Constraint Centric Scheduling Guide

Michael Sartin-Tarm† Tony Nowatzki† Lorenzo De Carli† Karthikeyan Sankaralingam† Cristian Estan*
†University of Wisconsin-Madison *Broadcom
(tjn@cs.wisc.edu, msartintarm@wisc.edu, lorenzo@cs.wisc.edu, karu@cs.wisc.edu, cristian@estan.org)

Abstract—The advent of architectures with software-exposed resources (Spatial Architectures) has created a demand for universally applicable scheduling techniques. This paper describes our generalized spatial scheduling framework, formulated with Integer Linear Programming, and specifically accomplishes two goals. First, using the "Simple" architecture, it illustrates how to use our open-source tool to create a customized scheduler and covers problem formulation with ILP and GAMS. Second, it summarizes results on the application to three real architectures (TRIPS,DySER,PLUG), demonstrating the technique's practicality and competitiveness with existing schedulers.

1 Introduction

Spatial architectures, which provide energy-efficient computation by exposing elements of their execution to the software, are likely to play an important role in future architectures. Compilers for spatial architectures typically use heuristic-based algorithms, like those for RAW, TRIPS, Wavescalar, and CCA [2], [5]–[7], [9]. However, such approaches can be complex and difficult to maintain or implement, lack insight into the optimality of solutions, and lack portability across architectures.

To address these deficiencies, Nowatzki et al. have designed a spatial scheduling framework, formulating the problem using integer linear programming (ILP) [8]. Their solution is advantageous in that the formulation is general enough to be applied to a variety of architectures, and their constraint-based approach finds solutions with guaranteed optimality bounds.

This paper describes, from a developer's perspective, the spatial scheduling tool we have created, which leverages the framework of [8]. Written in the GAMS language [1], this tool's source is at http://www.cs.wisc.edu/vertical/ilp-scheduler, and this paper serves as its usability guide. Our implementation is attractive in that it is extremely concise – the core scheduler implementation (the ILP constraints) for three real spatial architectures is less than 50 lines of GAMS code. Also, the simplicity of GAMS enables the formulation to be easily followed and augmented with only modest ILP background.

To demonstrate the use of our framework, this paper defines a scheduler for an abstract architecture: the "Simple" architecture. This approach allows us to focus on scheduling concepts without the distraction of architecture-specific nuances.

The structure of this paper is as follows: §2 gives an overview of our approach, and describes background information on our example architecture and ILP. §3 details the GAMS-based ILP Scheduler Interface used by the compilers. §4 illustrates the application of our general framework to the "Simple" architecture. §5 highlights scheduler performance from three architectures and considers limitations, and §6 concludes.

2 Overview and Background

This section presents three key pieces of background relevant to understanding our general scheduling framework. We first describe the role of the scheduler in a compilation system and highlight our approach, then describe the "Simple" spatial architecture that we use as a running example. Finally, we give a high-level overview of Integer Linear Programs and the GAMS modeling language.

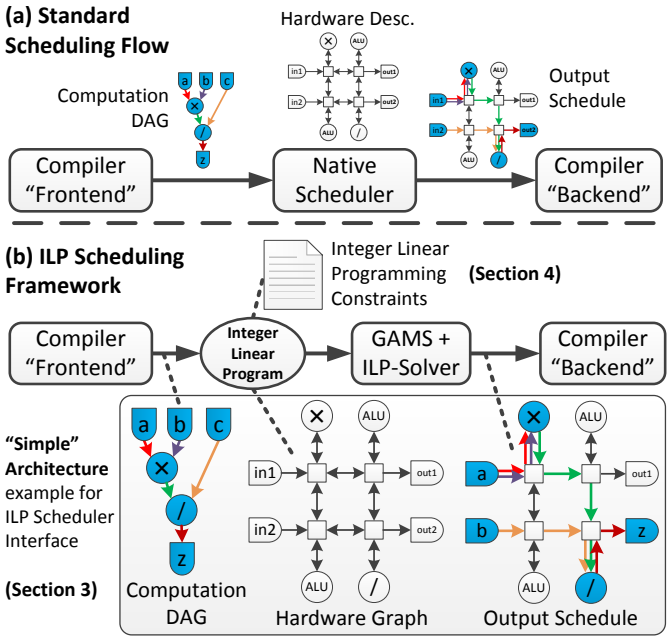

Figure 1. Overview of Standard and ILP Scheduler Flows

2.1 The Scheduler's Role

In a compilation system, the role of a spatial-architecture scheduler is to determine the hardware configuration of abstract units of computation. The scheduler's goal is to find a legal and optimal placement and ordering of events, where optimal is usually defined as minimizing the overall execution time. Figure 1(a) shows how a typical scheduler is fed a directed acyclic graph (DAG), representing the computation, from the "frontend" of the compiler. The scheduler then uses some knowledge of the hardware's topology and capabilities to create the output schedule, again in some native format. Finally, the compiler "backend" uses this information to generate the architectural configuration.

Our framework requires only modest changes, as shown by Figure 1(b). Here, the scheduler is replaced by an ILP solver with a GAMS frontend (described in section 2.3). The interface we provide, called the ILP Scheduling Interface (ISI), is a GAMS language based format for representing computation DAGS, hardware graphs, and output schedules. The flow is as follows: first, the compiler "frontend" produces an ISI computation DAG, based on the native format. Second, GAMS reads the computation DAG, hardware graph, and the ILP constraints, which together form an Integer Linear Program. Finally, the GAMS runtime interacts with an ILP solver to determine the optimal solution, and the ISI output schedule

is converted to the native format by the compiler "backend".

From a developer's perspective, the work required in using our framework is 3-fold: i) in creating compiler modules which can emit ISI computation DAGs and read ISI output schedules (§3), ii) one-time effort in creating the GAMS hardware graph describing the hardware topology and resource capabilities (§3), iii) one-time effort in selecting the constraints from our framework which enforce the appropriate architecture primitives (§4). The generalized constraint formulation, which aids developers in quickly writing ILP based schedulers, is the intellectual core of this work.

2.2 The "Simple" Architecture

To demonstrate our framework, we introduce the "Simple" architecture. Figure 1 shows an example software graph, hardware graph, and example schedule for our architecture. First, the software graph, which can be arbitrary computation DAGs, is composed of five node types: Inputs, Outputs, ALU, DIV, and multiply. The hardware graph has correspondingly typed nodes, and adds router nodes to create a 2×2 grid interconnect. The following five features describe the architecture.

1) **Heterogeneous Nodes:** Typed computation nodes.
2) **Custom Routing:** Messages between nodes may traverse an arbitrary sequence of routers.
3) **Dataflow Execution:** Execution units fire whenever their input operands are ready.
4) **Dynamic resource arbitration:** Resources arbitration decided on the fly, by first come first serve.
5) **Objective:** Lowest run-time latency.

The execution model of the architecture is straightforward. Inputs arrive dynamically at a cycle number which is unknowable at scheduling time. Links take one cycle to traverse and can carry multiple messages per cycle. Execution units can only perform a single operation per cycle, but are fully pipelined.

2.3 GAMS and ILP Primer

Integer Linear Programs (ILP) are algebraic models of systems used for optimization [10]. They are composed of: 1) decision variables describing the possible outcomes, 2) linear constraints describing the solution space, 3) a linear objective which describes the solution desirability. A short tutorial on ILP modeling and solving techniques is here: http://wpweb2.tepper.cmu.edu/fmargot/introILP.html. We utilize ILP for this work because it offers significant expressability with reasonable solution times.

GAMS is a language that can express optimization problems of many classes, including ILP. Using GAMS is a matter of converting the mathematical formulation into equivalent GAMS variable and constraint declarations. The language also provides mechanisms for formatted reading and writing, and interfaces with underlying solvers, which are completely encapsulated from the modeler. An extensive GAMS guide is here: http://www.gams.com/dd/docs/bigdocs/GAMSUsersGuide.pdf, while an example for the syntax of GAMS formulations is here: http://www.gams.com/docs/example.htm.

3 ILP Scheduler Interface (ISI)

We begin describing our scheduling framework with the details of the ISI which is exposed to the compiler in terms of the hardware graph and computation DAG.

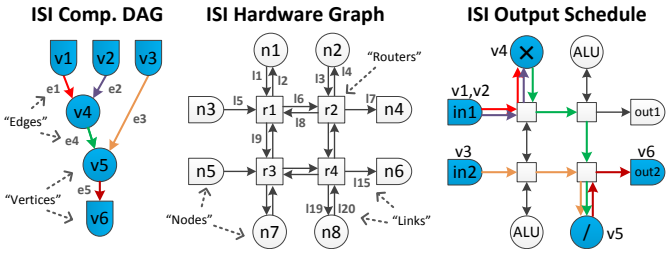

Figure 2. Notation for the ILP Scheduler Interface

Listing 1. GAMS based ISI Hardware Graph

```
1  Set n "Nodes"   /n1, n2, n3, n4, n5, n6, n7, n8/;
2  Set r "Routers" /r1, r2, r3, r4/;
3  Set l "Links"   /l1, l2, l3, [etc]/;
4  * Node Connections
5  Set Hnl(n,l)          /n1.l1, n2.l3, n3.l5, .../;
6  Set Hln(l,n)          /l2.n1, l4.n2, l7.n4, .../;
7  * Router Connections
8  Set Hrl(r,l)          /r1.l2, r1.l9, r2.l4, .../;
9  Set Hlr(r,l)          /l1.r1, l3.r2, l6.r2, .../;
10 Set K "Operation Kind" /IN,OUT,ADD,SUB,MULT/;
11 Set kindN(K,n)     /IN.n3, ADD.n2, SUB.n2, .../;
```

Listing 2. GAMS based ISI Computation Dag

```
1 Set v "Vertices" /v1, v2, v3, v4, v5, v6/;
2 Set e "Edges" /e1, e2, e3, e4, e5/;
3 Set Gve(v,e)  /v1.e1, v2.e2, v3.e3, v4.e4, .../;
4 Set Gev(e,v)  /e1.v4, e2.v4, e3.v5, e4.v5, .../;
5 parameter delta(e)       /e5 3, e4 1, e3 1, .../;
6 Set kindV(K,v)     /DIV.v5, MULT.v4, IN.v3, .../;
```

3.1 Specifying the Hardware Graph

The ISI hardware graph describes the spatial topology, and would be written once for a particular architecture. An example in Listing 1 corresponds to the depiction in Figure 2. This graph is composed of three sets: one representing mappable *nodes*, and *routers* and *links* which form the network. Lines 1-3 define these sets.

Links are explicit in our model because they are mapped to directly. As such, we need to specify the connections between nodes/routers and links, accomplished through four parameters: H_{nl},H_{ln},H_{rl},H_{lr}. Here H_{nl} describes the outgoing links of a node, while H_{lr} describes the incoming links of a router. Lines 4-9 define these parameters; ellipsis indicate truncation.

Finally, we specify the heterogeneous capabilities of the hardware through two sets, shown in lines 10-11. The set K lists the possible node "kinds", which represent fundamental operations like "add" or "multiply". The parameter $kindN$ describe the operation capabilities of each node.

3.2 Specifying the Computation DAG

The computation DAG, produced by the compiler, is modeled similar to the hardware graph as shown in Listing 2. Sets for *vertices*, representing individual operations, and *edges*, representing data communication, are defined in lines 1-2. The DAG is built through the sets Gve, for vertex to edge connections, and Gev edge to vertex connections, as shown in lines 3-4.

The latencies of the computations described by vertices are captured by $delta(e)$, which defines for each edge the delay between when the source vertex is activated and the message represented by the edge is sent(line 5). Finally, vertex types in the same way as hardware node types, with $kindV$(line 6).

4 Applying the ILP Formulation

We have created a general scheduling framework in GAMS which formulates common spatial scheduling constraints. The

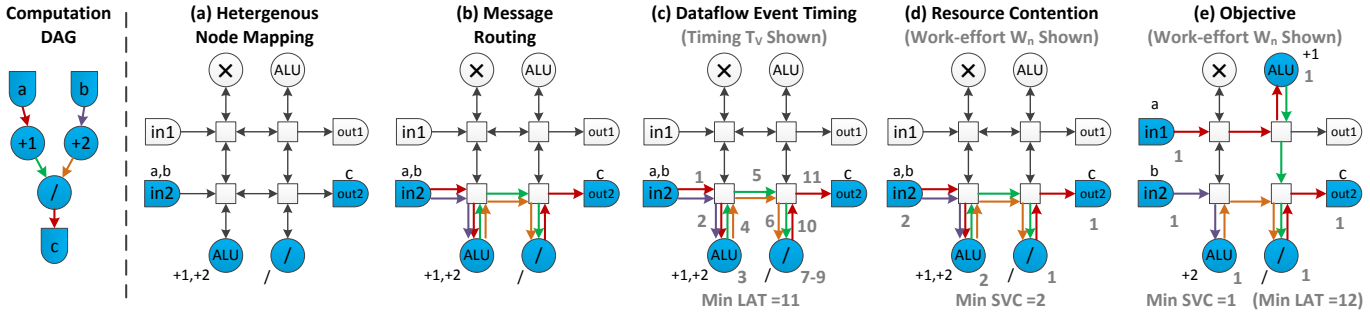

Figure 3. Applying scheduling constraints to the "Simple" Architecture

Listing 3. Scheduling Constraints for "Simple" Architecture

```
1  binary variable    Mvn(v,n);
2  c1_map_all_v(K,v)$kindV(K,v)..
                    sum(n$(kindN(K,n)), Mvn(v,n))=e=1;
3  c2_map_valid(K,v)$kindV(K,v)..
                  sum(n$(not kindN(K,n)), Mvn(v,n))=e=0;

4  binary variable    Mel(e,l);
5  c3_map_edge_src(v,e,n)$Gve(v,e)..
              Mvn(v,n) =e= sum(l$Hnl(n,l), Mel(e,l));
6  c4_map_edge_dst(e,v,n)$Gev(e,v)..
              Mvn(v,n) =e= sum(l$Hln(l,n), Mel(e,l));
7  c5_router_fwd(e,r)..     sum(l$Hlr(l,r), Mel(e,l))
                       =e= sum(l$Hrl(r,l), Mel(e,l));

8  positive variable Tv(v), LAT;
9  c6_timing(v1,e,v2)$(Gve(v1,e) and Gev(e,v2))..
          Tv(v2) =g= Tv(v1)+delta(e)+sum(l,Ml(e,l));
10 c7_max_cycle(v)..  LAT =g= Tv(v);

11 positive variable Wn(n), SVC;
12 c8_work_effort(n).. sum(v,Mvn(v,n)) =e= Wn(n);
13 c9_calc_svc(n)..                SVC =g= Wn(n);

14 Model   problem1  / c1, c2, c3 ... c9 /;
15 solve   problem1  using mip minimizing SVC;
16 c10..   SVC =e= SVC.l;
17 Model   problem2  / c1, c2, c3 ... c9, c10 /;
18 solve   problem2  using mip minimizing LAT;
```

specific ILP scheduler formulation for each architecture builds upon the general framework by selecting constraints to enforce the required behavior and features, and may extend the framework with new constraints where necessary. In this section, we first describe the constraints of Listing 3, which apply to the "Simple" architecture, and then describe aspects of our general framework which are beyond the scope of this architecture.

4.1 Heterogeneous Node Mapping

The most basic task of any spatial scheduler is the mapping between computations vertices and hardware resource nodes. We describe this mapping with M_{vn}, a set of binary variables, one for each combination of elements in n and v. Each element indicates if vertex v should be mapped to node n.

Two constraints enable node mapping. The first ensures that each vertex is mapped to one compatible node. We do this by enforcing that, for all vertices, the sum of M_{vn} over all nodes with corresponding type is equal to one. The second ensures all incompatible vertex/node pairs are illegal, and is performed similarly. The mathematical notation for both follow.

$$\forall v, K|kind(K,v) \quad \Sigma_{n|kindN(K,n)} M_{vn}(v,n) = 1 \tag{1}$$

$$\forall v, K|kind(K,v) \quad \Sigma_{n|not\ kindN(K,n)} M_{vn}(v,n) = 0 \tag{2}$$

Lines 1-3 of Listing 3 show the corresponding GAMS syntax. To explain c1, the syntax before the two dots ".." indicates for which elements in the specified sets our formula on the right of the dots applies. This constraint applies for all elements of K and v. Here, the dollar syntax intuitively means *such that* the following expression holds. Therefore "$kindV(k,v)" in c1 means that we are only considering the vertices of the current "Kind".

Consider the GAMS equation notation in line 2. The first argument of "sum" is the summation index, and the second is the summation expression. Note that in GAMS, the index and the index's bounds are synonymous. Next, "=e=" indicates expression equality, while "=l=" means less-than, etc. Constraint c2 is expressed similarly. Figure 3(a) shows a legal mapping of vertices to nodes after applying the above constraints.

4.2 Message Routing

In the "Simple" architecture, since the routing paths are flexible, we must model arbitrary communication mapping. This is performed through the binary variable M_{el}, which describes the mapping between edges in the computation graph, and links in the hardware graph, akin to the vertex/node mapping. Valid paths must start at the correct node, end at the correct node, and be fully connected in between. The GAMS code in lines 4-7 enforce these properties.

To explain, c3 enforces that edges must start at their mapped node's location. Intuitively, it does this by enforcing that if the corresponding vertex is mapped to a node, the edge should be mapped to one of the node's outgoing links. Mathematically, we equate the vertex/node mapping $M_{vn}(v,n)$ with the sum of edge/link mapping on outgoing links $\Sigma_{l|H_{nl}(n,l)} M_{el}(e,l)$. Constraint c4 works similarly for incoming edges/links. Constraint c5 enforces that all paths are continuous, by ensuring incoming router edges are also outgoing router edges. The constraint achieves this by equating the total edge/link mapping for incoming and outgoing router links for each edge. Figure 3(b) shows a legal mapping of edges to links after applying the above constraints.

4.3 Dataflow Event Timing

The timing model for our "Simple" architecture is based on dataflow execution ordering. This means that when all messages for a vertex have arrived, the computation can begin. First, we create a variable indicating the firing time of any vertex, T_v. Constraint c6 computes T_v for each vertex by selecting the maximum latency of each incoming edge. The latency of an edge is computed as the sum of: 1) the T_v of the source vertex; 2) the delay before the edge is ready ($delta(e)$); 3) the number of links traversed between source and destination nodes. We also compute the total latency LAT, with c7. Figure 3(c) shows that the total latency for our example schedule is 11 cycles.

4.4 Dynamic Resource Contention

Even though minimizing the overall latency is the primary goal of the scheduler, the dataflow timing T_v does not consider the delay due to resource constraints. To model contention, we need two new concepts, *work-effort* and the service interval, SVC. Work-effort is simply the amount of work carried out by each hardware resource. We use this to calculate the service interval.

The intuitive meaning of SVC is the minimum time between pipelined instances of the same computation. For instance, if two computational vertices are mapped to the same node, then it would only be possible to achieve a pipelined throughput of 1 computations every 2 cycles. Minimizing SVC will equalize work-effort across nodes. For the "Simple" architecture, we can compute the work-effort W_n using c8, which sums up the number of vertices mapped to a single node. Constraint c9 computes SVC as the maximum of all work-effort values. Figure 3(d) shows that the SVC for our example schedule is 2 cycles.

	General Constraints	Extra Constraints	Dynamic Equations	Average Solve (sec)
Trips	11	4	2.8 K	31
DySER	11	2	150 K	159
PLUG	9	5	23 K	120

Table 1. Number of Equations and Solve Times

4.5 Objective

The objective of a scheduler defines its ultimate goal. So far, we have seen that we need two distinct goals: minimizing dataflow latency (LAT) and service interval (SVC). Our framework, leveraging GAMS, allows us to easily specify and optimize for multiple objectives, as long as some priority can be assigned to each value.

Since the "Simple" architecture benefits more from resource distribution than just dataflow latency, we solve the problem two times, first for SVC, then for LAT, fixing the value of SVC to its best value after the first solve. The GAMS code of lines 14-18 shows this procedure, and Figure 3(e) shows the optimal schedule. Note that one cycle latency sacrifice enables an optimal service interval.

4.6 Other General Constraints

Though we have shown how the "Simple" architecture uses our general constraints to construct a scheduler, it does not elicit every possible architectural feature modeled in our general framework. We highlight some of these below.

- **Generalizations of Work-Effort:** The way work-effort is calculated may depend on the architecture. We provide constraints to model link work-effort and operation-specific work-effort.
- **Routing Restrictions:** Some architectures only allow certain routing patterns, like dimension ordered routing. We provide a general mechanism for such restrictions.
- **Multi-Destination Messages:** These messages are unique in that they may share routing resources along communication paths. We provide variables and equations for modeling resource usage (work-effort) for this capability.

Building an architecture specific scheduler is a matter of choosing the appropriate constraints given knowledge of the architecture. Though there are a number of constraints, each enforces a specific architectural primitive, and they are simple to compose.

5 Discussion

We evaluate our approach in terms of three questions: i) is this feasible for disparate spatial architectures? ii) does the solver run in practical time? iii) how does it compare to existing schedulers? We conclude by discussing the limitations of our approach.

5.1 Results

Feasibility: To test our framework on non-trivial architectures, we built three schedulers on top of our general framework for TRIPS [2], DySER [4] and PLUG [3], and integrated this scheduler into their compilers. The number of general framework versus architecture-specific constraints is shown in the first two columns of Table 2. Only a few architecture-specific constraints were required in each case. Moreover, our ILP based framework was able to model the primitives of three real architectures, producing correct schedules for all computation DAGs considered.

Practicality: We show the number of dynamic equations generated by GAMS, based on the constraints and ISI graphs, and also the average solution time in the third and fourth columns of table 2. Naturally, architectures which require more dynamic equations are prone to longer solution times. Even with many dynamic equations, the overall solution times are still practical, in the range of seconds to a few minutes.

	Micro Benchmarks	Standard Benchmarks
Trips	-0.8% (uBench)	1.5% (EEBMC)
DySER	420.0% (throughput)	7.1% (Parboil)
PLUG	-	2.1% (plug bench)

Table 2. Average Percentage Speedup

Performance: Table 2 shows the geometric mean speedup of our ILP scheduler versus the best known native scheduler for each architecture. TRIPS achieves only marginal speedups (or slight slowdowns) because the native scheduler implementation uses additional knowledge about cache banks that we did not have access to. The DySER ILP based scheduler always outperforms the native version, achieving an average of 7% improvement, and up to 4× on a throughput microbenchmark. The PLUG benchmarks are also an average of 2% better in terms of overall performance. Overall, we find that our scheduling approach is competitive with or outperforms native schedulers.

5.2 Limitations

In this section, we discuss the limitations of our approach, and offer possible solutions.

Architectural Generality: Though our general framework supports the primitives of many spatial architectures, it is possible that an architecture might contain some unique feature. In this case, it will be necessary to define new constraints or variables, potentially adding to or replacing some of the originals. Since our formulation was designed with the principle of generality, we have found such extensions to be simple and straightforward.

Handling Dynamic Latencies: Our model assumes common case latency values for events with variability, rather than considering the probability distribution of the latencies of such events. We argue this limitation is fundamental to any static scheduler, and other native schedulers like TRIPS, RAW, WaveScalar, DySER, and PLUG take the same approach. An extension to our model could employ stochastic programming, where several latency scenarios would be considered for inputs known to have variation.

6 Conclusion

This paper presents an overview of our constraint-centric spatial architecture scheduler and serves as a guide to our framework. It simultaneously introduces essential ILP modeling concepts and the GAMS language. Our approach is practically attractive in that the implementation and maintenance is relatively simple, due to the decreased complexity in specifying ILP constraints in contrast with heuristic-based algorithmic tuning. Additionally, our solutions are competitive with, or outperform heuristic-based algorithms.

Though we have demonstrated applicability for three diverse architectures, it is yet unclear what spatial architectures could render our framework ineffective. On the other hand, our techniques will be relevant in other domains in graph mapping, like memory controller scheduling, process scheduling in the context of an operating system, or job scheduling/partitioning in large heterogeneous systems. We believe that the infrastructure we created can serve as the basis and inspiration for future research.

References

[1] General algebraic modeling language, http://www.gams.com/.

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

[3] 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 '09*.

[4] 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.

[5] 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-VIII*.

[6] 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-XII*.

[7] 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 '04*.

[8] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *PLDI '13,* **To Appear**.

[9] 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 '08*.

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