DynaSpAM: Dynamic Spatial Architecture Mapping using Out of Order Instruction Schedules

Feng Liu Heejin Ahn Stephen R. Beard Taewook Oh David I. August
Princeton University
{fengliu,heejin,sbeard,twoh,august}@princeton.edu

Abstract

Spatial architectures are more efficient than traditional Out-of-Order (OOO) processors for computationally intensive programs. However, spatial architectures require mapping a program, either statically or dynamically, onto the spatial fabric. Static methods can generate efficient mappings, but they cannot adapt to changing workloads and are not compatible across hardware generations. Current dynamic methods are adaptive and compatible, but do not optimize as well due to their limited use of speculation and small mapping scopes. To overcome the limitations of existing dynamic mapping methods for spatial architectures, while minimizing the inefficiencies inherent in OOO superscalar processors, this paper presents DynaSpAM (Dynamic Spatial Architecture Mapping), a framework that tightly couples a spatial fabric with an OOO pipeline. DynaSpAM coaxes the OOO processor into producing an optimized mapping with a simple modification to the processor's scheduler. The insight behind DynaSpAM is that today's powerful OOO processors do for themselves most of the work necessary to produce a highly optimized mapping for a spatial architecture, including aggressively speculating control and memory dependences, and scheduling instructions using a large window. Evaluation of DynaSpAM shows a geomean speedup of 1.42× *for 11 benchmarks from the Rodinia benchmark suite with a geomean* 23.9% *reduction in energy consumption compared to an 8-issue OOO pipeline.*

1. Introduction

Out-of-Order (OOO) processors deliver high performance by dynamically training speculative units, such as branch and memory dependence predictors, to expose more instruction level parallelism for scheduling. However, even if a program enters a relatively fixed execution pattern, the processor cannot take full advantage of the predictable behavior and unnecessarily spends energy on regenerating a schedule for the same instruction trace [27]. Moreover, data communication between functional units occurs through explicit centralized structures, such as the physical register file and bypass network, even though the dataflow is well known and could occur through highly efficient dedicated data paths for the repeated code sequence [4, 17, 18, 50, 49].

In contrast, spatial architectures map computation across a grid of Processing Elements (PEs) and build specialized data connections between them to fulfill dependences. Fixing instruction assignments to PEs obviates the need to separate instruction execution into multiple pipeline stages (fetch, decode, rename and issue) and direct communication from producers to consumers obviates the need for the bypass network and register file [4, 17, 18]. However, considering hardware resources is critical when generating mappings for spatial architectures in order to make effective use of PEs and minimize the datapath usage between them.

A class of reconfigurable spatial architectures, such as Programmable Functional Units (PFUs) [3, 6, 10, 12, 17, 18, 37, 47, 48] and Coarse-Grained Reconfigurable Fabrics (CGRFs) [16, 28, 29, 36, 41, 46], uses static techniques, which offer a large scope, for mapping. The large scope enables the mapping generator to consider more instructions simultaneously, and thus produce mappings that achieve more efficient resource utilization. Executed a priori, static methods cannot make use of information gathered at runtime to optimize their mappings for changing workloads. Additionally, programs that are statically mapped to a particular reconfigurable fabric cannot run effectively on a processor without the fabric, and may not be compatible with different fabric generations.

Dynamic mapping methods can overcome the adaptability and compatibility issues that static methods face. However, due to the small instruction scopes and lack of speculation, current dynamic techniques fail to make the best use of routing resources in spatial architectures [10]. This might lead to increased execution time and energy consumption, as well as, in some cases, an inability to produce a feasible mapping.

To address this problem, we present DynaSpAM, Dynamic Spatial Architecture Mapping. The insight of DynaSpAM is that OOO processors excel at utilizing speculation and contain large instruction windows, thus combining OOO resources with the execution efficiency of spatial architectures leads to a more effective system. DynaSpAM efficiently and transparently integrates with, and utilizes the resources of, an OOO processor to dynamically map large instruction traces to a

ISCA'15, June 13-17, 2015, Portland, OR, USA
 ISBN 978-1-4503-3402-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750414

reconfigurable fabric. The contributions of this paper are:

- A design for a reconfigurable spatial fabric that supports speculative execution by reusing existing hardware prediction units in the host processor.
- A dynamic resource-aware mapping technique for reconfigurable spatial fabrics that leverages the existing scheduling logic in the host processor to provide a large mapping scope with only a small hardware cost.
- An overall architecture framework that contains trace detection, mapping, and acceleration to transparently offload hot program traces to a reconfigurable spatial fabric dynamically. Results show a 1.42× speedup for 11 benchmarks with a 23.9% reduction in energy without any binary modification.

2. Background and Motivation

2.1. Spatial Architectures

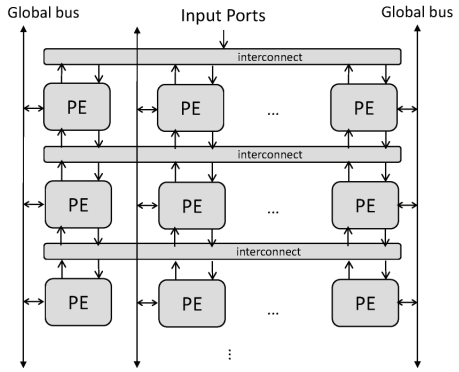

Figure 1: A typical spatial architecture organization.

While OOO processors provide excellent single core performance for general purpose programs, they are not energy efficient due to the complex instruction pipeline and centralized data communication required to deliver instructions and propagate data to the functional units [19, 24, 35]. Specializing hardware, using techniques such as CCA [10], DySER [17], or SGMF [46], can reduce power consumption. These techniques distribute computation to spatial elements and build the data dependences within the configurable circuit connections, and thus save energy on instruction delivery and data communication. This type of computer architecture is usually referred to as a *reconfigurable spatial architecture*.

A typical organization of a reconfigurable spatial architecture is shown in Figure 1. Normally, the reconfigurable spatial architecture is constructed as a grid of PEs and each PE has connections to its neighbors through an internal interconnect network. Different spatial architectures have different internal interconnect designs with different latency and complexity trade-offs. For example, in one row of PipeRench [16], a PE is connected only to its adjacent PEs, while the PEs in the same row of CCA [10] cannot communicate with each other at all. Due to the two-dimensional physical arrangement of PEs in spatial architectures, the distribution of external input and output ports are heterogeneous, i.e. the number of input and output ports directly accessible to each PE varies based upon it's location. For example, only the PEs in the top row of CCA can directly access inputs (from the register file or data bus) [10], while only half of the PEs at the edge of DySER can directly access inputs [17]. Transferring input data to internal PEs costs extra cycles and routing resources, or, in some cases, is impossible. Obviously, this resource heterogeneity should be considered when mapping instructions to PEs to get achieve good performance. A global bus can connect PEs and partially reduce the heterogeneity to provide flexibility in instruction mapping.

2.2. Limitations of Current Mapping Techniques

For both static and dynamic mapping techniques, the core problem is to satisfy the following three types of constraints when instructions are selected and placed to the PEs of a spatial fabric: *functionality*, *communication-resource*, and *timing* constraints [33, 22, 23] . The description of each constraint and our methods to satisfy them are elaborated in Table 1. *Functionality* and *communication-resource* constraints determine the feasibility of the mapping. *Timing* constraints determine the mapping (scheduling) quality, which is usually expressed as an optimization objective. In an OOO processor, *functionality* and *timing* constraints are resolved via instruction wake-up and selection logics. The *communication-resource* constraint is handled by communicating data through a centralized register file or bypass network. In a spatial fabric, the *communication-resource* constraint becomes especially important since the internal datapath and external input port resources are limited. Thus, the placement and routing decision by one instruction may conflict with that of another instruction. Consequently, a mapping technique for spatial architectures should be globally *resource-aware*. Static mapping methods are effective and *resource-aware* as they can map across a large enough scope of instructions to consider requirements from all instructions at the same time and thus produce a global mapping solution.

However, static mapping methods are not adaptable to changing workloads nor are they compatible across hardware generations. Dynamic mapping can overcome these limitations. Potentially, dynamic mapping techniques can be developed from existing dynamic optimization techniques, which use a method similar to trace caches [15, 39] to collect retired instructions and perform optimization. Unfortunately all existing dynamic techniques are naïve in the sense that they follow strict program order and consider only a single instruction at a time and thus make locally optimal decisions given the current constraints. For example, DIF [14, 32] places retired instructions in the *first* VLIW instruction word that can access its ready operands; and CCA [10] dynamically maps in-order instructions from the writeback stage to the *first* available functional unit that can receive its operands from the cross-bar. In the following two subsections, we elaborate on the limitations

Constraint	Description	Solution
Functionality	Specialized PEs in spatial fabrics can perform only select operations	OOO pipeline's Instruction selection logic (reused)
Communication-resource	Limited communication resources to provide operands	*Resource-aware* scheduling logic (new design)
Timing	Instructions should start as early as possible while respecting dependencies	OOO pipeline's Instruction wake up logic (reused)

Table 1: Mapping constraints and solutions

of current dynamic mapping techniques.

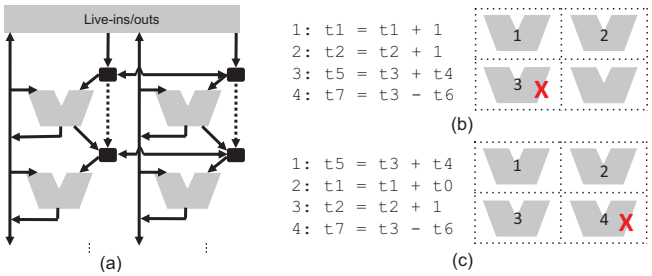

Figure 2: (a) for two spacial architecture settings (with and without dotted line), (b) and (c) show examples where a naïve method fails to create efficient schedules.

2.2.1. Limited Mapping Scope Limited and heterogeneous resource distribution in spatial architectures complicate the procedure of mapping instructions. Due to the limitation of a small mapping scope, naïve methods greedily satisfy the constraints of each instruction one at a time, and may create inefficient mappings or fail to produce a feasible schedule.

Figure 2 shows the importance of a large mapping scope when generating a mapping for reconfigurable spatial architectures. Figure 2(a) without dotted lines shows an example spatial architecture. While each row of PEs has the same capability, each has a unique set of input connections. PEs in the first row can get two operands from live-ins at the same time. PEs in the second row can receive only one live-in, which comes directly from a global bus such as in PipeRench [16].

Figure 2(b) shows how one naïve mapping results in a schedule failure. In the example code, the first two instructions have one live-in operand, and the following two instructions have two live-in operands. There is no dependence between these four instructions, so they can be scheduled independently. If a mapping generator can see all four instructions, it would map instruction 3 and 4 to the PEs in the first row and instruction 1 and 2 to the second row, thereby allowing all four instructions to be executed in a single cycle. However, the naïve mapping will place instruction 1 and 2 in the first row, resulting in a scheduling failure for instruction 3 and 4 because of the *communication-resource* constraint.

The *communication-resource* constraint failure can be resolved by adding more routing resources. For example, in Figure 2(a), if extra data paths (shown by dotted lines) are applied to allow for forwarding operands from one row to the next, the mapping shown in Figure 2(b) is feasible. However, forwarding requires extra cycles to complete and thus leads to lower performance. In this example, it two cycles are complete the computation with forwarding.

Figure 2(c) is another example demonstrating the deficiencies of naïve mapping. In this example, placing instructions 1 and 2 on the first row is reasonable as both of them require two source operands from live-ins. Instruction 4 also takes two source operands from live-ins, but there are no unallocated PEs in the first row. Fortunately, instruction 1 and 4 share the same source operand, `t3`, thus routing resources for `t3` can be reused by both instruction 1 and 4. However, in the naïve schedule shown in Figure 2(c), the PEs adjacent to instruction 1 are occupied, causing one extra datapath to be used and requiring two cycles to complete the bypass. With a larger mapping scope, the mapping generator will swap instruction 3 and 4 to make bypassing `t3` take only one cycle.

Although increasing the scope helps in finding efficient mappings, adding extra hardware logic to hold this scope is costly. The key insight of DynaSpAM is to avoid the cost by leveraging the scheduling logic of the host OOO processor. Since an OOO processor's scheduling logic is already equipped with a large instruction window and dependence analysis features, reusing them can allow a reconfigurable spatial architecture to generate efficient mappings with little to no additional hardware cost.

2.2.2. Limited Speculation Support Even with *resource-aware* mapping techniques, speculation support is necessary to relax *timing* constraints and achieve good mapping results. There are three kinds of program dependences that prohibit executing related instructions out of order without speculation: register dependence, control dependence and memory dependence. In DynaSpAM, register dependences are naturally handled by use of the data-flow execution model, in which one instruction starts to execute when its operands are ready or communicated from its producers by the physical datapath connections. However, spatial architectures must rely on speculation to break control and memory dependences in order to gain the freedom necessary to produce mappings that perform well.

Control Dependence Speculation With assistance from compilers, control speculation has been exploited for spatial architectures by forming enlarged basic blocks statically after profiling and checking their validity at runtime [18]. As a pure dynamic method, DynaSpAM utilizes the branch predictor of OOO processors to dynamic select code sequences across multiple basic blocks speculatively and execute them on the spatial architecture as fat atomic instructions.

Memory Dependence Speculation Properly handling memory instructions through the use of load/store (LDST) units, which properly respect memory order, in spatial architectures is a complex problem. Static mapping techniques for spatial architectures, such as Tartan [30] and SGMF [46], add explicit

control edges, by converting memory dependences to register dependences, between aliasing memory instructions to ensure that dependences are respected. In WaveScalar [42], a dataflow technique, all memory instructions are statically identified by two IDs: the sequence number of the instruction within the wave (trace), and a wave number indicating the wave (trace) invocation. All issued memory instructions from the fabric are reassembled in the memory system and executed in "total load-store order". This method requires new LDST units and is overly conservative. OOO pipelines intelligently break memory dependences using high confidence speculative techniques, such as Store-Sets [9]. Confidence is built by recording alias information during execution. DynaSpAM makes similar use of memory speculation to increase freedom in mapping and executing memory instructions.

Misspeculation Handling All side effects of speculation need to be kept from the architectural state of the host pipelines. Usually output buffers need to be inserted between the host pipeline and the fabrics to hold the live-outs and store values from the spatial fabric for performance [18]. These buffers can also serve as the synchronization points for starting new computation on the host pipeline or fabric, and enforces in-order execution between them. OOO pipelines verify the validity of speculation at the end of the execution by committing the instructions from re-order buffers (ROB) in order. To fully exploit the control and memory dependence speculation of OOO processors and enable true out-of-order execution between the host pipeline and the spatial fabric, the spatial fabric in DynaSpAM fuses its datapaths with the host pipeline more tightly through the ROB.

3. DynaSpAM Framework

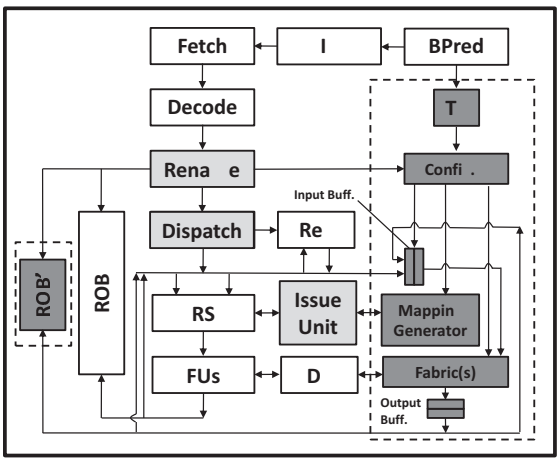

Figure 3: The overall view of the DynaSpAM architecture.

3.1. Overview

This section presents DynaSpAM, a tightly coupled accelerator architecture that applies a novel *resource-aware* mapping to automatically accelerate repeated execution traces from the host OOO pipeline on a spatial fabric for efficient execution. Figure 3 shows the overall architecture. DynaSpAM is designed as an enhancement of high performance OOO processors and does not fundamentally change the structure and execution of the original OOO pipeline.

Trace acceleration in DynaSpAM can be divided into three phases:

Trace Detection The detection phase identifies hot traces for acceleration. *T-Cache*, a trace cache like structure, recognizes recurring instruction sequences across multiple basic blocks. Upon commit of a branch instruction, T-Cache consults an internal history buffer that tracks the previous three branch results[1]. T-Cache then builds an index consisting of the PC of the earliest branch instruction and the three results stored in the buffer and increments a saturation counter using this index. If the counter for this trace is larger than a preset threshold value, a flag in T-Cache for the entry is set to indicate the trace is hot.

Trace Mapping Upon receipt of a branch instruction, the fetch unit retrieves the predictions for the next three branches from the *branch predictor* to build an index into the T-Cache in order to determine if the predicted coming trace is hot. If the trace is hot, the fetch stage grabs instructions until it reaches the fourth branch instruction, as DynaSpAM only tracks three branch instructions, or a preset instruction count limit and marks instructions in the sequence as *trace instructions*. These trace instructions are processed as normal instructions in the fetch, decode, and rename stages. When they arrive at the dispatch stage, the following process occurs :

1. the first trace instruction stops in the dispatch stage until all on-the-fly instructions that have been dispatched to the host OOO pipeline drain through the pipeline back-end;
2. the *mapping generator* adds the resource-aware priority policy to the issue unit. When the issue unit schedules an instruction to an OOO functional unit, it simultaneously maps this instruction to a PE on the fabric, routes operands from the producers and thus generates a *configuration*. Note that no instructions execute on the fabric during this phase;
3. the mapping process finishes when the last trace instruction completes and writes back in the OOO pipeline. The configuration for the spatial architecture is stored in the *configuration cache*. The configuration cache is indexed similarly to the T-Cache, but contains less entries to save space. Any mis-predicted branches or pipeline squash will abort the mapping process.

See Section 4 for a more detailed discussion. With the completion of the trace mapping, the entry in configuration cache is marked as mapped, and a saturation counter for the entry is set to zero and increased if the trace is predicted again by the fetch stage. The counter filters out traces that only appear a few times but could trigger reconfiguration of

[1] As suggested by [45]

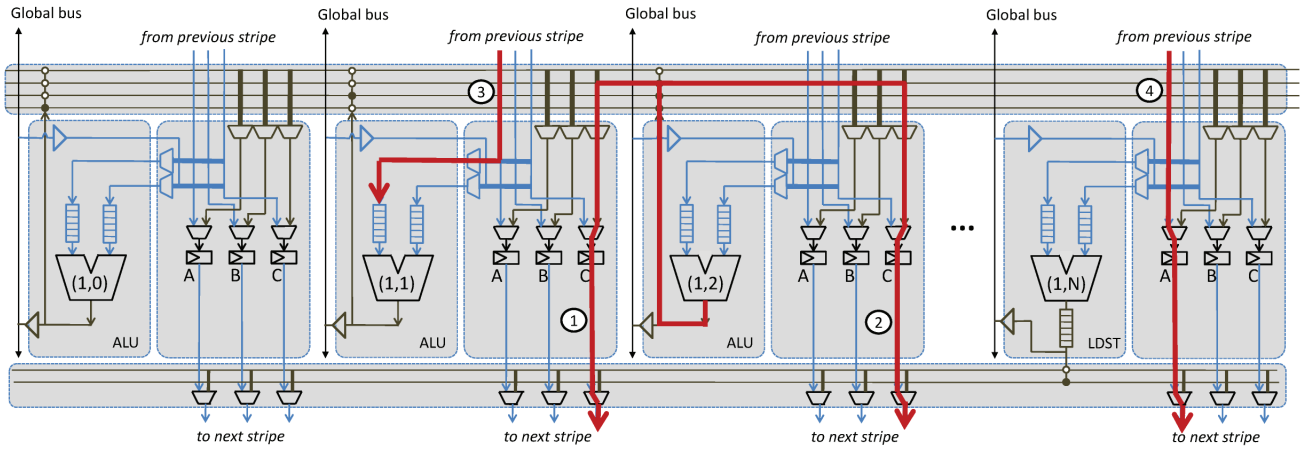

Figure 4: The functional units, pass registers, and interconnect of a stripe of Figure 1. Wires with different colors do not interact with each other. Units A, B and C are pass registers; ALU represents an ALU unit; LDST represents a load/store unit; buffers on ALU input represent the FIFOs of input operands; the output buffer on the LDST unit represents the memory reservation buffer.

the fabric and its associated overhead. Once the saturation counter reaches a threshold value, the entry is marked ready and begins. To prevent frequent reconfiguration, the saturation counters in both the T-Cache and configuration cache are periodically cleared to prevent traces that execute infrequently from occupying the spatial fabric.

Trace Offloading Before offloading begins, the fabric is configured using the mapping result stored in the configuration cache. Concurrently, the rename stage renames the live-ins and live-outs of the trace and sends the ready register values to the fabric. After offloading begins, the fetch unit is directed to the end of the trace and begins normal operation. This allows the host OOO processor to execute concurrently with DynaSpAM's fabric. If the same trace is executed back-to-back, the dependent live-outs from one invocation are forwarded directly using the global bus to the input ports of the fabric for future invocations, which allows for pipelined execution. During execution, the fabric sends live-out values to the ROB and bypass network of the host OOO pipeline.

3.2. Design and Integration of Spatial Fabric

Prior work contains various designs for reconfigurable spatial architectures; however, none of them are designed to dynamically map and offload hot traces from OOO processors. To enable this feature, we both optimize the complex internal interconnect design in existing reconfigurable spatial fabrics for dynamically detected traces and integrate the new hardware structures of DynaSpAM with the existing OOO pipeline.

Acyclically Connected Spatial Fabric To tailor the reconfigurable spatial fabric for dynamically detected traces, DynaSpAM adopts the high level design of the reconfigurable fabric shown in Figure 1. Internally, the DynaSpAM fabric is organized as stripes. Each stripe contains an array of PEs, which are connected by a circuit wired interconnect, as shown in Figure 4. Each PE contains one functional unit, a set of pass registers and multiplexers. Each PE can contain a different type of functional unit, and the set of functional units in a stripe can differ from other stripes. Each functional unit gets its operands from either the pass registers of the previous stripe (for dependences within a trace) or the global bus (for dependences between trace invocations).

Given that the data dependences found within traces are acyclic, the communication allowed within the DynaSpAM fabric is also acyclic. There are three types of connections provided by DynaSpAM: intra-stripe, local inter-stripe and global inter-stripe. Intra-stripe connections (as shown in ① and ② in Figure 4) are completed by the interconnect, and route intermediate results to the next stripes; local inter-stripe connections receive inputs from the previous stripe (such as ③) and can bypass values to the next stripe by storing results in a pass register (as shown in ④). Global inter-stripe connections are used to communicate live-ins and live-outs with the external OOO pipeline and between trace invocations.

Each PE in DynaSpAM can be power-gated independently according to the configuration to save both dynamic and static power consumption during execution. In Section 5, evaluation of DynaSpAM shows that the configuration of PEs are constant for long periods of time, which is sufficient to tolerate the latency between transistor sleep and wake-up across multiple configurations.

Integration into the Host OOO Pipeline In addition to the changes to the issue unit discussed in Section 3.1, the rename, dispatch, and reorder-buffer (ROB) units in the host OOO pipeline require augmentation to enable seamless execution of the program in both the OOO pipeline and the fabric.

The rename unit renames live-in and live-out registers for the trace and reserves entries in input and output FIFOs in the spatial fabric. Separate entries in the FIFOs represent separate invocations of the trace. The oldest entries in the input FIFOs work like reservation station entries to receive live-in values from the OOO pipeline. Live-out FIFOs broadcast live-out

values from the fabric to the OOO pipeline.

An extra index field is added in the main ROB to allow entries to point to a side ROB (ROB′), which contains the renamed live-out values, branch results, and memory stores of a trace invocation. Such an entry can only commit when all live-outs and branch results are obtained from the output FIFOs. This essentially means that the trace is treated as a fat atomic instruction by the host OOO pipeline. The ROB′ commits or squashes (if there is a branch mis-speculation or memory order violation) the entry when it reaches the top.

When the trace is committed or squashed, the ROB′ broadcasts the information to all pipeline stages. The number of live-ins and live-outs that the rename and ROB′ can handle are encoded as constraints in the dynamic mapping phase. If the number of live-ins or live-outs for a trace exceeds either number, a valid mapping cannot be completed.

Intra- and Inter-Trace Memory Ordering DynaSpAM utilizes the aggressive speculative LDST issue techniques in the OOO processors and allows certain LDST instructions from the fabric to be executed out-of-order. To achieve this, DynaSpAM keeps a simplified version of the memory instructions, consisting of only their PC, type, and their relative ordering, in the configuration. When a trace invocation is dispatched, the simplified memory instructions are sent to a memory dependence prediction unit similar to Store-Sets [9], which is used by modern processors to speculatively predict the memory instructions that alias. Memory operations that execute in the fabric consult the unit to determine if they can execute, or if they must stall in order to respect a memory dependence. If the dynamic memory dependence prediction unit mis-speculates and causes a memory violation, the trace is squashed in the ROB and re-executed after updating the offending dependence in the prediction unit.

Since load operations from a LDST unit on the fabric can receive responses out of order, DynaSpAM adds a reservation buffer to each LDST unit to hold all the in-flight loads as shown in the LDST unit in Figure 4.

4. Dynamic Mapping for Spatial Fabric

This section presents a design to enable the dynamic mapping of a trace onto a reconfigurable spatial fabric using *resource − aware* OOO scheduling.

4.1. Resource-Aware Scheduling

Scheduling Frontier The *scheduling frontier* is the set of unallocated PEs that are directly connected to those that have been allocated. Initially, the *scheduling frontier* consists of PEs that can access live-ins directly. At the end of each scheduling step, the *scheduling frontier* moves along the data paths of the allocated PEs. Figure 5 shows the arrangements of PEs in three different spatial fabrics and possible positions of the *scheduling frontier* before a scheduling step. CCA [10] and DynaSpAM fabrics have no cyclic data paths between rows (or stripes), thus their *scheduling frontiers* are straight, while DySER [17] has a complex data path network and its *scheduling frontier* can be irregular.

Algorithm 1: Resource-Aware Scheduling Algorithm

```
   INPUT  : scheduleCycle
   OUTPUT: A schedule of ready instructions to function units: Selected[:]
1  rowIdx ← SchedulingFrontierIdx(scheduleCycle);
2  if rowIdx is Invalid then
3      SCHEDULE_FAIL;
4  FabricPEsVec ← SchedulingFrontierPEs(scheduleCycle);
5  OOOFUsVec ← FabricToOOO(FabricPEVec);
6  ReadyInstsVec ← ReservationStation(scheduleCycle);
7  foreach FU ∈ OOOFUsVec do
8      foreach Inst ∈ ReadyInstsVec do
9          PriorityScore[FU, Inst] ← PriorityGen(FU, Inst, rowIdx);
10 foreach FU ∈ OOOFUsVec do
11     selectedInst =
           PriorityEncoder(PriorityScore[FU, :], HostPriorityPolicy);
12     Selected[FU] = selectedInst;
13     UpdateTables(FU, selectedInst);
```

Scheduling Insights As discussed in Section 2, naïve mapping techniques are not globally resource-aware, and thus do not generate efficient mappings. Building a standalone scheduling unit for dynamic spatial fabric mapping would be prohibitively expensive, thus leveraging the existing microarchitecture of the OOO pipeline is desirable. To support its own scheduling, the OOO reservation station provides the following capabilities:

- *Instruction Buffering* A large instruction window that can easily contain instructions from a large trace;
- *Data Dependence Analysis* Instructions marked as *ready* in the instruction window are known to have their operands available, and are independent of all other ready instructions;
- *Instruction Assignment* An issue unit can select ready instructions for multiple functional units by using priority rules, (referred to as *HostPriorityRule*), such as oldest-first, in its *Priority Encoder*.

One of the key insights of DynaSpAM is that these are the same set of features needed to support dynamic mapping for spatial fabrics, and thus we can equate the placement of trace instructions to PEs in the *scheduling frontier* with the instruction scheduling for the OOO functional units. However, the mapping of instructions to the fabric has additional resource considerations, such as the location of producers, data path availability, and the cost of allocating new paths. These considerations can be represented as a *priority score* that indicates both the feasibility and efficiency of mapping an instruction to a PE on the fabric. If we build a one-to-one mapping between the functional units in the OOO pipeline and the PEs in the *scheduling frontier* (Step A in Figure 5), then selecting an instruction with the highest priority score (Step B in Figure 5) for a functional unit on the OOO pipeline (Step C1 in Figure 5) also maps it to a PE in the *scheduling frontier* (Step C2 in Figure 5). After mapping, the resource information, which is contained in a set of status tables, can

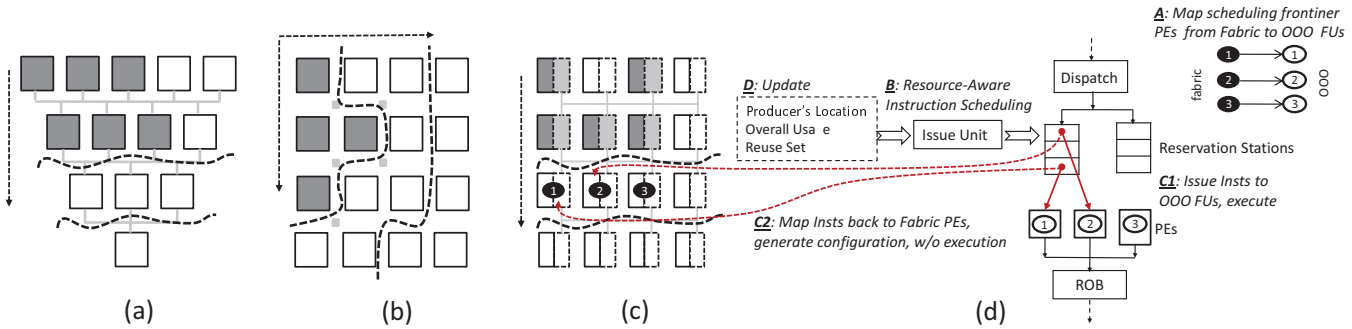

Figure 5: PEs and possible scheduling frontiers in (a) CCA; (b) 4x4 DySER; and (c) DynaSpAM fabric.

be updated (Step D in Figure 5) as discussed in Section 4.2. We call this resource-aware instruction scheduling.

Scheduling Algorithm The high-level Resource-Aware Scheduling Algorithm is presented in Algorithm 1. The input of the algorithm is the current *clock cycle*. First, the *scheduling frontier* is identified as *rowIdx* (Line 1) and the PEs in the frontier are identified as *FabricPEsVec* (Line 4). Since the *scheduling frontier* in DynaSpAM is always aligned to a stripe, the *scheduling frontier* identifier is the stripe index of the fabric that is currently being mapped. Line 5 maps these PEs to the OOO functional units. Line 6 selects the ready instructions from the reservation station, and Lines 11 and 12 use the original select logic to assign instructions to OOO functional units. This reuses the instruction wakeup (data dependence checking) and the instruction select logic in the host OOO pipeline. The new instruction schedule can differ from the schedule generated by the original host priority rule; however, we expect that this kind of priority change does not cause a significant performance change [5].

Special Issues There are two special issues that can occur during scheduling. First, the number of PEs in the current *scheduling frontier* can be greater than that in the OOO pipeline or the issue width of the OOO pipeline, thus some PEs cannot be mapped to in the current scheduling cycle. This problem can be overcome by dividing one scheduling step into multiple cycles. An extra field must be added to each entry of the reservation station to identify the ready instructions that become ready in the middle of a scheduling step, but cannot be issued until the current scheduling step ends. Second, the issue unit must pause if there are OOO functional units that have not finished execution at the start of a scheduling cycle. Otherwise, the *scheduling frontier* could proceed before all PEs in a stripe have been scheduled.

Lines 9 and 13 generate the priority scores for each pair of instruction and functional unit by consulting to a set of status tables (Line 9), and then update these tables after the instructions are scheduled (Line 13).

4.2. Priority Score Generation

Priority Score A *priority score* is a ranking for the placement of an instruction to a PE in the *scheduling frontier*. This priority can be used by the *Priority Encoder* in the issue unit to grant an issue request from the ready instructions [35].

The *priority score* of a fabric can be customizable. DynaSpAM contains five priority scores (shown in Table 2) to represent levels of mapping feasibility and routing score (a higher score means lower routing cost).

Instructions that require two live-ins to be feasibly placed have the highest priority with PEs in the first row, as only these units have two input ports. If the instruction does not require two live-ins, the mapping algorithm prefers to put instructions where they can enjoy more data path reuse, i.e. the producer's value has been routed nearby, with priority levels 0-2 representing the amount of data of available for reuse. Priority level -1 represents a PE that cannot provide enough resources to route its operands. Thus this instruction should not be scheduled to this PE.

The scheduling algorithm is not tied to any particular priority scoring mechanism; the scheduler should use a scoring mechanism that takes into account the resource constraints of the particular spatial architecture that is being mapped to. For example, in CCA [10] data used in one row cannot be reused in the same row, thus there is no routing cost preference. In DySER [17], there are multiple possible data paths that can route the same data for one PE, thus the routing latency should be considered.

Generation *PriorityGen*, a hardware module within the mapping generator, generates priority scores by consulting the current state of mapping, which is stored in three lookup tables: Producer Table (*ProdTable*), Overall Datapath Usage Table (*OverallUsage*), and Datapath Reuse Set Table (*ReuseSet*). The priority generation algorithm is shown in Algorithm 2. For each operand of an incoming instruction the algorithm checks:

1. *ProdTable* to obtain the location of the operands producers (Line 5);
2. *ReuseSet* to determine if the required operand can be obtained directly from the pass registers from the previous stripe, in which case it does not need to add a new route from the producers (Line 9) [2];

[2]In the current implementation, live-in values are not added to the *ReuseSet* and must be acquired from the global bus on each use.

Category	Priority Level	Description
Feasibility	3	Two operands are live-ins thus requiring two input ports.
Routing Score	2	Two operands are not live-ins, and can be provided by *ReuseSet*.
	1	Only one operand can be provided by *ReuseSet*, while the other can be routed.
	0	None of the operands can be provided by *ReuseSet*, but they can be routed.
	-1	One of the operands can not be provided by *ReuseSet* and can not be routed.

Table 2: Priority Scores for different connection status of the producers.

Algorithm 2: **MODULE** PriorityGen

```
   INPUT  : OOOFU, Inst, rowIdx
   OUTPUT: PriorityScore[:,:]
1  FabricPE ← OOOToFabric(OOOFU)
2  canReuse ← 0; canRoute ← 0; needInputs ← 0;
3  foreach op ∈ Inst.ops do
4      livein ← False;
5      ProdLoc ← ProdTable(op);
6      if ProdLoc does not exist then
7          livein ← True;
8          needInputs++;
9      else if op ∈ ReuseSet(FabricPE) then
10         canReuse++;
11     else if OverallUsage(ProdLoc, FabricPE) ≠ ∅ then
12         canRoute++;
13 if needInputs == 2 then
14     if FabricPE can provide two InputPorts then
15         PriorityScore[OOOFU, Inst] ← 3;
16     else
17         PriorityScore[OOOFU, Inst] ← −1;
18 else
19     if Inst.ops_num == canReuse == 2 then
20         PriorityScore[OOOFU, Inst] ← 2;
21     else if Inst.ops_num == canRoute then
22         ScorePriority[OOOFU, Inst] ← 0;
23     else if Inst.ops_num == canReuse + canRoute then
24         PriorityScore[OOOFU, Inst] ← 1;
25     else
26         PriorityScore[OOOFU, Inst] ← −1;
```

Algorithm 3: **MODULE** UpdateTables

```
   INPUT : OOOFU, Inst
   INOUT : ProdTable, ReuseSet, OverallUsage
1  FabricPE ← OOOToFabric(OOOFU);
2  ProdTable(Inst.dest) ← FabricPE;
3  foreach op ∈ Inst.ops do
4      ProdLoc ← ProdTable(op);
5      if ProdLoc exists & op ∉ ReuseSet(FabricPE) then
6          newDatapath ← SELECT OverallUsage(ProdLoc, FabricPE);
7          foreach FU ∈ newDatapath do
8              add op to ReuseSet[FU];
9              OverallUsage(FU, newDatapath) ← USED;
```

3. otherwise, *OverallUsage* to determine if there are available data paths to route the required data (Line 11);
4. otherwise, the instruction can not be assigned to the corresponding location, since there are no enough data path resources to route its operands (Lines 17 and 26).

Lines 19-24 summarize the scores from different operands, and gives corresponding priority scores. The priority scores are stored in a two dimensional table, called *PriorityScore*, for each pair of ready instructions and functional unit.

After one instruction is selected for a functional unit and issued, all the status tables are updated as shown in Algorithm 3.

Additionally, a *Live-Out Table*(LOT) is used to track functional units that produce live-outs and their corresponding architectural registers, and to configure the output ports of the fabric [10]. Upon advancing the *scheduling frontier*, a value is considered a potential live-out if its architectural register is not re-defined within the stripe, and will be automatically routed to the next stripe to increase the probability of reuse. A table called *Last Used Location* is used to track this information. If a potential live-out value is killed, the *Last Used Location* table is consulted, and any routing that was unnecessarily propagated for the killed live-out is removed.

4.3. Mapping Example

Figure 6 demonstrates mapping a short trace onto the Dyna-SpAM fabric and shows the additional hardware logic needed to support the mapping process. In this example, a trace with 9 instructions needs to be mapped to a fabric, as shown in Figure 4. *ProdTable* is a content addressable memory, or CAM, that maps the physical registers to locations on the fabric. *ReuseSet*, contains the physical registers which have values been stored in the pass registers of the previous stripe. It is also a CAM. *OverallUsage* tracks the overall data path usage across the whole fabric and is used to determine if there are enough resources to allocate a new data path to route the data. It can be implemented as a bitmap.

At the start of mapping, the trace instructions have been renamed and placed in the reservation station as per normal program execution. In cycle 0, four instructions are ready within the reservation station, and all status tables are empty. Three of the instructions (0, 1, and 3) generate Priority 0 for all functional units, indicating none of them can reuse the pass registers in the previous stripe to receive their operands. However, instruction 7 requires two input ports and thus has Priority 3 for all functional units. Instruction 0, 1, 7 are selected and placed in the corresponding functional units. Three entries of *ProdTable* are updated using the renamed destination registers of all instructions.

Since there are no available PEs in the *scheduling frontier*, the issue unit moves the *scheduling frontier* forward and begins the placement in cycle 1. In this cycle, instructions 2, 4, 6, and 8 become ready. No instruction can reuse data from the pass registers in the previous stripe and are assigned Priority 0 for all functional units. Thus the original oldest first priority policy selects instruction 2, 3, and 4. The data tables are updated as follows: *ProdTable* adds new destination registers with the producer location; *ReuseSet* records the data in the current pass registers; and *OverallUsage* records the data path usage after this cycle of mapping.

In cycle 2, the final 3 instructions are ready. Instruction

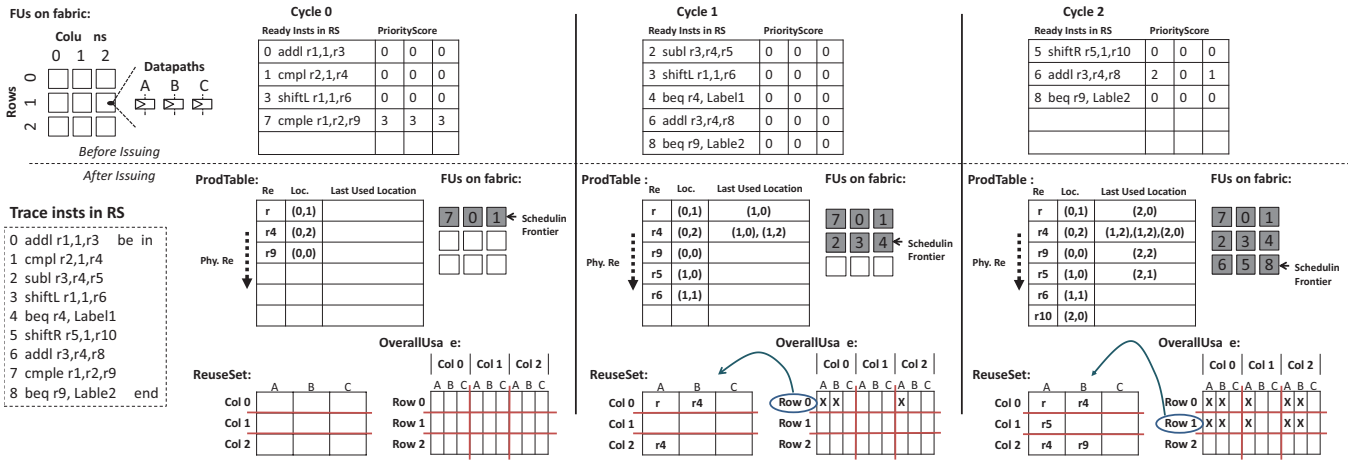

Figure 6: An example illustrates how instruction scheduling is impacted by the location information of the fabric.

5 and 8 have Priority 0 for all functional units since they cannot reuse any available data. Instruction 6 has Priority 2 for functional unit 0 as the pass registers of the previous row hold both of its operand values, r3 and r4. If Instruction 6 is placed there, no new data path routing is required. Mapping ends with instructions 6, 5, 8 being placed respectively.

The final mapping could not be obtained by the naïve method due to its limited instruction windows. For example, if the instructions were placed in program order, Instruction 7 would not be placed in the first row, resulting in an infeasible schedule, and Instruction 6 would not be able to reuse the data path from Instruction 2, resulting in an inefficient schedule.

5. Evaluation

5.1. Methodology

To evaluate the design of DynaSpAM framework, we deploy a comprehensive methodology involving an architectural simulator for performance, power simulators for efficiency, and CAD tools to estimate area.

Benchmarks DynaSpAM is evaluated using eleven programs from the Rodinia benchmark suite [7], representing computationally intensive workloads, to evaluate the system for trace detection, mapping and, acceleration. An overview of these benchmarks is shown in Table 3. We evaluate using the OpenMP version of all benchmarks, with OpenMP pragmas disabled to enable a sequential version. All the benchmarks are compiled with -O3 flag.

Performance Simulation The GEM5 2.0 [2] simulation framework was used for performance evaluation. All IO and initialization phases are skipped in the kernels to capture the main computation. The baseline is an OOO processor with the system configuration shown in Table 4. We implement the DynaSpAM subsystem in conjunction with the baseline OOO pipeline with the same configurations. We measure and compare the kernel performance for each benchmark.

Energy and Area Estimation We used McPAT v1.2 [26] to model the energy of DynaSpAM using execution statistics from the performance simulation. Power for the configuration cache was estimated separately using CACTI [31].

We used functional units from OpenSparc T1 [34] and implemented the datapath for the DynaSpAM fabric in Verilog. The fabric design was synthesized using Synopsys Design Compiler [43] with a 32nm generic cell library to estimate area.

5.2. Experimental Results

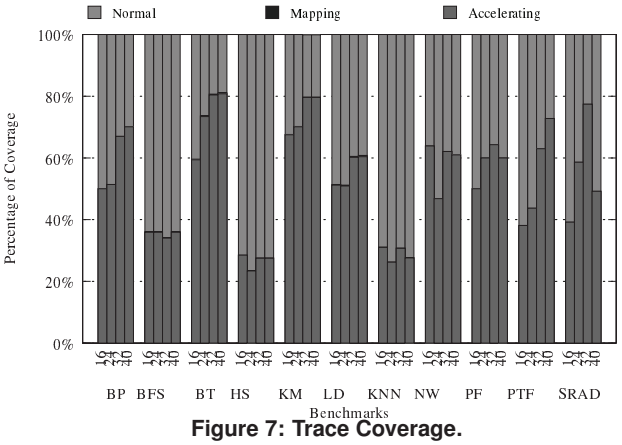

Figure 7: Trace Coverage.

Trace Coverage and Configuration Lifetime Figure 7 shows the percentage of dynamic instructions executed on the host OOO pipeline, the percentage of instructions that execute during the mapping phase, and the percentage that are accelerated and run on the fabric. We evaluate with pre-set trace lengths ranging from 16 to 40 instructions. From the figure, we observe that a small fraction of instructions are executed during the mapping phase for all programs. Generally, traces with longer lengths have higher coverage. However, if a trace contains only a few instructions from a block, it will force more instructions to run on the host pipeline. As

Benchmark Name	Domain	Kernel	Description
Back Propagation (BP)	Pattern Recognition	bpnn_train_kernel	Machine learning algorithm to train the weights of nodes of a layered neural network
Breadth-First Search (BFS)	Graph Algorithms	BFSGraph	Breadth-first search on a graph
B+ Tree (BT)	Search	kernel_cpu	Search in a B+ tree
Hotspot (HS)	Physics Simulation	compute_tran_temp	Estimate processor temperature based on power simulation
Kmeans (KM)	Data Mining	kmeans_clustering	Clustering algorithm for data-mining
LU Decomposition (LD)	Linear Algebra	lud_base	Matrix decomposition
K-Nearest Neighbors (KNN)	Data Mining	main	Finding the k-nearest neighbors from an unstructured data set
Needleman-Wunsch (NW)	Bioinformatics	runTest	Nonlinear global optimization method for DNA sequence alignments
PathFinder (PF)	Grid Traversal	run	Shortest path finder on a 2-D grid using dynamic programming
Particle Filter (PTF)	Medical Imaging	particleFilter	Statistical estimator of the location of a target object given noisy measurements
SRAD (SRAD)	Image Processing	main	Diffusion method for ultrasonic and radar imaging applications based on PDEs

Table 3: Programs tested from the Rodinia Benchmark Suite

Parameter	Setting
Fetch Unit	16-entry return stack; 4K-entry BTB Branch Predictor
Caches	64KB, 2-way, 2-cycle ICache; 64KB, 2-way, 2-cycle L1D; 2MB, 8-way, 20-cycle L2D (64-byte blocks for all caches)
Window Size	192-entry ROB; 256-entry physical RF; 8-wide issue
Execution Units	4 Int ALUs; 1 Int MUL/DIV; 4 Floating ALUs; 1 Floating MUL/DIV; 2 LDST units
Memory Unit	128-entry load queue; 128-entry store queue
Fabric	8-entry buffers; same execution units as OOO per strip; 16 strips; 3 pass regs per FU; 16 Live-in FIFOs, 16 Live-out FIFOs
Config. Cache	16-entry, direct mapped, 16-byte blocks, 3-bits saturation counter, threshold value 4

Table 4: Evaluation system parameters

Benchmark Name	Mapped Traces	Offloaded Traces	Avg. Config. Lifetime (Invocation)		
			1 fabric	2 fabrics	4 fabrics
BP	2	2	6505.5	13013.0	13013.0
BFS	24	10	6.4	8.5	63.9
BT	4	3	197.4	246.8	987.0
HS	11	2	1065.0	2130.0	2130.0
KM	1	1	2750.0	2750.0	2750.0
LD	9	5	81.8	334.4	7690
KNN	4	3	2750.0	2750.0	2750.0
NW	1	1	13276.0	13276.0	13276.0
PF	2	1	6514.0	6514.0	6514.0
PTF	2	2	46.2	9240.0	9240.0
SRAD	1	1	33574.0	33574.0	33574.0

Table 5: Detected Traces and Average Configuration Lifetime

an example, imagine a single block with 33 instructions that executes in a loop. At a trace length of 32, 32/33 instructions execute on the fabric and 1/33 instructions executes on the host OOO pipeline. At 40 instructions, the trace enters a new block and 40/66 instructions execute on the fabric and 26/66 instructions execute on the fabric, thereby reducing coverage. NW with 24 instructions, and SRAD with 40 instructions are examples of this effect at work. Addressing this via more intelligent instruction selection is a goal of future work. We use a trace length of 32 instructions for all the following experiments.

Table 5 shows the number of traces that are detected and mapped successfully (mapped trace), and the number traces that are actually offloaded (offloaded trace). Some of the traces are mapped but never offloaded due to their low frequency of execution. The last three columns of Table 5 show the average configuration lifetime, which starts when the fabric is configured by one trace and ends when the fabric is reconfigured by another trace. From Table 5, we find that the average configuration lifetime is above 40 trace invocations with 1 on-chip fabric for all programs except BFS, which has only 6.4 invocations per configuration. Investigating BFS reveals that there are many unbiased control branches within the loop. Multiple fabrics can be used to reduce reconfiguration times and increase efficiency. We modeled architectures with 2 and 4 fabrics and use a least-recently-used (LRU) policy to manage reconfiguration. The experiment results show that with 4 fabrics, BFS's average configuration life time is 64 invocations, and reaches 2045 invocations with 8 fabrics (not shown in the table).

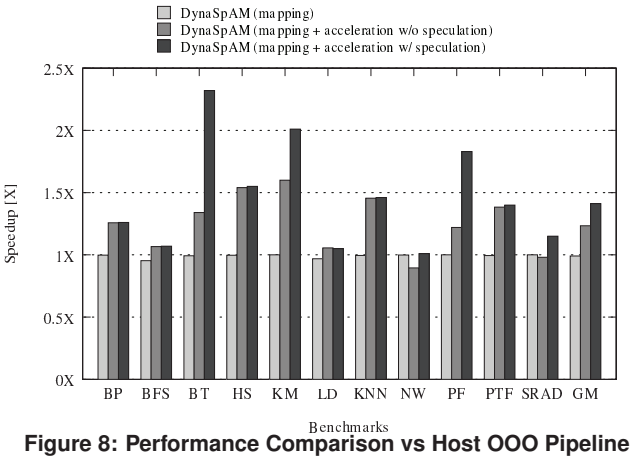

Figure 8: Performance Comparison vs Host OOO Pipeline

Performance We compared the performance of DynaSpAM with three different configurations to the baseline OOO pipeline. DynaSpAM with "mapping" only maps the detected traces but does not offload them to the fabric. DynaSpAM with "mapping + acceleration w/ speculation" both maps and offloads the traces to the fabric while using memory speculation. DynaSpAM with "mapping + acceleration w/o speculation" maps and offloads the detected traces while conservatively preserving all load-store and store-store orderings.

Recall that mapping overhead comes from two sources: 1) time draining the pipeline backend when the trace mapping starts; and 2) the cost of pausing instruction issue for long latency functional units during mapping. The simulation results show that the overhead of mapping is small, and causes less than 3% slowdown for all the benchmarks.

Without memory speculation, DynaSpAM produces a 1.23× geomean performance and causes slowdown in two programs, NW and SRAD, large fraction of dynamic memory instructions. With memory speculation enabled, DynaSpAM produces a 1.42× geomean performance improvement without

Module names	Area($\mu\ m^2$)	Module names	Area($\mu\ m^2$)
sparc_exu_alu	4660	fpu_add	34370
sparc_mul_top	47752	fpu_mul	62488
sparc_exu_div	11227	fpu_div	13769
data_path	4717	fifo	848

Table 6: Area Comparison for different components

ever causing program slowdown. This shows the importance of effectively using memory speculation.

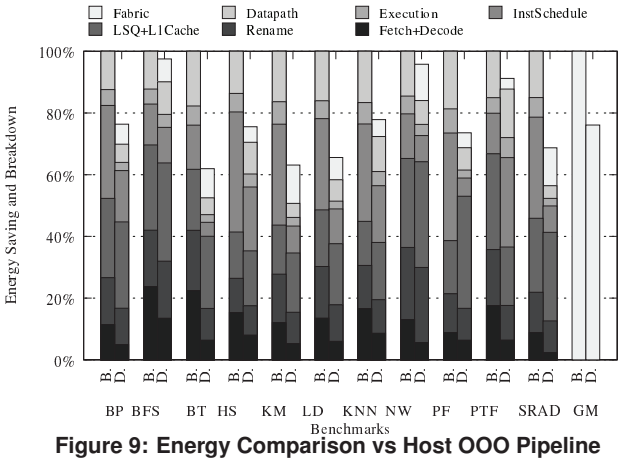

Figure 9: Energy Comparison vs Host OOO Pipeline

Energy We measured the energy consumption in the simulation of both the baseline processor and DynaSpAM. Figure 9 demonstrates the energy consumption of different components to show the energy increase/decrease in each component. The overall energy consumption is reduced by 2.5%-36.86%, with geomean 23.9%. For each benchmark, it is clear that the energy consumption from Fetch, Rename, Instruction Scheduling (InstSchedule), and the bypass networks (Datapath) are reduced. On the other hand, power consumption for the memory system is increased, since DynaSpAM cannot reduce memory activity. The energy consumption of the fabric includes both the functional units and datapaths, which is greater than the energy consumed by Execution on the OOO pipeline but smaller than the sum of Execution, Datapath, and InstSchedule.

Area To match the frequency of the OOO pipeline, we used functional units from an industry grade design, OpenSparc T1, to build the fabric. The datapath and FIFO buffers are designed separately. Table 6 shows that the size of each datapath block, containing pass registers and multiplexers and the area of FIFOs, compared to modules from OpenSparc T1. It is shown that datapath block is almost as large as an OpenSparc T1 integer ALU, and that the area of FIFOs are much smaller. The overall fabric size is $2.9mm^2$ with 8 stripes (A 2-core AMD Bulldozer is $30.0mm^2$ (including cache) at this technology node). The area of the configuration cache is obtained from CACTI, and the number is $0.003mm^2$.

Work on DynaSpAM has primarily focused on the feasibility and applicability of dynamic mapping, and has not focused on optimizing area usage. In future work, research will be done to adjust the number of functional units according to instruction type distributions of the benchmarks.

6. Related Work

Research into spatial architecture has been an active area for quite a long time, and different techniques have been proposed. Table 7 summarizes the differences between DynaSpAM and prior work.

Programmable Functional Units Programmable functional units [3, 6, 12, 17, 18, 37, 47, 48] such as OneChip, Chimaera and PRISC only consider short program traces or subgraphs, and usually do not include memory operations. In these techniques, only energy consumed by communicating intermediate results within the trace can be reduced. BERET [18] classifies a set of common subgraph patterns for the superblocks in general purpose programs, and builds corresponding specialized hardware modules for each pattern. These works employ compiler techniques to extract and map subgraphs to the special functional units. CCA [10] requires static subgraph extraction, but performs dynamic mapping.

Reconfigurable Spatial Co-Processors Another group of designs target larger instruction sequences. Garp adapted the VLIW compilation technique to generate pipelined datapaths on a fine-grained reconfigurable fabric [20]. ADRES [28] applies the same technique, but on a coarse-grained reconfigurable fabric with regular local connections between functional units. DIM [1] performs dynamic mapping, but like CCA, the mapping is naïve and in program order. Tartan [4, 30] compiles entire programs onto spatially connected functional units, which operate completely asynchronously. Elastic CGRAs [21] uses a similar design but focuses more on gate-level implementations. The SGMF architecture [46] supports dynamic spatial dataflow execution and uses buffers in front of each functional unit to execute multiple invocations simultaneously. These techniques all require a static compilation to map instructions to the fabric, and their control edges for memory operations are conservative. DynaSpAM dynamically maps detected hot traces, using memory speculation, from the OOO execution without compiling.

DynaSpAM's fabric design is similar to the fabric of PipeRench [16] in terms of the stripe organization and communication channels. The major difference is that our design is dataflow-based and is tailored the interconnect to match the shape of trace by reducing the connections between functional units in the same stripe, while PipeRench is not dataflow based and contains dense connections.

Spatial General-Purpose Processors RAW [44] supports both ILP and streaming instructions by routing operands between architecturally-exposed functional units over a point-to-point scalar operand network. TRIPS [40] and its successors such as TFlex [25] and T3 [38] use a compiler to find hyperblocks, and schedule each hyperblock in functional units individually. WaveScalar [42] uses a similar pipelined model

Reconfigurable Execution Engine	Compiler Effort		Hardware Feature					Target Instruction Range
	Placement Routing Not Required	Binary Compatible	Dynamic Mapping	Resource-aware Scheduling	Pipeline Execution	Dataflow	\w MEM	
PRISC [37], Chimaera [48]	×	×	×	×	×	×	×	Subgraph
DySER [17]	×	×	×	×	✓	✓	×	Subgraph
ADRES [28],PipeRench [16]	×	×	×	×	✓	✓	✓	Kernel
BERET [18]	✓	×	×	×	✓	✓	✓	Subgraph
SGMF [46]	×	×	×	×	✓	✓	✓	Kernel
Tartan [30], WaveScalar[42]	×	×	×	×	✓	✓	✓	Whole Program
CCA [10, 11]	✓	✓	✓	×	×	×	× [3]	Subgraph
DynaSpAM	✓	✓	✓	✓	✓	✓	✓	Kernel

Table 7: Comparison between DynaSpAM and other in-core reconfigurable computation engine

as our work to execute "waves," which are control flow graphs. It requires a new ISA to encode the global sequence of memory operations, which allows for dynamic reassembly to preserve program order.

Dynamic Trace Detection and Execution In addition to dynamic trace construction with trace cache [15, 39], many techniques optimize dynamically formed traces for high efficient backends. DIF [14, 32] dynamically compacts retired instructions for repeated execution on a VLIW engine. HBA [13] and Yoga [45] select only hot traces and build VLIW/In-Order instruction streams for the retired instruction. CCA [10] dynamically maps instruction streams to spatial functional units with consideration given to placement but not resources. None of these techniques actively generate mappings during instruction scheduling as in DynaSpAM. I-COP [8] builds a standalone coprocessor to complete binary optimization for incoming instruction streams, but our work leverages existing micro-architecture features in OOO pipeline.

7. Conclusions

Reconfigurable spatial architectures are more efficient than OOO processors. Static mapping methods rely heavily on profiling and compiler techniques to explore instruction and loop level parallelism. Dynamic methods can overcome the disadvantages of static methods, such as the inability to adaptive to different workloads and lack of compatibility; however, they have not had access to large enough instruction scope to generate efficient mappings. We have presented DynaSpAM, a framework to dynamically detect, map and accelerate hot instruction sequences from an OOO pipeline on a spatial fabric. Specifically, DynaSpAM leverages existing features from the OOO pipeline and actively guides instruction issue to generate efficient mappings for the fabric. This new method is both low-cost and efficient. Experimental results, a geomean 1.42× speedup with 23.9% energy consumption reduction for 11 benchmarks from Rodinia Benchmark Suite, demonstrate the potential gains from symbiotic combination of an OOO pipeline and spatial dataflow architecture.

[3] Note that CCA traces can contains stores and loads to spilled values which can be eliminated during the mapping. But generally the accelerator does not contain memory ports.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. We thank Yungang Bao for his comments and feedback during this work. We also thank the anonymous reviewers for their insightful comments. This research was funded in part by National Science Foundation grants 1047879 and 1439085 and by DARPA contracts FA8750-10-2-0253 and HR0011-13-C-0005. All opinions, findings, conclusions, and recommendations expressed throughout this work are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] A. Beck, M. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Design, Automation and Test in Europe*, March 2008, pp. 1208–1213.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[3] A. Bracy, P. Prahlad, and A. Roth, "Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth," in *In Proc. of the 37th Annual International Symposium on Microarchitecture*, 2004, pp. 18–29.

[4] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 14–26.

[5] M. Butler and Y. Patt, "An investigation of the performance of various dynamic scheduling techniques," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992, pp. 1–9.

[6] J. E. Carrillo and P. Chow, "The effect of reconfigurable units in superscalar processors," in *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, ser. FPGA '01, 2001, pp. 141–150.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[8] Y. Chou and J. P. Shen, "Instruction path coprocessors," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 270–281.

[9] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th annual international symposium on Computer architecture*. IEEE Computer Society, 1998, pp. 142–153.

[10] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An architecture framework for transparent instruction set customization in embedded processors," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05, 2005, pp. 272–283.

[11] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08, 2008, pp. 389–400.

[12] C. Ebeling, D. Cronquist, P. Franklin, J. Secosky, and S. Berg, "Mapping applications to the rapid configurable architecture," in *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, Apr 1997, pp. 106–115.

[13] C. Fallin, C. Wilkerson, and O. Mutlu, "The heterogeneous block architecture," SAFARI Group, Department of Electrical and Computer Engineering, Carnegie Melon University, Tech. Rep., 2014.

[14] M. Franklin and M. Smotherman, "A fill-unit approach to multiple instruction issue," in *in Proceedings of the 27th Annual International Symposium on Microarchitecture*, Nov 1994, pp. 162–171.

[15] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors," in *Proceedings 31th Annual IEEE/ACM International Symposium on Microarchitecture*, December 1998, pp. 173–181.

[16] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "Piperench: A co/processor for streaming multimedia acceleration," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999, pp. 28–39.

[17] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *International Symposium on High Performance Computer Architecture*, 2011, pp. 503–514.

[18] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *International Symposium on Microarchitecture*, 2011, pp. 12–23.

[19] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of ISCA*, 2010.

[20] J. Hauser and J. Wawrzynek, "Garp: a mips processor with a reconfigurable coprocessor," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997, pp. 12–21.

[21] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic cgras," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2013, pp. 171–180.

[22] Z. Huang and S. Malik, "Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2001, pp. 735–.

[23] Z. Huang and M. Sharad, "Exploiting operation level parallelism through dynamically reconfigurable datapaths," in *Proceeding of 39th Design Automation Conference*, 2002, pp. 337–342.

[24] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, 2003, pp. 93–.

[25] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," in *IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 381–394.

[26] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture.*, Dec 2009, pp. 469–480.

[27] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?" in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 241–252.

[28] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix." in *Proceedings of the Conference on Field Programmable Logic*, vol. 2778, 2003, pp. 61–70.

[29] E. Mirsky and A. DeHon, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, Apr 1996, pp. 157–166.

[30] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: Evaluating spatial computation for whole program execution," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 163–174.

[31] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," HP Laboratories, Tech. Rep., 2009.

[32] R. Nair and M. E. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 13–25.

[33] 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 *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013, pp. 495–506.

[34] Oracle, "Opensparc t1 microprocessor," http://www.oracle.com/technetwork/systems/opensparc/index.html.

[35] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997, pp. 206–218.

[36] M. Quax, J. Huisken, and J. van Meerbergen, "A scalable implementation of a reconfigurable wcdma rake receiver," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3*, 2004, pp. 30 230–.

[37] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27, 1994, pp. 172–180.

[38] B. Robatmili, D. Li, H. Esmaeilzadeh, S. Govindan, A. Smith, A. Putnam, D. Burger, and S. W. Keckler, "How to implement effective prediction and forwarding for fusable dynamic multicore architectures," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 460–471.

[39] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 24–34.

[40] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *International Symposium on Computer Architecture*, 2003, pp. 422–433.

[41] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on*, vol. 49, no. 5, pp. 465–481, May 2000.

[42] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 291–303.

[43] Synopsys, "Synopsys Design Compiler," http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler.

[44] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams," in *International Symposium on Computer Architecture*, 2004, pp. 2–12.

[45] C. Villavieja, J. A. Joao, R. Miftakhutdinov, and Y. N. Patt, "Yoga: A hybrid dynamic vliw/ooo processor," High Performance Systems Group, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, Texas 78212-0240, USA, Tech. Rep., 2014.

[46] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for gpgpus," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, 2014, pp. 205–216.

[47] R. Wittig and P. Chow, "Onechip: an fpga processor with reconfigurable logic," in *IEEE Symposium on FPGAs for Custom Computing Machines*, Apr 1996, pp. 126–135.

[48] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "Chimaera: A high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 225–235.

[49] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," in *2013 IEEE International 3D Systems Integration Conference (3DIC)*, Oct 2013, pp. 1–7.

[50] Q. Zhu, K. Vaidyanathan, O. Shacham, M. Horowitz, L. Pileggi, and F. Franchetti, "Design automation framework for application-specific logic-in-memory blocks," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, July 2012, pp. 125–132.