Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures

Ramadass Nagarajan Sundeep K. Kushwaha Doug Burger Kathryn S. McKinley
Calvin Lin Stephen W. Keckler

Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart

Abstract

Technology trends present new challenges for processor architectures and their instruction schedulers. Growing transistor density will increase the number of execution units on a single chip, and decreasing wire transmission speeds will cause long and variable on-chip latencies. These trends will severely limit the two dominant conventional architectures: dynamic issue superscalars, and static placement and issue VLIWs. We present a new execution model in which the hardware and static scheduler instead work cooperatively, called Static Placement Dynamic Issue *(SPDI). This paper focuses on the static instruction scheduler for SPDI. We identify and explore three issues SPDI schedulers must consider—locality, contention, and depth of speculation. We evaluate a range of SPDI scheduling algorithms executing on an* Explicit Data Graph Execution *(EDGE) architecture. We find that a surprisingly simple one achieves an average of 5.6 instructions-per-cycle (IPC) for SPEC2000 64-wide issue machine, and is within 80% of the performance without on-chip latencies. These results suggest that the compiler is effective at balancing on-chip latency and parallelism, and that the division of responsibilities between the compiler and the architecture is well suited to future systems.*

1 Introduction

Technology trends are forcing dramatic changes in architectures and their accompanying software. Architects are proposing heavily partitioned execution substrates with many ALUs to deal with limits in pipeline depth, clock rate growth [13, 30], and slowing of on-chip wires [1, 12]. To achieve high performance, these architectures must execute many instructions in parallel and tolerate multi-cycle on-chip delays between dependent producing and consuming instructions. The solution this paper explores is to expose placement of instructions on a grid of ALUs to the compiler, thereby allowing schedulers to expose parallelism and to minimize the physical distances that operands must travel. Instruction scheduling is a mature area that researchers have studied extensively in the context of conventional VLIW and superscalar architectures. Researchers have particularly focused on VLIW schedulers because of the central role that they play in obtaining good VLIW performance.

However, conventional architectures and their schedulers are ill equipped to solve these emerging problems. VLIW architectures [7, 9, 14, 18, 26], including EPIC architectures [29], use an execution model in which the compiler chooses both the order and the ALU on which to issue each instruction. Despite techniques to move instructions across branches, such as predication [22], trace scheduling [8], and region formation [11], instruction schedulers often cannot find enough instructions to pack into each VLIW instruction. In addition, instructions with unpredictable latency, such as load misses, stall the entire VLIW execution engine.

Conversely in the superscalar execution model, the hardware dynamically chooses from a finite issue window both the execution order and the placement of instructions onto ALUs. Centralized superscalar processor are limited because of the quadratic hardware required for both the issue width and window size [24, 1] to check for data dependences and broadcasts results. While researchers have proposed clustered (partitioned) superscalar architectures to improve their scalability, the hardware instruction steering algorithms are limited by the necessarily local scope of the dynamic placement hardware. In addition, the instruction-set-architecture cannot encode instruction placement, which cripples the scheduler's effectiveness.

We observe that the scheduling problem can be broken into two components: *instruction placement* and *instruction issue*. VLIW processors use both static issue (SI) and static placement (SP), so we classify them as using an SPSI scheduling model. Out-of-order superscalar architectures, conversely, use a DPDI model: they rely on both dynamic placement (DP), since instructions are dynamically assigned to appropriate ALUs, and dynamic issue (DI).

For VLIW processors, the static issue is the limiting factor for high ILP, whereas for superscalar processors, poor dynamic placement limits their ability to handle growing wire delays. On the other hand, static placement makes VLIW a good match for partitioned architectures, and dynamic issue permits superscalar processor to exploit parallelism and tolerate uncertain latencies. This paper studies scheduling algorithms for machines that combine the strengths of the two scheduling models, coupling static placement with dynamic issue (SPDI).

Explicit Dataflow Graph Execution (EDGE) instruction set architectures present an SPDI scheduling model [3]. These architectures have one defining characteristic: *direct instruction communication*. The ISA directly expresses the dataflow graph of program segments by specifying in an instruction the physical location of its consumers. Thus, instructions communicate directly with one another in the common case, rather than conveying the dependences through a shared name space, such as a register file. In an EDGE architecture that uses the SPDI scheduling model [1] instructions execute in dataflow order, with each instruction issuing when its inputs become available. SPDI EDGE architectures thus retain the benefits of static placement. At the same time, they permit dynamic issue, without the need for complex associative issue window searches, large multiported register files, and bypass networks.

The SPDI scheduling model relies on the compiler to select the ALU on which an instruction will execute, but it allows the hardware to issue the instruction dynamically as soon as its operands become available. The SPDI scheduler must balance the reduction of communication latencies—on a topology with non-uniform latencies to different resources—with the exposure of instruction-level parallelism by both minimizing ALU contention and maximizing *instruction density*, the number of instructions that can be fit into the issue window. We show that on one SPDI EDGE machine (the TRIPS Architecture [27]), a surprisingly simple greedy scheduling algorithm can yield good results with an appropriate set of placement heuristics, improving performance by 29% over a naive greedy scheduler. We also show that this simple algorithm outperforms a much more complex scheduling algorithm, the recently developed convergent scheduling framework [20] both tuned with the same heuristics. On at 64-wide issue machine, we achieve 5.6 IPC and are within 80% of the performance of an idealized machine without on-chip latency. We conclude that by separating placement and issue, SPDI schedulers for EDGE architectures can mitigate ILP losses due to wire-dominated communication on a high-ILP substrate.

The remainder of this paper is organized as follows. Section 2 draws the distinction between VLIW and EDGE/SPDI scheduling and describes the variant of the TRIPS architecture that provides context for this paper. Section 3 describes the baseline top-down greedy scheduling algorithm and presents instruction placement optimizations to reduce communication latency, while exploiting parallelism. Section 4 presents our evaluation methodology. Section 5 evaluates different placement optimizations and determine a good set of heuristics for an EDGE SPDI scheduler, computes a performance upper bound, compares our scheduler with convergent scheduling, and explores schedule density versus latency minimization. We contrast the EDGE SPDI scheduler with previous work in Section 6.

2 Static-Placement Dynamic-Issue Schedulers

An SPDI scheduling model couples compiler-driven placement of instructions with hardware-determined issue order, striking a balance between the runtime flexibility of superscalar processors (DPDI) and the hardware simplicity of VLIW processors (SPSI).

To demonstrate the principal differences between VLIW and SPDI scheduling and execution, Figure 1 compares these two models using a simple two-ALU architecture. The left column shows a sequence of instructions and its corresponding dataflow graph. The VLIW scheduler assigns the instructions to time slots within the ALUs and uses NOP placeholders to enforce the pipeline hazards, which in this case are due to a predicted one-cycle cache hit latency on the load `(LD)` instructions. Assuming that both `LD` instructions miss in the cache and incur a two-cycle miss penalty, the VLIW processor must stall the pipeline at or before the consuming instruction, resulting in a nine-cycle execution time for this sequence. The SPDI scheduler instead packs the instructions, eliminating the VLIW placeholders, and relies on the dynamic issue logic to determine when instructions should issue. At runtime, the issue logic only stalls those instructions that depend on a delayed data value, thus allowing the two `LD` instructions to execute concurrently and to overlap their miss handling times. The resulting execution time is only seven cycles.

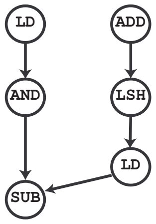
While both VLIW and SPDI schedulers place instructions, the SPDI execution model provides certain advantages to the EDGE architecture. First, hardware hazard detection logic enables the scheduler to produce more space-efficient schedules; NOP placeholders are not required to reserve issue time slots. Second, the scheduler need not consider unpredictable instruction latencies, as instruction issue on different execution units is decoupled, allowing independent instructions to continue execution even if other instructions must wait for their producers to complete. In

[1]EDGE architectures that use other execution models are certainly possible; for example, WaveScalar [32] can be characterized as a DPDI EDGE architecture. The RAW architecture [34] is arguably an SPSI EDGE architecture, since it supports direct communication from one ALU to another on a remote tile.

Original Program

```
LD R1, R2, #0
AND R1, R1, R4
ADD R3, R4, R5
LSH R3, R3, #3
LD R6, R3, #0
SUB R7, R1, R6
```

Dataflow Graph



VLIW Schedule

```
LD R1, R2, #0     ADD R3, R4, R5
                  LSH R3, R3, #3
AND R1, R1, R4    LD R6, R3, #0

SUB R7, R1, R6
```

VLIW Execution

```
cycle 1  LD R1, R2, #0     ADD R3, R4, R5
      2                    LSH R3, R3, #3
      3
      4
      5  AND R1, R1, R4    LD R6, R3, #0
      6
      7
      8
      9  SUB R7, R1, R6
```

SPDI Schedule

```
LD R1, R2, #0     ADD R3, R4, R5
AND R1, R1, R4    LSH R3, R3, #3
SUB R7, R1, R6    LD R6, R3, #0
```

SPDI Execution

```
cycle 1  LD R1, R2, #0     ADD R3, R4, R5
      2                    LSH R3, R3, #3
      3                    LD R6, R3, #0
      4
      5  AND R1, R1, R4
      6
      7  SUB R7, R1, R6
```

Figure 1. VLIW versus SPDI scheduling and execution.

fact, the SPDI model allows independent instructions on the same execution unit to issue in any order as long as its operands are ready, further freeing the scheduler to make judicious placement decisions that improve performance and reduce contention. While the SPDI model eases certain aspects of scheduling, partitioned execution substrates impose additional demands.

2.1 Execution Substrate

The TRIPS architecture is an instantiation of an EDGE ISA that employs the SPDI execution model. Figure 2 shows the components of a 4×4 TRIPS processor [27] with 16 instruction execution units connected via a thin operand routing network. The diagram illustrates a 2-D mesh routing network. Passing values between ALUs in the mesh requires a delay that depends on the distance traveled in the network. The instruction cache, register file, and data cache are on the perimeter of the ALU array. Each ALU includes an integer unit, a floating point unit, an operand router, and an instruction buffer (storage) for multiple instructions and their operands. The scheduler specifies to the architecture which instructions to place in each ALU's buffer. Instructions in a buffer may execute in any order after the operands arrive.

The TRIPS compiler generates hyperblocks [22] and schedules each hyperblock independently. Figure 3 shows the instructions of a hyperblock scheduled onto a 2×2 execution node array (Figure 1 shows the same hyperblock.) Three *read* instructions obtain the register values and forward them to their consuming instructions. Similarly a *write* instruction places values produced by the hyperblock into a register. Figure 3a shows the placement of each instruction, and Figure 3b shows the encoding. Instructions do not encode their source operands — they encode only the physical locations of their dependent instructions. For example, the `add` instruction placed at location [0,1,0], upon execution, forwards its result to the `LSH` instruction placed at location [1,1,0].

The hardware maps the hyperblock to the execution array, reads the input registers from the register file, and injects them into the appropriate ALUs. These values trigger the firing of instructions, which on completion then distribute their results to consumer ALUs through the operand network. Instructions that produce register outputs write their values back to the register file. The hardware transmits temporary values that are only live within a block directly from producer to consumer, without writing them to the register file. Address computations for load and store instructions execute within the grid and then transmit the addresses (and data values for stores) to the data cache banks. The cache banks send the loaded values back into the execution array via the operand network.

Each ALU contains a fixed number of instruction buffer slots. We refer to corresponding slots across all ALUs collectively as a *frame*. A 4×4 grid with 128 instruction buffer entries at each ALU thus has 128 frames of 16 instructions each. A subset of contiguous frames constitutes an *architecture frame* (A-frame), into which the compiler schedules all instructions from a hyperblock. For example, dividing 128 frames into 8 A-frames composed of 16 physical frames each allows the scheduler to map a total of 256 instructions (per hyperblock) at once to the ALU array.

Similar to control speculation in superscalar processors, the TRIPS microarchitecture speculatively selects subsequent hyperblocks to execute and maps and executes them concurrently with the non-speculatively executing hyperblock. These blocks are mapped into the A-frames not used by the non-speculative hyperblock. The number of instructions spanning the non-speculative and speculative A-frames corresponds to the size of the architecture's dynamic scheduling window. In superscalar processors, this window is centralized and relatively small (80-100 instructions), while in a TRIPS-like microarchitecture this window is distributed and can range from hundreds to tens of thousands of instructions.

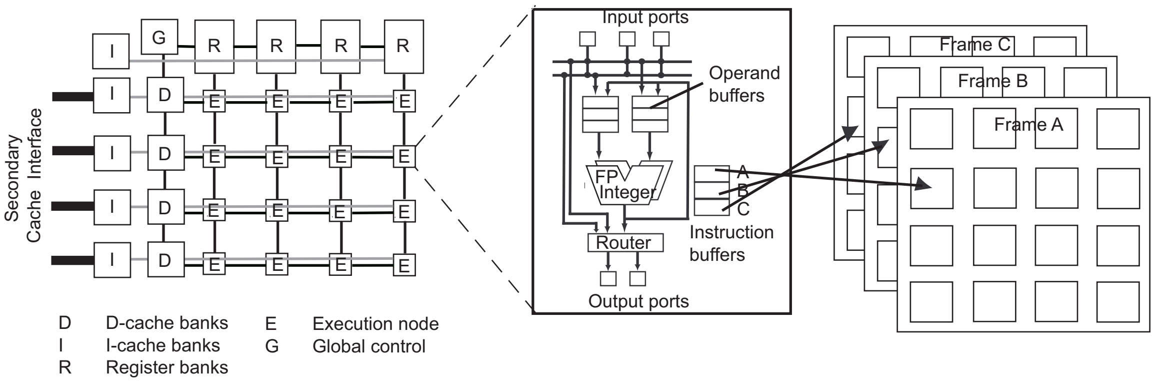


Figure 2. Example $4{\times}4$ **TRIPS Processor.**

Since the number of frames must be fixed, the hardware may present a trade-off between A-frame size and the number of A-frames available for speculation. In the prototype TRIPS architecture, A-frames sizes are fixed, with 8 frames per 128-instruction A-frame. In this paper, we explore a variant of the TRIPS architecture in which the compiler may specify the number of frames in an A-frame. In this variant, the microarchitecture can map as many hyperblocks onto the hardware as the total number of frames can support. For example, if four successive hyperblocks used three, four, five and six frames respectively, a machine with 16 frames can concurrently execute only three of those blocks. We simulate a finite number of A-frames, however, so the hardware can be running concurrently at most the same number of hyperblocks as there are A-frames.

This variant TRIPS architecture exposes the trade-off of frames per hyperblock to the scheduler. If the scheduler chooses to use more frames for a given hyperblock, with the goal of improving the execution time on a per-hyperblock basis, the degree of speculation may be throttled, since the same number of instructions will be allocated in more frames, permitting fewer total hyperblocks to be running concurrently. Conversely, an emphasis on deeper speculation (more speculative hyperblocks executing concurrently) reduces the number of frames available per hyperblock, which may degrade per hyperblock execution time by increasing communication latencies, but may improve overall performance. We explore this tradeoff further in Section 5.3.

2.2 SPDI Scheduler Duties

Since the architecture exposes the interconnection topology and delay, the scheduler has several additional duties:

Placement for Locality: The topology and connectivity of the network combined with the physical distances among the ALUs, the register file, and the cache banks determine run-time operand latency. Maximizing performance of this architecture requires that the scheduler select an instruction mapping that minimizes communication latencies among

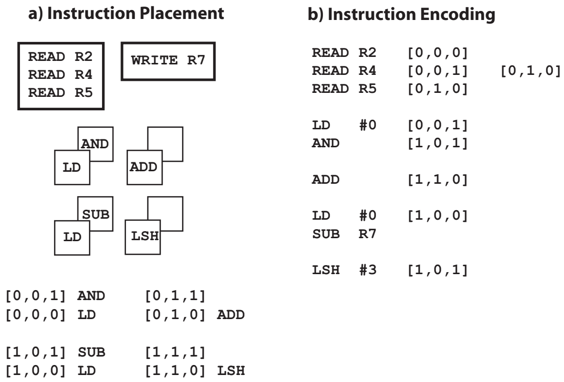


Figure 3. TRIPS Instruction Encodings

ALUs, the register file, and cache banks for dependent instructions.

Contention: Since the execution units can issue only one instruction per cycle and since the inter-ALU operand network has limited bandwidth, issue and network contention can degrade performance. The scheduler can reduce contention by spreading independent instructions across the ALUs, but must balance this benefit against the goal of reducing communication latencies.

Effective window size: Since the number of instruction slots (frames) is fixed, the hardware and software must balance the architecture frame size with the number of speculative frames. Larger architecture frames may enable lower communication latencies since it gives the scheduler freedom in placing instructions. More instructions per architecture frames allows the architecture to speculatively execute additional frames to exploit ILP. The challenge for the scheduler is to create low cross-chip latency schedules while allowing enough speculative frames for ILP. While this requirement may seem somewhat specific to block-structured EDGE architectures, the interaction between scheduling and speculation is likely to exist in other future systems as well.

3 SPDI Scheduling Algorithms

The SPDI scheduling algorithm takes as input a group of instructions and a processor model description. The model includes execution latencies for different types of instructions, interconnect topology, and communication latencies. The scheduler outputs an assignment of instructions to ALUs. We first describe a simple extension of a VLIW scheduler that serves as the baseline SPDI scheduler. We then augment the algorithm with several SPDI heuristics.

3.1 Base Algorithm

The baseline SPDI scheduling algorithm resembles a simple VLIW scheduler. It computes the initial set of ready instructions, all of which can issue in parallel. It prioritizes instructions in the ready set and selects the highest priority instruction for scheduling. After it schedules an instruction *i*, it adds to the ready set any of *i*'s children whose parents have all been scheduled. It selects the next instruction for scheduling and iterates until completion. While a SPSI VLIW scheduler assigns an instruction to an ALU and a time slot, a SPDI scheduler assigns each instruction to an ALU without specifying a time slot. Figure 4(a) shows the basic algorithm.

The algorithm first determines the number of instruction slots needed to schedule a group of instructions. For our TRIPS derivative, the scheduler determines the number of A-frames needed to accommodate the hyperblock. For example, a hyperblock with 150 instructions requires 3 frames on an 8×8 array of ALUs (because $2*(8\times8) < 150 < 3*(8\times8)$). It then sorts the instructions in a top-down greedy order as described in the previous paragraph. For example, the sorted order for instructions shown in Figure 1 would be `ADD, LD, LSH, AND, LD, and SUB`. The scheduler obtains this order by sorting the instructions based on their depths from the root instructions in the dataflow graph (DFG) and breaking ties by further sorting the instructions based on their heights from the bottom instructions in the DFG. To compute the depths and heights, the scheduler assumes static instruction latencies and cache hits. The sorted order of instructions is represented by the list `S` in Figure 4.

For the unscheduled instruction *i* with the highest priority, the scheduler computes the set of legal instruction slots, *R*, which the interconnection topology defines. Using a mesh interconnect, all instruction slots are reachable from all others, but simpler networks with limited reachability are possible. If no instruction slot is available, the scheduler adds to the total pool by increasing the number of frames by one, and starts over.

For every legal instruction slot *rs*, the scheduler computes the following expression:

$$ReadyTime(i,rs) = \max_{\forall p}\{CompleteTime(p,rs_p) + Distance[rs_p,rs]\}$$

The term p denotes a parent of i. $CompleteTime(p,rs_p)$ refers to the expected time at which p will produce its results at rs_p, and $Distance[rs_p,rs]$ denotes the number of communication hops required to route p's result to rs. $ReadyTime(i,rs)$ is simply the earliest time at which i can issue at rs. To schedule i, the scheduler chooses the instruction slot rs_{min} at which $ReadyTime$ is minimum. In case of ties, it selects the location at the right most column in the top most row.

The scheduler then marks *i* as scheduled and updates the following expression:

$$CompleteTime(i,rs_{min}) = ReadyTime(i,rs_{min}) + Latency(i)$$

The process iterates until all instructions are scheduled.

3.2 Scheduler Optimizations

We now augment the base scheduler with five heuristics that balance increasing parallelism and reducing latency. The first goal attempts to place independent instructions on different ALUs, while the second attempts to schedule consumers physically close to producers. Figure 4(b) shows the scheduling algorithm augmented with the five heuristics described below.

Critical path priority (C): This heuristic attempts to prioritize instructions by critical path first over parallelism, with the intuition that reducing the communication latencies along the static critical path will improve overall performance. It achieves this by sorting the instructions based on the maximum depth of any of their descendants. In case of ties, it further sorts them based on their heights in the dataflow graph. For example, the sorted order for instructions in Figure 1 would be `ADD, LSH, LD, SUB, LD, and AND`. This strategy provides the advantage of first selecting every instruction on the critical path for scheduling.

Load balancing (B): To reduce contention for an ALU and maximize parallelism, the scheduler places on different ALUs instructions that it expects to be ready at the same time. The scheduler keeps track of the cycles when an ALU is expected to be busy executing instructions and when it is free. These estimates do not bind the dynamic issue of instructions; instead they help the compiler judiciously place instructions and reduce contention. For the candidate instruction *i* and instruction slot *rs*, the scheduler computes the term $Contention(i,rs)$, which denotes any additional delay cycles between when *i* is ready and when it can be issued from *rs*. The scheduler then computes $CompleteTime(i,rs)$ as shown in Figure 4(b).

Data cache locality (L): The scheduler reduces load latencies by placing loads and consumers of loads close to the data caches. It augments the dataflow graph with a pseudo-memory instruction *m* and fixes the placement of *m* at a position *M* that is one hop away from the rightmost column of the execution array. For example, a dependence edge in the DFG, $A \rightarrow B$, where A is a load instruction, is changed to $A \rightarrow m \rightarrow B$, where m is the pseudo-memory instruction.

```
Input: Hyperblock Instruction Set H, ALU set A
Output: H -> A

#Frames = ceil(|H|/|A|)
S = top_down_greedy_sort(H)
foreach instruction i in sorted list S {
     R = find_legal_instruction_slots(i)
     if |R|=0 {
          #Frames++, Reschedule()
     } else {
          foreach rs in R {
               compute ReadyTime(i,rs)
          }
          E = sort_by_ready_time(R)
          Schedule(i) = first_element(E)
          S = S-{i}
     }
}
```

(a) Base Algorithm

```
Input: Hyperblock Instruction Set H, ALU set A
Output: H -> A

#Frames = ceil(|H|/|A|)
S = top_down_criticality_sort(H)          C
foreach instruction i in sorted list S {
     R = find_legal_instruction_slots(i)
     if |R|=0 {
          #Frames++, Reschedule()
     } else {                                                        B
          foreach rs in R {
               compute ReadyTime(i,rs)
               IssueTime(i,rs) = ReadyTime(i,rs)+Contention(i,rs)
               CompleteTime(i,rs) = IssueTime(i,rs)+Latency(i)
               Score(i,rs) = CompleteTime(i,rs)+Lookahead(i,rs)*0.5
          }
          E = sort_by_score(R)
          Schedule(i) = first_element(E), H = H-{i}                 O
          S = top_down_criticality_sort(H)           R
     }
}
```

(b) Scheduling Algorithm with Optimizations

Figure 4. Scheduling Algorithms.

$CompleteTime(M,m)$ is calculated by summing the latency to route the address to the cache and the latency of a cache hit.

Register Output (O): Instructions that produce register outputs ought to be placed close to the register file. However, prior placed instructions may constrain the placement of these instructions by occupying these desired locations. To avoid this situation, the lookahead heuristic, in anticipation of a later but a better use, avoids using instruction slots closer to the register file for instructions that do not produce register outputs.

$$Lookahead(i,rs) = \frac{df(i,o)}{Distance(rs,Reg)} + \frac{Distance(rs,Reg)}{df(i,o)}$$

In the above expression, $df(i,o)$ denotes the dataflow distance between instruction i and the nearest output instruction o. $Distance(rs,Reg)$ denotes the distance of instruction slot rs from the register file. $Lookahead(i,rs)$ is minimized at a location rs, where $Distance(rs,Reg)$ is equal to $df(i,o)$. For every dataflow chain, the heuristic thus attempts to place instructions across slots by starting farthest from the register file and gradually moving closer to the register file.

The scheduler computes a final score for an instruction i at a location rs as shown in Figure 4. In this score, the lookahead factor is weighted by a factor of 0.5, which offered the best performance in our experiments. Finally, it schedules i at the instruction slot with the lowest score.

Critical path re-ordering (R): Since the position of an instruction can alter critical paths through the hyperblock, the scheduler re-computes the critical path and re-prioritizes the unscheduled instructions. It then selects the next instruction to schedule and iterates until completion

4. Evaluation Methodology

To compare performance across different schedules, we measure instructions per cycle (IPC) using a cycle-by-cycle timing simulator. We model a TRIPS-like architecture with an 8×8 array of ALUs. We model a total of 128 reservation stations at each ALU, separate 64KB two-way, three-cycle primary instruction and data caches, a 13-cycle miss penalty to a 2MB two-way L2 cache, a 132-cycle main memory access penalty, 0.5 cycles per hop in the ALU array, a 250Kb exit branch predictor, and a 20-cycle branch misprediction penalty. Optimistic assumptions include no modeling of TLBs or page faults, simulation of a centralized register file, and no issue of wrong path instructions to the memory system. We leave these additions for future work. We also assume an oracular load/store disambiguation mechanism. Experiments with a realistic disambiguation mechanism show trends similar to results presented in this paper, and run on average 18% slower than the oracular mechanism.

We added a TRIPS scheduler to the Trimaran compiler tool set, which is based on the Illinois Impact compiler [4]. Trimaran applies aggressive VLIW optimizations and produces code in the Elcor intermediate format, based on the Hewlett-Packard PD [33] instruction set. In addition to the usual set of classic optimizations, Trimaran incorporates many VLIW-specific optimizations such as control flow profiling, trace scheduling, loop unrolling, peeling, software pipelining with modulo scheduling, speculative hoisted loads, predication with acyclic scheduling of hyperblocks and control height reduction. We have not yet tuned these optimizations for our TRIPS-like architecture.

We target a set of architectural models by using a scheduler that converts the Elcor output into TRIPS code. The scheduler models the connectivity and latencies of the 2-D Mesh network. We assume the latencies to be 0.5 cycle per hop (0.25 cycle wire and 0.25 fan-in/fan-out). The scheduler does not model contention in the network but the simulator does.

The experiments use the SPEC2000 [31] and Media-

bench	base	C	CR	CRB	CRBL	CRBLO	RBLO	convergent	Upper bound
mcf	0.86	0.95	0.87	0.90	0.85	**0.98**	0.91	0.70	1.01
adpcm	1.16	1.07	1.07	1.11	**1.19**	1.18	1.17	0.96	1.47
compr	1.36	1.34	1.31	1.36	1.44	1.42	**1.45**	1.33	1.75
parser	1.36	1.35	1.34	1.35	1.43	1.45	**1.49**	1.24	1.79
gzip	1.79	1.78	1.78	1.80	1.92	1.92	**1.96**	1.58	2.44
twolf	1.88	1.84	1.86	1.88	1.99	**2.02**	**2.02**	1.82	2.40
m88ksim	2.29	2.36	2.35	2.25	2.47	**2.50**	2.49	2.15	3.22
bzip2	2.54	2.30	2.63	2.65	2.88	**2.91**	2.86	2.71	3.39
equake	2.57	2.53	2.62	2.70	2.68	**2.86**	**2.86**	2.63	3.19
turb3d	3.61	3.66	3.74	4.28	**4.77**	4.59	4.00	4.25	6.59
hydro2d	3.88	3.54	3.87	4.22	**4.24**	4.20	3.99	4.02	6.24
mpeg2	3.95	3.27	3.34	3.93	3.97	3.99	**4.06**	3.38	4.96
art	5.14	4.88	4.95	4.99	5.17	**5.24**	5.23	4.66	5.72
ammp	5.38	4.86	4.96	5.76	5.93	6.12	**6.25**	5.60	7.09
vortex	5.56	5.71	5.77	5.93	6.38	6.54	**6.57**	6.09	7.87
tomcatv	9.46	7.49	9.84	11.44	12.06	**14.44**	13.21	13.70	18.39
swim	9.96	7.00	12.39	13.87	14.57	**16.19**	15.26	10.20	21.33
mgrid	10.99	8.24	11.46	13.47	14.95	15.08	**17.90**	13.48	19.20
dct	11.10	10.70	13.53	**17.13**	16.06	16.41	15.78	14.24	20.45
HMEAN	2.52	2.45	2.49	2.59	2.69	**2.78**	2.74	2.36	3.33
AMEAN	4.37	3.87	4.61	5.18	5.38	**5.64**	5.61	4.85	7.09

Table 1. Performance improvements from scheduler optimizations.

bench [19] benchmark suites. The Trimaran front end currently compiles only C benchmarks, so we convert a number of the SPECFP benchmarks to C, and we present results for all of the SPEC benchmarks that the Trimaran tools successfully compile.

5 Results

In this section we demonstrate the effectiveness of different compiler heuristics and show how the scheduler can balance reduced latency with better speculation depth to attain better performance. We also compare the SPDI scheduling algorithm with an implementation of the convergent scheduling framework [20] and find that the framework does not offer any improvements over the best set of SPDI scheduler heuristics.

5.1 Scheduler Evaluation

This section evaluates the scheduler heuristics from Section 3, using the TRIPS-like microarchitecture with the 2-D mesh inter-ALU network described earlier. We start with the baseline TRIPS scheduler, *Base*, which is similar to a greedy VLIW scheduler, and then progressively add the optimizations discussed in Section 3.2. Table 1 presents the performance results of these optimizations applied in different combinations. The column labeled *convergent* shows the results for Convergent scheduling, described in Section 5.2. The last column shows the results for a configuration where all communication latencies are zero. This column represents a loose upper bound on performance for the schedule of a given hyperblock. Across each row, the table shows the best-performing (non-upper bound) configuration in bold.

We examine the effect of instruction priority in the scheduling algorithm by comparing the greedy ordering (Base), the criticality ordering (C), and criticality with recomputation of the critical path during scheduling (CR). In the absence of other optimizations, greedy outperforms criticality ordering. However, recomputing the critical path at every iteration improves criticality order, yielding the best performance on nine of the 19 benchmarks. This heuristic achieves 5.5% improvement over the baseline and 19% improvement over the critical-path ordering. We also observed in other experiments that recomputing instruction priorities with greedy ordering performed slightly worse than CR. Recomputing the critical paths after each instruction placement minimizes dilation of any secondary critical paths and yields consistently better performance.

Augmenting the scheduler with a contention model to improve load balance across the ALUs (CRB) improves performance significantly in high ILP benchmarks—*tomcatv, swim, mgrid*, and *dct*. This optimization explicitly attempts to migrate independent instructions to different ALUs, but only after first optimizing the critical path. The interaction of this optimization with the critical path heuristic shows that it is easier for the scheduler to exploit parallelism after minimizing latency than vice versa. Intuitively this result makes sense because given a minimal-latency placement, the scheduler can hide the latency of instructions that have ILP with those already placed. If the scheduler places parallel instructions first, they can force non-essential latencies for critical instructions. Averaged across the entire benchmark suite, this optimization improves performance by 12.4% over critical path re-computation (CR).

The locality-aware optimizations bias the placement of load instructions (CRBL) and instructions that write to the register file (CRBLO). Table 1 shows that they also consistently improve performance, with large gains on 12 benchmarks. Further analysis reveals that there were significant reductions in load-to-use latencies. Lookahead optimizations for register outputs (CRBLO) offer additional improvements of 5% on the average. Placing the output instructions closer to the register file reduces latencies on critical paths that span block boundaries.

An optimal schedule for a given hyperblock tries to place instructions on ALUs such that back-to-back instructions on the critical path always execute in successive cycles. To estimate the performance of this schedule, we simulated the CRBLO schedules on a TRIPS configuration where all ALU-to-ALU communication latencies are zero. These results are shown in the last column of Table 1. We find that the performance achieved by the CRBLO schedules are within 80% of the upper bound, proving that a simple set of heuristics is fairly effective in achieving near-optimal schedules. We note the upper bound shown here is specific to a given set of hyperblocks. A different compilation strategy that results in different hyperblocks could improve performance well beyond this upper bound.

5.2 Comparisons with Convergent Scheduling

The scheduling algorithms described thus far tightly integrate all heuristics within one single algorithm. A competing approach that decouples different scheduler optimizations through a flexible interface is *Convergent Scheduling* [20], which has been proposed as a framework for clustered architectures. The framework composes independent phases that each address a particular constraint. All phases share a common interface that contains the current spatial and temporal preferences for a scheduling unit. A phase operates by analyzing the current preferences and modifying it by applying its heuristics. The scheduler applies the phases successively, one or more times and in any order until it converges on a final schedule. Lee et al. show that this scheduling approach works well on clustered-VLIW and RAW architectures [20].

We implemented the convergent scheduling algorithm for TRIPS and compared the results with our approach. Unlike VLIW architectures, temporal scheduling preferences are not required for TRIPS, so only spatial preferences are communicated between the different phases. We apply the following phases successively for each block.

- Parallelism optimizations: noise introduction, load balancing for parallelism.
- Placement optimizations: preplacement, preplacement propagation for loads, load consumers, and instructions that need to access the register file.
- Latency optimizations: communication minimization, critical path strengthening, path propagation.

We extensively tuned confidence measures for the placement optimizations and latency optimizations. Table 1 shows the results for the best performing convergent scheduling heuristics. We see that convergent scheduling performs less well than the best TRIPS scheduling algorithm, achieving 15% lower IPC on average. A dynamic critical path analysis reveals that while convergent scheduling reduces latencies for loads, for load consumers, and for instructions that frequently access the register file, it typically does so at the cost of increased ALU to ALU operand communication latencies, which the TRIPS scheduler usually avoids.

bench	Sparse-5	Sparse-200	Dense-500	Dense-∞
mcf	1.04	**1.06**	0.98	0.98
adpcm	1.16	**1.18**	**1.18**	**1.18**
compr	**1.48**	**1.48**	1.42	1.42
parser	**1.49**	1.48	1.45	1.45
gzip	1.96	**2.00**	1.92	1.92
twolf	**2.05**	2.04	2.03	2.02
m88ksim	2.39	2.49	**2.50**	**2.50**
bzip2	2.87	**2.96**	2.91	2.91
equake	2.74	**2.89**	2.86	2.86
turb3d	4.55	4.62	**4.67**	4.59
hydro2d	**4.62**	4.18	4.20	4.20
mpeg2	3.79	**3.99**	**3.99**	**3.99**
art	4.79	5.23	5.23	**5.24**
ammp	5.78	6.11	**6.12**	**6.12**
vortex	6.04	6.30	6.51	**6.54**
tomcatv	9.75	13.49	**14.44**	**14.44**
swim	13.50	**16.19**	**16.19**	**16.19**
mgrid	9.98	**15.13**	15.08	15.08
dct	16.29	**17.17**	16.41	16.41
HMEAN	2.76	**2.84**	2.78	2.78

Table 2. Trade-offs of scheduling for utilization versus communication

5.3 Code Density Optimizations

This section evaluates the trade-off between communication latencies within a block and ILP from increased speculation depth. As described in Section 2, the use of a small number of frames for a block enables the runtime to map and execute a higher number of speculative blocks, exploiting cross-block parallelism at the expense of longer latency for a single block. Fewer frames result in denser schedules and have the benefit of good instruction memory performance, increasing I-cache hit rates and improving I-cache bandwidth utilization. By contrast, the use of more frames creates more opportunities to schedule critical path instructions on the same ALU, thus minimizing communication latencies along the critical path. We apply a density optimization heuristic that explores this trade-off and automatically determines the best number of frames for scheduling.

The density optimization attempts to schedule a block with the minimum number of frames. It iteratively increments the number of frames by one and reschedules the block until the post-schedule critical path length is within a threshold factor of the unscheduled critical path length or until the schedule remains unchanged in successive iterations. Table 2 shows the performance obtained for different values of the threshold factor. The first column shows the performance with the sparsest schedules (threshold = 5) while the last column shows the performance with the densest schedules (threshold = ∞). Threshold defines the maximum percentage increase from the unscheduled critical path length.

Table 2 shows that a threshold factor of 200 achieves best or near-best performance on most benchmarks. It is significantly better than the sparsest schedules on 9 benchmarks and worse in only one (hydro2d). It is significantly better than the densest schedules in four benchmarks (*compress, dct, gzip, mcf*) and worse in one (*tomcatv*), because the architecture tolerates latencies well within a block. These results show that the scheduler's ability to carefully trade-off increases in the critical path with density is key to attaining the best performance.

6 Related Work

Instruction scheduling is a mature and well-researched area. In this section, we discuss only prior art related to static scheduling.

VLIW: A classic VLIW scheduler [5, 9] uses a bottom-up-greedy (BUG) approach to pack parallel instructions into a long instruction. VLIW schedulers take an aggressive approach to building large basic blocks, including predication [2, 22, 28], but use similar BUG heuristics to select and place instructions in the schedule. Our scheduler also operates on predicated hyperblocks, but it considers additional constraints, such as on-chip latencies between functional units and variable latencies from the memory hierarchy.

Partitioned VLIW: The SPDI scheduling problem bears the most resemblance to scheduling for a partitioned VLIW [15, 17, 20, 23, 25, 35, 10]. For RAW, which uses a 2-D VLIW execution model, the convergent scheduler handles complexity by computing an ALU preference for each scheduling heuristic [20]. These preferences are weighted, and because of the execution model parallelism is favored over latency. Ozer et al. solve the scheduling part of VLIW cluster assignment and leave the register assignment to a later phase using the UAS approach [23]. They find that placing critical paths in the same cluster is best in their setting. The CARS approach is similar to UAS but performs register allocation concurrently with scheduling and has lower algorithmic complexity [15].

In comparison to VLIW approaches, EDGE architectures using SPDI scheduling provide two major features that improve the effectiveness of the scheduler. First, the scheduler is freed from some of the burden of register allocation due to direct communication between ALUs, which reduces register pressure. This dedicated storage for temporaries allows the scheduler to focus solely on instruction placement and scheduling. Second, the SPDI execution model handles variable and unknown instruction and memory latencies at runtime, freeing the scheduler to achieve other goals such as instruction density, high instruction concurrency, and low instruction communication overhead.

Superscalar: The superscalar execution model is designed to tolerate variable and unknown latencies. However, compile-time schedulers can improve performance by ensuring that the hardware scheduler sees a set of instructions that exhibit both concurrency and latency tolerance. For instance, the balanced scheduler of Kerns and Eggers spreads ILP to cover variable latencies in a superscalar execution model, resulting in a schedule with available ILP spread evenly between load instructions [16]. Later work spread ILP to the loads with the highest latencies [21]. Clustered superscalar processors present to the hardware scheduler the same problems of load balancing and latency tolerance that clustered VLIWs present to software schedulers. Farkas attempts to reduce intercluster communication in a clustered superscalar using compile-time cluster assignment [6]. However, performance gains are underwhelming, in part because the scope of scheduling is limited to basic blocks, rather than the larger regions of instructions available with hyperblock formation techniques.

7 Conclusions

Conventional architectures sit at opposite ends of the spectrum with regard to their demands on the scheduler. While superscalar architectures can improve some schedules through dynamic scheduling hardware and can see some benefit from good instruction schedulers, performance is ultimately constrained by the limited instruction window size. VLIW architectures require the compiler to place every instruction and schedule every latency. In the face of variable memory latencies, the VLIW scheduler always fails. A hybrid approach that allows the scheduler to place instructions for good locality while also allowing the hardware to dynamically execute the instructions (overlapping instruction latencies and other unknown latencies) produces better performance and scalable hardware. Such approaches will become even more important as technology trends make communication more critical due to increased wire delays.

We have implemented and evaluated a scheduler for such an emerging architecture. Because the hardware dynamically executes the instructions, the scheduler is freed from the burden of precise scheduling constraints. Instead its job is to expose the concurrency in the instruction stream and place the instructions to minimize communication overheads. This EDGE scheduling algorithm is able to automatically schedule into the appropriate number of instruction slots, while still reducing average operand latency and balancing the load across the ALUs, thus eliminating hot spots where too many independent instructions have been placed. The SPDI execution model uses this schedule to opportunistically exploit ILP to tolerate variable latencies in its distributed issue window.

We have evaluated our scheduler on a 64-issue processor and shown the interplay between hardware constraints and the scheduler's capabilities. We show that for the EDGE architecture that we study, a simple scheduler with well-chosen heuristics outperforms a more sophisticated scheduler. We show that iteratively updating the estimated critical path during the instruction placement process provides a 19% boost in performance over a single priority listing. By balancing load after scheduling critical path instructions, parallelism can be exploited without introducing unnecessary latencies, improving performance by an additional 12%. We demonstrate that accounting for distances, not just between ALUs, but also to the register file and cache banks, is critical as it improves performance by 9%. A linear application of all these heuristics performs within 80% of an optimistic upper bound.

Combining the strengths of static scheduling with the advantages of dynamic issue will be critical to achieving high performance in emerging wire-dominated technologies, especially as power ceilings limit the ability of RISC and CISC-like architectures to issue instructions dynamically.

Acknowledgments

This research is supported by the Defense Advanced Research Projects Agency under contracts F33615-01-C-1892 and F33615-03-C-4106, NSF grants EIA-0303609, CCR-9985109, CCR-9984336, CCR-0085792, CCR-0311829, ACI-0313263, three IBM faculty partnership awards, and a grant from the Intel Research Council. Any opinions, findings and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 45–54, July 1998.

[3] 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 the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[4] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275, May 1991.

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

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30rd International Symposium on Microarchitecture*, pages 149–159, December 1997.

[7] J. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the Tenth Annual International Symposium on Computer Architecture*, pages 140–150, June 1983.

[8] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[9] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, June 1984.

[10] 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.

[11] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide-issue processors. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 266–276, January 1998.

[12] M. Horowitz, C.-K. K. Yang, and S. Sidiropoulos. High-speed electrical signaling: overview and limitations. In *IEEE Micro*, pages 12–24, January 1998.

[13] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 14–24, May 2002.

[14] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, September/October 2000.

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

[16] D. R. Kerns and S. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 278–289, June 1993.

[17] 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.

[18] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[19] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, December 1997.

[20] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 111–122, November 2002.

[21] G. Lindenmaier, K. S. McKinley, and O. Temam. Load scheduling with profile information. In A. Bode, T. Ludwig, and R. Wismüller, editors, *Euro-Par 2000 – Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 223–233, Munich, Germany, Aug. 2000. Springer-Verlag.

[22] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, June 1992.

[23] 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.

[24] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

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

[26] B. Rau. Dynamically scheduled VLIW processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 80–90, December 1993.

[27] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, May 2003.

[28] M. Schlansker, S. Mahlke, and R. Johnson. Control CPR: A branch height reduction optimization for EPIC architectures. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 155–168, June 1999.

[29] M. S. Schlansker and B. R. Rau. EPIC: Explicitly parallel instruction computing. *IEEE Computer*, 33(2):37–45, 2000.

[30] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 25–34, May 2002.

[31] Standard Performance Evaluation Corporation. *SPEC CPU 2000, http://www.spec.org/osg/cpu2000*, April 2000.

[32] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 291–302, 2003.

[33] V.Kathail, M.Schlansker, and B.R.Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, February 2000.

[34] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, 30(9):86–93, September 1997.

[35] 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.