Stanford University

Concurrent VLSI Architecture Group Memo 127

Guaranteeing Forward Progress of Unified Register Allocation and Instruction Scheduling

Jongsoo Park and William J. Dally

Concurrent VLSI Architecture Group

Computer Systems Laboratory

Stanford University, Stanford, CA 94305

Email: {jongsoo, dally}@cva.stanford.edu

March 31, 2011

Abstract

Increasingly demanding computation requirements and tighter energy constraints have motivated distributed and/or hierarchical register file (DHRF) organizations as a mean to efficiently sustain a sufficient ALU utilization in processors targeting embedded applications with many ALUs. Compared to conventional centralized register file organizations, DHRFs lead to tighter coupling between register allocation and instruction scheduling: since latencies to register files are non-uniform, register allocation affects access latencies, thus in turn affects instruction scheduling. To avoid this phase ordering problem, researchers have proposed performing instruction scheduling and register allocation simultaneously.

While these unified register allocation and instruction scheduling algorithms address the phase ordering problem, they can be susceptible to scheduling deadlocks: the scheduling algorithm cannot make any forward progress due to its previous decisions. Previous unified algorithms either ignore the scheduling deadlock issue or rely on architectural assumptions with respect to connectivity between register files and functional units.

In this report, we present a unified register allocation and instruction scheduling algorithm that guarantees forward progress. Our algorithm continuously checks an invariant that is formulated as a maximum flow problem. We prove that our unified register allocation and instruction scheduling guarantees to make forward progress when the invariant is maintained.

We have implemented our unified algorithm in the compiler targeting an early version of Stanford ELM architecture. Since only 6 simple applications are considered during our preliminary evaluation, the results should be viewed as a proof of concept, and we mostly focus on algorithm description and its correctness proof rather than analysis of evaluation results. Nevertheless, we believe that our algorithm can be applied to other architectures with non-uniform register file organizations such as reconfigurable architectures, and a more extensive evaluation of our algorithm is expected in the future.

1 Introduction

Embedded applications such as baseband modem processing have demanding computation requirements and energy restrictions [21]. In order to satisfy the computation requirements, we need to keep a sufficient number of ALUs busy, but this poses a challenge of providing a sufficient bandwidth to the ALUs in an energy efficient manner. Although a centralized register file provides a simple solution, it is not energy efficient due to its requirement with respect to many ports and due to increasing gap between global wire delays and transistor speeds [22]. Distributed [5, 20, 23] and hierarchical [25, 2] register file organizations are therefore proposed to avoid energy-hungry multi-ported register files and scaling bottlenecks from wire delays.

By distributing register files and exploiting data locality, majority of required bandwidth can be provided to ALUs from their local register files. Although accesses to remote register files introduce additional latencies, these remote accesses happen infrequently and exposing wire delay to the compiler is more scalable approach than hiding it considering the increasing gap between wire delays and transistor speeds. The register organization of emerging reconfigurable architectures [14,13,22,17] can also be viewed as a distributed one since each reconfigurable functional unit has its own local register file, which is connected to remote register files through an interconnection network.

Hierarchical register files [25,3,2] also improve the energy efficiency of delivering operands to ALUs. We can keep the capacity of register files that are close to ALUs small. Due to the locality of data access, shallow register files that are close to ALUs (e.g., L0 register files shown in Figure 1) will provide most of the required bandwidth, while farther register files with larger capacity and higher unit access energy will be accessed infrequently. In addition, we can reduce the number of ports of the larger register files exploiting their infrequent access [2] and expose forwarding registers so that operands can be explicitly forwarded [2].

Realizing the full benefit of *distributed and/or hierarchical register file organization* (DHRF) however requires effective register allocation and instruction scheduling by the compiler. Conventional separate allocation and scheduling is considerably ineffective due to their tighter coupling. Even in the context of conventional unified register file organizations, register allocation and instruction scheduling ordering has been one of the primary phase

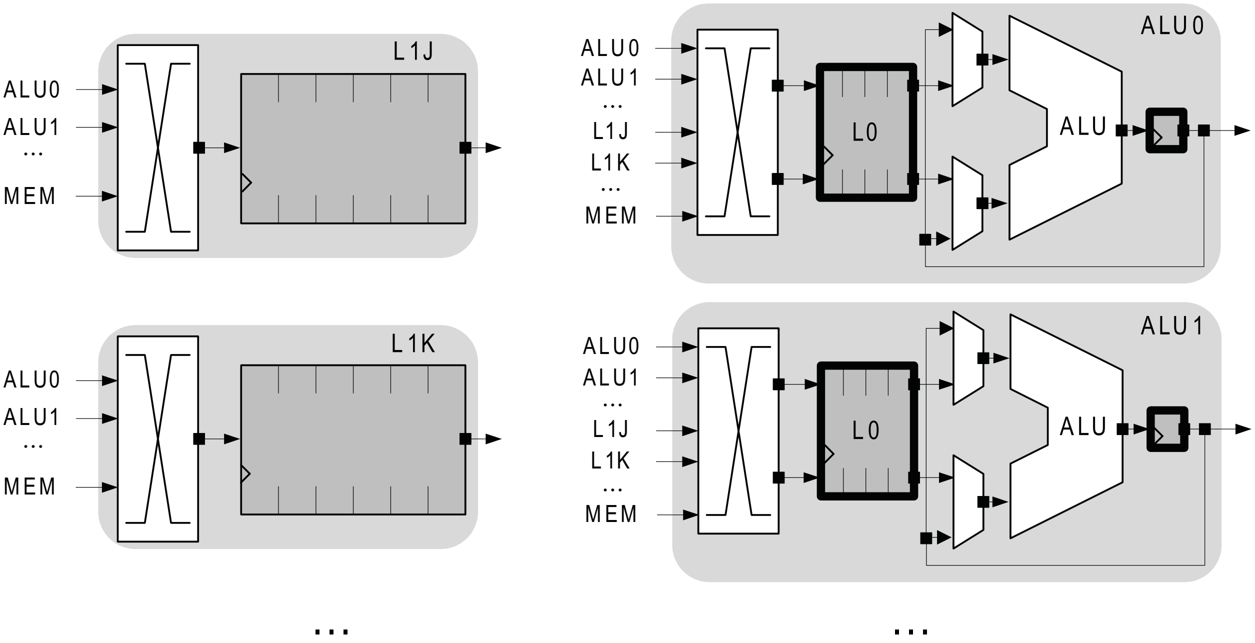


Figure 1: An example of distributed hierarchical register file organization. Similar to VLIW processors with distributed register file organization [5], L1 register files (L1J and L1K) can be accessed by multiple functional units. We add another level of memory hierarchy, L0, which captures the short term operand locality: each functional unit has its own local L0 register file which has thick outlines in this figure. These L0 register files can be viewed as pipeline registers between the register read stage and the execution stage that are exposed to the compiler. In addition, the pipeline register that holds output of each function unit is exposed to the compiler so that temporal variables that are produced and consumed by back-to-back instructions can be explicitly forwarded without occupying register files.

ordering problem in the compiler literature [15, 4, 18]: allocating register first may decrease instruction-level parallelism that can be found by instruction scheduling, whereas scheduling instructions first may increase register pressure, thereby forcing register allocation phase to unnecessarily spill variables. DHRF aggravates this phase ordering problem since it tightly couples decisions in space domain (allocation) and those in time domain (scheduling). The location where a variable is allocated affects the latency of accessing the variable from functional units. Conversely, the time when an instruction is scheduled affects feasible locations of its operands due to non-uniform latencies.

In order to address this phase ordering problem, researchers have proposed performing

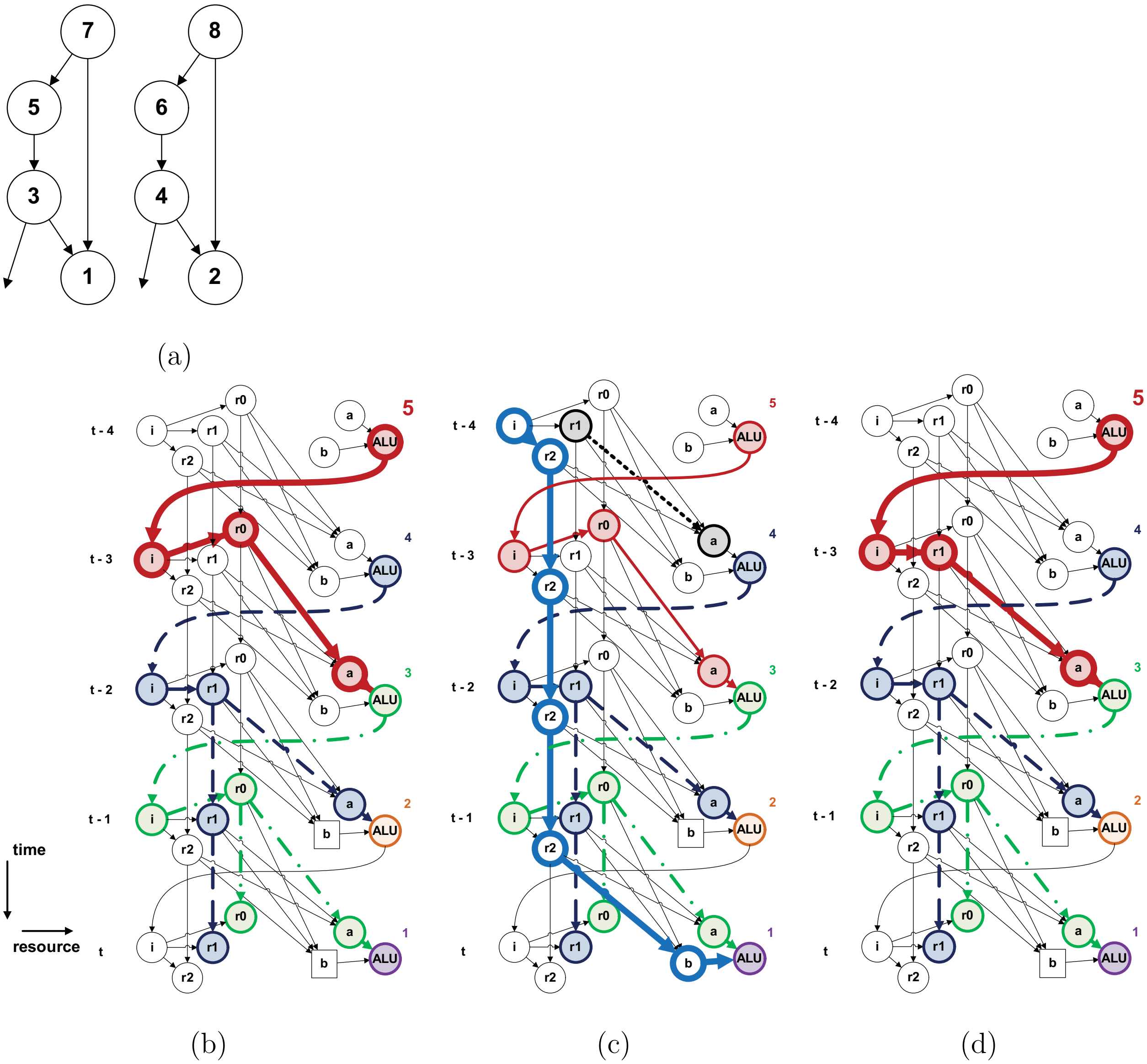


Figure 2: (a): The data flow graph of a simple example. (b)-(d): Schedules for the example. We use a simplified architecture and omit some resources for illustration. Each ALU has 1 L0 register file, which has 3 registers r0-2, 1 input port i, and 2 output ports (merged with ALU input ports a and b). Squares are consumers which yet to have an incoming path. (b) is after scheduling instructions 1-5, and (c) is after finding a path from instruction 7 to 1. We cannot find any path from instruction 8 to instruction 2 ($\langle t-1, b\rangle$) in (c) because we used $\langle t-3, r0\rangle$ for instruction 5 in (b). (d) is an alternative schedule for instructions 1-5 that admits a feasible schedule for instruction 8.

instruction scheduling and register allocation simultaneously [12, 19, 17]. By merging two tightly coupled phases, the phase ordering problem is addressed, but careful consideration must be taken to guarantee forward progress of the unified algorithm. With instruction scheduling and register allocation unified, a resource commitment from a previous scheduling/allocation decision can prevent a feasible scheduling of the current instruction. Such cases are called *scheduling deadlock* in the literature [10, 8], and Figure 2 shows an example when we schedule the data flow graph shown in Figure 2(a). To avoid clutter, we assume a simple architecture with one ALU and 3 registers `r0-2` with 1 input and 2 output ports. In Figure 2(b)-(d), resources are duplicated at each time step to represent at which time a resource is used by an instruction. In Figure 2(b), we have scheduled instructions 1-5. Each path in Figure 2(b) denotes which path along DHRFs are used to transfer an operand from its producer to consumers. For example, the path $\langle t-4, \text{ALU}\rangle \rightsquigarrow \langle t-2, \text{ALU}\rangle$ specifies that `r0` is used at time step $t-3$ to transfer the result of instruction 5 to 3. In Figure 2(c), we have scheduled a path from instruction 7 to 1. But, we cannot find any path from instruction 8 to 2 because we used $\langle t-3, r0\rangle$ for instruction 5 in Figure 2(b). Figure 2(c) shows an alternative schedule for instructions 1-5 that admits a feasible schedule for instruction 8.

A natural question that arises from the example above is how we can avoid scheduling deadlocks and guarantee forward progress of unified allocation and scheduling algorithms. A simple backtracking when we encounter a deadlock is not satisfactory since it leads to an exponential running time in the worst case. This report presents a method for guaranteeing forward progress of unified algorithms, which induces a unified allocation and scheduling algorithm called *path finding scheduling* (PFS).

In PFS, we simultaneously schedule instructions and allocate registers by finding paths of operands to the ALUs. We model hardware resources and their connectivity with a graph data structure and find paths through the graph duplicated at each time step. Scheduling deadlocks are avoided by maintaining an invariant throughout the algorithm. The invariant is formulated as a maximum flow problem and ensures that existence of paths required for operands yet undelivered.

We have implemented PFS in our compiler back-end targeting an initial design of the elm architecture [1, 6]. We evaluate 6 kernels from embedded domain with our cycle-accurate simulator to verify that PFS successfully avoids deadlock while maintaining a polynomial time complexity.

The remainder of this report is structured as follows: Section 2 overviews PFS, and Section 3 details how scheduling deadlocks are avoided in PFS. Section 4 presents an evaluation of PFS. Section 5 reviews related work, and Section 6 concludes.

2 Algorithm Overview

Figure 3 provides an overview of the PFS algorithm, whose higher-level structure is similar to that of list scheduling [9]. We schedule one basic block at a time starting from the most frequently executed ones (line 2). Within a basic block, we visit each instruction in a reverse topological order (line 3): i.e., we visit consumers before their producer (e.g., instruction 1 and 5 before 7). Among the instructions following a reverse topological order, we select one on the critical path. The instructions of the data flow graph in Figure 2(a) are numbered according to such ordering. When we visit an instruction i, we find every available $\langle$time, ALU$\rangle$ pair m_i on which i can be executed (line 4). We give a precedence on m_is with later cycles as a heuristic to reduce the length of schedules (lines 6 and 10). For each m_i, we find a path from m_i to consumers of i. This is done by path-finding, the central part of PFS.

The high level structure of PFS is similar to that of list scheduling in that both schedule instructions in a topological order and use a tie-breaking heuristic such as critical-path-first. PFS is also similar to unified assign and schedule (UAS) [16] in that both assign instructions to functional units during instruction scheduling. However, PFS has two important distinc-

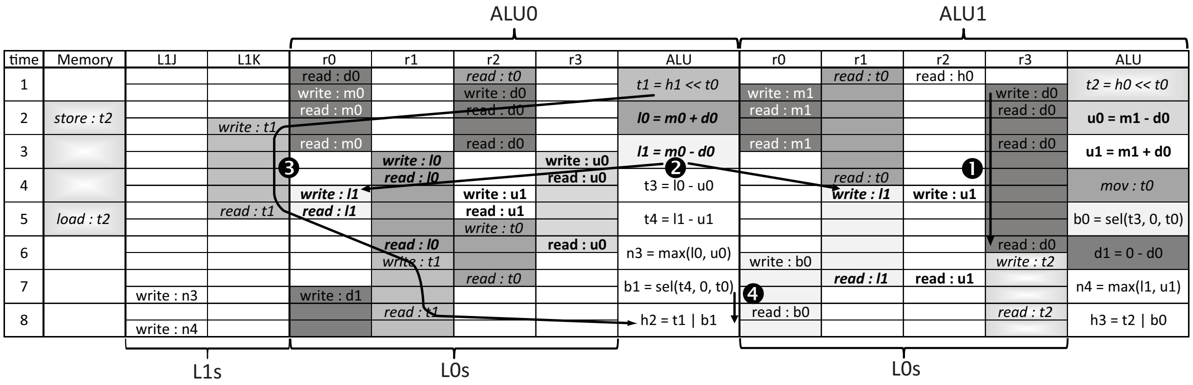
```
schedule = $\emptyset$
for each basic block in order of frequency
  for each instruction $i$ in a reverse topological order
    $P = \emptyset$, $M = \{m_i | \langle \text{time, ALU} \rangle\ i \text{ can be executed}\}$
    latest = latest time among $M$
    for each $t$ from latest to (latest - threshold)
      for each $m_i \in M$ such that the time step of $m_i$ is $t$
        path = GREEDY-PATH($i$, $m_i$)
        if VALID(schedule + path) then $P \cup = \{\text{path}\}$
      if $P \neq \emptyset$ then break
    if $P == \emptyset$ then $P \cup = \{\text{SAFE-PATH}(i, \text{schedule})\}$
    schedule += a min-cost path among $P$
```

Figure 3: Pseudo-code of the overall algorithm.

Table 1: Specific architecture parameters used for the Viterbi decoder example in Figure 4 and our evaluation.

ALU	2 ALUs
L0	4 registers per ALU. 2 in-ports and 2 out-ports
L1	2 L1s. Each L1 has 16 registers, 1 in-port, and 1 out-port

tions from them. First, *path-finding* models general data movement across distributed and hierarchical resources in finer details. UAS assigns instructions to functional units and inserts copy instructions between distributed register files when necessary, but register allocation is deferred to a separate phase. On the other hand, in PFS, register allocation is performed simultaneously and the data movement is not restricted to a copy between distributed register files. Figure 4 shows a scheduled code snippet from the inner loop of a Viterbi decoder [24]. The path denoted as ❸ in Figure 4 represents the data movement ALU → L1 → L0 → ALU. Note that, in this example, a register is allocated to a variable only for the duration when a path traverses the register: along the path ❸, an L0 register `ALU0.r1` is allocated to variable `t1` from time step 6 to 8. Consequently, path-finding naturally models migration between different levels of the register file hierarchy (i.e., spilling) and between ALUs. Furthermore,



				ALU0					ALU1				
time	Memory	L1J	L1K	r0	r1	r2	r3	ALU	r0	r1	r2	r3	ALU
1				read : d0		*read : t0*		*t1 = h1 << t0*		*read : t0*	read : h0		*t2 = h0 << t0*
				write : m0		write : d0			write : m1			write : d0	
2	*store : t2*			read : m0		read : d0		***l0 = m0 + d0***	read : m1			read : d0	**u0 = m1 - d0**
			write : t1										
3				read : m0		read : d0		***l1 = m0 - d0***	read : m1			read : d0	**u1 = m1 + d0**
					write : l0		**write : u0**						
4					***read : l0***		**read : u0**	t3 = l0 - u0		*read : t0*			*mov : t0*
				write : l1		**write : u1**				***write : l1***	**write : u1**		
5	*load : t2*		*read : t1*	***read : l1***		**read : u1**		t4 = l1 - u1					b0 = sel(t3, 0, t0)
						write : t0							
6					***read : l0***		**read : u0**	n3 = max(l0, u0)				read : d0	d1 = 0 - d0
					write : t1				write : b0			*write : t2*	
7						*read : t0*		b1 = sel(t4, 0, t0)		***read : l1***	**read : u1**		n4 = max(l1, u1)
		write : n3		write : d1									
8					*read : t1*			h2 = t1 | b1	read : b0			*read : t2*	h3 = t2 | b0
		write : n4											

Figure 4: Scheduled code snippet from the inner loop of a Viterbi decoder. The ⟨time step, resource⟩ pairs holding the same variable and the variable's producer have the same font and background color. Individual L1 registers are not shown for simple illustration.

path-finding can be extended to find a multi-cast tree as ❷ in Figure 4.

Second, each step of PFS systematically ensures that every unscheduled instruction can find a feasible schedule. Let us look again at the example shown in Figure 2 in more detail. We define a consumer as an *open consumer* when its incoming path is not yet scheduled. Figures 2(b)-(d) denote open consumers as squares. We have two open consumers in Figure 2(b), $\langle t-1, b\rangle$ and $\langle t, b\rangle$, which correspond to the edge from 8 to 2 and the edge from 7 to 1 shown in Figure 2(a), respectively. However, we cannot find disjoint incoming paths to both open consumers: an incoming path to $\langle t, b\rangle$ must pass $\langle t-2, r2\rangle$ as shown in Figure 2(c), but we cannot find such a path for $\langle t-1, b\rangle$ without $\langle t-2, r2\rangle$ (incoming paths to $\langle t-1, b\rangle$ through $\langle t-2, r0\rangle$ are blocked by $\langle t-3, r0\rangle$ and $\langle t-2, \mathrm{i}\rangle$). Therefore, the algorithm will make no progress when it tries to schedule instruction 8 (the producer of 2). Although an algorithm may backtrack to a previous instruction, backtracking can result in traversing back and forth along an exponentially large search space. The danger of exponential time complexity from backtracking is illustrated in the example shown in Figure 2: we cannot make forward progress with respect to instruction 8 not because of decisions made for the instruction scheduled immediately before. We call this non-progressing case *scheduling deadlock*, which is discussed in [10,8] in the context of RISC processors. Note that a scheduling deadlock does not happen from a cyclic dependency. Its meaning is that a scheduling algorithm cannot make any progress without backtracking because of a previous decision made in the algorithm.

A simple way to avoid scheduling deadlock is allocating registers before scheduling. However, if we allocate small L0s before scheduling, we are significantly restricted by the dependencies introduced by the allocation. On the other hand, if we allocate L1s before scheduling and conservatively enforce each instruction to read every source operand from L1s, we make little use of the register file hierarchy.

PFS by default does not pre-allocate any resource to avoid deadlocks, optimistically assuming that deadlocks will not happen. PFS finds a minimum cost path using the `GREEDY-PATH` function as shown in line 8 of Figure 3 pursuing the quality of schedule as much as possible without worrying about scheduling deadlocks. This optimism is complemented by checking the existence of disjoint incoming paths to open consumers, which is done by a function called `VALID`. If it detects that paths to an open consumer are blocked as in the case shown in Figure 2(b), `VALID` function in line 9 of Figure 3 returns `false`. In this case, PFS rolls back the path found by `GREEDY-PATH` and invokes `SAFE-PATH` (line 11), which always preserves the existence of disjoint paths. However, this is a rare case in practice, as will be shown in Section 4.

We define the terminology used in `GREEDY-PATH` as follows: An architecture is modeled by a *resource graph* G_R. Each vertex $r \in G_R$ represents a resource, which has latency $l(r)$ as its property. Cost $c(r)$ is another property which represents, for example, the scarcity or energy consumption of the resource. PFS produces energy-efficient code by finding paths with minimum costs. If a resource can hold data indefinitely, we call it a *storage resource* (e.g., registers and memory). Edges of the resource graph G_R represents the connectivity between resources. We define a *schedule graph* based on a resource graph as follows:

Definition 2.1 Schedule graph *G_S is a directed graph whose vertex set is a duplication of the vertex set of resource graph at each time step. Therefore, for each resource r in the resource graph, we have ..., $\langle t-1, r\rangle$, $\langle t, r\rangle$, $\langle t+1, r\rangle$, ... in G_S. The edge set of schedule graph is constructed as follows.*

For each time step t and edge from $r1$ to $r2$ in the resource graph G_R, we add a **move edge** *from $\langle t, r1\rangle$ to $\langle t+l(r1), r2\rangle$ in the schedule graph G_S.*

For each time step t and storage resource r in the resource graph G_R, we add a **storage edge** *from $\langle t, r\rangle$ to $\langle t+1, r\rangle$ in the schedule graph G_S.*

Figures 2(b)-(d) are examples of schedule graphs with some vertices omitted. A move edge represents data from a resource $r1$ at time t moving to a resource $r2$ at $t + l(r1)$ (e.g., $\langle t-1, ALU\rangle \rightarrow \langle t, \text{i}\rangle$ in Figure 2(b)). A storage edge represents data at a storage resource r that can be held there for multiple time steps (e.g., $\langle t-1, r0\rangle \rightarrow \langle t, r0\rangle$ in Figure 2(b)).

We implement GREEDY-PATH by running Dijkstra's shortest path algorithm to find a minimum cost path from a producer vertex to a consumer vertex in a schedule graph G_S. Since resource graph G_R is conceptually duplicated from -∞ to ∞ time steps, schedule graph G_S must be represented implicitly: vertices and edges are created on the fly when the algorithm actually visits them.

The next section details how invariant checking is implemented in the VALID function.

3 Invariant Preservation

The VALID function verifies that every unscheduled producer i can 1) find a ⟨time, ALU⟩ pair m_i on which i can be executed *and* 2) find disjoint paths to its consumers. For example,

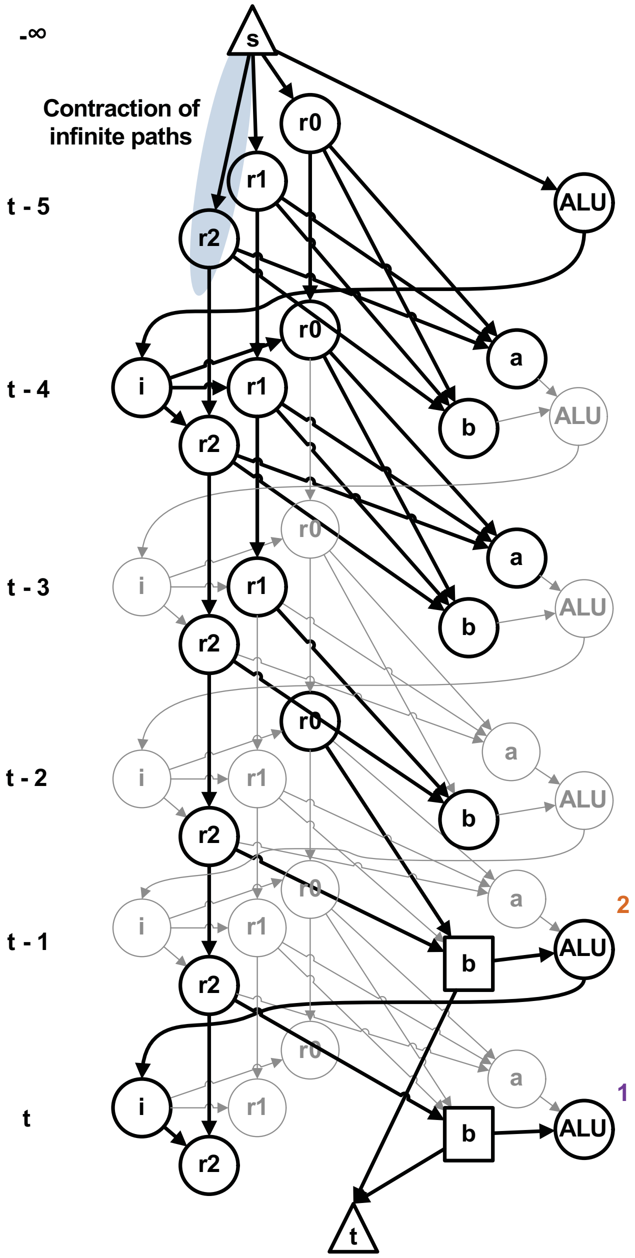


Figure 5: Invariant graph corresponding Figure 2(b). Gray vertices are used by instructions that are already scheduled, thereby not belonging to the invariant graph. Two squares $\langle t-1, b\rangle$ and $\langle t, b\rangle$ are open consumers.

VALID returns false at the case in Figure 2(b), where we cannot find disjoint paths to $\langle t-1, b\rangle$ and $\langle t, b\rangle$.

We formulate the invariant check as a maximum flow problem. A more accurate and straightforward formulation would be a vertex disjoint paths problem, but, it is a special case of an integer multi-commodity max-flow problem, which is NP-hard. The time complexity of this straightforward formulation arises from distinguishing paths from different producers. The basic idea of relaxing the NP-hard problem to a maximum flow problem is eliminating this distinction: we use a single vertex as the starting point for each incoming path to open consumers. In Figure 5, this single vertex is denoted as an auxiliary source s. The source vertex s is conceptually located at -∞ time. We also add an auxiliary sink t and connect each open consumer to t. We remove vertices used by instructions that are already scheduled and assign unit capacity to remaining vertices. We call the graph constructed by this procedure an *invariant graph*. If the max-flow value of the invariant graph is equal to the number of open consumers, we know that the number of paths with unit flow is equal to the number of open consumers [7], thus ensuring the existence of disjoint paths. As shown in Figure 5, we contract infinitely many vertices toward -∞ time direction in s so that the size of the invariant graph is proportional to the number of time steps of a basic block. We use the augmenting path maximum flow algorithm [7], and its time complexity is $O(km)$ for unit capacity graphs as in our case, where k is the number of open consumers and m is the number of edges in the invariant graph. The following formally defines invariant graph and proves the correctness of this max-flow formulation.

Definition 3.1 *Let t_{min} be the earliest time when at least one resource is used by an instruction (e.g., $t-4$ in Figure 2(b)).*

$u = \langle t, r\rangle \in G_S$ is an **exposed vertex** *if $t \geq t_{min}$, and is not already used by scheduled instructions.*

$u = \langle t, r \rangle \in G_S$ *is a* **boundary vertex candidate** *if it is not an exposed vertex but is unused and adjacent to an exposed vertex (e.g., vertices at* $t - 5$ *in Figure 5).*

A **boundary vertex set** V_B *is a maximal set of boundary vertex candidates such that, for any vertex u at -∞ time, its members have vertex disjoint paths from u.*

Definition 3.2 *We define a directed graph constructed by the following algorithm,* `BUILD-INVARIANT-GRAPH`*, as* **invariant graph** G_I*.*

```
procedure BUILD-INVARIANT-GRAPH($G_S$)
01  $V_E$ = exposed vertices of $G_S$
02  $V_B$ = a boundary vertex set of $G_S$
03  $G_I$ = the subgraph of $G_S$ induced by $V_E \cup V_B$
04  for each $\langle t, r \rangle \in V(G_I)$
05      $cap(\langle t, r \rangle)$ = ($r$ is a memory) ? $\infty$ : 1

06  $W$ = subset of $V_B$ occupied by live-in variables to the current basic block
07  $W$ ∪= subset of $V_B$ unreachable from the memory in $G_R$ - $W$
08  $G_I$ -= $W$

09  add auxiliary source $s$ with $\infty$ capacity
10  add edges $\{\langle s, u \rangle \,|u \in (V_B - W)\}$
11  add auxiliary sink $t$ with $\infty$ capacity
12  add edges $\{\langle c, t \rangle$ |$c$ is an open consumer$\}$
```

Vertices representing the memory have ∞ capacity (line 5) because they are contractions of conceptually infinite memory locations. The auxiliary source has ∞ capacity (line 9) because it is a contraction of an infinite number of vertices toward the -∞ time direction. Other vertices have unit capacity to be used exclusively. We need lines 6-8 since decreasing t_{min} does not release the resources held by live-ins.

Corollary 3.1 *If the maximum flow value on G_I is equal to the number of open consumers, it implies that every unscheduled producer i can 1) find a $\langle$time, ALU$\rangle$ pair m_i on which i can be executed and 2) find disjoint paths to its consumers.*

Proof of Corollary 3.1. Given an open consumer c (e.g., $\langle t, b\rangle$ in Figure 5), assume that there is a vertex disjoint path from a boundary vertex u to c, say p_1 (e.g., $\langle t-5, r2\rangle \rightsquigarrow \langle t-1, r2\rangle \rightarrow \langle t, b\rangle$), where u is a boundary vertex $\in V_B - W$. There always exists such a vertex disjoint path if the max-flow value on G_I is equal to the number of open consumers.

Since $u \in V_B - W$, u is reachable from the memory in $G_R - \{$subset of V_B occupied by live-ins$\}$, thus there are an infinite number of paths from s to u in G_S, which are contracted in the edge from s to u (e.g., $s \rightarrow \langle t-5, r2\rangle$). In addition, even after assigning one incoming path for each boundary vertex other than u, we still have infinite number of available paths from s to u by the definition of boundary vertex set. Since we use a reverse topological order, we can indefinitely decrease the execution time of c's producer, say i. Therefore, we have an infinite number of paths $m_i \rightsquigarrow u$ such that i can be executed at m_i. Among these infinite paths, we can choose a vertex disjoint path, say p_2.

Concatenation of p_1 and p_2 is again a vertex disjoint path. □

The `VALID` function executes max-flow on an invariant graph, and returns `false` if and only if the max-flow value is less than the number of open consumers. For example, `VALID` function returns `false` for the invariant graph in Figure 5 because we can augment only one path from s to t.

With the `VALID` function using max-flow, we can check if `GREEDY-PATH` returns a path that violates the invariant. However, `GREEDY-PATH` can continuously violate the invariant in a pathological case. Although this case is rare, we need another function `SAFE-PATH` to guarantee the termination of PFS in a polynomial time. The pseudo-code of `SAFE-PATH` is as follows.

```
procedure SAFE-PATH(instruction $i$, schedule)
1 $G_I$ = BUILD-INVARIANT-GRAPH($G_S$ of schedule)
2 MAX-FLOW($G_I$, $s$, $t$)
3 $G$ = a subgraph of $G_S$ corresponds to all the augmented paths
    resulted from MAX-FLOW except for those going to consumers of $i$
4 $G_S$ -= $G$
5 for each $m_i = \langle$time, ALU$\rangle$
6   path = GREEDY-PATH($i$, $m_i$)
7   if VALID(schedule + path) return path
```

The basic idea of this `SAFE-PATH` implementation is that when we schedule an instruction, we yield vertices for disjoint paths to other consumers, then use remaining vertices. For example, after `VALID` returns `false` at the case shown in Figure 2(b), `SAFE-PATH` yields $\langle t-3, r2\rangle$ and $\langle t-3, r0\rangle$ to instructions 1 and 2, then it finds a path from instruction 5 to 3 through $\langle t-3, r1\rangle$ as shown in Figure 2(d). Note that `SAFE-PATH` inverts the priority between the current instruction and instructions that will be scheduled later. Thus, it likely finds a sub-optimal path for the current instruction, but it is executed rarely as will be shown in Section 4. The correctness of `SAFE-PATH` can be proved as follows:

Corollary 3.2 *The pseudo-code of* `SAFE-PATH` *is correct.*

Proof of Corollary 3.2. Let C be the set of open consumers except consumers(i), i.e., consumers of i. Since G, a subgraph of G_S, contains augmented paths to C, it solely satisfies the invariant with respect to C. `GREEDY-PATH` in line 6 finds a path without G. Therefore, G_S of "schedule + path" in line 7 contains G, thus satisfying the invariant with respect to C.

After line 4, G_S contains augmented paths to consumers(i), so it satisfies the invariant with respect to consumers(i). Also, since we use a reverse topological order, we can indef-

Table 2: Kernels used in evaluation. "# of inst." is counted in the LLVM IR. "max # of inst." denotes the number of instructions in the biggest basic block.

Kernel	Description	# of BBs / # of inst. / max # of inst.
aes	Advanced Encryption Standard	5 / 65 / 30
conv2d	2D convolution with 3×3 Sobel filter.	13 / 69 / 35
crc32	IEEE 802.3 Cyclic Redundancy Check	7 / 36 / 17
dct	8×8 2D Discrete Cosign Transform	7 /112 / 52
huffman	Huffman encoder used in JPEG	14 / 74 / 15
viterbi	Viterbi decoder in the GSM standard	18 /124 / 22

initely decrease the execution time of i. Therefore, for at least one m_is enumerated in line 5, `GREEDY-PATH` must be able to find vertex disjoint paths to consumers(i) and the invariant still holds with respect to $C \cup C_{new}$, where C_{new} are new open consumers created by i's operands.

An exponential number of iterations in line 5 can be avoided by doubling the time step after an appropriate constant number of iterations. □

4 Results

We have implemented PFS in our compiler back-end written in Java. We compile C source code using the LLVM [11] compiler infrastructure with -O3 option, which applies comprehensive traditional optimizations. Our compiler back-end transforms platform-independent LLVM intermediate representation (IR) into our IR.

We evaluate 6 embedded kernels shown in Table 2 using our cycle-accurate simulator. The same set of kernels are hand-assembled in [1, 6] for its architecture evaluation. We use the architecture configuration shown in Table 1 in Section 2.

PFS executes `SAFE-PATH` only twice when compiling these 6 kernels, once in `huffman` and once in `viterbi`. We executed `SAFE-PATH` after trying `GREEDY-PATH` for each ⟨time, ALU⟩

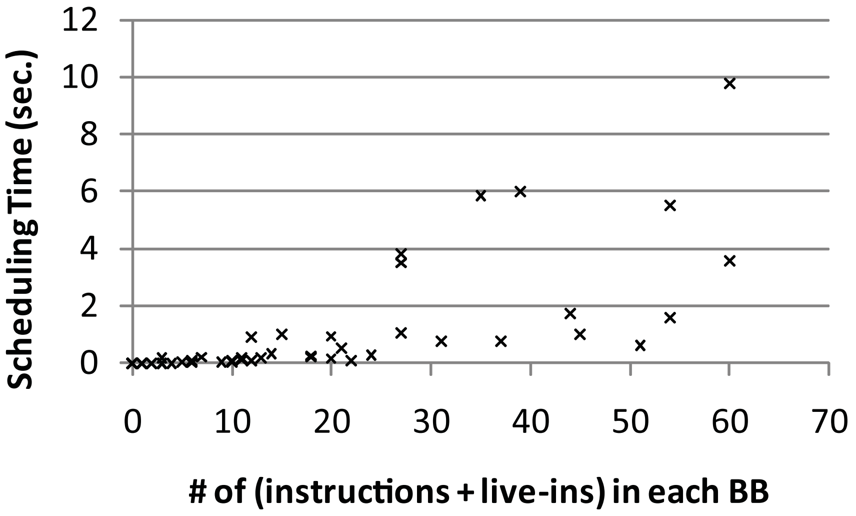


Figure 6: Scheduling times for basic blocks with different sizes.

pair at the latest cycle: i.e. we used 0 as the threshold value of line 6 in Figure 3. In code produced by PFS, only 4 L0 registers per ALU provide 63% of the total access, showing that there is abundant locality suitable for L0s and that PFS efficiently exploits the locality. Explicit bypassing between ALUs constitutes 8% of the total access count. Figure 6 demonstrates that PFS scales well as basic block sizes increase.

5 Related Work

Unified assign and schedule (UAS) [16] and PFS both simultaneously assign ALUs and schedule instructions. Path-finding can be seen as a generalization of inserting copies done in UAS. PFS finds data movement not only between ALUs but also between different levels of the register file hierarchy. Moreover, PFS unifies *register* allocation with instruction scheduling, while UAS unifies *functional unit* allocation with instruction scheduling.

Communication scheduling is proposed in [12] for an architecture with limited connectivity. The term open consumer in PFS was adopted from the concept named *opening* in [12]. However, the forward progress of communication scheduling relies on the fact that the target architecture provides the all-to-all connectivity between distributed register files.

A bypass-aware scheduling algorithm is proposed in [17] for the ARM ISA in order to maximize the number of bypassing and save the energy consumed for reading register files. PFS targets an architecture in which the compiler can explicitly control bypass paths, hence having more optimization opportunities by completely avoiding the allocation of the bypassed variable to a register — we not only can save energy consumed for reading register files, but also can save write energy and capture more data in register files by reducing register pressure.

In [19], the authors describe a scheduling algorithm for processors with customized pipelines that uses path-finding for simultaneous scheduling and allocation. However, their algorithm assumes that register files are fully connected to every functional unit, thereby not needing to avoid scheduling deadlocks.

Park el al. [17] describes a scheduling algorithm for reconfigurable architectures that integrate placement and routing, which is similar to PFS's integration of register allocation and instruction scheduling. They also point out routing failures due to resource commitments of already scheduled instructions. They suggest a method for proactively avoiding routing failures that associates resource and time pairs with occupancy probabilities that represent the probabilities of being occupied by instructions to be scheduled. However, due to its probabilistic nature, this method only reduces the likelihood of scheduling deadlocks and cannot guarantee the forward progress.

A hierarchical register file organization is introduced in [25], but it is mainly concerned with capturing more data in register files to increase instruction-level parallelism. On the other hand, DHRF focuses on energy efficiency, thus exposing existing pipeline registers as small register files attached to each ALU instead of adding larger higher-level register files.

6 Conclusion

This report presents path-finding scheduling algorithm that effectively utilizes small lower-level register files in a distributed and hierarchical register organization. Its termination in a polynomial time is ensured by an invariant formulated as a maximum flow problem, and its effectiveness is achieved by optimistically finding minimum cost paths. Evaluation of larger applications remains as future work.

References

[1] J. Balfour, W. J. Dally, D. Black-Schaffer, V. Parikh, and J. Park. An Energy-Efficient Processor Architecture for Embedded Systems. *Computer Architecture Letters*, 7(1), 2008.

[2] J. Balfour, R. C. Harting, and W. J. Dally. Operand Registers and Explicit Operand Forwarding. *Computer Architecture Letters*, 8(2):60–63, 2009.

[3] D. Black-Schaffer, J. Balfour, W. J. Dally, V. Parikh, and J. Park. Hierarchical Instruction Register Organization. *Computer Architecture Letters*, 7(2):41–44, 2008.

[4] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. In *International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 122–131, 1991.

[5] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *International Symposium on Microarchitecture (MICRO)*, pages 292–300, 1992.

[6] W. J. Dally, J. Balfour, D. Black-Schaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient Embedded Computing. *IEEE Computer*, 41(7):27–32, 2008.

[7] L. Ford, Jr. and D. R. Fulkerson. Maximal Flow Through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[8] P. B. Gibbons and S. S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architectures. *ACM SIGPLAN Notices*, 2(7):11–16, 1986.

[9] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[10] J. L. Hennessy and T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):422–448, 1993.

[11] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Aanalysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[12] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Owens. Communication Scheduling. In *International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 82–92, 2000.

[13] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures. In *International Conference on Field-Programmable Technologies*, pages 166–173, 2002.

[14] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architectures with Configurable Instruction Distribution and Deployable Resources. In *IEEE Symposium on FPGAs for Custom Computing Machines*, page 157, 1996.

[15] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[16] E. Özer, 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 (MICRO)*, pages 308–315, 1998.

[17] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. seok Kim. Edge-centric Modulo Scheduling for Coarse-Grained Reconfigurable Architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 166–176, 2008.

[18] S. S. Pinter. Register Allocation with Instruction Scheduling: a New Approach. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 248–257, 1993.

[19] M. Reshadi and D. Gajski. A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths. In *International Conference on hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 21–26, 2005.

[20] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. Register Organization for Media Processing. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 375–386, 2000.

[21] O. Silvén and K. Jyrkkä. Observations on Power-Efficiency Trends in Mobile Communication Devices. In *Proceedings of the 5th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2005.

[22] 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 (ISCA)*, page 2, 2004.

[23] J. H. Tseng and K. Asanović. Energy-Efficient Register Access. In *Symposium on Integrated Circuits and Systems Design*, page 377, 2000.

[24] A. J. Viterbi. Error Bounds for Convolutional Cdoes and an Asymptotically Optimum Decoding Algoritm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[25] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level Hierarchical Register File Organization for VLIW Processors. In *International Symposium on Microarchitecture (MICRO)*, pages 137–146, 2000.