Appears in the 12^{th} International Conference on Architectural Support for Programming Languages and Operating Systems

A Spatial Path Scheduling Algorithm for EDGE Architectures [*]

Katherine E. Coons Xia Chen Sundeep K. Kushwaha[†] Doug Burger Kathryn S. McKinley

Department of Computer Sciences, The University of Texas at Austin
[†] Qualcomm, Austin, Texas

Abstract

Growing on-chip wire delays are motivating architectural features that expose on-chip communication to the compiler. EDGE architectures are one example of communication-exposed microarchitectures in which the compiler forms dataflow graphs that specify how the microarchitecture maps instructions onto a distributed execution substrate. This paper describes a compiler scheduling algorithm called *spatial path scheduling* that factors in previously fixed locations - called anchor points - for each placement. This algorithm extends easily to different spatial topologies. We augment this basic algorithm with three heuristics: (1) local and global ALU and network link contention modeling, (2) global critical path estimates, and (3) dependence chain path reservation. We use simulated annealing to explore possible performance improvements and to motivate the augmented heuristics and their weighting functions. We show that the spatial path scheduling algorithm augmented with these three heuristics achieves a 21% average performance improvement over the best prior algorithm and comes within an average of 5% of the annealed performance for our benchmarks.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—optimization, compilers

General Terms Algorithm, Performance

Keywords Instruction scheduling, path scheduling, simulated annealing, EDGE architecture

1. Introduction

Growing on-chip wire delays will make communication a growing and significant factor in future microprocessor design. The recent decline of frequency scaling implies that most future performance gains will come from increased exploitation of concurrency. These two trends work in opposition, however, because it is becoming more difficult to exploit concurrency as communication overheads increase. One approach researchers are pursuing is the explicit or implicit exposure of on-chip communication to the software. The most popular current approach, chip multiprocessors (CMPs), requires the programmer or compiler to explicitly specify concurrency and either the communication or synchronization.

Alternatives include Explicit Dataflow Graph Execution (EDGE) architectures, which aim to exploit fine-grained concurrency within a single thread. EDGE architectures break a program into a sequence of multi-instruction blocks that must each commit atomically. Within each block, the ISA explicitly encodes instructions' dependences in a statically constructed dataflow graph and their execution placement in a distributed substrate. This encoding enables out-of-order execution with lower per-instruction energy overheads, as no renaming, associative issue, or multi-ported register files are required to execute instructions within a block.

A key resultant challenge is how to map EDGE dataflow graphs onto a hardware substrate to minimize the effects of communication latencies while taking advantage of the available concurrency. In the TRIPS prototype EDGE architecture, the compiler assigns instruction numbers that determine placement on the ALU substrate. The TRIPS prototype microarchitecture contains a four-by-four array of arithmetic units, each one holding up to eight instructions from a 128-instruction block. The microarchitecture places each instruction according to its statically assigned number within the block (ranging from 0 to 127). When assigning these numbers, the scheduler attempts to balance communication, by placing dependent instructions in proximity, and concurrency, by placing independent instructions on different functional units.

To exploit instruction-level parallelism, the TRIPS microarchitecture implements out-of-order execution. By assigning IDs to instructions, the TRIPS scheduler *statically places* (SP) each instruction on the array of ALUs, and the hardware *dynamically issues* (DI) instructions when their operands are ready. SPDI differs from the VLIW approach, which uses static placement and static issue, and the out-of-order superscalar approach, which uses dynamic placement and dynamic issue. The schedulers in this paper place instructions, but do not determine issue order. The SPDI execution model creates challenges for the scheduler since it must statically estimate dynamic resource conflicts and critical paths.

This paper describes an algorithm called *spatial path scheduling* (SPS) that reasons explicitly about path routing distances when mapping a dataflow graph to the ALU topology. The algorithm also exploits the fact that some locations are known or partially known even before the first instruction is in place. For example, a chain of dependent instructions must follow a path on the chip that is in part determined by the physical locations of the registers and cache banks. Figure 1 shows these initial locations in the TRIPS prototype microarchitecture, which has four register banks above the top row of the four-by-four ALU array, and a column of four

[*] This work is supported by DARPA F33615-03-C-4106, NSF EIA-0303609, NSF ITR CCR-0085792, NSF CCR-0311829, DARPA NBCH30390004, NSF CCF-0429859 and IBM. Any opinions, findings and conclusions are the authors' and do not necessarily reflect those of the sponsors.

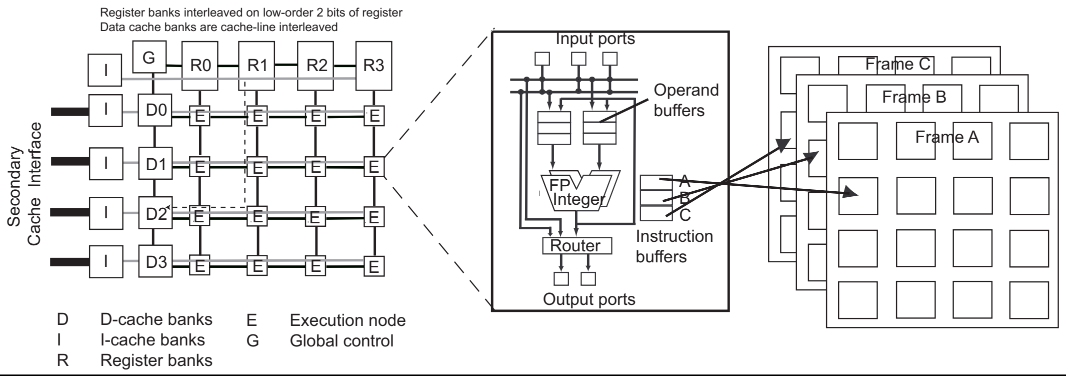

Figure 1. 4×4 TRIPS Processor

data cache banks to the left of the array. For example, a set of dependent instructions that reads a register in bank 1, computes an intermediate value, and stores that value in data cache bank 2, must traverse at least 5 operand network links, as shown by the dotted path in Figure 1.

The basic SPS algorithm computes criticality based on routing distances using all known *anchor points*, i.e., fixed positions for operations in the block, such as register accesses. We augment this basic SPS algorithm with heuristics to model contention on the ALUs and network links, estimate inter-block (global) critical paths, and provide lookahead for planning path routes based on the number and location of instructions on the path.

We compare SPS with a previously published greedy list scheduling algorithm for TRIPS (GRST) [17] using a cycle-accurate, validated simulator with hand-optimized kernels drawn from SPEC2000, EEMBC, Livermore Loops, MediaBench, and C libraries. The basic SPS algorithm improves performance by 14% on average and up to 46% over GRST, the previous best algorithm.

Unfortunately, finding ideal schedules for comparisons is computationally intractable. We thus use simulated annealing [12] to approximate ideal schedules. Because our evaluation function is full simulation of the program, which is prohibitively time-consuming if unconstrained, we show how to further prune the annealed search space. We use information from the annealed schedules to motivate and weight heuristics that we add to SPS. The resulting algorithm improves performance by 7% over the basic SPS algorithm and 21% over GRST. This scheduler is on average within 5% of the annealed schedules.

By exploiting anchor points, SPS easily generalizes to many different topologies. One simply provides the microarchitecture and topology in an abstract form; i.e., location, number, and spatial relationship of microarchitectural resources such as PEs, caches, and register files. While this paper demonstrates SPS's effectiveness for the TRIPS ISA and microarchitecture, we believe it is applicable to schedulers for other partitioned architectures such as WaveScalar [23], and may be useful for clustered VLIWs and RAW [14, 25].

2. Background

This section explains the basic TRIPS architecture, the instruction scheduling problem, and the previous best scheduler.

2.1 TRIPS Architecture and Scheduling Problem

Figure 1 shows a TRIPS microarchitecture, instantiated as a 4×4 array of ALUs. Each ALU has eight issue slots per block, for a maximum of 128 instructions per block. The processor maps each block onto the substrate, executes it as a unit, and unmaps the block after it produces all of its outputs (up to 32 stores, up to 32 register writes, and a single branch decision). The register reads and writes are not included in the 128-instruction count, but are part of the block header. Mapping a block onto the array has a fixed per-block cost, and thus the blocks must be mostly full of useful instructions to maximize performance. The microarchitecture supports speculation via next-block prediction, allowing up to eight blocks in flight. Other work describes the architecture in more detail [4] and describes how the compiler produces correct blocks that meet the maximum number of instructions, maximum number of dynamic load/stores, and register banking constraints [22].

The instruction scheduler maps a dataflow graph that encodes the dependences among instructions onto 128 *slots* that represent the (4×4) ALU array and eight reservation stations per ALU. (ALUs have a total of 64 reservation stations; eight instructions for each of eight potentially in-flight blocks.) The scheduler encodes the instructions in *target* form, which specifies the physical location and operand position of each consuming instruction in the producing instruction. The 64-entry reservation station at each ALU is out-of-order and issues at most one instruction per cycle, selecting the oldest *ready* instruction. An instruction is ready after all of its operands arrive. Instructions that produce values internal to a block need not access memory or registers. Each hop between adjacent ALUs, register banks, and cache banks adds one cycle to the routing latency, and the deterministic Y-X routing function routes north/south to the correct row and then east/west to the correct column. As with a conventional architecture, the registers' names exactly specify their physical location in a bank. When a new block is mapped to the array, the microarchitecture injects all register reads into the array, routing their values to the dependent instructions in the block. The microarchitecture routes operands for register writes to the appropriate register bank when the value is produced.

2.2 Greedy scheduling for TRIPS

A static local scheduling algorithm considers the instructions $\mathcal{I}$ in a block, which may be a basic block or a predicated hyperblock that obeys the TRIPS architectural constraints. It builds a directed acyclic dataflow graph (DFG) that describes the dependences among instructions. The scheduling problem for TRIPS differs from the scheduling problem for static issue architectures because it is legal to place any of a block's 128 instructions into any of the 128 positions in a block. The goal of the scheduler, however, is to minimize the completion time of the block by exploiting instruction level parallelism, minimizing static routing latency between pairs of dependent instructions, and minimizing dynamic latencies, such as contention, whenever possible.

Figure 2 shows a *greedy* list scheduling algorithm for TRIPS (GRST) [17]. GRST uses a ready set consisting of instructions

```
Input: $\mathcal{I}$ - instructions in a block, $\mathcal{G}$ - array of ALUs
output: $\mathcal{A}$ - a mapping $\mathcal{I} \to \mathcal{G}$
1:  $S$ += $\{j\}$, $\forall\ j \in \mathcal{I}$ with no or only register inputs (ready set)
2:  $S$ = top_down_criticality_sort($S$)
3:  for $i$ = most critical instruction in sorted list $S$ do
4:      bestCost = 0; bestSlot = none
5:      issueSlots = find_legal_instruction_slots($i$)
6:      for all slot in issueSlots do
7:          issue($i$, slot) = ready($i$, slot) + ALU Contention($i$, slot)
8:          pCost($i$, slot) = complete($i$, slot) +
                                  lookahead($i$, slot) * 0.5
9:          bestCost = min(bestCost, pCost($i$, slot))
10:         bestSlot = node for bestCost
11:     end for
12:     $\mathcal{A}$ += schedule($i$, bestSlot)
13:     $\mathcal{I} = \mathcal{I}$ - $\{i\}$
14:     $S$ += $\{j\}$, $\forall j \in \mathcal{I}$ with all parents $\in \mathcal{A}$
15:     $S$ = top_down_criticality_sort($\mathcal{S}$)
16: end for
```

Figure 2. GRST: The Greedy List Scheduling Algorithm

whose inputs have already been scheduled. It initializes the ready set to instructions with no inputs (i.e., constant-generating instructions) or with only register inputs. Like a VLIW list scheduler, it sorts the instructions in a top-down fashion, putting instructions with the smallest depth in the DFG into the ready set first. It prioritizes instructions to schedule based on their depth from the root instructions in the DFG, and then based on their height from the leaf instructions in the DFG. The depth and height calculations include delays for multicycle instructions and communication latencies, such as those to the register and cache banks.

To schedule instruction i, GRST computes the unscheduled issue slots. For each of these slots, it estimates the *ready* time for i on an ALU by determining when i's operands will be available (line 7). The ready time includes the routing delay of the operand(s) to the ALU from other ALUs. Previous work [17] augments GRST with heuristics to guide its placement decisions. We use this augmented version as a baseline, and refer to it as GRST. The following heuristics guide GRST placement decisions:

Critical path ordering (C) and reordering (R): GRST prioritizes instructions along the critical path first by sorting the instructions based on the maximum depth of any of their descendents in the DFG and then by their height in the DFG. The scheduler recomputes the critical path after each instruction placement (this step includes inter-ALU latency costs between scheduled instructions) and re-prioritizes the instructions in the ready set.

Load balancing (B): GRST estimates *completion time* based on its estimates of the issue and completion time of other instructions on the same ALU (lines 7 & 8). It places an instruction in the slot that minimizes its completion time, avoiding scheduling independent instructions that may issue at the same time on the same ALU.

Data cache locality (L): GRST assumes loads and stores hit in the L1 data cache and are equally likely to go to any cache bank. To reduce latency and contention, it places load instructions and their consumers close to the data cache banks by inserting a non-executable pseudo-instruction between the load and its consumers in the DFG. It places the pseudo-instruction in the cache and models a one-hop latency to it from the left side of the ALU array.

Register output (O): To place instructions that produce register outputs close to the register file, GRST prevents other instructions from occupying these slots by penalizing instructions that do not lead to register-producing instructions.

The register output heuristic attempts to place instructions that produce register outputs along a balanced path that ends at the register file. It uses a fairly *ad hoc* weighting function to balance paths for register outputs, by adding the following *lookahead* penalty (line 8) to an instruction's placement cost:

$$\begin{aligned} lookahead &= rowDistance/graphDistance \\ &\quad + graphDistance/rowDistance \end{aligned}$$

where *rowDistance* is the number of rows from the register bank (minimum 1), and *graphDistance* is the number of instructions in the DFG between this instruction and a write instruction. This function is minimized when *graphDistance* equals *rowDistance*, which pushes instructions that produce register outputs as far from the register file as their dataflow distance from the register write. This optimization improves performance because placing register output instructions near the register banks is important. However, it is insufficient for a wider array of spatial constraints.

Unfortunately, in the actual TRIPS implementation there are more constraints on the scheduler. For example, the TRIPS prototype breaks up the centralized register file and data cache modeled in the previous simulation study [17] into four discrete banks, as shown in Figure 1. Prior work showed that the GRST algorithm achieved close to ideal, communication-free performance [17]. This earlier study was limited because it used binary rewriting of Itanium binaries, modeled the microarchitecture at a higher level than the actual implementation, and did not model all costs, including partitioned register bank access. The increased simulation detail puts more pressure on the scheduler, creating a renewed gap between an ideal schedule and the output of GRST. Closing this gap and finding a general algorithm to handle an arbitrary set of spatial constraints were key motivations for the SPS algorithm.

3. Simulated Annealing

Computing optimal schedules to understand the performance potential for scheduling is unfortunately NP-complete. Even for a single full block, exhaustive evaluation requires 128! possible schedules and is impractical. Worse, an optimal schedule would consider global information, since multiple in-flight blocks may interfere with one another, for a total search space of $128!^b$ combinations, where b is the number of blocks in the program. Simulated annealing provides a method for finding good solutions in a large search space in tractable time.

Simulated annealing is a probabilistic method that approximates a global optimum in a large search space by searching for good solutions, but occasionally using worse alternatives to avoid becoming trapped in local minima. The problem must be expressed in terms of a set of possible states, an objective function E(s) that evaluates the *energy* of a particular state s, a transition function that moves from the current state s with energy e to a neighbor state s' with energy e', an *annealing planner*[1] that decreases a global time-varying parameter, T, and a function P(T, e, e') that represents the probability of transitioning from s to s'.

3.1 Instruction scheduling with simulated annealing

Since any static cost function is imperfectly correlated with performance, we use a software simulator for the objective function E(s). The set of possible states consists of all legal mappings of instructions to physical locations. A transition from the current state

[1] The simulated annealing literature calls the planner the annealing scheduler, but that term is overloaded here.

Input: initial placement of instructions
output: the best placement of instructions by simulated annealing

```
1:  $T$ = $init\_temperature$
2:  while temperature is higher than freeze temperature do
3:      while equilibrium not reached do
4:          Select a critical block
5:          Select latency type proportionately based on the distribution of delay types in the chosen block
6:          if $latency\_type$ == $alu\_contention$ then
7:              Move one instruction away from the node or exchange with another instruction on another node
8:          else if $latency\_type$ == $network\_contention$ then
9:              Move one instruction whose operand passes this link to a different but nearby location
10:         else if $latency\_type$ == $route\_delay$ then
11:             Find the instruction causing the routing delay
12:             Move it closer to its children
13:         end if
14:         $cost$ = Compute_cost_use_simulator($placement$)
15:         if $cost < last\_cost$ then
16:             Accept the placement
17:             if $cost < best\_cost$ then
18:                 Save_best_placement($placement$)
19:             end if
20:         else
21:             Accept or reject based on probability equation of temperature
22:         end if
23:     end while
24:     $T = T * cooling\_rate$
25:     $placement$ = Restore_best_placement()
26: end while
```

Figure 3. Guided Simulated Annealing

to a neighbor state involves swapping the locations of pairs of instructions or moving instructions to empty slots. The number of instructions the annealer statistically favors moving at each iteration is proportional to the temperature. After the annealer swaps or moves instructions to form a new schedule, it measures the cycle count of the application with the new schedule. If the result beats the previous schedule, the annealer accepts the new schedule. If the result is worse, the annealer accepts it based on the following probabilistic function:

$$rand < e^{\frac{-(newCost - oldCost)*Constant}{oldCost*temp}}$$

where *rand* is a random number between 0 and 1, *newCost* and *oldCost* are the cycle times of the new and current schedules, respectively, and *temp* is the temperature, which is scaled by the *cooling rate* each time the system reaches equilibrium. The cooling rate determines how the temperature decreases as annealing proceeds. We use a constant factor.

3.2 Guided simulated annealing

While simulated annealing is designed to prune large search spaces, our lengthy evaluation function makes it too time consuming (days to months) to search the space blindly. We therefore move only instructions that contribute cycles to the program's critical path. To find the critical path, we use a tool called `tsim-critical` [18] that implements Fields et al.'s critical path modeling methodology [7]. This tool captures a dynamic event trace from the processor simulator, builds the program dependence graph, and emits critical path information such as the number of cycles each instruction and block contribute to the critical path through the program. It also divides these contributions into different delay types: ALU contention, link contention, and operand network routing delay. We use this information to combine random and guided transitions, as illustrated in Figure 3. For all results, we start with the best compiled schedule and low to moderate temperatures.

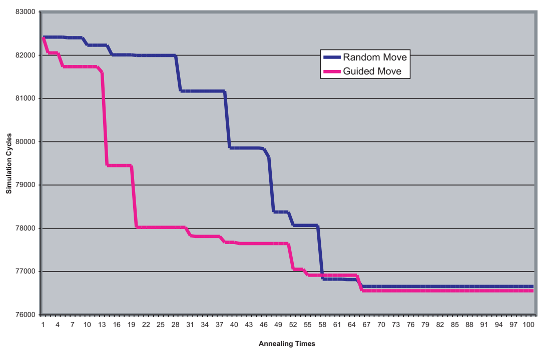

Figure 4. Guided versus Unguided Annealing on memset

As a case study, Figure 4 compares guided versus unguided simulated annealing for a hand-coded version of *memset* from the C libraries. The GRST algorithm requires 102,326 cycles to execute. We carefully hand-scheduled its inner loop and produced a version that executes in 82,416 cycles. We then used this version as the starting point for simulated annealing. Figure 4 plots the performance in simulated cycles (y-axis) of the current best schedule against the number of evaluation function iterations (x-axis) for guided and unguided annealing. Guided annealing drops the cycle counts much more quickly than with fully random perturbations, making intermediate results with guided annealing more useful to detect better opportunities for scheduling.

4. Spatial Path Scheduling

This section describes basic SPS, and the next section shows how we refine SPS with the insights from simulated annealing.

A key deficiency of GRST is that it does not consider all of the potential *anchor* points on a path. An *anchor* is an instruction whose placement is constrained because it accesses a known spatial location: a register bank for a read or write, a cache bank for a load or store, the global control tile for a branch, or a specific execution tile for an instruction the scheduler has already placed. Consider a path $i_1 \dots i_a$ where i_a is an anchor point. For all paths in a legal DFG, the leaf instructions must be anchor points (stores, branches, or register writes); otherwise the path is dead. Intermediate instructions may also be anchor points. For example, at line 8 in Figure 6, the cost of placing the current instruction i_j, where $1 \leq j < a$, at some position p depends not only on $i_1 \dots i_{j-1}$ (the instructions scheduled so far) but on the position of the anchor point i_a. The basic SPS algorithm computes the cost of the entire path for each potential position of i_j, and selects the position for i_j that minimizes this cost. This capability allows the scheduler to place instructions in any order, whether their parents are scheduled or not. Each scheduled instruction then becomes an anchor point, and the algorithm naturally adapts to factor in the cost of routing operands to and from that instruction.

Consider scheduling the DFG in Figure 5(a) on a simplified 4×4×1 ALU array. The path in this example starts at register bank 2, ends at register bank 1, and has four intervening instructions. GRST produces the schedule in Figure 5(b) with a network routing

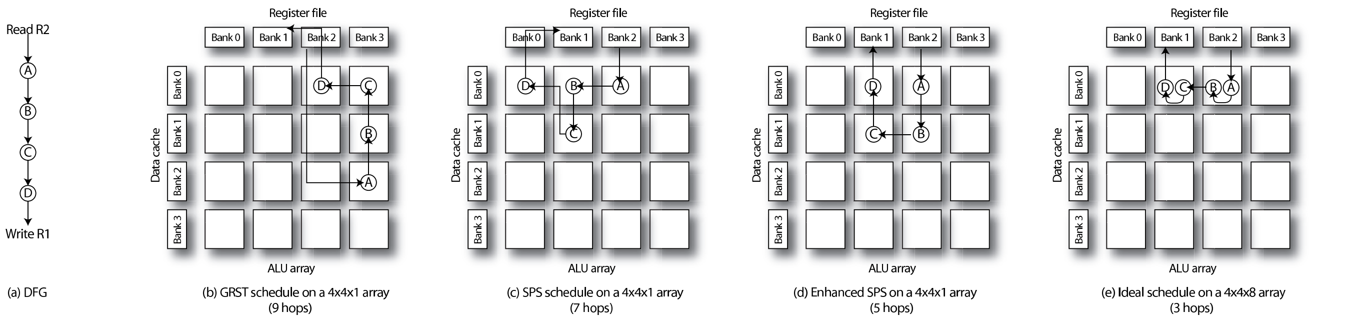
Read R2
Write R1
Register file
Bank 0
Bank 1
Bank 2
Bank 3
Data cache
ALU array
(a) DFG
(b) GRST schedule on a 4x4x1 array (9 hops)
(c) SPS schedule on a 4x4x1 array (7 hops)
(d) Enhanced SPS on a 4x4x1 array (5 hops)
(e) Ideal schedule on a 4x4x8 array (3 hops)

Figure 5. Path Scheduling Example

latency of nine cycles because it does not use anchors, penalizes instructions on paths leading to registers, and breaks ties to the right-hand side to minimize unnecessary ALU contention closer to the caches. Using the basic SPS algorithm yields the better schedule in Figure 5(c) with seven cycles of routing latency. To obtain the ideal five cycle latency schedule for a $4\times4\times1$ ALU array in Figure 5(d) the scheduler must consider the *volume* of instructions between the anchor points as well as the anchors (described in Section 5.4). Figure 5(e) shows the ideal schedule on a TRIPS prototype $4\times4\times8$ array.

4.1 Basic SPS algorithm

We define the *open list* to contain candidate instructions to schedule next. Since an instruction may occupy any slot, it is legal to include all unplaced instructions in the open list. However, to reduce compilation time, we restrict the list to unplaced instructions with no inputs or at least one scheduled parent. This restriction should not reduce the scheduler's effectiveness because SPS evaluates the entire path and its anchors when computing an instruction's criticality and best location. In contrast to the ready set in GRST and other list schedulers, the SPS open list does not require that both parents of an instruction be placed before considering it for scheduling.

For each instruction i in the open list, SPS computes the set of all legal physical locations. For each location $slot$, SPS computes a *placement cost* associated with placing i at $slot$, as shown in Figure 6, keeping track of the minimum cost $slot$ (lines 9-10). SPS schedules the instruction i in S whose lowest placement cost is the highest of the lowest placement costs across all instructions in the open list (i.e., we choose the maximum of the minimums). It then adds any missing children of the scheduled instruction to the open list. This process repeats until all instructions are scheduled.

The placement cost is the expected latency of the longest path in the DFG that passes through instruction i, including both execution and routing latencies. SPS uses the following formula to compute placement cost (*pCost*):

$$\begin{aligned} pCost(i, slot) &= inputLatency + execLatency \\ &\quad + outputLatency \end{aligned} \quad (1)$$

The *inputLatency* is the maximum of each parent instruction p's completion time plus the communication delay from p to i; i.e., the time when the last input operand arrives and the instruction is ready to fire. The *execLatency* is the number of cycles necessary to execute i with no contention. The *outputLatency* is the maximum expected number of cycles from i to any output-producing leaf instruction in the DFG. The cost of the path from i to the latest executing DFG leaf instruction is the sum of the execution latencies of the instructions along that path plus static routing delays between any anchor points on that path.

The algorithm selects the most critical instruction by choosing the instruction whose lowest placement cost is the worst of all instructions in the open list. Lines 9 and 10 in Figure 6 find the minimum cost for each instruction, and lines 12 and 13 then select the instruction with the maximum cost. This function subsumes GRST critical path re-ordering by recomputing the critical path for each instruction in the open list every iteration.

4.2 Anchor points with SPS

Previous algorithms require candidate instructions to have all parents scheduled. SPS naturally accounts for routing latencies between all known anchor point instructions. In addition, each time it places an instruction, that instruction becomes an anchor point. Instructions can therefore be placed in any order and the scheduler naturally incorporates the effects of each placement when placing subsequent instructions.

Since effective addresses for loads and stores are not known until runtime, the scheduler cannot determine their target cache bank. We estimate these latencies using the horizontal routing latency to the cache banks but do not model the vertical routing latency since the cache bank is unknown. We support cache bank location hints, but we currently only use them in hand-assembled code.

This algorithm will support any microarchitectural topology, and will work well when known anchor points are provided to the algorithm before placement. By identifying a greater number of target banks for loads and stores, the compiler will enable the scheduler to generate a superior placement.

Spatial path scheduling (SPS) thus reasons explicitly about path anchor points when computing the cost of placing instructions, which naturally encompasses four of the five heuristics from GRST. This basic SPS algorithm provides an average improvement of 14% over GRST, as described in Section 6. Simulated annealing results, however, show an additional 12% improvement over the basic SPS algorithm, and the next section introduces several heuristics to help close this gap.

5. Extension heuristics to SPS

Analyzing annealed schedules reveals three key shortcomings of the base SPS algorithm. First, it does not account for local or global ALU or network link contention. Second, the critical path is local to a TRIPS block - it does not model global effects (i.e., late arriving block inputs or critical block outputs) that may determine which path is critical at runtime. Finally, it does not account for extra routing delays required when the number of instructions between two anchor points is greater than the instruction capacity of the ALUs along that path.

To address these issues, we extend SPS with an improved load balancing heuristic that approximates global (cross block) and local (within a block) ALU and network link contention. We use the annealed schedules to discover heuristics for approximating these unknown resource conflicts. We augment the critical path calcu-

```
Input: $\mathcal{I}$ - instructions in a block, $\mathcal{G}$ - array of ALUs
output: $\mathcal{A}$ - a mapping $\mathcal{I} \to \mathcal{G}$
 1: $S$ = root_nodes($\mathcal{I}$)
 2: while $S$ is not empty do
 3:     inst = null; instCost = 0; instSlot = none
 4:     for all instructions $i$ in $S$ do
 5:         issueSlots = find_legal_instruction_slots($i$)
 6:         iCost = 0; iSlot = none
 7:         for all slot in issueSlots do
 8:             pCost($i$, slot) = inputLatency($i$, slot) +
                    execLatency($i$) + outputLatency($i$, slot)
 9:             iCost = min(iCost, pCost($i$, slot))
10:             iSlot = slot with minimum iCost
11:         end for
12:         if iCost $>$ instCost then
13:             inst = $i$; instCost = iCost; instSlot = iSlot
14:         end if
15:     end for
16:     $\mathcal{A}$ += schedule(inst, instSlot)
17:     $\mathcal{I}$ = $\mathcal{I}$ - $\{inst\}$
18:     $S$ += get_top_nodes($\mathcal{I}$)
19: end while
```

Figure 6. Spatial Path Scheduling (SPS) Algorithm

lations with limited global information to account for critical dependences between blocks. We use annealed schedules to discover which inter-block dependences increase the critical path length and what static calculations best approximate their importance. Finally, we add a heuristic that explicitly reasons about the volume of instructions and issue slots on a path to anchor point i_a. We tune these heuristics with simulated annealing data, and show that they are synergistic, providing total performance greater than the contributions of each.

5.1 Static metrics

We mined data from the annealed schedules by calculating static data about the annealed schedules, comparing that data to data produced with various scheduler settings, and searching for correlations. These correlations guide the SPS augmentations by suggesting appropriate weighting functions and revealing trends in annealed path latencies. First, we use the critical path tool to determine which blocks contribute the most cycles to the critical path, since those are the blocks most heavily optimized by simulated annealing. For each critical block we derive data including the following:

1. Number of instructions
2. Available ILP
3. Maximum ALU utilization
4. Maximum link utilization
5. Longest static path length

The number of instructions, available ILP, and maximum ALU and link utilization provide insights to weight ALU and link contention metrics. To estimate available ILP, we divided the highest-latency unscheduled path, including intrinsic delays, by the number of instructions in the block. Comparing the highest latency path length between annealed and scheduler-generated schedules guided both how we weight latency and contention, and how we optimize loops. In addition, we found the highest latency path between each input and each of its dependent outputs to determine which paths were more heavily optimized by the annealer.

5.2 Contention modeling

ALU contention: We provide a load balancing heuristic to penalize scheduling instruction i on an ALU in which a previously scheduled independent instruction may cause a resource conflict. We add this penalty to the placement cost of i. This heuristic is similar to the load balancing heuristic used in GRST, but we extend it to approximate global effects and determine more precisely the set of instructions that are likely to cause resource conflicts.

To alleviate local (intra-block) contention, we keep track of when instructions are expected to fire and penalize an instruction by increasing its path length accordingly after detecting a potential conflict. We allow a cycle of slack in either direction for the ready time to account for imperfect estimates.

Although intuitively it makes sense to place dependent instructions on the same ALU to minimize communication delays, annealed schedules sometimes break up dependent paths to prevent resource conflicts between different invocations of the same block in a loop. The local contention metric does not handle these conflicts, so we introduced a global (inter-block) contention heuristic. Initially, we added the sum of issue slots consumed by all instructions scheduled on the same ALU to the placement cost of an instruction on that ALU. Comparisons with annealing data revealed that this simple metric was overly conservative. We improved this optimization by eliminating penalties for instructions that cannot conflict.

To avoid penalizing instructions unnecessarily, we identify more cases in which two instructions on the same ALU cannot cause a resource conflict. A local resource conflict clearly cannot occur between dependent instructions, nor can one occur between instructions on opposite predicate paths. A global resource conflict occurs when instructions from different blocks are ready to use a resource simultaneously. A global conflict cannot occur between two dependent instructions that form a loop carried dependence unless the successor instruction is on a predicated path. A global conflict is more likely to occur due to this block's immediate predecessors or successors (even if they are a separate invocation of the same block), and cannot occur between blocks that cannot be in flight simultaneously. We do not penalize an instruction if other instructions on the same ALU meet these criteria and are therefore unlikely to conflict.

Link contention: Estimating network link contention is less straightforward than estimating ALU contention because each instruction may communicate with multiple other instructions and consume multiple network links along each path. In addition, the scheduler may not know statically which data cache bank a load or store instruction will access and thus will not know which network links it will use. To account for link contention without explicitly knowing each instruction's link usage, the scheduler tracks the number of cycles during which each link is busy for each block. The link contention penalty for instruction i is the number of network links i uses that are consumed for more cycles than a threshold. .

Weighting function: When comparing the schedules generated by the SPS algorithm and by the annealer, we observe that two static metrics are proportional to the size of the gap between annealed and SPS performance (e.g., a more full block has a higher performance gap):

1. The fullness of the block
2. The ratio of the most critical path to the average path

Full blocks are less affected by ALU utilization penalties because the instructions are naturally distributed across the ALU array due to lack of unused slots. Full blocks depend more on good link utilization, however, because more instructions typically mean more communication between instructions. Thus, we introduce a fullness factor that we correlate directly with link contention and inversely with ALU contention:

$$\textit{fullness} = \textit{instructions} / \textit{maxInstructions} \quad (2)$$

where *instructions* is the number of instructions in the block under consideration, and *maxInstructions* is 128 for the TRIPS prototype.

If a resource conflict occurs at runtime the annealer will eventually move the less critical instruction away to allow the more critical instruction to execute without conflict. To approximate this behavior, when the scheduler considers placing instruction i_j on an ALU with instruction i_k and the ratio of instruction i_k's placement cost to i_j's exceeds a threshold, we increase the penalty to twice the number of issue slots consumed by i_k. This moves less critical instructions that can afford extra latency to different ALUs. The fullness factor described above will dampen this effect when the block is full enough that it is not useful.

Finally, we augmented the global ALU contention and the network link contention metrics by the criticality of the instruction, observing that critical instructions should be optimized for communication latency while non-critical instructions should be optimized for utilization. We calculate criticality as follows:

$$\textit{criticality} = \textit{pathLength} / \textit{criticalPathLength} \quad (3)$$

where *pathLength* is the length of the longest path through the DFG that includes this instruction, and *criticalPathLength* is the length of the longest path through the DFG. Because this factor varies inversely with the importance of utilization penalties, we use *1 - criticality* as a factor for utilization.

Further comparisons with simulated annealing revealed that this metric worked better for low concurrency blocks. A block with high concurrency often has no clear critical path, yet utilization is very important in optimizing these blocks because they have abundant available parallelism. We adjusted the criticality measurement based on the concurrency in the block as follows:

$$\textit{criticality} = (\textit{pathLength} / \textit{criticalPathLength}) / \textit{concurrency} \quad (4)$$

where concurrency is equal to the number of instructions in the block divided by the latency of the highest-latency unscheduled path through the block (i.e. IPC on an ideal machine). This modified metric significantly improved the effects of utilization in highly concurrent benchmarks. Thus, the final utilization penalty can be described as follows:

$$\begin{aligned} utilPenalty = {} & localALUCntn \quad (5) \\ & + globalALUCntn * (1 - fullness) * (1 - criticality) \\ & + globalLinkCntn * fullness * (1 - criticality) \end{aligned}$$

Local ALU contention (*localALUCntn*) is not scaled because it is more precise - we estimate it based on the exact expected execution time. Using these weighting functions improves performance by an additional 3% compared to SPS without them.

5.3 Global register prioritization

Optimizing the longest path through the block is not always the best decision, particularly in the presence of loops. The annealed schedules showed that paths with register reads and writes were often optimized differently than expected based only on local information. To address this issue, we prioritize registers expected to arrive late by increasing their path length. We first prioritized paths containing loop-carried dependences before considering other paths by significantly increasing their path length by a constant amount. We analyzed the scheduled path length of the annealed results and the SPS results and still found that certain paths were more heavily optimized in the annealed schedules, even if those register outputs were not loop-carried dependences.

In the presence of next-block prediction, a block can begin to speculatively execute before previous blocks commit. As a result, instructions that depend on registers from the previous block are often a bottleneck. The paths most aggressively optimized in the annealed schedules were sometimes those that formed the longest path through multiple blocks. The lengths of paths in immediately neighboring blocks affect the importance of register inputs and outputs. By approximating a global critical path in the presence of register dependences in loops SPS improved by an additional 4% compared to the previous results. The scheduler prioritizes as follows:

1. Schedule smaller loops (measured in number of blocks) before larger loops.
2. Schedule loop-carried dependences before other instructions.
3. Augment the placement cost function for each instruction with the length of the longest path through predecessor and successor blocks with which it forms a register dependence.

The last heuristic approximates a global critical path by extending the path through three blocks instead of one. Using profile information would improve this optimization, but is left for future work.

5.4 Path volume scheduling

As shown in Figure 5(d), SPS needs to consider the number of instructions together with the distance between two anchors. Given one source anchor, i_1, and one destination anchor, i_a, and $i_2 \ldots i_{a-1}$, the minimum number of ALUs in which to schedule n instructions is n/(*issue slots per ALU*) if all slots are free. During scheduling, however, some issue slots may already be occupied. The path volume heuristic attempts to find the best physical path from i_1 to i_a.

To find the best path, the scheduler performs a depth first search with *iterative deepening* [13] of possible paths to find the shortest path from source to destination ALU that accommodates all n instructions. Iterative deepening attempts to solve a search problem with a given depth bound. If it succeeds, it returns the result, and if it fails, it increases the depth bound and repeats. The scheduler initializes its depth bound to the minimum communication delay between the source and destination ALUs, and increases it until it finds a path with enough available instruction slots to accommodate all n instructions. The search returns the difference between the number of links traversed in the solution path and the number of links in the minimum latency path. We add this value to the path's latency. This lookahead value prevents the scheduler from unnecessarily penalizing locations along the best available path because they are not on the path with the lowest minimum communication delays. This heuristic is generalizable to any mapping intended to minimize communication latencies.

5.5 Modified placement cost: Putting it all together

SPS combines the above heuristics to calculate a final placement cost. This cost determines both which instruction to schedule next and where to schedule it. The placement cost initially contains the total known path length of the most critical path through instruction i_j, including all execution and communication latencies along that path, as shown in Equation 1. We then add the result of local and global contention modeling to the total placement cost. This calculation estimates critical path delay due to resource conflicts for the ALU in question, rather than just the path length through a single block. We augment the input and output latencies with global register prioritization costs. This prioritizes instructions that have delayed inputs due to critical read instructions such that they are scheduled first, making it more likely they will be placed in their ideal spot. This step also helps determine which are the most critical anchors. Finally, the volume optimization adds an intrinsic delay to the total placement cost.

Revising Equation 1 (described in Section 4.1) with these heuristics yields the following placement cost of instruction i at location $slot$:

$$\begin{aligned} pCost(i, slot) \quad = \quad & inputLatency + utilPenalty \\ & + execLatency + outputLatency \\ & + additionalRoutingCost \end{aligned} \tag{6}$$

Here, *inputLatency* and *outputLatency* include the global register prioritization cost of block inputs (register reads) or block outputs (register writes) along the longest path through i. This information percolates up the graph from the leaf instructions and down the graph from the root instructions. The *utilPenalty* is computed with Equation 6. Finally, *additionalRoutingCost* returns the number of additional routing cycles necessary between the most critical upward anchor and $slot$, and between $slot$ and the most critical downward anchor. SPS calculates this revised placement cost for each instruction at each legal location. If multiple anchor points may be critical, then the algorithm will calculate this cost for multiple pairs of upward and downward anchor points.

5.6 Complexity

For a fixed-size ALU array, the time complexity of both basic SPS and GRST is O(i^2), where i is the number of instructions in the block. Varying the number of functional units as well, GRST's complexity becomes O(i^2 + i*n) where n is the number of functional units. SPS has O(n*i^2) complexity because it considers each location for each instruction at each step, whereas GRST only finds the best location for the most critical instruction at each step. The volume optimization adds a b^n term where b is the branching factor (constant). Reducing the impact of this factor would be relatively simple with pruning, but doing so was unnecessary for the array sizes considered so far.

6. Results

We tested SPS and its heuristics on a low-level simulator verified within 5% on average of the TRIPS prototype RTL. This simulator closely models delays in the TRIPS prototype including communication and contention delays within the array of ALUs and the operand network.

We selected benchmarks with varying levels of concurrency and memory behavior to find results applicable across a variety of application domains. Because the simulator is prohibitively slow, we report results for isolated kernels from SPEC2000, EEMBC, Livermore Loops, MediaBench, and C libraries. We created the SPEC kernels by profiling the SPEC2000 benchmarks and extracting functions and loops that account for approximately 90% of the program execution time. We used checkpointing to find appropriate inputs to these functions, and modified the data set size so that the kernels run in tractable time on the cycle-accurate simulator.

The scheduling problem is most interesting for blocks with medium to high instruction counts. Dense blocks require some subset of instructions to incur significant communication delays traversing the ALU array. They also require careful balancing of ALU and network link contention. When blocks are not full, block overheads often dominate the critical path, making it hard to evaluate the performance of the instruction scheduler. To provide dense blocks, we focus on hand-coded versions of these benchmarks. SPS improves over GRST by 21% on average on these benchmarks, and up to 52%. Section 6.4 contains results using compiler-generated code from the EEMBC suite. These blocks are not yet as full as the hand-coded benchmarks, and thus are less sensitive to scheduling. SPS still improves performance by 17% on average and up to 37% over GRST, however.

6.1 Comparison with GRST

The baseline for all results is the GRST algorithm applied to the TRIPS prototype configuration. Column 1 in Table 1 provides the IPC with GRST on the hand-coded microbenchmarks for reference. The SPS algorithm provides a 14% improvement on average over GRST on hand-optimized benchmarks, and the annealed results show a 26% improvement over GRST.

The Livermore Loops kernels show lower performance improvements than other benchmarks because they make extensive use of libraries that were not recompiled with different algorithms. The annealed results all use a single set of libraries created with the best scheduler on-hand at the time we began annealing. When we recompile the libraries with SPS, we improve over simulated annealing. However, to preserve a fair comparison between scheduler heuristics and annealed results, we exclude scheduling the libraries in these experiments.

6.2 Tuning SPS

The augmented SPS algorithm provides heuristics to address global interactions and lookahead. In isolation, the path volume, contention modeling, and critical register prioritization heuristics each provide less than 4% improvement over the basic SPS algorithm. We present percent improvement using each of the heuristics in isolation in columns 4-6 of Table 1. The heuristics are synergistic, combining to provide a 7% improvement. The path volume heuristic, in particular, does not improve average performance over base SPS without global register prioritization. The volume heuristic is most beneficial when applied to critical global paths and their anchor points; global register prioritization reveals these critical anchor points. The contention heuristic also performs better with register prioritization in place, because its weighting function is based on instruction criticality.

Register prioritization using only loop-carried dependences performed 3% better than the basic SPS algorithm with all other heuristics in place. Adding additional register prioritization as described in Section 5.3 was necessary for the full 7% performance improvement.

Using the static metrics described in Section 5.1 to analyze the annealed results, we developed functions to restrict and weight the effects of the contention heuristics. Using the number of instructions in the block, the available ILP, the critical path length through the block, and the maximum ALU and link utilization was necessary to achieve the best possible improvements in contention. With all other heuristics in place, the initial contention optimization provided a 4% improvement over the original SPS algorithm. We see the additional 3% (for a total 7% improvement) only when using the contention heuristics described in Section 5.2.

Profiling could potentially improve global register prioritization and global ALU contention significantly. Without a weighted control flow graph, the scheduler uses loop nest depth to determine which block is the most likely predecessor or successor to a given block. We leave incorporating profile information for future work.

With all heuristics in place, the final scheduler improves over the previous best algorithm by 21%, improves over the basic SPS algorithm by 7%, and is within 5% of the annealed results.

6.3 Comparison to annealed results

We used the annealed results as a target point to analyze the quality of the scheduler's placement choices. The annealed results presented here represent weeks to months of annealing time, with many results converging, but they may be stuck in local minima.

The annealer we implemented is limited because it considers only the blocks from a single source file (which is why libraries are omitted). Currently, most of the benchmarks that we evaluate are contained within a single file. For programs in multiple files,

Benchmark	GRST IPC	% Percent cycle count improvement over GRST% Base SPS	Contention	Register	Volume	Combined SPS	Annealed	Annealed IPC
EEMBC								
a2time01_hand	2.4	12.2	16.9	8.0	12.4	18.6	18.9	3.1
Spec2000 Microbenchmarks								
ammp_2_hand	2.7	7.2	13.3	30.5	9.3	35.0	36.8	4.4
art_1_hand	4.1	9.3	10.6	32.9	15.0	34.0	35.4	6.4
art_2_hand	2.8	32.4	41.3	46.8	26.9	48.6	52.2	5.9
art_3_hand	2.1	24.9	47.6	24.6	25.8	49.1	50.3	4.4
bzip2_1_hand	1.4	46.6	52.2	48.4	47.9	52.2	52.2	3.0
equake_1_hand	2.4	0.2	-4.3	1.4	0.9	6.8	10.4	2.7
gzip_1_hand	0.6	26.1	19.0	25.0	27.0	14.6	49.7	1.3
gzip_2_hand	3.9	-1.6	3.2	-1.5	-2.9	11.5	13.1	4.6
matrix_1_hand	2.9	1.8	3.2	4.6	5.4	8.3	19.6	3.7
parser_1_hand	1.2	46.6	42.5	44.4	42.2	49.7	49.4	2.4
transpose_GMTI_hand	2.9	25.3	29.6	26.2	24.3	29.9	33.7	4.4
vadd_hand	5.0	5.0	7.7	4.7	5.1	8.7	18.2	6.2
Livermore Loop Kernels								
cfar_hand	1.6	10.7	9.5	12.5	10.3	15.6	16.4	1.9
conv_hand	4.8	-2.0	0.6	11.3	-7.3	10.0	19.0	6.0
ct_hand	3.7	14.5	14.5	19.6	12.2	13.9	18.9	4.6
db_hand	0.5	7.9	9.1	9.1	8.2	8.8	10.1	0.6
genalg_hand	1.2	10.0	12.3	13.0	12.4	16.8	16.8	1.5
pm_hand	1.7	7.8	10.1	9.5	9.8	13.5	14.7	2.0
qr_hand	1.3	3.1	3.2	3.4	2.8	3.3	3.7	1.4
svd_hand	1.0	4.5	4.5	4.4	4.8	5.1	5.2	1.1
C libraries								
memchr_hand	1.1	20.7	27.6	23.5	25.6	28.2	37.8	1.8
memcpy_hand	2.8	12.2	27.4	22.6	11.5	21.8	32.4	4.2
memset_hand	3.0	33.0	32.0	23.0	24.3	34.0	39.7	5.0
strcmp_hand	2.2	9.5	8.6	7.3	12.1	15.8	24.5	3.0
MediaBench								
sha_hand	0.9	12.8	14.3	15.0	14.4	15.6	18.4	1.1
Average		**14.6**	**17.6**	**18.1**	**14.6**	**21.9**	**26.8**	

Table 1. Percent improvement in cycle count of hand-coded benchmarks over GRST.

we chose the one that contributed the most cycles to the program's critical path. For all other input files, we used a consistent set of schedules for the other files. In most cases, the functions in the annealed file consumed the majority of the critical path cycles. The Livermore Loops are an exception. They consist of multiple source files and use library calls extensively. Since we anneal only one file, we do not see as much improvement on them: 3% to 19% versus 10% to 52% for the other benchmarks. Compiling the annealed files together with libraries scheduled by SPS sometimes significantly improves or degrades the results. This result occurs because instructions from blocks in libraries may execute simultaneously with instructions from the annealed source file. Applying the annealer again would adapt the schedule to alleviate any resource conflicts that occur.

Some of the memory-intensive benchmarks such as vadd and matrix_1 show a significant gap between the best scheduler and the annealed results. In some cases, the annealer will produce a schedule that SPS can never achieve without help from the compiler. In these benchmarks, loads and stores representing array accesses are regular and consistently map to one or more cache banks. The annealer places critical loads and their consumers in the appropriate location based on their cache bank. The cache banks use a simple round robin address mapping, and thus if the compiler knows the base address and array dimensions, it can also predict the load/store cache bank. We have experimented with array base address alignment and compiler-inserted cache bank hints for predictable loads, and these results match the annealer's for vadd and matrix_1.

The memset hand-coded benchmark provides cache bank hints, which allows the best SPS schedule to come closer to the annealed results (5%, compared to 10% and 11% for vadd and matrix_1, respectively). Most other hand-coded benchmarks do not provide these hints, and none of the compiler-generated benchmarks use them.

Annealed performance on gzip_1 is 35% better than the fully augmented SPS schedule. This unusual result is due to the dependence predictor in the TRIPS prototype. TRIPS allows speculative execution of memory operations and initiates a pipeline flush after detecting a dependence violation. The TRIPS implementation uses dependence prediction to help prevent these expensive flushes. For a set period after a load causes a dependence violation, it will conservatively wait for all prior stores before executing. For gzip_1, when minimizing latency and contention the scheduler enables a dependent load to issue before its corresponding store, triggering a pipeline flush, initiating conservative memory ordering, and degrading performance. This particular performance anomaly would be solved with a more accurate dependence predictor and is beyond the scope of the scheduler.

6.4 Cross Validation

Because the heuristics were driven by data from the hand-coded schedules that we annealed, we ran the same set of tests on the EEMBC benchmark suite using compiler-generated code to see how well the heuristics perform on new benchmarks. We ran all of the EEMBC benchmarks except the consumer benchmarks cjpeg and djpeg, which took too long to simulate. Table 2 shows these re-

		% Percent cycle count improvement over GRST %					
Benchmark	GRST IPC	Base SPS	Contention	Register	Volume	Combined SPS	SPS IPC
Automotive							
a2time01	0.8	7.4	8.2	6.5	6.2	9.7	0.9
aifftr01	1.1	17.3	20.2	17.9	19.7	19.2	1.4
aifirf01	0.9	9.6	12.1	9.5	9.8	12.1	1.0
aiifft01	1.1	18.2	20.4	18.0	20.5	22.1	1.4
basefp01	0.8	7.5	9.3	7.7	5.7	11.1	0.9
bitmnp01	1.3	10.5	14.6	11.9	13.5	16.0	1.5
cacheb01	0.7	8.4	8.0	7.6	10.0	8.3	0.7
canrdr01	0.8	13.6	17.5	13.4	17.8	17.3	1.0
idctrn01	1.4	11.0	12.6	12.3	11.2	16.2	1.7
iirflt01	0.7	5.7	7.7	7.9	6.4	10.9	0.8
matrix01	0.9	12.5	14.8	11.7	13.2	12.9	1.0
pntrch01	0.8	8.5	13.1	7.2	8.4	11.3	0.9
puwmod01	0.8	13.5	17.4	13.8	18.1	17.3	1.0
rspeed01	0.8	12.2	15.9	12.1	16.0	16.1	1.0
tblook01	0.8	11.3	13.3	10.1	12.3	14.4	0.9
Networking							
ospf	0.9	14.9	15.4	12.2	15.3	16.8	1.1
pktflow	1.0	13.3	12.8	11.5	13.7	18.8	1.3
routelookup	0.9	17.4	17.8	14.6	18.6	19.5	1.1
Office							
bezier01	1.1	13.0	16.3	12.9	16.8	17.3	1.3
dither01	1.5	18.5	19.6	23.9	17.4	21.8	2.2
rotate01	1.1	16.5	18.6	18.8	19.6	20.2	1.4
text01	0.8	15.8	16.5	17.6	19.3	17.9	1.0
Telecom							
autcor00	1.2	8.3	9.2	11.0	7.5	10.6	1.4
conven00	1.8	16.1	16.0	14.7	16.4	13.7	2.1
fbital00	1.2	34.2	35.3	35.0	35.9	37.2	1.9
fft00	2.1	10.7	22.2	18.9	18.8	34.5	3.2
viterb00	0.9	28.1	36.4	41.8	37.3	36.5	1.4
Average		**13.9**	**16.3**	**14.8**	**15.8**	**17.8**	

Table 2. Percent improvement in cycle count of compiler generated code for EEMBC over GRST.

sults. The compiler does not yet produce blocks that are as tightly-packed as the blocks in the hand-coded benchmarks. The average dynamic number of instructions fetched per block fetched is 45% larger for the hand-coded benchmarks. Because the resulting blocks have proportionally more block overhead, placement has a smaller impact on their performance. Still, the SPS algorithm improves 13% over GRST for the compiler-generated EEMBC benchmarks, and with all heuristics in place SPS improves 17% over GRST.

We tested the algorithm with and without the annealing-inspired heuristics on the compiler-generated codes. The contention heuristics produce a 2% improvement, and global register prioritization produces a 3% improvement. The heuristics are slightly less synergistic than on the hand-coded benchmarks, but still improve and apply to this wider set of applications.

7. Related work

This section compares the TRIPS scheduling problem and solution to the scheduling problems and solutions for other architectures that use *static* spatial and/or temporal compile-time scheduling. We also compare spatial instruction scheduling to ASIC and FPGA place and route algorithms.

7.1 Greedy list scheduling for TRIPS

The path scheduling algorithm described here naturally subsumes the heuristics in the prior scheduling work for TRIPS [17], and substantially improves performance. GRST uses a list scheduling approach augmented with heuristics that account for data cache and register accesses, load balancing, and critical path prioritization. Its ability to identify and optimize the critical path is limited, however, because it does not account for routing delays between anchor points along a path, take global effects into account, or choose the next instruction to schedule by directly comparing placement costs. The GRST study used binary rewriting and a higher-level simulator with simplifications such as a shared register file and a unified L1 cache, which substantially dampened the influence of the scheduling algorithms.

7.2 Static scheduling

Classic VLIW schedulers focus solely on minimizing the dependence height of the final schedule. At runtime, instructions that do not fire must still check their predicate before subsequent instructions can execute due to the static issue execution model [6, 8]. These approaches thus schedule the critical path bottom up and do not consider registers or other physical processor layout constraints.

The most closely related static (compiler) scheduling problem is the one posed by architectures such as partitioned VLIW [9, 10, 11, 19, 21, 26] and RAW [14, 25], which both consider register resource constraints and layout. Ozer et al. solve the scheduling part of VLIW cluster assignment first with a later phase performing register assignment based on the cluster assignment of dependent instructions [19]. Ozer et al. also find that placing critical paths in the same cluster is best in a VLIW compiler. Another VLIW example is the CARS approach, which is similar to Ozer et al. but performs register allocation concurrently with scheduling and has lower algorithmic complexity [10]. The typically modest number of VLIW partitions and unlimited instructions per partition impose a different structure on VLIW schedulers. In particular, the paths are shorter and less constrained compared with TRIPS scheduling problem.

RAW's execution model is essentially a 2D VLIW with independent sequencers [14, 25]. Lee et al. introduce a convergent scheduler that is similar to the simulated annealing scheduler, but explores the space in a model-driven way rather than accepting random poor placements occasionally. The convergent scheduler deals with the complexity of the 2-D scheduling problem by computing an ALU preference for each instruction for a selection of scheduling heuristics. These preferences are weighted to determine the final schedule. Because of the static-issue execution model, parallelism is favored over latency.

7.3 Simulated annealing

Because most problems in compiler optimization are NP-complete, researchers on compiler optimizations and scheduling are increasingly turning to machine learning and statistical techniques such as simulated annealing [12, 2] to find better heuristics. Sweany and Beaty apply simulated annealing to scheduling straight-line (a single basic block) code for the Alpha 21164, which is 4-way issue, and evaluate the schedules with a static metric: the reduction in the schedule length. Their test suite consists of small blocks (e.g., 20–40 instructions on average). The TRIPS scheduling problem is more complex and requires guidance for simulated annealing to efficiently search the much larger space.

Swanson et al. suggest a real-time, on-line simulated annealing approach to optimize reconfigurable hardware dynamically [24]. They suggest that dynamically reorganizing instructions would allow the configuration to adjust to changing program behavior, and to route computation around defective hardware.

Moss et al. [16] use exhaustive search for small basic blocks (on the order of ten instructions) in a supervised learning approach that generates heuristics for a basic block scheduler for the Alpha. Exhaustive evaluation is too expensive for even a single block for TRIPS, but we do consider global effects with simulated annealing. Because the TRIPS prototype is more sensitive to scheduling, we consider the entire program (versus individual basic blocks in isolation), and we evaluate schedules using simulated cycles (versus statically).

Betz and Rose present Versatile Place and Route (VPR) [3], a placement and routing tool for FPGA research that uses simulated annealing for placement. Because latencies and resource conflicts are known for place and route, the algorithm can use an easy to evaluate static cost function as the objective function for simulated annealing. Because runtime latencies and resource conflicts exist with dynamic issue, a static cost function is not sufficient.

7.4 Dynamic scheduling

The TRIPS architecture combines dataflow execution within a block by forwarding temporary values directly from producers to consumers and out-of-order speculation across blocks by using sequential memory semantics and register communication between blocks, together with commit and rollback logic.

Compilers for classic dataflow machines performed no static scheduling [1, 5]. These machines dynamically schedule instructions as the dynamic dataflow graph unfolds. They sidestep sequential memory semantics by relying on a functional programming model in which programs could only ever produce exactly one input to an operand. The WaveScalar [23] dataflow architecture breaks the program up into dataflow waves to preserve sequential memory semantics. These waves have constraints, but are not as resource constrained as TRIPS blocks.

Mercaldi et al. present a static instruction placement performance model for WaveScalar and analyze its correlation with actual performance [15]. A similar static cost function is not sufficient for simulated annealing in our setting because we attempt to find near-optimal schedules and use them to discover the appropriate heuristics. Using a static cost function for simulated annealing would not be useful for discovering heuristics, and without perfect correlation to real performance it would not be useful for exploring the performance upper bound. A static metric with good correlation could be used to guide annealing, however, or to work in combination with the perfect evaluation function to reduce annealing time, which is a direction for future work.

7.5 ASIC and FPGA place and route

Spatial instruction scheduling is similar to place and route because both problems involve mapping a graph of operations onto a two dimensional substrate under a set of constraints. When evaluating place and route decisions, however, the latency and resource conflicts are known statically. Spatial instruction scheduling for a dynamic issue machine like TRIPS or WaveScalar presents new challenges because resource conflicts can only be determined probabilistically.

Paulin and Knight present Force-Directed Scheduling for ASIC synthesis [20], an algorithm that bears some similarities to SPS. The space is constrained in different ways, however. SPS considers dynamic issue orders with probabilistic contention modeling. Its heuristics map a graph onto a predefined topology, while an ASIC scheduler manipulates many variables including the hardware configuration. An ASIC scheduler is guided by external constraints including area and power budget, whereas a spatial instruction scheduler operates under no equivalent restrictions.

8. Conclusions

Spatial Path Scheduling is designed to map DFGs of instructions to a distributed, potentially irregular compute substrate. It improves over list scheduling by explicitly modeling instruction placement for all instructions on a path. We extended the base SPS algorithm with three heuristics: a contention heuristic that captures the runtime and global effects of contention in the ALUs and networking links; a critical register prioritization heuristic that accounts for inter-block dependences; and a path volume heuristic that plans out potential routes for long paths in a DFG and avoids overly constrained local minima. The combined SPS algorithm achieved a 21% speedup over the best previous algorithm for this architecture. The SPS algorithm intrinsically captures and augments the heuristics from GRST while providing flexibility to adapt to new topologies and constraints.

We also implemented a simulated annealing scheduler that uses program criticality information for faster convergence. The simulated annealer perturbs schedules periodically in an attempt to avoid finding itself in local minima. Although the annealer does not necessarily generate optimal schedules, the schedules it produces upon convergence were typically close to the best schedules it produced across many different starting points. We used them to tune the SPS heuristics, achieving average performance within 5% of the annealed results.

This performance bound, however, only applies to generating the best schedule for a fixed set of instructions. More opportunities for performance improvement remain if the scheduler can collude with other compiler phases to generate graphs that are more amenable to better scheduling. Three examples of this effect are register allocation, fanout tree insertion, and hyperblock formation. Since allocated registers serve as pre-schedule anchor points, allocating registers to minimize critical path lengths can improve performance significantly, as we have seen in a few hand-tuned examples. Fanout trees for distributing produced values to many consumers in a block can have many different topologies, each of which will provide a different set of opportunities or costs to the scheduler. Finally, the hyperblock formation algorithm in the compiler attempts to fill blocks as close to 128 instructions as possible to minimize per-block overheads. However, it may be that in some cases, overly full blocks constrain the scheduler too much, and that better performance may be possible if the hyperblock generator leaves space in those blocks to provide flexibility to the scheduler. Coupling the scheduler to these compiler phases is a focus of our current research efforts.

Acknowledgments

We would like to thank Simha Sethumadhavan, Sadia Sharif, Mike Bond, Boris Grot, Bert Maher, Walter Chang, and our reviewers for their helpful comments. We would also like to thank Bert Maher and Behnam Robatmili for the use of their condor cycles.

References

[1] K. Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[2] S. J. Beaty and P. H. Sweany. Instruction scheduling using simulated annealing. In *International Conference on Massively Parallel Computing Systems*, Colorado Springs, CO, Apr. 1998.

[3] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 213–222, London, UK, 1997. Springer-Verlag.

[4] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and others. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, pages 44–55, July 2004.

[5] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *International Symposium on Computer Architecture*, pages 126–132, New York, NY, USA, 1975.

[6] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1986.

[7] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[8] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *ACM Symposium on Compiler Construction*, Montreal, Canada, June 1984.

[9] E. Gibert, J. Sanchez, and A. Gonzalez. Effective instruction scheduling techniques for an interleaved cache clustered VLIW processor. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 123–133, 2002.

[10] K. Kailas, K. Ebcioglu, and A. K. Agrawala. CARS: A new code generation framework for clustered ILP processors. In *International Symposium on High-Performance Computer Architecture*, pages 133–143, Jan. 2001.

[11] C. Kessler and A. Bednarski. Optimal integrated code generation for clustered VLIW architectures. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 102–111, June 2002.

[12] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[13] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[14] W. Lee, D. Puppin, S. Swanson, and S. Amarasinghe. Convergent scheduling. In *International Symposium on Microarchitecture*, Istanbul, Turkey, Oct. 2002.

[15] M. Mercaldi, S. Swanson, A. Peterson, A. Putnam, A. Schwerin, M. Oskin, and S. Eggers. Modeling instruction placement on a spatial architecture. In *SPAA '06: Proceedings of the Symposium on Parallel Architectures and Applications*, 2006.

[16] J. Moss, P. E. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. Brodley, and D. Scheeff. Learning to schedule straight-line code. In *Neural Information Processing Systems – Natural and Synthetic*, Denver, CO, Dec. 1997.

[17] R. Nagarajan, D. Burger, K. S. McKinley, C. Lin, S. W. Keckler, and S. K. Kushwaha. Instruction scheduling for emerging communication-exposed architectures. In *The International Conference on Parallel Architectures and Compilation Techniques*, pages 74–84, Antibes Juan-les-Pins, France, Oct. 2004.

[18] R. Nagarajan, X. Chen, R. G. McDonald, D. Burger, and S. W. Keckler. Critical path analysis of the TRIPS architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2006.

[19] E. Ozer, S. Banerjia, and T. M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *International Symposium on Microarchitecture*, pages 308–315, December 1998.

[20] P. G. Paulin and J. P. Knight. Force-directed scheduling in automatic data path synthesis. In *DAC '87: Proceedings of the 24th ACM/IEEE conference on Design automation*, pages 195–202, New York, NY, USA, 1987. ACM Press.

[21] Y. Qian, S. Carr, and P. Sweany. Optimizing loop performance for clustered VLIW architectures. In *The International Conference on Parallel Architectures and Compilation Techniques*, pages 271–280, Charlottesville, VA, Sept. 2002.

[22] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, Manhattan, NY, Mar. 2006.

[23] S. Swanson, K. Michaelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Symposium on Microarchitecture*, December 2003.

[24] S. Swanson, K. Michelson, and M. Oskin. Configuration by combustion: Online simulated annealing for dynamic hardware configuration. In *ASPLOS X Wild and Crazy Idea Session*, 2002.

[25] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, pages 86–93, Sept. 1997.

[26] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Software and hardware techniques to optimize register file utilization in VLIW architectures. In *Proceedings of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Systems (IWACT)*, July 2001.