X(E)) =1
Optimal Mapping
T(v1)=0, T(v5)=7
Latency LAT=7
X(e1)=0, X(e3)=0, X(e2)=0, X(e4)=0, …
Lat Mismatch= MIS=Max(X(E)) =0

Figure 7. Optimizing performance.

5. Architecture-specific modeling

In this section, we describe how the general formulation presented above is used by three diverse architectures. Figure 8 shows schematics and H graphs for the three architectures.

5.1 Architecture-specific details for TRIPS

Computation organization → Placement of computation: Figure 8 depicts the graph H we use to describe a 4-tile TRIPS architecture. A tile in TRIPS is comprised of nodes $n \in N$ denoting a functional unit in the tile and $r \in R$ representing its router - the two are connected with one link in either direction. The router also connects to the routers in the neighboring tiles. The functional unit has a self-loop that denotes the bypass of the tile's router to move results into input slots for operations scheduled on the same tile.

Network organization → Routing data: Since messages are dedicated and point-to-point (as opposed to multicast), we use constraints modeling each edge as consuming a resource and contributing to the total utilization. The TRIPS routers implement dimension-order routing, i.e. messages first travel along the X axis, then along the Y axis. TRIPS uses the $I(L, L)$ parameter, which disallows the mapping of certain link pairs, to invalidate any paths which are not compatible with dimension-order.

HW timing → Managing timing of events: We can calculate network timing without any additions to the general formulation.

Concurrent HW usage → Utilization: TRIPS allows significant level of concurrent hardware usage which affects both the latency and throughput of blocks. Specifically, the maximum number of vertices per node is $MAX_N = 8$. The utilization on links is used to finally formulate the objective function.
Extensions: For TRIPS, the scheduler must also account for control flow when computing the utilization and ultimately the service interval for throughput. Simple extensions, as explained below, can in general handle control flow for any architecture and could

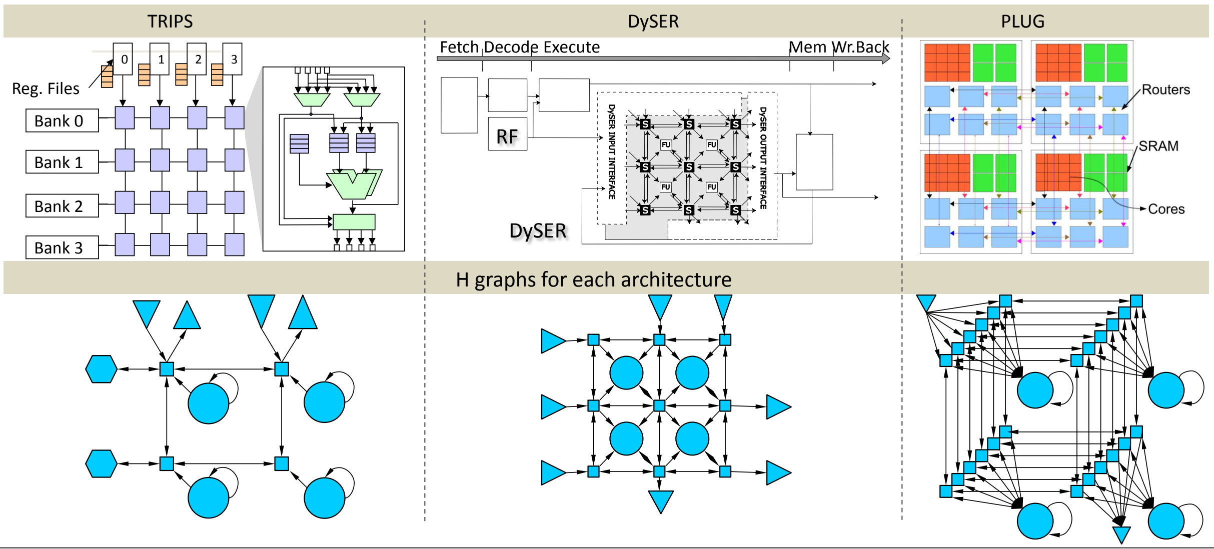


Figure 8. Three candidate architectures and corresponding H graphs, considering 4 tiles for each architecture.

Architecture	Description	ILP Modeling and scheduler responsibility	ILP Constraints
Compute HW organization	16 titles, 6 routers per tile	Each tile is 7 nodes in *H*, one in *N*, and six in *R*.	Gen. ILP framework
	4 mem-banks per tile	Handled with utilization	Gen. ILP framework
	32 cores per tile	Handled with utilization	Gen. ILP framework
Network HW organization	2D nearest neighbor mesh	Node *n* connected to *r*; *r* connected to 4 neighbors	Gen. ILP framework
	Dimension order routing	*I(L,L)* configure for dimension-order routing	Gen. ILP framework
	Multicast messages deliver values at every intermediate node on path	Enforce mapping of multicast messages to link to computation node	Eq. 25
HW timing and synchronization	Code-scheduling of network send instructions	Variables for send timing and receive time	Δ(E); Γ(E)
	Send instructions scheduled to avoid n/w conflicts	Variables for delaying send/receive timing by padding with no-ops	Eq. 26, 27a-c, 28
Concurrent HW usage	4 mem-banks per tile	Manage utilization (MAX_N = 4)	Gen. ILP framework
	Dedicated network per message	Manage utilization (MAX_L=1)	Gen. ILP framework
	Mutually exclusive activation of nodes in G	Concept of vertex bundles and utilization refined	Eq. 29, 30, 31
	Code-length limitations (maximum is 32) handled for all code on tile	Manage utilization and combine with vertex bundles	Eq. 32, 33, 34

Table 4. Description of ILP model implementation for PLUG

belong in the general ILP formulation as well. Let P be the set of control flow paths that the computation can take through G. Note that $p \in P$ is not actually a path through G, but the subset of its vertices and edges activated in a given execution. Let $A_v(P, V)$ and $A_e(P, E)$ be the activation matrices defining, for each vertex and edge of the computation, whether they are activated when a given path is taken or not. For each path we define the maximum utilization on this path $W_p(P)$. These constraints are similar to the original utilization constraints (10, 14), but also take control flow activation matrices into account.

$$\forall l \in L, p \in P, \qquad \Sigma_{e \in E} M_{el}(e,l) A_e(p,e) \leq W_p(p) \qquad (21)$$
$$\forall n \in N, p \in P, \quad \Sigma_{v \in V} M_{vn}(v,n) \Delta(v) A_v(p,v) \leq W_p(p) (22)$$

And an additional constraint for calculating overall service interval:

$$SVC = \Sigma_{p \in P} W_p(p) \qquad (23)$$

Note that this heuristic provides the same importance to all control-flow paths. With profiling or other analysis, differential weighting can be implemented.

Objective formulation: For the TRIPS architecture, we empirically found that optimizing for throughput is of higher importance, in most cases, then for latency. Therefore, our strategy is to first minimize the SVC, add the lowest value as a constraint, and then optimize for LAT. The following is our solution procedure, where numbers refer to constraints from the formulation:

$$min\ SVC\ s.t.\ [1, 2, 3, 4, 5, 6, 7, 8, 10, 14, 15, 16, 17, 18, 21, 22, 23]$$
$$min\ LAT\ s.t.\ [1, 2, 3, 4, 5, 6, 7, 8, 10, 14, 15, 16, 17, 18, 21, 22, 23]$$
$$and\ SVC\ =\ SVC_{optimal}$$

5.2 Architecture-specific details for DySER

Computation organization → Placement of computation: We model DySER with the hardware graph H shown in Figure 8; heterogeneity is captured with the $C(V, N)$ compatibility matrix.

Network organization → Routing data: We use bundle-link mapping constraints to model multicast, and constraint 9 to prevent fictitious cycles. Since the network has the ability to perform multicast messages and can route multiple edges on the same link, we use the bundle-link mapping constraints. Since there is no ordering constraint on the network, we need to prevent fictitious cycles.

HW timing → Managing timing of events: No additions to the general formulation are required.

Concurrent HW usage → Utilization: Since DySER can only route one message per link, and max one vertex to a node, both MAX_L and MAX_N are set to 1.

Objective Formulation: DySER throughput can be as much as one computation G per cycle, since the functional units themselves are pipelined. However, throughput degradation can occur because of the limited buffering available for messages. The utilization defined in the general framework does not capture this problem because it only measures the usage of functional units and links, not of buffers. Unlike TRIPS, where all operands are buffered as long as needed in named registers, DySER buffers messages at routers and at most one message per edge is buffered at each router. Thus, two paths that diverge and then converge, but have different lengths, will also have different amounts of buffering. Combined with backpressure, this can reduce throughput.

Computing the exact throughput achievable by a DySER schedule is difficult, as multiple such pairs of converging paths may exist - even paths that converge after passing through functional units affect throughput. Instead we note that latency mismatches always manifest themselves as extra buffering delays $X(e)$ for some edges, so we model latency mismatch as MIS:

$$\forall e \in E, X(e) \leq MIS \tag{24}$$

Empirically, we found that external limitations on the throughput of inputs is greater than that of computation. For this reason, the DySER scheduler first optimizes for latency, adds the latency of the solution as a constraint, then optimizes for throughput by minimizing latency mismatch MIS, as below:

$$min\, LAT\; s.t.\; [\,1,2,3,4,5,6,8,9,11,12,13,14,15,16,17,18,24]$$

$$min\, MIS\; s.t.\; [\,1,2,3,4,5,6,8,9,11,12,13,14,15,16,17,18,24] \quad and\, LAT \;=\; LAT_{optimal}$$

5.3 Architecture-specific details for PLUG

The PLUG architecture is radically different from the previous two architectures since all decisions are static. Our formulation is general enough that it works for PLUG with only 10 predominantly simple additional constraints. In the interest of clarity, we summarize the key concepts of the PLUG architecture, corresponding ILP model, and additional equations in Table 4. The grayed rows summarize the extensions, and this section's text describes them.

Computation organization $\rightarrow$ Placement of computation: See Table 4, row 1. No additional constraints required.

Network organization $\rightarrow$ Routing data: See Table 4, row 2. *Additional constraints:* Multicast handled with edge-bundles: Let $B_{multi} \subset B_e$ be the subset of edge-bundles that involve multicast edges. The following constraint, which considers links through a router to a node, then enforces that the bundle mapped to the router link must also be mapped to the node's incoming link.

$$\forall b \in B_{multi}, l, r, l', n | H(l,r) \& H(r,l') \& H(l',n), \quad M_{bl}(b,l) \leq M_{bl}(b,l') \tag{25}$$

HW timing $\rightarrow$ Managing timing of events: See Table 4, row 3. *Additional constraints:* We need to handle the timing of send instructions. We use $\Delta(E)$ and the newly-introduced $\Gamma(E)$ to respectively indicate the relative cycle number of the corresponding send instruction and use instruction.

Network contention is avoided by code-scheduling the send instructions with NOP padding to create appropriate delays and equalize all delay mismatch. $\delta(E)$ denotes sending delay added, and $\gamma(E)$ denotes receiving delay added. To model the timing for PLUG, we augment equation 8 as follows:

$$\forall v_{src}, e, v_{dst} | G(v_{src}, e) \& G(e, v_{dst}),$$
$$T(v_{src}) + \Sigma_{l \in L} M_{el}(e,l) + \Delta(e) + \delta(e) = T(v_{dst}) + \Gamma(e) + \gamma(e) \tag{26}$$

Because the insertion of no-ops can only change timing in specific ways, we use two constraints to further link $\delta(E)$ and $\gamma(E)$ to $\Delta(E)$ and $\Gamma(E)$. The first ensures that the scheduler never attempts to pad a negative number of NOPs. The second ensures that sending delay $\delta(E)$ is the same for all multicast edges carrying the same message.

To implement these constraints we use the following 4 sets concerning distinct edges e, e': $SI(e,e')$ has the set of pairs of edges arriving to the same vertex such that $\Gamma(e) < \Gamma(e')$, $LIFO(e,e')$ has, for each vertex with both input and output edges, the last input edge e and the first output edge e', $SO(e,e')$ has the pairs of output edges with the same source vertex such that $\Delta(e) < \Delta(e')$, and $EQO(e,e')$ has the pairs of output edges leaving the same node concurrently.

$$\forall e, e' | SI(e,e'), \gamma(e) \leq \gamma(e') \tag{27a}$$
$$\forall e, e' | LIFO(e,e'), \gamma(e) \leq \delta(e') \tag{27b}$$
$$\forall e, e' | SO(e,e'), \delta(e) \leq \delta(e') \tag{27c}$$
$$\forall e, e' | EQO(e,e'), \delta(e) = \delta(e') \tag{28}$$

Concurrent HW usage $\rightarrow$ Utilization: See Table 4, row 4. *Additional constraints:* PLUG groups nodes in G into "super-nodes" (logical-page), and programmatically only a single node executes in every super-node. This mutual exclusion behavior is modeled by partitioning V into a set of vertex bundles B_v with $B(V, B_v)$ indicating to which bundle a vertex $v \in V$ belongs. We introduce $M_{bn}(b,n)$ to model the mapping of bundles to nodes, enforced by the following constraints:

$$\forall v, b_v | B(v, b_v), n \in N, \quad M_{bn}(b_v, n) \geq M_{vn}(v, n) \tag{29}$$
$$\forall b_v \in B_v, n \in N, \quad \Sigma_{v \in B(v,b_v)} M_{vn}(v,n) \geq M_{bn}(b_v, n) \tag{30}$$

We then define the utilization based on the number of vertex bundles mapped to a node. We also instantiate edge bundles b_e for all the set of edges coming from the same vertex bundle and going to the same destination. Since all the edges in such a bundle are *logically* a single message source, the schedule must equalize the receiving times of the message they send. Let $B_{mutex} \subseteq B_e$ be the set of edge-bundles described above. Then we add the following timing constraint:

$$\forall e, e', b_x \in B_{mutex} | B(e, b_x) \& B(e', b_x), \quad \gamma(e) = \gamma(e') \tag{31}$$

Additionally, architectural constraints require the total length in instructions of the vertex bundles mapped to the same node to be ≤ 32. This requires defining, for each bundle, the maximum bundle length $\lambda(b_v)$ as a function of the last send message of the vertex. This length can then be constrained to be ≤ 32.

To achieve this, we first define the set $LAST(B_v, B_e)$, which pairs each vertex bundle with its last edge bundle, corresponding to the last send message of the vertex. This enables to define the maximum bundle length $\lambda(b_v)$ as:

$$\forall e, b_e, b_v | LAST(b_v, b_e) \& B(e, b_e), \quad \Delta(e) + \delta(e) \leq \lambda(b_v) \tag{32}$$

We finally define $Q(B_v, N)$ as the required number of instructions on node n from vertex bundle b_v and limit it to 32 (the code-snippet length).

$$\forall b_v, n \in N, \quad Q(b_v, n) - 32 * M_{bn}(b_v, n) \geq \lambda(b_v) - 32 \tag{33}$$
$$\forall n, \quad \Sigma_{b_v \in B_v} Q(b_v, n) \leq 32 \tag{34}$$

Objective Formulation: For PLUG, the smallest service interval is achieved and enforced for any legal schedule, and we optimize solely for latency LAT.

$$min\, LAT\; s.t.\; [1,2,3,4,5,6,7,11,12,13,14,$$
$$15,16,17,18,25,26,27,28,29,30,31,32,33,34]$$

6. Implementation and Evaluation

In this section, we describe our implementation of the constraints in an off-the-shelf ILP solver and evaluate its performance compared to native specialized schedulers for the three architectures.

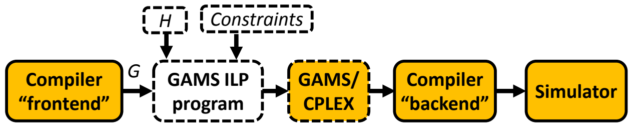
6.1 Implementation

We use the GAMS modeling language to specify our constraints as mixed integer linear programs, and we use the commercial CPLEX solver to obtain the schedules. Our implementation strategy for

	TRIPS	DySER	PLUG
Benchmarks	• Same as prior TRIPS scheduler papers [9]: SPEC microbenchmarks and EEMBC • Full SPEC benchmarks can't run to completion on simulator and don't stress scheduler (since blocks are small)	• DySER data-parallel workloads since they produce large blocks and complete code from compiler [25]. • Additional throughput microbenchmark (details below)[1] • SPEC2006 and PARSEC in [25] are not usable because they don't produce high quality code on compiler. [25] used a trace-simulation based approach (not compiler-based code-generation).	• PLUG benchmarks from [13]
Native scheduler	• Optimized SPS scheduler [9]	• Specialized greedy algorithm in toolchain & hand scheduled [26]	• Hand scheduled [13]
Metric	• Total execution cycles for program	• Total execution cycles for program	• Total execution cycles for lookups

1. DySER "throughput" microbenchmark: This performs the calculation $y = x - x^{2i}$ in the code-region. Paths diverge at the input node x, into one long path which computes the x^{2i} with a series of i multiplies, and along a short path which routes x to the subtraction. This pattern tends to cause latency mismatch because one of these converging paths naturally takes less resources.

Table 5. Tools and methodology for quantitative evaluation



"frontend": passes in the compiler that produce pre-scheduled code; "backend": passes that convert scheduled code into binary.

Figure 9. Implementation of our ILP scheduler. Dotted boxes indicate the new components added.

prioritizing multiple variables follows a standard approach: we define an allowable percentage optimization gap (of between 2% to 10%, depending on the architecture), and optimize for each variable in prioritized succession, finishing the solver when the percent gap is within the specified bounds. After finding the optimal value for each variable, we add a constraint which restricts that variable to be no worse in future iterations.

Figure 9 shows our implementation and how we integrated with the compiler/simulator toolchains [1, 13, 25]. For all three architectures, we use their intermediate output converted into our standard directed acyclic graph (DAG) for G and fed to our GAMS ILP program. We specified H for each architecture. To evaluate our approach, we compare the performance of the final binaries on the architectures varying only the scheduler. Table 5 summarizes the methodology and infrastructure used.

6.2 Results

Is this ILP-based approach implementable? Yes, it is possible to express the scheduling problem as an ILP problem and implement it for real architectures. Considering the ILP constraint formulation for the general framework, our GAMS implementation is around 50 lines of code.

Result-1: A declarative and general approach to expressing and implementing spatial-architecture schedulers is possible.

Is the execution time of standard ILP-solvers fast enough to be practical? Table 6 (page 10) summarizes the mathematical characteristics of the workloads and corresponding scheduling behavior. The three right-hand columns respectively show the number of software nodes to schedule, the amount of single ILP equations created, and the solver time.[2] There is a rough correlation between the workload "size" and scheduling time, but it is still highly variable.

The solver time of the specialized schedulers in comparison is typically on the order of seconds or less. Although some blocks may take minutes to solve, these times are still tractable, demonstrating the practicality of ILP as a scheduling technique.

Result-2: Our general ILP scheduler runs in tractable time.

Are the output solutions produced good? How do they compare against the output of specialized schedulers? Figure 10 (page 10) shows the performance of our ILP scheduler. It shows the cycle-count reduction for the executed programs as a normalized percentage of the program produced by the specialized compiler (higher is better, negative numbers mean execution time was increased). We discuss these results in terms of each architecture.

Compared to the TRIPS SPS specialized scheduler (a cumulated multi-year effort spanning several publications [9, 12, 40]), our ILP scheduler performs competitively as summarized below.[3]

Compared to SPS		
(a) Better on 22 of 43 benchmarks	up to 21%	GM +2.9%
(b) Worse on 18 of 43 benchmarks	within 4.9% **(typically 2%)**	GM -1.9%
(c) 5.4%, 6.04%, and 13.2% worse on ONLY 3 benchmarks		
Compared to GRST		
Consistently better, up to 59% better; GM +30%		

Groups (a) and (b) show the ILP scheduler is capturing the architecture/scheduler interactions well. The small slowdowns/speedups compared to SPS are due to dynamic events which disrupt the scheduler's view of event timing, making its node/link assignments sub-optimal, typically by only 2%. After detailed analysis, we discovered the reason for the performance gap of group (c) is the lack of information that could be easily integrated in our model. First, the SPS scheduler took advantage of information regarding the specific cache banks of loads and stores, which is not available in the modular scheduling interface exposed by the TRIPS compiler. This knowledge would improve the ILP scheduler's performance and would only require changes to the compatibility matrix $C(V, N)$. Second, knowledge of limited resources was available to SPS, allowing it to defer decisions and interact with code-generation to map movement-related instructions. What these results show overall is that our first-principles based approach is capturing all the architecture behavior in a general fashion and arguably aesthetically cleaner fashion than SPS's indirect heuristics. Our ILP scheduler consistently exceeds by appreciable amounts a previous generation TRIPS scheduler, GRST, that did not model contention [40], as shown by the hatched bars in the figure.

On DySER, the ILP scheduler outperforms the specialized scheduler on all benchmarks, as shown in Figure 10, for a 64-unit DySER. Across the benchmarks, the ILP scheduler reduces *individual* block latencies by 38% on average. When the latency of DySER execution is the bottleneck, especially when there are dependencies between instances of the computation (like the needle benchmark), this leads to significant speedup of up to 15%. We also implemented an extra DySER benchmark, which elucidates the importance of latency mismatch and is described in Table 5.

[2] For TRIPS, the per-benchmark number of DAGs can range from 50 to 5000, and the metrics provided are average per DAG. For DySER, #DAGs is 1 to 4 per benchmark, and PLUG is always 1.

[3] We did not run on SPEC benchmarks for three reasons: prior TRIPS scheduler work uses this set; TRIPS simulator does not have sim-point etc. to meaningfully simulate TRIPS benchmarks; TRIPS compiler does not produce good enough code on SPEC (10-15 inst blocks only) to make scheduler a factor [12, 23]. Using TRIPS hardware was impractical for us.

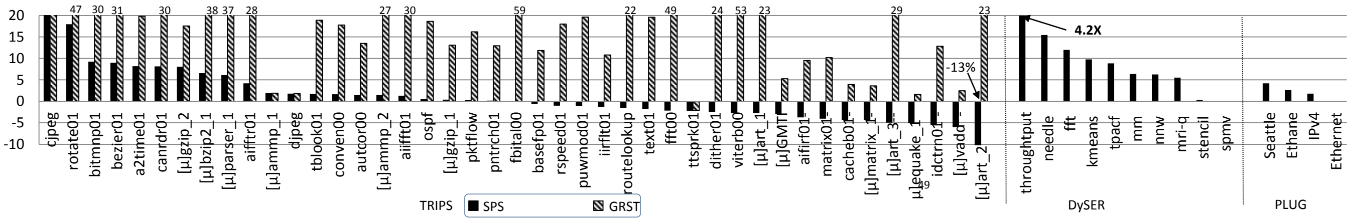


Figure 10. Normalized percentage improvement in execution cycles of ILP scheduler compared to specialized scheduler.

(a) TRIPS

	Applications	#nodes	#eqns	Solver time (sec)	Applications	#nodes	#eqns	Solver time (sec)
Microbenchmarks [9]	ammp_1	17	3744	76	gzip_1	23	4480	1
	ammp_2	8	1593	<1	gzip_2	22	4506	111
	art_1	22	4547	74	matrix_1	19	3797	18
	art_2	27	5506	76	parser_1	33	7248	174
	art_3	33	7042	20	transp_GMTI	20	4159	115
	bzip_1	13	2655	10	vadd	30	7313	315
	equake_1	24	4455	3				
EEMBC [9]	a2time01	11	1914	5	ttsprk01	11	1993	8
	aifftr01	12	2173	25	cjpeg	12	2280	3
	aifirf01	11	1933	7	djpeg	12	2277	1
	aiifft01	11	2025	1	ospf	10	1778	3
	basefp01	10	1863	6	pktflow	10	1774	3
	bitmnp01	9	1535	3	routelookup	10	1747	3
	cacheb01	27	2745	76	bezier01	10	1788	2
	canrdr01	10	1871	8	dither01	10	3579	4
	idctrn01	11	1947	3	rotate01	10	1910	5
	iirflt01	11	2080	2	text01	10	1781	3
	matrix01	11	1426	2	autcor00	10	1746	2
	pntrch01	10	1819	5	conven00	10	1758	4
	puwmod01	10	1779	3	fbital00	9	1699	3
	rspeed01	10	1816	7	fft00	10	1808	5
	tblook01	10	1818	4	viterb00	10	1870	5
					TRIPS Avg.	14	2832	31

(b) DySER

	Applications	#nodes	#eqns	Solver time (sec)
Data-Parallel Benchmarks [26]	fft	20	120250	365
	mm	32	159231	77
	mri-q	19	98615	66
	spmv	32	155068	72
	stencil	30	153428	74
	tpacf	40	211584	368
	nnw	25	169197	102
	kmeans	40	232399	218
	Needle	34	181686	183
	Throughput	9	45138	62
	DySER Avg.	**28**	**152660**	**159**

(c) PLUG

Applications	#nodes	#eqns	Solver time (sec)	
Ethernet	18	25603	57	PLUG benchmarks [13]
Ethane	11	13905	14	
IPv4	12	38741	384	
Seattle	16	14531	26	
PLUG Avg.	14	23195	120	

Table 6. Benchmark characteristics and ILP scheduler behavior.

The specialized scheduler tries to minimize the extra path length at each step, exacerbating the latency mismatch of the short and long paths in the program. The ILP scheduler, on the other hand, pads the length of the shorter path to reduce latency mismatch, increasing the potential throughput and achieving a 4.2× improvement over the specialized scheduler. Finally, we also compared to manually scheduled code on an *16-unit* DySER (since hand-scheduling for 64unit DySER is exceedingly tedious). The ILP scheduler always matched or out-performed it by a small ($< 2\%$) percentage.

The ILP scheduler matches or out-performs the PLUG hand-mapped schedules. It is able to both find schedules that force $SVC = 1$ and provide latency improvements of a few percent. Of particular note is solver time, because PLUG's DFGs are more complex. In fact, each DFG represents an *entire* application. The most complex benchmark, IPV4, contains 74 edges (24 more than any others) split between 30 mutually exclusive or multicast groups. Despite these difficulties, it completes in tractable time.

Result-3: Our ILP scheduler outperforms or matches the performance of specialized schedulers.

Modeling Concerns	Approach
Can constraints be modeled precisely?	• Most constraints are directly expressible • Nonlinear constraints do not arise • Logical operations can be modeled using known techniques [24, 29, 30]
Can other constraint theories be used?	• Constraints can be specified in SMT theory • Our Z3[14] implementation shows that SMT solving takes significantly longer than ILP
Implementation Concerns	
Can dynamic effects (cache misses, network contention) be taken into account?	• We optimize for best-case scenario (same approach as existing specialized schedulers) • Stochastic techniques can optimize concurrently for multiple scenarios (future work)
How does implementing objective functions compare to implementing scheduling heuristics?	• Declarative formulation is more intuitive & simplifies implementation. • Example: the TRIPS SPS scheduler uses several indirect heuristics to model utilization, whereas utilization directly captures it
Productivity Concerns	
Is a deep understanding of architecture required?	• Full understanding of the target architecture is required regardless of the approach • We streamline process by defining sets of responsibilities & direct modeling of behavior

Table 7. Feasibility concerns

7. Discussion and Conclusions

Scheduling is a fundamental problem for spatial architectures, which are increasingly used to address energy efficiency. Compared to the architecture-specific schedulers, which are the current state-of-the-art, this paper provides a general formulation of spatial scheduling as a constraint-solving problem. We applied this formulation to three diverse architectures, ran them on a standard ILP solver, and demonstrated such a general scheduler outperforms or matches the respective specialized schedulers. Some potential limitations and concerns about our work are outlined in Table 7, but do not detract from its central contributions.

We conclude with a discussion of some broader extensions and implications of our work. Specifically, we discuss the possibility of improving the scheduling time through algorithmic specialization, and how our scheduler delivers on its promises of compiler-developer productivity/extensibility, cross-architecture applicability, and insights on optimality.

Specializing ILP Solvers: While the benefits of using Integer Linear Programming come at the cost of additional scheduler execution time, we suspect that there may be further opportunities for improvement. One strategy is to specialize the solver's algorithms to the problem domain. The Network Simplex algorithm for the minimum cost flow problem is a widely known example. To create a specialized algorithm for our problem, a detailed investigation of the constraints from a solver design perspective would be required, as well as the modification of existing ILP solvers. This is one critical direction for future work.

Responsibility	RAW	WaveScalar	Neural Processing Unit
Placement	Homogeneous Cores (Tiles)	Homogeneous Processing Elements	8 Processing Elements, 1 Shared Bus
Routing	2-D grid, unconstrained routing	Hierarchical Network. First two levels are fully connected, last level grid uses dynamic routing	Responsibility Not Applicable -- Broadcast bus used for communication.
Timing	In-order execution inside tile, dataflow between tiles (Secondary list scheduler orders inter-stream events)	Data-flow execution, and dynamic network arbitration; network latency varies by hierarchy level	Fully Static execution. "No-ops" between bus events maintain synchronization.
Utilization	Many instructions per tile. Shared network links	64-Instuctions/PE; Shared Network Links	Shared Processing Elements
Objective	Latency & Throughput	Contention & Latency	Latency

Table 8. Applicability to other Spatial Architectures

Formulation Extensibility: In our experience, our model formulation was easily adaptable and extensible for modeling various problem variations or optimizations. For example, we improved upon our TRIPS scheduler's performance by identifying blocks with carried-loop cache dependencies (commonly the most frequently executed), and extended our formulation to only optimize for relevant paths.

Application to Example Architectures: Table 8 shows how our framework could be applied to three other systems. For both WaveScalar and RAW, we can attain optimal solutions by refraining from making early decisions, essentially avoiding the drawbacks of multi-stage solutions. For WaveScalar, our scheduler would consider all levels of the network hierarchy at once, using different latencies for links in different networks. For RAW, our scheduler would consider both the partitioning of instructions into streams, and the spatial placement of these instructions simultaneously.

As a more recent example, NPU [20] is a statically-scheduled architecture like PLUG, but uses a broadcast network instead of a point-to-point, tiled network. Instead of using the routing equations for communication, the NPU bus is more aptly modeled as a computation type. Timing would be modeled similarly to PLUG, where "no-ops" prevent bus contention, allowing a fully static schedule.

Insights on Optimality: Since our approach provides insights on optimality, it has potentially broader uses as well. For instance, in the context of a dynamic compilation framework, even though the compilation time of seconds is impractical, the ILP scheduler still has significant practical value – it enables developers to easily formulate and evaluate objectives that can guide the implementation of specialized heuristic schedulers.

Revisiting NPU scheduling, we can observe another potential use of ILP models, specifically in designing the hardware itself. For the NPU, the fifo depth of each processing element is expensive in terms of hardware, so we could easily extend the model to calculate the fifo depth as a function of the schedule. One strategy would be to first optimize for performance, then fix the performance and optimize for lowest maximum fifo depth. Doing this across a set of benchmarks would give the best lower-bound fifo depth which does not sacrifice performance.

Finally, while our approach is general, in that we have demonstrated implementations across three disparate architectures and shown extensions to others, a somewhat open question remains on "universality": what spatial architecture organization could render our framework ineffective? This is subject to debate and is future work. Overall, our general scheduler can form an important component for future spatial architectures.

Acknowledgments

We thank the anonymous reviewers for comments. Thanks to Mark Hill and Daniel Luchaup for their valuable insights and comments on the paper. Support for this research was provided by NSF under the following grants: CCF-0845751, CNS-0917213, and CNS-0917238.

References

[1] Trips toolchain, http://www.cs.utexas.edu/ trips/dist/.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*.

[3] S. Amarasinghe, D. R. Karger, W. Lee, and V. S. Mirrokni. A theoretical and practical approach to instruction scheduling on spatial architectures. Technical report, MIT, 2002.

[4] S. Amellal and B. Kaminska. Functional synthesis of digital systems with tass. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(5):537 –552, 1994.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *PPOPP 1991*.

[6] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *ISCA 2010*.

[7] S. S. Battacharyya, E. A. Lee, and P. K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[8] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.

[9] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, 2004.

[10] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO 2004*.

[11] J. Cong, K. Gururaj, G. Han, and W. Jiang. Synthesis algorithm for application-specific homogeneous processor networks. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(9), Sept. 2009.

[12] K. Coons, X. Chen, S. Kushwaha, K. S. McKinley, and D. Burger. A Spatial Path Scheduling Algorithm for EDGE Architectures. In *ASPLOS 2006*.

[13] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam. Plug: Flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *SIGCOMM 2009*.

[14] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[15] A. Deb, J. M. Codina, and A. Gonzales. Softhv: A hw/sw co-designed processor with horizontal and vertical fusion. In *International Conference on Computing Frontiers 2011*.

[16] A. E. Eichenberger and E. S. Davidson. Efficient formulation for optimal modulo schedulers. In *PLDI 1997*.

[17] J. R. Ellis. *Bulldog: a compiler for vliw architectures*. PhD thesis, 1985.

[18] D. W. Engels, J. Feldman, D. R. Karger, and M. Ruhl. Parallel processor scheduling with delay constraints. In *SODA 2001*.

[19] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA 2011*.

[20] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO 2012*.

[21] K. Fan, H. h. Park, M. Kudlur, and S. o. Mahlke. Modulo scheduling for highly customized datapaths to increase hardware reusability. In *CGO 2008*.

[22] P. Feautrier. Some efficient solutions to the affine scheduling problem. *International Journal of Parallel Programming*, 21:313–347, 1992.

[23] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An evaluation of the trips computer system. In *ASPLOS 2009*.

[24] G. J. Gordon, S. A. Hong, and M. Dudík. First-order mixed integer linear programming. In *UAI 2009*.

[25] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy efficient computing. *IEEE Micro*, 33(5), 2012.

[26] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA 2011*.

[27] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO 2011*.

[28] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

[29] J. N. Hooker. Logic, optimization and constraint programming. *INFORMS Journal on Computing*, 14:295–321, 2002.

[30] J. N. Hooker and M. A. Osorio. Mixed logical-linear programming. *Discrete Appl. Math.*, 96-97(1), Oct. 1999.

[31] Z. Huang, S. Malik, N. Moreano, and G. Araujo. The design of dynamically reconfigurable datapath coprocessors. *ACM Trans. Embed. Comput. Syst.*, 3(2):361–384, May 2004.

[32] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *PLDI 2002*.

[33] K. Kailas and A. Agrawala. Cars: A new code generation framework for clustered ilp processors. In *HPCA 2001*.

[34] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI 2008*.

[35] A. Kumar, L. De Carli, S. J. Kim, M. de Kruijf, K. Sankaralingam, C. Estan, and S. Jha. Design and implementation of the plug architecture for programmable and efficient network lookups. In *PACT 2010*.

[36] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *ASPLOS 1998*.

[37] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Instruction scheduling for a tiled dataflow architecture. In *ASPLOS 2006*.

[38] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Modeling instruction placement on a spatial architecture. In *SPAA 2006*.

[39] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, M. Budiu, and S. C. Goldstein. Tartan: Evaluating spatial computation for whole program execution. In *ASPLOS 2006*.

[40] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler. Static placement, dynamic issue (spdi) scheduling for edge architectures. In *PACT 2004*.

[41] E. Özer, S. Banerjia, and T. M. Conte. Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. In *MICRO 31*.

[42] J. Palsberg and M. Naik. Ilp-based resource-aware compilation, 2004.

[43] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *PACT 2008*.

[44] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing 1991*.

[45] N. Satish, K. Ravindran, and K. Keutzer. A decomposition-based constraint optimization approach for statically scheduling task graphs with communication delays to multiprocessors. In *DATE 2007*.

[46] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *MICRO 2003*.

[47] M. Thuresson, M. Sjalander, M. Bjork, L. Svensson, P. Larsson-Edefors, and P. Stenstrom. Flexcore: Utilizing exposed datapath control for efficient computing. In *IC-SAMOS 2007*.

[48] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS 2010*.

[49] H. M. Wagner. An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly*, 6(2):131–140, 1959.

[50] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: RAW Machines. *Computer*, 30(9):86–93, 1997.

[51] M. Watkins, M. Cianchetti, and D. Albonesi. Shared reconfigurable architectures for cmps. In *FPGA 2008*.

[52] L. A. Wolsey and G. L. Nemhauser. *Integer and Combinatorial Optimization*.