SAT-Based Compilation to a non-vonNeumann Processor

Samit Chaudhuri
Wave Computing
Campbell, CA
samit@wavecomp.com

Asmus Hetzel*
Amazon Development Center
Berlin, Germany
ahhetzel@amazon.com

ABSTRACT

This paper describes a compilation technique used to accelerate dataflow computations, common in deep neural network computing, onto Coarse Grained Reconfigurable Array (CGRA) architectures. This technique has been demonstrated to automatically compile dataflow programs onto a commercial massively parallel CGRA-based dataflow processor (DPU) containing 16000 processing elements. The DPU architecture overcomes the von Neumann bottleneck by spatially flowing and reusing data from local memories, and provides higher computation efficiency compared to temporal parallel architectures such as GPUs and multi-core CPUs. However, existing software development tools for CGRAs are limited to compiling domain specific programs to processing elements with uniform structures, and are not effective on complex micro architectures where latencies of memory access vary in a non-trivial fashion depending on data locality. A primary contribution of this paper is to provide a general algorithm that can compile general dataflow graphs, and can efficiently utilize processing elements with rich microarchitectural features such as complex instructions, multi-precision data paths, local memories, register files, switches etc. Another contribution is a uniquely innovative application of Boolean Satisfiability to formally solve this complex, and irregular optimization problem and produce high-quality results comparable to hand-written assembly code produced by human experts. A third contribution is an adaptive windowing algorithm that harnesses the complexity of the SAT-based approach and delivers a scalable and robust solution.

Categories and Subject Descriptors
Hardware → Emerging architectures, Hardware → Emerging languages and compilers, Hardware → Emerging tools and methodologies.

General Terms
Algorithms, Performance, Design.

Keywords
Dataflow, CGRA, Neural Networks, Parallel Processors, Compilers, Satisfiability, Scheduling.

1. INTRODUCTION

With growing interest in deep learning, there is renewed focus on new computing architectures that can address the limitations of traditional von Neumann computing on deep-learning workloads. These new architectures often require new compilation techniques.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICCAD'17, Nov 13–17, 2017, Irvine, CA. USA.
Copyright 2017 ACM 1-58113-000-0/00/0010 ...$15.00.

This paper describes a compilation technique for a massively-parallel dataflow processor (DPU), to efficiently execute the neural networks that form the core of most modern deep-learning applications. The 16nm DPU processor uses self-timed logic to reach speeds of up to 10GHz, and contains 16000 independent Processing Elements (PEs) that can issue a total of 180 trillion of 8-bit integer operations per second [1].

The DPU processing substrate of 16K PEs resembles a Coarse Grained Reconfigurable Array (CGRA) such as RaPID [3] or ADRES [5]. As in case of CGRAs, the PEs are tightly coupled and proceed in lock step. Unlike CPUs, the instructions in the PEs are statically scheduled and not reordered at run time. It is well known that CGRA architectures spatially flow and reuse data from local memories to provide higher computation efficiency compared to temporal parallel architectures such as GPUs and multi-core CPUs [2]. However, unlike conventional CGRAs, the DPU instructions are used to perform dataflow computation, which works well with machine-learning workloads where an application is declared as a graph of communicating agents. The agents are statically scheduled, placed, and routed on a region of the DPU, and are executed concurrently. Unlike traditional token-based dataflow, the agents are data driven: if data is not present, the agent skips computation in that cycle. This approach is very different from multi-kernel dataflow-controlled computation explored in Mosaic [7] where the CGRA is enhanced with dynamic communication channels common in MPPA architectures such as ASAP2 [8] and RAW [9]. In this way DPU uses spatial CGRA as the base architecture for computation efficiency, and adapts it to dataflow computation to execute deep neural network applications with full efficiency.

The programming environment for CGRAs often provide a C-like language with directives that are used by the programmer to specify inner loops, placement etc. However, the techniques for code generation vary. For example, ANDRES uses the DRESC [4] compiler to place and route operations inside an outer scheduling loop. While DRESC is built for a specific micro architecture, the SPR [6] compiler takes a more generic approach of combining a VLIW-style scheduler with FPGA-like place and route algorithms that can be adapted to broader class of CGRAs. In summary, all the existing approaches use an ensemble of scheduling, placement, and routing heuristics to solve the combined problem with some creative feedback scheme to address the interdependencies of the sub problems.

The ensemble approaches neither guarantee optimal quality across sub problems, nor do they provide a measure of their blind spot around the optimal solutions. Our experience shows that although the ensemble approach produces assembly code that is correct-by-construction, the quality of the code can deviate significantly from assembly code produced by human experts on domain specific problems such as FIR, FFT, matrix multiplication, encryption, compression etc. These quality gaps are caused by two primary factors: human experts pay more attention to the performance bottlenecks of the program, and human experts exploit
Performance bottlenecks in programs are usually localized around small but critical code sections. These critical code sections must exploit all capabilities of the underlying processor architecture to generate optimal results. One of these bottlenecks is related to data locality. Compilers for von Neumann processors adhere to the von Neumann bottleneck and generate code that fetches data in a relatively uniform fashion from memory or register files, while leaving data locality to be handled by the cache subsystem. Instruction schedulers used in ensemble approaches are constrained by this limitation.

The DPU architecture removes the von Neumann bottleneck by providing rich flexibilities to exploit data locality. It allows operands to be stored in a variety of registers and memories and latencies of accessing the operands vary significantly depending on the spatial distance between the operands and the instructions. Additionally, often operations can be performed by alternative assembly code fragments; e.g., a 32-bit add can be performed using cascaded local 8-bit adds on 4 PEs, or using a single 32-bit add in the arithmetic co-processor. Depending on location of the operands, one alternative may outperform the other.

For high quality results, it is crucial to explore these flexibilities in context. However, ensemble methods that use a scheduler cannot account for locality-dependent latencies for memory access. Boolean Satisfiability (SAT) solvers show promise in this area due to their ability to perform combinatorial search under arbitrary and irregular constraints. This paper describes a SAT-based compilation technique that overcomes the limitations of prior ensemble approaches and consistently produces superior solutions that are comparable or even better than those produced by human experts.

SAT-based formulations have been a powerful tool to solve a variety of problems including formal verification [10, 11], FPGA routing [12], logic optimization [13], test-pattern generation [14], and detailed placement on time-multiplexed FPGAs [15].

However, we are not aware of any research to solve an industrial-scale compilation problem onto a spatial architecture that includes a wide range of spatial and temporal combinatorial sub-problems including scheduling, placement, and routing.

2. SW AND HW ARCHITECTURE

This section includes two sub sections that describe the programming environment and the device architecture respectively in sufficient detail needed to build a compilation model that is presented in the next section.

2.1 DPU Programming Model

The current programming environment supports a restricted set of C++ with user-defined directives for high-level programming. A compiler frontend converts the high-level program to Static Single Assignment (SSA) form and extracts the parallelism to create an intermediate dataflow representation called Flow Graph (FG).

A Flow Graph is an intermediate representation of a program where nodes represent operations and links represent bytes or bits. FG has a dataflow-like execution semantics where multiple operations execute concurrently whenever they have valid values available at their inputs. At each execution step each operation consumes the values from its inputs, and produces new values at its outputs. There is no need for dedicated clock signals or program counters to sequence the operations - the presence of valid data is all that is required to manage the orderly processing of information.

FG nodes perform a comprehensive set of primitive operations that include Boolean (logical or bit-wise), arithmetic, storage, and control functions sufficient to express programs that are meant to be executed on the DPU. Another important feature of FG is that the primitives are completely generic and are not tied to any details of the DPU architecture. The association between the primitives and machine instructions are established through node implementations. A node implementation describes a machine code segment that performs the operation at the node. The compiler assumes that each primitive operation is given one or more node implementations to choose from.

2.2 DPU Architecture

Each Processing Element (PE), shown in Figure 1, behaves like an independent processor with its own instruction and data RAM, a register file, and an 8-bit data path. However, unlike a RISC or VLIW processor that reads are operands from and to a register file, a PE can also fetch and forward operands from and to neighboring PEs. The implementation is tuned so that it is in fact faster to exchange data with neighboring PEs than to store it in its local register file or data RAM. Therefore, it favors dataflow computation by encouraging data to flow after it is processed by a PE. Each PE contains a logic unit that can execute compare, shift, rotate, and logical instructions.

As illustrated in Figure 2, 4 PEs are connected into a quad, and 4 quads (16 PEs) are grouped into a cluster. Additionally, each cluster contains a switch to support high speed local communication with PEs in other clusters. The cluster also contains two 32-bit arithmetic co-processors that can execute addition, subtraction, bit-manipulation, and multiply-accumulate with signed or unsigned saturation. Furthermore, the arithmetic units can also perform 8-bit or 16-bit operations in SIMD fashion, i.e. 4 8-bit, or 2 8-bit and 2 16-bit simultaneously. Latencies of such instructions vary with their complexity. The data paths are pipelined so that complex instructions can be issued at rates faster than their latencies.
Although the PEs can issue instructions once per machine cycle, different instructions operate at different latencies and dispatch rates. To support this high-speed instruction issue rate, the micro-architecture is finely tuned so that different instructions consume operands and produce results along different register transfer paths to prevent complex instructions from blocking simpler instructions. Aggressive pipelining along a multitude of register transfer paths entails the compiler to use a very detailed micro-architecture model and precisely account for every register-transfer delay stage.

The DPU chip contains 1024 clusters tiled in a 32x32 array. The switches allow each cluster to send data to any of its neighbors, or route data over a series of hops through the clusters.

3. COMPILATION MODEL

The compilation model ties the execution of a program to the architecture of the device both of which are described in the previous section. Modelling is crucial for efficient compilation and needs to represent the concrete micro architecture of the device with an abstraction that is both flexible and generic. A flexible model allows future extensions and modifications to incorporate new requirements coming from findings in the existing hardware, or from next generation architecture changes. A generic model is needed to efficiently run complex algorithms for high performance compilation. Our model is built upon the following key concepts:

A resource usually describes an elementary component of the micro architecture such as an 8-bit register, a 1-bit register, or an arithmetic-logic unit that can execute an instruction. A resource may also describe an abstracted hardware concept such as a specific part of an arithmetic pipeline. Abstracted hardware concepts are useful for modeling various contention requirements that the compilation process must satisfy.

Time is measured in a discrete unit called a sub-tick. The PEs in the DPU architecture allow a certain finite number of sub-ticks after which the program counter is reset, usually 256.

A bucket is a time multiplexed container of resources, so it provides a certain amount of resources for every sub-tick. Besides introducing the notion of time, buckets are useful to group certain resources together, specifically sets of equivalent registers.

The content of a resource is called a signal; a resource can contain at most one signal at any given sub-tick. A signal usually describes the byte or bit present in a specific component at some sub-tick, or an instruction generating some result. A signal may also describe an abstracted quantity useful for modeling contention rules, special time dependencies between instructions etc.

A reservation describes the occurrence of a specific signal s in a specific a resource r at a specific sub-tick t, and is denoted with the triple \((r, s, t)\).

A template describes a method of generating one or more signals in specific resources at specific sub-ticks out of other signals. A template contains

- **output reservations** that represent the outputs of the template as a set of reservations \([o_1, ..., o_i]\) that are generated by the template.
- **input reservations** that represent the prerequisites of the template as a set of set of reservations \([\{d^{1_1}, ..., d^{1_{m_1}}\}, ..., \{d^{n_1}, ..., d^{n_m}\}]\)

When a template is applied, certain signals are generated at specified location(s) in space and time and are described as output reservations. As a prerequisite, certain other signals must be present at certain other space/time locations. Input reservations capture these prerequisites where each prerequisite is represented as a set of choices from which at least one reservation must be present. The choices for a prerequisite reflect the flexibility that the input to a certain instruction may be fetched from various registers. Non-trivial and diverse latencies of register access, execution latency of instructions, and data transport in the architecture can all be implicitly modeled within such a template. For example, input reservations may be associated with very different sub-ticks depending how long it will take an instruction to read/write from/to the different registers.

Multiple types of templates are used. Most common templates, called node templates, are used to capture the computations performed by the nodes in the flow graph. Each node template is associated with a node and models its input output requirements as well as the machine instruction(s) needed to perform its computation. Additionally, all potential contentions relative to other templates are modeled using abstract resources listed in the output sets.

The compilation process explores the possibility of computing the flow-graph nodes at different times, on different PEs in multiple ways using different machine code fragments (e.g. cascading 4 8-bit adds vs a single 32-bit add) and the best choice depends on its context; e.g. criticality of the node, location of operands in registers of memory etc. These degrees of freedom are expressed by associating each flow graph node with a multitude of node templates each of which denotes an alternative implementation of the node.

Some other types of templates are

- **Routing templates** model machine instructions that move data between different registers. These templates have the same signal in their input and output reservations.
- **Input templates** model primary inputs of a flow graph. These templates have empty input reservations, and may only be realized in specific buckets that model the connectivity of a PE to the switch.
• **output templates** model primary outputs of a flow graph. These templates may only be realized in specific buckets that model the connectivity of a switch to a PE.

This generic and comprehensive model is sufficient to capture all the details and the degrees of freedom that need to be managed and explored by the compilation process. One primary advantage of such a comprehensive formulation is that it models the underlying hardware and architecture very accurately. Since the model preserves all information, it allows the compilation stage to fully utilize the capabilities of the underlying hardware. This is crucial for generating high-quality assembly code, which often exploits the special tweaks in the architecture.

### 3.1 Compilation Preprocessing

The first part of the compilation generates an intermediate database containing the compilation model as described in the prior section. Resources and buckets are extracted from a suitable XML-representation of the device. For every node in the flow graph, node templates are generated for all possible space/time locations and every possible elementary implementation. Additionally, all templates that correspond to pure movement of data are generated at all space/time locations for all signals. This potentially large database is significantly compressed by exploiting redundancy. For example, templates that are the result of a temporal translation can be encoded in full only once and then just kept in the database with their relative time offsets.

The latencies of the instructions, and the latencies of data transfer between the components of the micro architecture are all accounted for while constructing the templates. For example, consider a routing template, that moves data between different registers, with an output reservation (r, s, t). The input of this template is single set of alternative reservations \\{ \{ r_1, s, t_1 \}, ..., \{ r_n, s, t_n \} \}. In other words, signal s must occur in at least one of the r_i resources at sub-tick t_i which depends on the value of the latency between resources r and r_i. If the n resources belong to different buckets, then the values of t_i may vary depending on the latencies. Note that both resources may be identical in which case this encodes a rollover of data within the same register.

Due to the high redundancy in the templates, compilation preprocessing is very fast and needs little memory. Because of this, the current compiler does it on the fly each time it is invoked and needs no access to a pre-build database.

### 4. SAT MODEL

#### 4.1 Basic Model

A SAT model is derived from the compilation model as follows:

- For every possible template T_i, a Boolean variable t_i is introduced where t_i == true denotes that the template will be applied and assembly code according to this template will be generated.

- For every possible reservation D_i, a Boolean variable d_i is introduced where d_i == true denotes that the reservation is used in the compiled code (i.e. a specific signal will be present in a specific register at a specific time, an instruction will be executed at a specific time, or an abstract resource will be used at a specific time).

The constraints that link inputs and outputs of templates are Boolean dual Horn-clauses of the following type:

1. \[ \land \{ \land \{ t_{i_1} \} \}, ..., \land \{ t_{i_k} \} \} \] where the t_j denote all templates that have d_j as output, and

2. \[ \land \{ \land \{ d_{j_1} \}, ..., \land \{ d_{j_k} \} \} \] where the \{d_{j_1},...,d_{j_k}\} is any of the sets that are inputs to the template represented by t_i

3. \[ \land \{ o_{i_j} \} \] where o_j is any of the outputs of the template represented by t_i

The first set of constraints essentially means that a certain reservation can only be part of a solution of the SAT-problem if it is the output of some template that is applied in the solution. The second set of constraints denote that a template can only be applied if at least one reservation is part of the solution within every set of reservations that define one template input. The third set of constraints denotes that all output reservations of a template must be part of a solution when the template itself is applied.

Additional constraints are added to the SAT-model to ensure that resources are not oversubscribed in a solution. For each resource/sub-tick pair, there is an "at most k" constraint that constitutes an upper bound on the number of used resources within this sub-tick. For the vast majority of the cases, the value of k is 1; however, for register banks, the value of k is larger. (but practically k <= 8 in all cases). Such constraints are encoded using binomial encoding [16] for k==1, and pairwise cardinality networks for k > 1 [17]. These constraints are referred as capacity constraints in the sequel.

Additionally, a set of constraints is added that ensures that exactly one out of all templates that represent the same flow graph node must be applied in the solution, as well as exactly one out of all templates that represent the same primary input or primary output. These exactly-one type of constraints is called cloning constraints.

A solution of an instance of this SAT problem is decoded to an assembly program that describes a valid execution of the flow graph on the hardware device. A node template that is applied in a solution maps directly to a set of instructions that are executed by one or more PEs at specific times. For routing templates applied in the solution, each output resource that represents a register is decoded into an appropriate data transfer instruction in the compiled code. Operands for the hardware instructions that specify the specific source or target registers of the instruction can be derived by examining the variables for reservations that are set to "true" in a SAT-solution.

In this basic model, optimization towards a solution with low latency can be trivially achieved by repeatedly trying to solve the Boolean model while tying all variables representing reservations \( (r, s, t) \) with \( t > z \) to zero in the input to the SAT-solver. Varying over \( z \) by binary search will yield the optimal assembly code with respect to execution time.

Since this model is derived using simple general principles, it can cover almost all constraints that are imposed by an arbitrary micro architecture. The template mechanism is very expressive and allows modelling a large diversity of constraints. Specifically, contention constraints around the use of specific resources by different parts of the parallel executed code can be encoded easily by means of abstract resources used by templates.

This expressive and flexible model had an additional practical benefit: it enabled iterative simultaneous iterative refinement of hardware and software. The micro-architecture could be improved late in the design cycle, because the compiler could accommodate these improvements with local modifications in the templates. The
core SAT solver engine remained unaffected and could reliably produce new solutions.

4.2 Simplifying the SAT Model

The basic formulation presented in the previous section can be used to create SAT problem instances that can be solved by any open source or commercial SAT solver. We use a well-known and efficient solver called Minisat [18]. For wide application, it is important to simplify the SAT problem instances so that they do not exceed the solver’s capabilities. This section presents two techniques we use to simplify the SAT problem instances.

The first approach reduces the size of the model with a detailed As-Soon-As-Possible/As-Late-As-Possible (ASAP/ALAP) analysis to bound the mobility of the templates prior to constructing the SAT model. Compared to a conventional temporal analysis of the flow graph, it computes much more precise bounds by performing the analysis on the templates in context of the spatial and architectural data. For each potential resource/signal pair (r, s), a lower and an upper bound \( t_{\text{min}} \) and \( t_{\text{max}} \) are computed such that reservations \((r, s, t)\) can only be part of the SAT-solution when \( t \) is in the window \([t_{\text{min}}, t_{\text{max}}]\). Computed based on the template representations, these bounds accurately account for register-transfer latencies and are tighter than conventional spatially-unaware flow-graph based computations. These bounds help prune large numbers of variables and clauses that can be discarded prior to constructing the SAT model.

Note that ASAP/ALAP analysis on templates is based on two types of clauses presented in the previous section: types 1 clauses stating that any reservation must be the output of some template, and type 2 clauses stating that a template depends on other reservations as inputs.

The second technique is based on the observation that the SAT solver converges faster when, instead of starting with all the constraints and clauses, it works on them in stages, as if it were a cooking process that uses clauses and constraints as ingredients. In our set up, Minisat [18] is initially started without any capacity constraints because their encodings often create large numbers of additional variables and clauses. Once a solution is found by Minisat [18], all the capacity constraints are examined and the violated ones are encoded as a Boolean network and added to the SAT model. Then Minisat [18] is restarted incrementally and this process is repeated until no more violated capacity constraints are found in the solution.

Using these techniques, we can optimally minimize execution latency on compilation problems involving up to 16 PEs and 100 flow graph nodes. The next section presents another technique to scale this approach further to solve even larger problems.

4.3 Scaling with Adaptive Windowing

This section presents a technique called adaptive windowing to bound the overall complexity of a single SAT-problem so that larger programs can be compiled into high-quality assembly code in reasonable time.

SAT usually exhibits exponential runtime explosion at some point. Adaptive windowing breaks down the overall problem into a series of smaller sub problems working on a series of overlapping time windows. The size of each individual window is customized according to the complexity of the sub problem so that each sub problem can be solved by the SAT solver in a run time that is bounded by a constant.

We can use the analogy of weaving a carpet on a loom. The entire carpet is woven over several days by weaving a part of the carpet each day. However, the size of the woven part varies depending on the intricacy of the pattern in the part. Similarly, the SAT problem is solved over several time windows, where the size of the time window depends on the complexity of the flow graph in the time window.

To solve the SAT problem in each time window within bounded run time, the inherent all-or-nothing paradigm of SAT-based approaches must be avoided. A set up where compilation may result in either good or no solution is not workable. The SAT solver must make progress in each time window, just like the weaver must weave some part of the carpet each day.

We achieve this type of scalability and robustness with an iterative compilation process containing repeated SAT-based optimizations. The entire compilation task is broken into a series of packing sub problems on overlapping time windows. We define the time windows as overlapping sub-tick intervals \( I_0 = [l_0, u_0], I_1 = [l_1, u_1], ..., I_n = [l_n, u_n] \), where \( l_i \leq l_{i+1} \leq u_i \leq u_{i+1} \), and work on each window individually. The solutions to the packing sub problems are accumulated into a partial solution as a set \( Q \) of templates that have already been selected to be part of a final solution. The iterative compilation process starts with \( Q = \emptyset \), and in every iteration, solves a packing sub problem and accumulates the newly applied templates into \( Q \). Therefore, at any point in time, \( Q \) partitions the flow graph nodes into two sets: nodes with associated templates in \( Q \) are called realized nodes, and all other nodes are called unrealized nodes. The goal of the compilation process is to realize all nodes while minimizing overall latency.

Iteration \( i \) starts with a partial solution \( Q_{i-1} \), and tries to create a new partial solution \( Q_i \) by applying templates from time window \( I_i = [l_i, u_i] \), while adhering to the following 3 invariants:

- Invariant 1 ensures that the new partial solution \( Q_i \) remains legal to the global SAT model. Except for the cloning constraints, the templates in \( Q_i \) must satisfy all constraints of the SAT model in Section 4.1.
- Invariant 2 ensures that the prior partial solution \( Q_{i-1} \) remains restricted to the prior sub-ticks. For all reservations \((r, s, t)\) used in inputs or outputs of any template in \( Q_{i-1}, t \leq u_{i-1} \).
- Invariant 3 ensures that the additional templates applied in the current iteration can be “stitched” to the prior partial solution \( Q_{i-1} \). This basically means the input signals must be delivered to every applied template. In other words, for every reservation \((r, s, t)\) that is referenced as an input of a template associated with an unrealized node, there is either an unrealized node that has an output template \((r', s, t')\), or there is a reservation \((r', s, t')\) in an applied template where \( t' > l_i \). The first condition represents the case where signal \( s \) is generated by an instruction, and the latter represents the case where signal \( s \) is routed from another resource.

These invariants are used to construct the SAT model for iteration \( i \) as follows:

1. Build a SAT model as in Section 4.1 but restrict the model to the time window \([l_i, u_i]\), and relax the cloning constraints from "equal one" to "at most one". This step ensures invariant 1.
2. Add constraints to ensure that every template in \( Q_{i-1} \) is part of the solution. This step ensures invariant 2, and additionally ensures that \( Q_i \) grows monotonically.
3. Add constraints to ensure that invariant 3 holds w.r.t. the sub-tick \( u_i \) in any solution of the SAT model.
4. Add constraints so that all additional templates applied in this iteration are contained within the current time window. If a template is not in $Q_{\lfloor s \rfloor}$ but references a reservation $(r, s, t)$ with $t < l$ or $t > u$, then it is not part of the solution.

It is evident that such a SAT model for the time window $[l_0, u_0]$ can be constructed efficiently. For the time window $[l_1, l_1]$, the templates in $Q_{\lfloor s \rfloor}$ are set to "true", and the templates not in $Q_{\lfloor s \rfloor}$ are skipped. Hence the complexity of the SAT-model is bounded by the number of templates completely within $[l_1, u_1]$.

To avoid the all-or-nothing paradigm of SAT, this local model is solved by Boolean optimization. The integral linear objective function contains two parts. The first part contains a weighted number of additional node templates that are applied in the solution of the current sub-problem but are not part of $Q_{\lfloor s \rfloor}$. Since a node can be implemented by multiple alternative node templates, the weight of each node template reflects the average number of processor instructions in those choice templates. Optimization this part of the objective attempts to pack as many instructions as possible into the current time window. This part ignores the instructions inside the routing templates to allow easy movement of data to the inputs of the unrealized nodes.

To guide this optimization safely to low-latency compiled code, a second part is added to the linear objective to minimize the timing criticality of unrealized nodes in the solution. Timing criticality for each node template is inferred from the ALAP-analysis described in Section 4.2. These two parts are combined into one linear objective by a weighting factor that ensures that timing criticality dominates packing density in the objective until a cut-off point.

This integral linear objective can be efficiently encoded as pure SAT-clauses in various ways; one such method using base-n encodings is shown in Minisat+ [19]. We represent the two parts of the objective function in two different ways. We encode the first part of the objective, which counts the number of packed instructions, with at-most-k constraints using pairwise cardinality networks [17]. For the second part, we discretize timing criticality with a bit vector where each bit represents a range of criticality. This bit vector is used to create unary integral coefficients for the templates with criticality within the same range. This effectively provides exponentially increasing weights for timing critical templates.

As in [19], the SAT-problem is first solved without the objective. Then iteratively the objective is evaluated on the current solution, a constraint is added to force the solution to improve the current value of the objective by 1, and the solver is restarted incrementally. This process is repeated until no solution can be found (which implies the prior step produced the optimal solution), or until a certain amount of maximum effort has been spent. The effort is specified in terms of the maximum number of conflicts that the solver can discover.

This approach allows us to bound the size of the SAT-optimization problems in each step. Still occasionally some difficult compilation tasks can create problems with up to ten million variables which is usually far too big to be even nearly exploited by the SAT-solver. The key to success lies in the fact that the optimization starts with very easy solvable problems and incrementally works towards more difficult ones. A valid initial solution for every optimization problem is usually to pack only routing templates to transfer data from the applied templates to the upper boundary of the current time window for consumption by the unrealized nodes. There is usually an easy solution for this: just move content straight up in time without changing the occupied resource. Starting from this easy initial solution, the optimization then exploits the search space finding more and more dense configurations that pack additional non-routing templates into the time window. While doing so, the SAT-solver efficiently updates its internal database with learnt clauses, thus making the search process more and more effective. This helps obtain final solutions with very dense assignment of instructions to processors.

As time windowing is done across the entire spatial domain, there is still some limit in terms of spatial scaling. We currently restrict the compilation problems to up to 160 PEs at a time. Applications that need to use more PEs usually consist of several independent and potentially repeated sub-tasks. Such applications are currently manually partitioned, compiled independently and assembled together by a specialized linking step.

### 5. EXPERIMENTAL RESULTS

The SAT-based compiler software is in production and exhibits highly robust run times even when presented with large and complex compilation problems. We also have an earlier internal version of the compiler built using the ensemble method (scheduler + placer + router) that we use as a baseline for comparison. As a third data point, we use assembly code written by human experts with specialized domain knowledge. This section presents comparative experimental data collected from these 3 sources.

Table 1 summarizes the results of the experiments on 6 different programs. The first program, Shelby, is a streaming application used as an internal demonstration vehicle. Programs div implements half-precision fixed-point division, AES256 performs AES encryption with a 256-bit fixed key as specified in FIPS-197, LZRW3 compresses 1KB data using the LZRW3 algorithm, and exponent performs fixed-point exponent using an algorithm similar to the CORDIC method. Finally, the turbo program performs LTE turbo decoding using the BCJR algorithm with max-log map as in 3GPP Technical Spec TS 36.212 v8.3.0.

For each program, column 2 shows the size of the program in instruction count. Since our goal is to maximize performance, we use the latency metric in units of sub-ticks to compare the quality of the assembly code. The remaining 3 columns present the latency metric as produced by a human expert, the ensemble algorithm, and the SAT-based compiler respectively. To make the latency numbers directly comparable, all 3 implementations of a given program uses the same number of PEs.

The results in the Table show that the SAT-based compiler takes us closer to our original goal of producing extremely compact assembly code comparable to the ones produced by human experts. It also shows that the ensemble approach leaves a lot of room for improvement in the quality of the final assembly code because it cannot aggressively utilize the flexibilities present the microarchitecture for exploiting data locality. The SAT-based approach also proved to be very malleable, and allowed safe modifications late in the HW design cycle.

### 6. SUMMARY AND FUTURE WORK

The paper presented a SAT-based approach for compiling dataflow programs onto a CGRA-based non-vonNeumann processor architecture called DPU. We showed how the SAT formulation can explore a variety of memory access latencies during compilation and produce high quality assembly code comparable in quality to those produced by human experts. Additionally, we presented a technique called adaptive windowing to scale the SAT-based
approach to solve larger problems. The compilation strategy avoids the all-or-nothing trap of SAT solutions by ensuring forward progress while keeping track of the effort spent in the search.

**Table 1. Experimental Results**

<table>
<thead>
<tr>
<th>Program</th>
<th>Instruction Count</th>
<th>Latency in Sub-ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Human Expert</td>
</tr>
<tr>
<td>Shelby</td>
<td>186</td>
<td>19</td>
</tr>
<tr>
<td>div</td>
<td>617</td>
<td>NA</td>
</tr>
<tr>
<td>aes256</td>
<td>1250</td>
<td>127</td>
</tr>
<tr>
<td>lzrw3</td>
<td>1286</td>
<td>128**</td>
</tr>
<tr>
<td>exponent</td>
<td>1408</td>
<td>NA</td>
</tr>
<tr>
<td>turbo</td>
<td>6197</td>
<td>NA</td>
</tr>
</tbody>
</table>

* globally optimal solution

** code was incomplete; missing a few memory accesses

The compiler software is in production and exhibits highly robust run times even when presented with large and complex compilation problems. It also consistently outperforms our internal implementation of ensemble method both in run time and quality of the assembly code.

In near future, improving the run time of the compiler is of high interest, since compiler run time directly affects programmer productivity. There are multiple avenues of improvements to consider. It may be possible to speed up the learning rate of the SAT solver by supplying additional redundant constraints so that the solver does not have to learn them anymore. Further improvements can come from experimenting with variable ordering inside the solver, and from exploring different Boolean encodings.

7. ACKNOWLEDGMENTS

Our thanks to Alan Mishchenko for numerous discussions and suggestions on improving the SAT formulation.

8. REFERENCES


