Can Fine-Grain Multi-threading Subsume VLIW?*

Scott Pomerville
Northern Michigan University
Marquette, USA
spomervi@nmu.edu

Soner Önder
Michigan Technological University
Houghton, USA
soner@mtu.edu

Gang-Ryung Uh
Florida State University
Panama City, USA
guh@fsu.edu

David Whalley
Florida State University
Tallahassee, USA
dwhalley@fsu.edu

Abstract

We explore the question: "Can a fine-grain multi-threaded architecture form the basis for an efficient, VLIW style, statically scheduled architecture?" We illustrate that operations comprising a VLIW instruction can indeed be viewed as belonging to separate threads, such that the number of such operations is equivalent to the number of threads representing the program's semantics. On the other hand, a more efficient synchronization mechanism than data synchronization is needed to realize the lock-step execution model of VLIW processors. This synchronization is accomplished through the instruction space, by using a small number of bits in each instruction under the compiler control. We call the resulting architecture a "Synchronized Lane Architecture (SLA)".

The SLA approach makes embedding of no-operations in the code unnecessary in the majority of the cases, and the architecture can dynamically adapt to changing levels of ILP, while permitting code compiled for a narrow-width processor to run unmodified on a larger width processor.

We present this novel architecture paradigm, as well as the mechanism of transforming traditional VLIW code so that it can be executed by our SLA processor. We provide an evaluation of the new paradigm with respect to a conventional VLIW architecture, and demonstrate that the SLA approach delivers similar levels of performance to that of VLIW processors while providing significant energy and code size savings.

CCS Concepts: • **Hardware** → *Chip-level power issues*; • **Computer systems organization** → **Very long instruction word**; Multicore architectures; **Multiple instruction, single data**; Multiple instruction, multiple data; **Pipeline computing**; Reduced instruction set computing; *Embedded hardware*; *Embedded software*; • **Computing methodologies** → Simulation evaluation.

Keywords: synchronized lane architecture, static scheduling, very long instruction word, instruction-level parallelism, multiple program counters, program lanes

ACM Reference Format:
Scott Pomerville, Soner Önder, Gang-Ryung Uh, and David Whalley. 2026. Can Fine-Grain Multi-threading Subsume VLIW?. In *Proceedings of the 27th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '26), June 15–16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3814943.3816179

1 Introduction

Statically scheduled processors aim to extract instruction-level parallelism (ILP) under compiler control, hence requiring significantly simpler pipelines than other ILP processors. This approach yields significant power savings while extracting high levels of instruction-level parallelism (ILP) in domains such as regular scientific code and digital signal processing. The primary means of operation is to group independent operations into a single instruction word, then fetch and execute each instruction in a lock-step fashion. When there aren't enough independent operations to be packed into the instruction, no operations (*nops*) are substituted instead to maintain a fixed instruction length and a simple fetch/decode/execute pipeline.

Fine grain multi-threaded processors also execute multiple operations every cycle. Since each of the operations are coming from a different thread, typically these operations are independent of one another, just like simultaneously issued operations in a VLIW processor. Each thread of execution may be specified by the programmer using parallel programming, identified and extracted from loop iterations by the compiler or utilize a combination of both techniques.

Consider the execution of the code shown in Fig. 1 in a VLIW processor, where there are three instructions comprising seven operations. Each operation resembles a RISC instruction and is placed in consecutive memory locations. In the first cycle, the instruction containing i_0, i_1, i_2, and

*In homage to Nikhil and Arvind's seminal paper, "Can dataflow subsume von Neumann computing?" [12]

LCTES '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2721-4/2026/06
https://doi.org/10.1145/3814943.3816179

L0:			
PC_0: i_0	i_1	i_2	i_3
i_4	*nop*	*nop*	*nop*
i_5	i_6	i_7	*nop*
...	...	...	...

Figure 1. Typical VLIW code

i_3 is issued. In the next cycle, the instruction containing i_4 and three *nops* is issued. Finally, in the third cycle, the third instruction containing i_5, i_6, and i_7, along with another *nop* is issued.

L0:	L0_1:	L0_2:	L0_3:
PC_0: i_0	PC_1: i_1	PC_2: i_2	PC_3: i_3
i_4	*nop*	*nop*	*nop*
i_5	i_6	i_7	*nop*
...	...	...	...

Figure 2. VLIW code viewed as streams

Traditional multithreading often involves switching to different threads on different cycles, such as in simultaneous multi-threaded superscalar processors, but consider instead a processor that accomplishes the same task using parallel pipelines. Then, the same code can instead be represented by four streams of instructions shown in Fig. 2. Each stream can be treated as a separate thread with independent Program Counters (PCs), PC_0 through PC_3, starting at memory addresses L_0, $L0\_1$, $L0\_2$ and $L0\_3$ respectively, and moving in lock-step. Assuming an appropriate multi-threaded processor that maintains a lock-step fashion for each thread, thread 0 issues i_0, thread 1 issues i_1, thread 2 issues i_2 and thread 3 issues i_3. The PC for each thread is incremented by the length of an operation, then the process is repeated. Thread 0 issues i_4 and the other three threads issue *nops*. Finally, in the last cycle, thread 0 issues i_5, thread 1 issues i_6 and thread 2 issues i_7 while thread 3 issues another *nop*, yielding the same schedule as the VLIW processor provided that each stream continues in a lock step fashion and the control-dependencies of each of the operations are correctly maintained, since in VLIW code, individual operations forming an instruction have the same control-dependence as others in the pack. Hence, when a branch instruction is executed by a thread, all other threads need to jump (or fall-through) as the thread executing the branch.

Lock-step operation can easily be accomplished by stalling all other thread pipelines when a particular thread is stalled. This is necessary, because in a fine-grain multi-threaded processor consecutive operations are dependent only on operations from the same thread, i.e., i_4 can only be dependent on i_0, whereas in a VLIW code, i_4 may be dependent on any of the prior operations, i_0 through i_3, hence all other threads.

Notably, this fine-grain instruction-space controlled synchronization sidesteps two problems affecting VLIW processors, namely, wasting of resources through *nop* instructions when parallelism is low, and inability to execute code compiled for one architecture on a wider, more powerful architecture. It is important to note that both problems have been targeted by a number of techniques with varying degrees of success, such as compression schemes for eliminating nop instructions and instruction-templates for maintaining backward-compatibility by the Itanium architecture[6, 15]. We discuss these mechanisms in our related work section.

In order to see how these two problems can be naturally solved, consider that the synchronization mechanism can also suspend a given thread. Encoding when a thread should be suspended or resumed is easily accomplished by reserving a single bit of the instruction representation, which we refer to as the suspend-resume *(sr)* bit. A suspended thread does not fetch or execute until it is resumed. Since a suspended thread cannot resume operation on its own, we keep one thread, namely thread 0 always active and use that thread's *sr* bit to resume other threads.

We can see how the *sr* bit works in Fig. 3, where operations with the *.sr* mnemonic have the suspend-resume bit set. In the first cycle, thread 0 issues i_0, thread 1 issues i_1, thread 2 issues i_2, and thread 3 issues i_3. Since threads 1, 2, and 3 each issued an instruction with the *sr* bit set, they increment their PCs to *i6*, *i7*, and the *nop* instructions, respectively, but enter a suspended state. The next cycle, only thread 0 issues *i4*. Since the thread 0's *sr* bit is set, other threads are instructed to resume execution in the next cycle. Finally, just like in the last example, the final group of operations can issue with each thread issuing its own operation.

L0:	L0_1:	L0_2:	L0_3:
PC_0: i_0	PC_1: $i_1.sr$	PC_2: $i_2.sr$	PC_3: $i_3.sr$
$i_4.sr$			
i_5	i_6	i_7	*nop*
...	...	...	...

Figure 3. Code with thread suspend/resume

Having the ability for threads to dynamically start and stop allows us to eliminate the majority of *nops* in the program. Furthermore, when a thread is not active, the pipeline executing that thread can enter into a low-power state without the need to fetch and decode *nops* or access that pipeline's instruction cache (IC). Since pipelines which are inactive can remain in a suspended state, we can easily execute code compiled for a narrower machine on wider machines without any code modifications. Additionally, unlike VLIW code, the multi-threaded representation decouples parallelism from individual instruction representation. If the stalling behavior to maintain synchronization is controlled by instructions,

any threads which need to be lock-step can be kept lock-step, while other threads may be allowed to behave as in traditional multi-threaded processors.

In this paper, we propose a novel fine-grain multi-threaded instruction representation and a corresponding processor architecture to support it that we collectively refer to as a Synchronized Lane Architecture (SLA). In this design, multi-threading refers to this parallel instruction stream design, and not a more traditional design where the processor swaps between active threads within a single lane. We believe this architecture has significant promise in creating new avenues for finding ILP that were previously not accessible in traditional single-stream and wide-instruction representations. This paper, being the first step in that direction, evaluates a proof-of-concept of the lock-step approach demonstrating that, in addition to mechanisms of automatic program parallelization, code-generation mechanisms traditionally developed for VLIW processors can in fact be utilized to generate multi-threaded code for extracting general purpose ILP. On the other hand, even when emulating simple VLIW behavior, the SLA approach demonstrates native code size and instruction fetch energy reductions, but with comparable performance to that of a traditional VLIW. However, the SLA approach introduces new challenges that require both hardware and compilation support.

2 Synchronous Lane Architecture Overview

An SLA processor can be designed to allow up to *n* operations to simultaneously issue, one per each execution unit. We refer to each unit as a lane, which we label as 0, 1, ..., *n*-1. We refer to the collections of threads that are synchronously executed together as a *pack* of instructions. A *pack* of instructions can be seen as similar to a wide-instruction in a VLIW processor in function, but with operations in SLA packs placed into separate instruction streams instead of contiguous memory.

SLA lanes can be in one of three states, all of which are under complete compiler control:

- An *inactive (or halted)* state indicates that the lane has not started execution for that function or has been explicitly disabled. This is the initial state for all lanes except lane 0 when a function is called.
- A *suspended* state indicates that the lane's execution had started, but is currently suspended.
- An *active* state indicates that the lane's execution for that function has started and is currently active.

To prevent deadlocks, lane 0 is always active. However, the other lanes can be active, suspended, or inactive at various points during the execution of a program. When a lane is not active, the lane is left in a low-power state, since it does not need to fetch or process any instructions. If a lane does not need to provide an instruction in a given cycle, access to that lane's instruction cache (IC) is also suppressed.

Since the architecture is synchronous (i.e., lock-step), a miss in any IC associated with an active lane stalls all lanes. In our proof of concept, if multiple L1 ICs miss, each one lane pipelines their request to the L2 cache so that additional ports are not required to access the L2 cache at the cost of some performance.

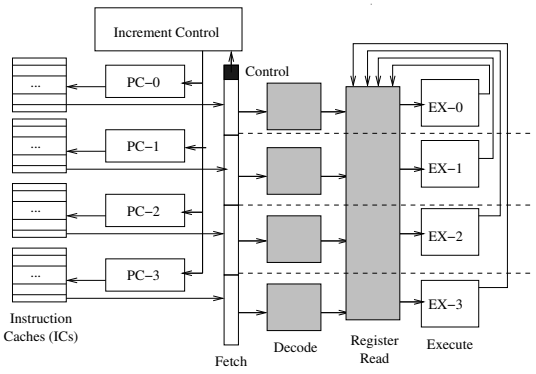

Figure 4. An Overview of the SLA Processor

Fig. 4 shows an overview of an SLA processor. Each lane has a dedicated PC, IC, and decoder. Delineated by the dotted lines, each lane looks and functions similar to that of a single issue pipelined processor in the pipeline frontend, but they share a single register file. In fact, an SLA processor can directly execute a single RISC style instruction stream in lane 0, since lane 0 is active when the processor is initialized. In this respect, code generated for a simple single issue pipelined processor can be considered to be a special case of SLA code where the code width is one operation wide.

A significant difference between an SLA processor and a VLIW processor is the increment control unit, which is responsible for maintaining lane status for each lane. These responsibilities include suspending, resuming, or, halting a particular lane. Since these actions need to take immediate effect, suspend-resume (*sr*) bits always occupy the most significant bit of the instruction, allowing immediate use of this information by the control unit as soon as an instruction is fetched and before it is decoded.

The SLA paradigm is orthogonal to many VLIW pipelined processor designs. In other words, any VLIW processor design can be converted to an SLA architecture by replacing the pipeline frontend with one which utilizes streams in the described manner. Therefore, we do not further discuss the details of the execution pipelines under the paradigm, except the necessary modifications, as needed.

3 The SLA ISA

Traditional VLIW processors employ an instruction encoding where each instruction has a number of operands and multiple opcodes describing each of the multiple operations to be performed by a single wide instruction. Since the SLA

architecture fetches individual operations using multiple program counters, each of the streams can fetch and execute RISC style instructions that describe the particular operation to be performed. In order to evaluate the SLA paradigm, we rely on a new instruction set architecture (ISA) and refer to this new ISA as our SLA ISA.

In the SLA ISA, the dedicated *sr* bit allows us to have two versions of the same instruction, one with the *sr* bit set and another with the *sr* bit clear. The former utilizes the assembly mnemonics of the instruction followed by `.sr` and the latter does not have the suffix. For example, our SLA add instruction would either be `add $2,$3,$4`, or, `add.sr $2,$3,$4`. When the *sr* bit is set for an instruction in any lane other than lane 0, then the bit indicates that this lane is suspended after the current instruction executes. Any subsequent instructions are not dispatched until lane 0 encounters an instruction with an *sr* bit set. In all lanes, except lane 0, the *sr* bit affects (suspends) only the lane in which it is executed and in lane 0, it affects (resumes) all other suspended lanes. Since *sr* bits are handled by their own lanes, any number of lanes can suspend in the same synchronized pack.

Table 1. SLA Synchronization Instructions

Inst	Arguments	Description
call callr	target_addr or addr_reg	Each Lane RT reg ← Lane PC. PLS Reg ← Each LS reg. PC[lane_0] ← target_addr. LS[lanes_1..N-1] ← inactive
callm	target_addr	A call instruction that does not save lane states.
return		Each Lane PC ← Lane RT reg Each LS reg ← PLS reg
swrt	rt_reg, st_addr_reg	mem[st_addr] ← rt_reg_val
lwrt	rt_reg, ld_addr_reg	rt_reg ← mem[ld_addr].
sas	st_addr_reg	mem[st_addr] ← PLS
las	ld_addr_reg	PLS ← mem[ld_addr]
colane	lane_id, target_addr	LS[lane_id] ← active PC[lane_id] ← target_addr.
halt		LS[my_lane] ← inactive
pb	target_addr	PB[my_lane] ← target_addr

An overview of all introduced synchronization instructions are given in Table 1, with a definition of each architectural effect. Likewise Table 2 shows new SLA registers to support SLA code generation. These SLA instructions and registers are referenced in the rest of this section.

3.1 Handling Function Calls

Lane 0 in the SLA paradigm is responsible for all control flow changes, including branches, jumps, and function calls. Since function calls must place the processor in a predictable state before any function invocation, SLA provides several functions to save the state of the processor and place it in a predictable state upon a function call. SLA primarily utilizes call and callr instructions, which are analogous to *jal* or *jalr* in MIPS, but all lanes other than lane 0 are marked as inactive while the lane 0 PC is set to the call target address.

Table 2. New SLA Registers in an N-wide SLA processor

Register/Use	Size	Location	Count
Prepare Branch (PB)	PC Size	Each Nonzero Lane	N-1
Lane Status (LS)	2 Bits	Each Nonzero Lane	N-1
Return Addr (RT)	PC Size	Register File	N
Prev Lane Status (PLS)	2*N Bits	Register File	1

When a *call* or *callr* instruction is encountered, the processor saves the processor state into a set of registers, which is equivalent to the return address register in conventional architectures. The special registers are called *RT* registers, and each lane is provided one to store the PC state at the point of the function call. Similarly, a previous lane status *(PLS)* register is provided to store the state of each lane.

For any non-leaf function, the compiler is responsible for scheduling instructions to store each of the *RT* registers in the function prologue and instructions to restore each of the *RT* registers in the function epilogue. Since these are not general-purpose, the ISA has special store (*swrt*) and load (*lwrt*) instructions to save and restore *RT* registers, respectively. Store-active-status (*sas*) and load-active-status (*las*) instructions are also provided to preserve the *PLS* register.

Lanes that are never enabled in a function never need to load or store an RT register. Since restoring the RT register to the previous state is handled by callee functions, if a callee modifies the RT register of a lane that is inactive in the caller function, the register will be restored to the proper state before a return occurs. The RT register repair being handled by the callee means that caller functions do not need to know the total width of the processor, and do not need to store or load values for lanes the function does not activate.

To restore an SLA processor to a state prior to a function call, the *return* instruction indicates for all lanes to restore their PC and active status using the *RT* and *PLS* registers.

3.2 Creating and Halting Instruction Streams

Since call instructions mark nonzero lanes as *inactive*, we provide *colane* instructions to activate new instruction streams. To minimize required changes in the branch target buffer (BTB) to support *colane* instructions, we enforce that *colane* instructions only be scheduled in lane 0, similar to branches. Each *colane* instruction specifies a starting target address as a displacement from the executing lane 0 PC, as well as the lane assigned to handle the new instruction stream. Note that a *colane* instruction to activate a lane that can be immediately suspended with a *nop.sr* instruction.

We also introduce a *halt* instruction. This instruction marks the lane in which it resides as *inactive* for the rest of the function execution. This instruction always has the *.sr* bit set, but also halts the lane, so the lane is not resumed on all further lane 0 instructions with their *.sr* bit set. Setting a lane to inactive frees up instruction space and decreases the number of times a lane is woken up, allowing for additional power savings.

3.3 Scheduling Branches and Jumps in Lanes 1 to *n*-1

A conditional branch or unconditional jump only contains enough information to affect the PC of the lane in which the transfer of control resides. Since branches and jumps can only be issued in lane 0, they behave identically in that lane compared to standard processors. For simplicity, we have each lane follow the same control flow, meaning that nonzero lanes do not need to perform the redundant truth operation for conditional branches. Instead, we provide an instruction that allows for nonzero lanes to update their PC when a branch or jump is taken in lane 0. The prepare-branch (*pb*) instruction is encoded as a jump instruction in nonzero lanes since regular jumps are only utilized in lane 0. Before a branch or a jump, each active or suspended lane other than lane 0 must execute a pb instruction to prepare the target address of the lane prior to the lane 0 branch. When a *pb* instruction is fetched, it sets the PB register associated with the lane in which the *pb* instruction resides. Upon lane 0 taking a branch, whether conditional or unconditional, each lane that is active or suspended sets the lane PC to the address stored in the PB register associated with its lane.

Lane 0	Lane 1	Lane 2	Lane 3
i0.1	i1.1	pb LN_2	pb.sr LN_3
i0.2	pb LN_1	i2.1.sr	-
j LN_0	i1.2	-	-
...	...	...	...

Figure 5. An example of *pb* instruction scheduling

Since *pb* instructions are independent of other instructions, they can be scheduled any time prior to the branch in lane 0, as shown in Fig. 5. Allowing for *pb* instructions to be scheduled prior to a branch also lifts the requirement for all lanes to be active at the time as the branch or duplicating branches for each lane.

4 Code Generation Strategy

We developed an assembly optimizer, *asopt* to perform transformations at the assembly level for this project. Use of an assembly optimizer allows us to leverage all the optimizations performed by a mature compiler like gcc while being able to perform low-level transformations in a much simpler infrastructure. We use *gcc* to generate MIPS assembly code, and also obtain information from *gcc* indicating which registers are passed as arguments to each function call and which registers are used as return values. The *asopt* to then expands each pseudo assembly instruction to be a one-to-one mapping between assembly and machine instructions, before translating the few instructions to their new representations as described in Section 3. We perform register lieveness analysis using the assembly and the function information from *gcc* before performing a number of optimizations that is concluded with VLIW scheduling of operations and filling slots after each branch from the fall-through successors.

As previously stated, there are minor differences between the MIPS ISA and our MIPS-like baseline ISA. Traditional MIPS processors support a *displacement addressing* mode for loads and stores. In order to initiate a long latency memory operation early in a pipelined data path, modern dynamic processors dynamically break a load (or store) with the displacement into two machine instructions[7]. where one instruction computes the memory address and the second performs the data access. The SLA processor also exploits this feature by (1) only supporting a *register deferred addressing* mode for loads or stores, and by (2) requiring the code generator to statically split memory instructions into one that calculates the effective address and the other to access memory with a register deferred addressing mode. These memory instructions access the data cache at the same time other instructions perform computations in execution units. We also take steps to always generate conditional branches to compare a single register to zero (e.g. *beqz* or *bnez*) due to fewer available instruction opcodes.

As an example of SLA code generation, consider the general layout of a 4-lane function in Fig. 6, taken from the function `makekey` in the 482.sphinx3 benchmark from spec2006. In our study, we allow the code to schedule floating-point (FP) and memory instructions in any lane, but similar to a VLIW processor, SLA processors can allow for asymmetric lanes where certain capabilities are limited to specific lanes, such as memory ports only in lanes 0 and 1.

The number of lanes to activate for a function is determined by the largest width of the VLIW packs without *nops* in the traditionally scheduled VLIW code. For `makekey`, the maximum width is 4. Vertical lines denote each lane, and instructions are aligned to show when they will execute in a pack together. Blank spots indicate suspended/inactive lanes. We then change all function calls and returns to *call* and *return* instructions. The *call* and *return* instructions are highlighted in pink

In order to create a separate instruction stream per lane, each nonzero lane is assigned unique labels by duplicating the lane 0 labels. Each lane's target labels are a copy of the label, with an extension of "__#" , where # is the lane number to provide a unique destination in each instruction stream.

For any non-leaf function, there must be an *swrt* instruction scheduled for each lane to store the return PC for that lane in the function prologue. However, only a single *sas* instruction is performed for the entire processor, since active status information requires only two bits per lane, supporting up to 16 lanes in a 32-bit word. For non-leaf functions, the epilogue prior to a return must contain *lwrt* instructions executed to restore the state of each lane. Similarly, there must be a single *las* instruction to restore the lane active status for each lane. Since `makekey` has a *call*, we insert stores (*swrt/sas*) to save the current values of the *RT* and *PLS* registers to its stack frame at the function prologue, and loads

```
makekey:                makekey__1:              makekey__2:              makekey__3:
colane makekey__1,1
colane makekey__2,2     addiu $sp,$sp,-64
colane makekey__3,3     addiu $3,$sp,24          addiu $7,$sp,44
addiu $10,$sp,40        pb lane1,$L52__1         addiu $8,$sp,32          pb lane3,$L52__3
addiu $9,$sp,36         addiu $2,$sp,20          pb lane2,$L52__2         addiu $1,$sp,28
sw $16,($2)             swrt $rt0,($1)           move $16,$5              swrt $rt3,($10)
sw $17,($3)             swrt $rt2,($9)           swrt $rt1,($8)           sas.sr ($7)
beqz.sr $6,$L52         move $2,$6               move $17,$4
$L46:                   $L46__1:                 $L46__2:                 $L46__3:
slt $1,$0,$16           pb.sr lane1,$L49__1      pb.sr lane2,$L49__2      halt
beqz.sr $1,$L49
sll $5,$16,1            move $4,$2               move $7,$17
addu $5,$5,$2           pb lane1,$L48__1         pb lane2,$L48__2
$L48:                   $L48__1:                 $L48__2:
lbu $3,($7)             addiu.sr $1,$4,1         nop.sr
andi $3,$3,0xf
addiu.sr $3,$3,65
sb $3,($4)              addiu $4,$4,2            nop
lbu $3,($7)             seq.sr $6,$4,$5          addiu.sr $7,$7,1
srl $3,$3,4
addiu.sr $3,$3,74
beqz $6,$L48            sb $3,($1)               nop
sll.sr $16,$16,1        nop                      nop
$L47:                   $L47__1:                 $L47__2:
addiu $1,$sp,28         addiu $3,$sp,20          addiu $6,$sp,36
addiu $4,$sp,24         addiu $5,$sp,44          addiu $7,$sp,32
addiu $8,$sp,40         lwrt $rt1,($7)           addiu $sp,$sp,64
lwrt $rt3,($8)          lw $17,($4)              lwrt $rt2,($6)
lwrt $rt0,($1)          addu $16,$2,$16          las ($5)
return                  sb $0,($16)              lw $16,($3)
$L52:                   $L52__1:                 $L52__2:                 $L52__3:
sll $4,$5,1             lalui $6,$LC0            addiu $5,$0,1            addiu.sr $7,$0,177
call.sr __ckd_calloc__  addiu $4,$4,1            laori $6,$6,$LC0
nop                     pb.sr lane1,$L46__1      pb.sr lane2,$L46__2      halt
b.sr $L46
$L49:                   $L49__1:                 $L49__2:
move $16,$0             pb.sr lane1,$L47__1      pb.sr lane2,$L47__2
b.sr $L47
```

Figure 6. Example of SLA generated code

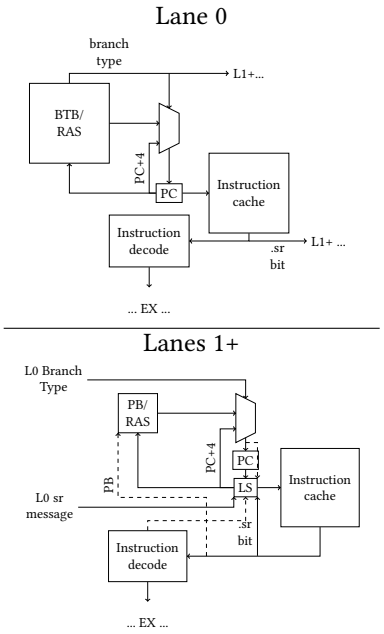

Figure 7. Design of the frontend pipeline of Lanes 0 and 1+

(*lwrt/las*) to restore these values at each function epilogue (return block). These instructions are highlighted in yellow.

Next, we insert lane specific *pb* instructions in nonzero lanes prior to jumps or branches in lane 0, with each PC reflecting the lane-duplicated label in tandem with the destination of lane 0. Duplicated labels and inserted *pb* instructions are marked in **bold**. Each *pb* instruction can be scheduled within the basic block of the same lane, as long as it is prior to the branch instruction in lane 0. Each *pb* instruction in a single basic block loop is moved out to its loop preheader as being shown in the loop at label $*L*48.

After making the changes previously described, we reapply VLIW scheduling using the maximum detected width of the function, keeping each *pb* instruction in its associated lane. Note that the *sb* and *lw* instructions scheduled in lanes 1 and 2 alongside the *return* instruction will execute prior to any instructions at the return point. Similarly, the instructions scheduled alongside the *call.sr* will execute before any instructions in the callee.

We next insert the underlined *colane* instructions at the function entry to provide the starting address of each lane within the function. With the exception of lane 0 (the master lane), we need to insert a *colane* instruction for each lane in which one or more instructions are scheduled within the function.

In the next step we detect where to insert *halt* instructions. We perform a simple analysis to detect when a lane will no longer perform any instructions outside of those required for synchronization. During this analysis, we omit the function epilogue, since epilogue instructions can be re-scheduled into other lanes. If no further useful instructions are detected, we place a *halt* instruction (marked with a double-underline) in the lane to disable unnecessary *sr* bit reactivations. We eliminate instructions in those lanes, and shift any remaining computations that are assigned to that lane in the function epilogue into other lanes.

We then introduce proper synchronization when lanes need to be suspended or resumed. We set the *sr* bit in a lane 0 instruction when there are more instructions in the next pack, and can contract the processor width using *sr* bits in those packs. Finally, we generate the code so that the instructions for each lane are contiguous in memory.

5 Architectural Design of an SLA processor

5.1 The SLA Pipeline Frontend

The SLA processor is designed to be similar to multiple simple 32-bit pipelined processors working in conjunction, but several small modifications to hardware must be done to maintain correctness. Architectural considerations for performance are outlined in further detail in Appendix A.

As shown in Fig. 7, each pipeline frontend is similar to a simple pipelined processor, with several added registers defined in Table 2. Lane 0 behaves nearly identically to a typical pipelined processor, but notifies other lanes when calls, returns, branches, and jumps are encountered, as well as when instructions with a set *.sr* bit are fetched from the Lane 0 IC. Nonzero lanes are also similar, but do not need to access the BTB. Instead, nonzero lanes simply access their lane's PB register, which is updated via *pb* instructions, and takes directional information from Lane 0.

Since a *colane* instruction is a transfer of control flow that activates another lane, we allow them update the BTB, storing the target lane and target address when relevant. To avoid requiring larger BTB entries or a dedicated structure, we limit colane instructions to lane 0. Note that unlike a traditional VLIW, in which any operation in the wide instruction can potentially contain branches or jumps, all transfers of control occur in lane 0. This simplifies BTB accesses since only lane 0 in the SLA processor needs to access the BTB. Aside from the normal BTB structures, the BTB needs to have some bits to indicate if the instruction in the BTB is a colane and the target lane number.

The return address stack (RAS) in each lane also behaves similar to a normal processor, though nonzero lanes must also store the previous lane status along with the destination PC for each lane.

To maintain the state of each nonzero lane, we introduce a 2-bit lane status *(LS)* register to represent lane states. Table 3 shows the valid states. The most significant bit is responsible for suppressing requests to resume execution, while the least significant bit is responsible for lane active status. When nonzero lanes receive a branch type from lane 0, they may also modify the LS register in their lane. When a *call* instruction is encountered, the value of the LS register for each nonzero lane is set to inactive after copying the value to the PLS register. When a *return* instruction is encountered, the value of the LS register is restored using the corresponding lane's RAS. The LS register is also updated to active and enabled when a colane instruction is sent from the BTB. If a *halt* instruction is decoded, the LS register is set to 00 in that lane. Since the *halt* instruction always has the *sr* bit set, the lane first sets the LS register to suspended when the halt is fetched, then the lane is set entirely to inactive once the instruction is decoded.

Table 3. LS States

Bit Configuration	LS State
0 0	Inactive. Suppress lane 0 wakeup requests.
1 0	Suspended. Wake up on lane 0 sr requests.
1 1	Active. Execute Instructions.

Note that since L1 ICs of SLA processors are separated, the SLA processor has separate tags associated with each operation in a pack similar to a traditional processor. The VLIW, however, only has one tag associated with an entire wide instruction. Since the SLA processor performs a tag comparison for each active lane, a SLA processor performs more IC tag comparisons than a VLIW processor on average.

5.2 The SLA Processor Pipeline Backend

Unique to the architecture are two other required registers that do not reside in the pipeline frontend, and are defined in Table 2. These two registers are the (*PLS*) register and the (*RT*) registers. These registers do not need to reside in the pipeline frontend. Since they are not speculatively updated, they can be updated similar to the return register for a jump-and-link instruction. The *PLS* register stores the concatenation of all previous lane status values upon a call instruction and is designed to maintain data for the processor to use upon a return if the RAS is insufficient. Similarly, each *RT* register is responsible for maintaining the lane's return address. Since the LS register works in conjunction with the PC, the processor must restore both the PC and LS register upon rollback. The PLS and RT registers inform rollbacks when a return is mispredicted.

5.3 Hardware Cost Analysis

Table 4. Hardware Overhead for each Nonzero Lane

Structure	Size
PC	32 bits
RAS	34 bits per entry
LS	2 bits
PLS	2 bits
RT	32 bits
Instruction Cache Tags	Variable

As shown in Table 4, hardware costs for each additional nonzero lane are smaller than a full separate frontend since branch predictions are handled by lane 0. Since nonzero lanes do not predict, they do not duplicate the BTB, instead replacing the entire structure with a PB register. The return address stack entries cost an additional two bits to store the previous status, so for an 8-entry RAS, the cost would be around 272 bits total. This means the cost of an entire pipeline frontend is dominated by the size of the additional introduced tags in the dedicated instruction cache.

5.4 Variable-Width Behavior in SLA

A notable property of explicit activation of lanes is that SLA processor to naturally support backwards compatibility with lower-width processors. In contrast, the PC increments in VLIW processors vary based on the width of the wide instruction. If a wider SLA processor is provided with code compiled for a narrower SLA processor, the program will function as though the wider SLA processor were the same width as the processor targeted by the compiler.

While the SLA processor is underutilized when lanes are inactive, the SLA processor natively supports narrower width code with no modifications. VLIW processors also underutilize their hardware when executing narrower code, but need ways to modify PC increments alongside structures to align wide instructions with the processor or deal with instructions that cross cache line boundaries when the width of the VLIW instruction is not cache aligned.

6 Methodology

Since the SLA design is a unique processor, we compare characteristics to a comparable VLIW processor design to examine its properties as a proof-of-concept processor.

We use the Fast/ADL simulator [13] to simulate both a conventional VLIW processor as the baseline and a modified processor supporting the SLA approach. Both of these processors use variants of the SLA ISA and are implemented as faithful cycle-accurate simulators.

Table 5. Processor Configuration

Branch Predictor	4096 entry direct-mapped BTB GShare predictor w/ 17-bit branch history.
L1 Partitioned IC	32KiB total, 64B line size, 4-way, VLIW: 11 cycle miss pen. SLA: 12 cycle miss pen.
L1 DC	32KiB, 64B line size, 4-way
L2 Unified Cache	1MiB, 64B line size, 16-way, 80 cycle miss pen.
5 stage integer pipeline	IF, ID, RF, EX, WB

Table 5 shows other details regarding the processor configuration that we utilize in our simulations. We support a separate L1 IC for each lane to allow simultaneous instruction fetches. The total L1 IC space for all lanes is 32KiB, but the actual partitions can be symmetric or asymmetric. In our simulations, the L2 cache uses a single shared port for all ICs so L1 IC requests are pipelined. The SLA can gain performance by adding more ports and banks into the L2 cache at the cost of higher energy usage, this was not modeled in our experiments.

We support a 5 stage integer pipeline, where the memory reference for a load instruction is executed in the EX stage due to no displacement with loads. While modern processors use deeper pipelines, both the SLA and VLIW processors are affected in the same way by the pipeline depth. In both the baseline VLIW and the SLA processors, misprediction penalties are increased by each stage that occurs prior to the misprediction detection, and commit latencies increase with each additional stage added to the processor as a whole.

We test two 4-wide SLA processors and two 8-wide SLA processors. The processors are identical, aside from the lanes and their IC configurations. We test both a symmetric and asymmetric IC allocation, and caches are set to store a total of 32KiB of IC space. The processor configurations are shown in Table 6. The 2KiB caches of the Wide-SLA-Asym configuration are set to be 32-byte lines instead of the 64-byte lines of other caches since these caches are quite small and lanes are often disabled much more frequently.

Table 6. KiBs allocated per SLA Lane Instruction Cache
*2KiB caches have 32 Byte lines

Description	L0	L1	L2	L3	L4	L5	L6	L7
SLA-4x8k	8	8	8	8	-	-	-	-
SLA-16-8-4-4	16	8	4	4	-	-	-	-
Wide-SLA-8x4k	4	4	4	4	4	4	4	4
Wide-SLA-Asym	8	4	4	4	4	4	2*	2*

6.1 Ensuring Equivalent Workloads

We ran our simulator on a variety of SPEC 2006 benchmarks on reference inputs. However, the instruction spaces of the VLIW and SLA architectures are significantly different, since both contain different instructions that do not directly contribute to a program's core computations but are needed for correctness. Therefore, traditional IPC cannot be used to compare performance. The instruction sets of the VLIW and SLA architectures are designed to be identical aside from the instructions that are unique to the SLA processor. To account for the fact that these new instructions do not work that is part of the core program, we classified each operation as either *meaningful* or *meta* instructions. In this context, an instruction is a single operation.

We classify *meta* instructions as operations that are not part of a program's core control flow or calculations. The VLIW (and occasionally the SLA) processor uses *nops* to maintain alignment, and the SLA memory instructions that load and store *RT* registers, as well as *colane* and *halt* do not directly contribute to the core computations of the program, so we mark these as meta instructions that should not be counted towards performance. The loads and stores to the *PLS* register correspond 1-to-1 with loads and stores to a VLIW return address register. Any non-*meta* instructions are considered *meaningful* instructions. During the lifetime of a function, while the number of *meta* instructions vary, both the VLIW and SLA processor execute the same number of meaningful instructions. To ensure that meaningful instructions were robust and properly classied, we tracked function calls and branches in various windows of meaningful instructions to ensure that the VLIW and SLA architectures performed the same number, as well as hand-comparing several functions. In any given function, *meaningful instructions* remain the same between VLIW and SLA architectures, and only *meta* instructions are different, ensuring both processors perform the same amount of meaningful work on each benchmark. In both cases, the actual rate of progress through a program is measured using Meaningful Instructions per cycle (MIPC), which is a representation of $MIPC = Instructions_{meaningful}/Cycles_{total}$. Since we count all cycles, but omit meta instructions, if a section of code

consists of only meta instructions, the MIPC for that section would be 0.

We allow each benchmark to warm up for 2 billion meaningful instructions, followed by gathering statistics for 10 billion meaningful instructions. Expansion from VLIW to SLA was completely mechanical using our assembly optimizer, and no additional code optimizations were performed once the SLA conversion occurred.

When performing power calculations, we use CACTI 7 [3] with 32 nm technology and 1 read port for the L1 ICs, with the port being 32 bits for each SLA lane, and 128/256 bits for the baseline 4-wide/8-wide VLIW. Similarly, the L2 cache is simulated with a read/write port instead, but it is otherwise identical between the two processors. We did not calculate the energy for BTB accesses, which SLA only accesses for lane 0, and VLIW accesses for all lanes.

7 Results

7.1 Performance Analysis

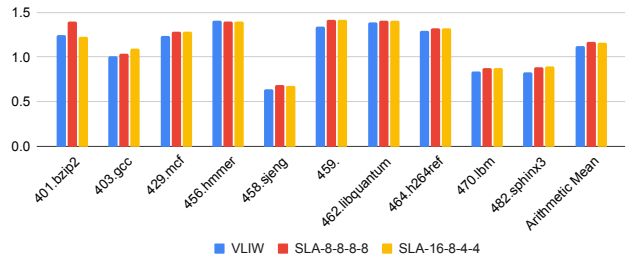

Figure 8. Meaningful IPC (Higher is Better)

As shown in Fig. 8, we find that meaningful IPC is similar to that of a baseline VLIW processor in nearly every benchmark. While the SLA processor introduces extra instructions to the stream, which can introduce new packs and lower MIPC, the SLA can better utilize the L1 ICs compared to the baseline VLIW processor to offset the increase in total packs.

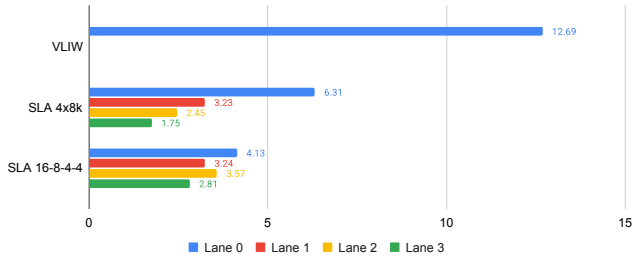

Figure 9. Arithmetic Mean Instruction Cache Misses Per 1000 Meaningful Instructions over Spec 06 (Lower is better)

We can see IC performance from Fig. 9. The L1 IC misses less often for SLA processors than the VLIW. Performance is highly sensitive to IC hit rates, meaning that even marginal improvements can make significant differences in terms of performance. SLA ICs store far fewer *nop* instructions, meaning that more of the cache is dedicated to storing *meaningful* instructions. While the total number of misses in SLA processors are marginally higher, across all four ICs (13.74 for SLA 4x8k, and 13.75 for SLA 16-8-4-4), many misses occur immediately after a taken branch. During these taken branches, multiple ICs often miss, allowing for opportunities to pipeline requests to the L2 cache, reducing the effective IC miss penalty. Furthermore, IC misses can be triggered by suspended lanes, which occurs after executing an instruction with its *sr* bit set, or when the PC is updated during a taken branch associated with a suspended lane. Suspended lanes can begin handling the IC miss while the lane execution is suspended without interrupting the execution of active lanes. Furthermore, L1 IC lines in lanes that are often suspended can functionally service more packs of operations before a miss occurs, as lanes do not need to represent as many *nops* in the L1 IC. Due to improved performance for ICs, we find that on average, meaningful IPC is within 0.04% for both configurations of the SLA processor and the VLIW baseline, despite the increase of around 1.2% in total packs executed.

7.2 Instruction-Cache Behavior

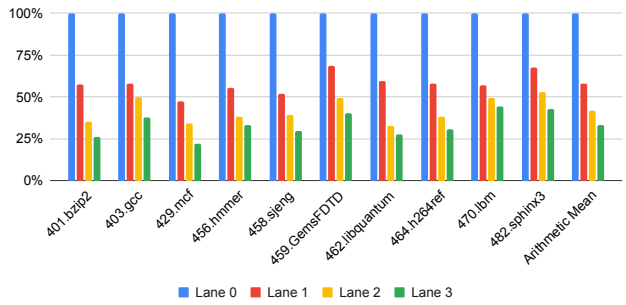

Figure 10. SLA 4x8k caches, # of accesses wrt lane 0

Fig. 10 shows how frequently that each lane accesses the L1 ICs for an SLA processor with the symmetric 4-8K cache configuration over the entire benchmark suite. We do not show the SLA 16-8-4-4 configuration since it behaves nearly identically (within 1% for each lane). When a lane is suspended or halted, that lane does not need to issue a cache request. Suspending lanes means that during periods of lower IPC, lane ICs can save significant energy on average. What we see is that, on average, lane 3 accesses its IC 35.97% as often as lane 0. Since each SLA lane IC is significantly smaller than the VLIW IC, individual lane IC power requirements are lower than the VLIW IC.

Since only one or two of the ICs are accessed in a majority of cycles, the SLA processor has potential to save significant energy with ICs, as seen in Fig. 11. 458.sjeng performs most poorly due to bad IC coverage coupled with many small, function calls. When a function is encountered for the first time, the SLA processor needs to perform an L2 cache access for each lane instead of the single access that the VLIW

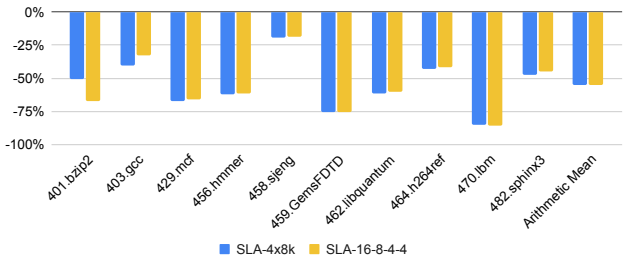

Figure 11. IC Energy Normalized to VLIW (Lower is Better)

requires. The IC hit rate for 458.sjeng is lowest in the benchmarks, and due to how poorly it caches, the SLA processor needs to more frequently access the L2 cache compared to the VLIW when a new function is encountered. While the SLA processor still saves energy with the L1 caches, the power requirements for additional L2 cache accesses offsets some of the power savings of the L1 caches. However, in the typical case, ICs in the SLA processor use significantly less power than the baseline VLIW processor, with an average mean reduction in power being 55.17% for SLA-4x8K and 55.29% for SLA-16-8-4-4.

7.3 Lane Power Analysis

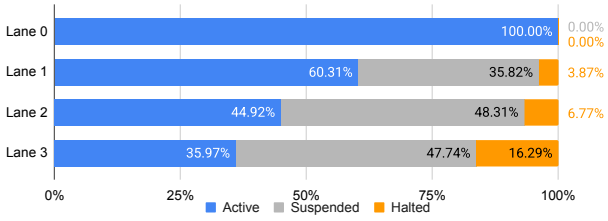

Figure 12. SLA 4x8k - Avg Distribution of Lane Status

Fig. 12 shows the properties of each of the lanes. Lane 1 only executes, on average, 60.31% of cycles, while suspended or halted the rest of the time. Lanes 2 and 3 execute even less, needing to be active in under half of the total cycles. Compared to a VLIW, more power is used on cycles where all lanes are active, but power consumption can be reduced when any lanes are turned off, and the simultaneous utilization of all lanes in the entire processor occurs under 35.97% of the time. Though difficult to measure due to dependency on how the pipeline frontend is implemented, lanes that are suspended have to check only for incoming messages from lane 0, meaning that when in a suspended state, power savings extend beyond just the instruction cache.

7.4 SLA Scalability

Fig. 13 shows how MIPC compares in 8-wide processors. During points of limited MIPC, the VLIW processor performs worse at higher widths since the IC hit rate dramatically decreases, and increased parallelism does not compensate

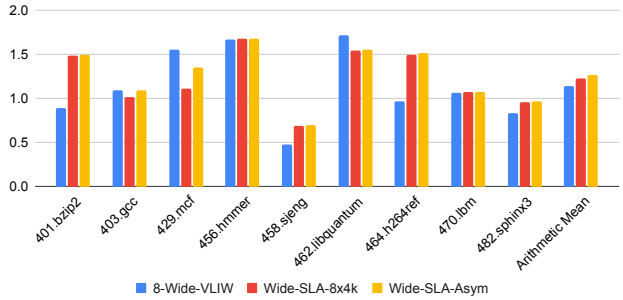

Figure 13. 8-Wide Meaningful IPC

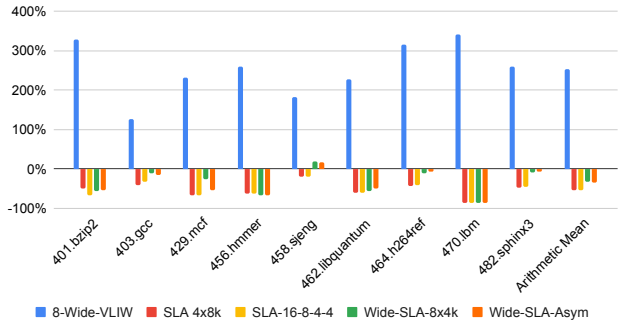

Figure 14. 8-Wide Cache Power Statistics (Normalized to 4-wide VLIW)

for the increase in *nop* instructions. For instance, in 401.bzip, the 4-wide VLIW has a 98.72% IC hit rate, but hit rate drops in the 8-wide VLIW to 91.58%, which is a 7.14% drop in hit rate. However, the hit rate of the asymmetric SLA processor is impacted less than a VLIW. In SLA-16-8-4-4, the lane 0 hit rate of 401.bzip2 was 99.96%, while the lane 0 hit rate of Wide-SLA-Asym only decreased to 98.28%, a 1.68% drop in hit rate. All other caches for the asymmetric cache configurations degraded less than lane 0, meaning the SLA processor is able to maintain performance even during periods of low MIPC. Since the discrepancy of lane activity is higher for the wider-width processors, we find the asymmetric configuration tends to outperform the symmetric configuration.

Since *nops* are not needed to be stored in caches, we find that IC performance scales better than VLIW processors. Fig. 14 demonstrates the IC power usage of an 8-wide VLIW vs SLA configurations. Both 8-wide SLA processors use less IC power than the 4-wide VLIW. Since ICs of unused lanes can stay in a low power state for significant lengths of time, the Wide-SLA-8x4 requires 20.35% less IC energy, and the Wide-SLA-Asym uses 42.95% less IC energy than the baseline 4-wide VLIW. The asymmetric design performs better due to the L0 IC having additional resources, meaning that it has fewer requests to the L2 cache.

7.5 Binary Encoding Size

Another byproduct of eliminating nops from VLIW binaries is smaller binary sizes. In smaller systems where storage is at a premium, such as in the case of ROM chips, smaller code

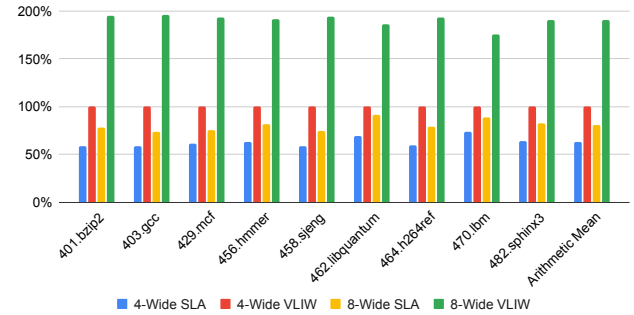

Figure 15. Number of Encoded Operations (Normalized to 4-Wide VLIW)

size can impact cost. Fig. 15 shows the number of operations required to encode the binaries for each of the benchmarks, excluding libraries. On average, a 4-wide SLA binary contains 63.50% of the operations of an equivalent VLIW binary.

While the 8-wide VLIW requires 190.85% of the encoded operations compared to the 4-wide VLIW, since width increases the overhead of *nops*, the 8-wide SLA processor configurations, maintain a significantly smaller binary since introduced instructions only affect lanes that are active in the lifetime of a function. Since nops are widely eliminated, the 8-wide SLA configuration requires a smaller binary than the 4-wide VLIW configuration, encoding just 80.74% of the operations of the 4-wide VLIW binary.

8 Related Work

Multi-PC and Distributed VLIW Designs: The approach of increasing granularity was addressed by research into clustered VLIWs. To mitigate wire delays, from centralized resources, Zhong et al. proposed a variation of the VLIW processor design, called the distributed control path VLIW *(DVLIW)* [20]. The DVLIW processor design similarly distributed fetch, decode, and distribution logic, and each cluster was provided its own PC. The design focused on mitigating wire delay and improving VLIW scalability through clustering resources, not changing the representation to target control flow.

Static Code Compression: There have been many techniques to reduce the code size of VLIW architectures by the use of compression. Some architectures compress *nop* operations in the VLIW by setting special bits within the VLIW, such as the Philips Trimedia and TigerSHARC processors [1, 2], which is also referred to as variable length VLIW encoding [16, 18]. Some of these architectures will compress the VLIWs in memory and have the *nop* operations explicitly represented in the IC or have hardware dedicated to realigning packs of instructions that cross cache boundaries and variable PC increments informed by the size of previous packs [2, 4, 8]. VLIW compression techniques introduce additional complexity to reduce code size, including dedicated hardware and pipeline stages for realigning packs, that SLA processors sidestep since operation sizes are consistent regardless of the number of lanes active, and most *nop* operations are not required in SLA representations. Other techniques to remove nops have involved on-the-fly decoding of compressed binaries using dedicated hardware[11, 14], using template bits to indicate when back-to-back packs are independent and can be executed in parallel [15], or have supported compact operations, which take up less space than normal operations inside of a single pack, allowing for some packs to have more operations than traditionally allowed [16, 17].

Instruction Cache Power: There are many cache improvements that have been targeted for traditional processor designs, and many are not mutually exclusive to the SLA processor. A notable technique is like drowsy ICs [10], which reduce power via reducing cache leakage, and while the original work does this speculatively, the SLA processor can instead inform this using lane states. Similarly, there are many techniques involving tools to predict either the cache way or subdividing ICs to only access parts of the cache that likely contain the data [5, 9, 19]. The IC design in the SLA processor gains efficiency using similar principles by having an IC for each lane that is responsible for that specific lane's instructions.

9 Conclusion

We provide a brand new architectural paradigm in the form of the SLA architecture, including the hardware and corresponding program representation. The SLA representation allows for programs to be represented as a set of separate instruction streams capable of synchronizing at an instruction level and provides a foundation for further work in parallel program representation and execution. Furthermore, unlike VLIW, SLA circumvents the need for explicit alignment via *nops*, increases instruction cache performance, improves backwards compatability of binaries, and allows for new scheduling opportunities. Our SLA proof-of-concept shows how the native representation can be robust against low-IPC sections of code, reduces code size, simplifies porcessor scalability, and introduces opportunities for power savings, without the need for additional mechanisms and while maintaining similar performance and believe it has significant potential as a new design space.

Acknowledgements

This work was supported in part by the US National Science Foundation (NSF) under grants CCF-1900788, CCF-1901005, OISE-2103103, OISE-2103105, CCF-2211353, and CCF-2211354. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of the NSF. We would also like to thank the reviewers, whose valuable time, effort, and feedback helped improve the clarity and quality of the overall work.

Appendix A. Processor Performance Considerations

This appendix provides more detailed information on the SLA implementation and several performance considerations that are not necessary for correctness, or the core ideas presented in the paper.

When a branch instruction immediately follows a PB instruction, the PB instruction would be decoded at the same point that the BTB is being accessed, meaning the PB register is stale by the point of a control flow change from a jump or taken branch. To ensure the PB register is correct by the end of the fetch stage, we propose pre-decoding *pb* instructions at the point of insertion into an L1 IC. When a line is filled in the L1 IC for a nonzero lane, we examine the opcodes of the instructions loaded in for any *pb* instructions. When a *pb* opcode is detected, we formulate the target address using the address in the *pb* instruction and the most significant bits of the PC and then mark a bit for the L1 IC entry called the PB-target *(PBT)* bit. Only the lower 31 bits are stored to preserve the *sr* bit of the instruction. The most significant bit can be reconstructed using the lane PC when the instruction is fetched. When a PB instruction is fetched, the *sr* bit is handled the exact same way as all other instructions, but the remaining fetched data comprising the target address is simply moved into the PB register for that lane.

Speculation is supported similar to a normal processor, but special considerations must be made to support speculation of lane updates. Upon a rollback, each lane must restore both the lane PC and status, since lanes could have been enabled or disabled on a wrong path.

LS registers can be repaired using the state of the processor at EX using similar mechanisms to PC, but lane PCs need to be calculated with the repaired LS states. If a load excepts, a similar step needs to be taken to ensure the pack properly replays. Program counters also need to be rolled back to the state that the prediction is made, including verifying suspended and halted lanes did not do so speculatively.

However, PB registers do not need to be restored. Since PB instructions are always scheduled prior to the next branch or jump instruction, the PB registers will be correct by the time they are read and used, even when incorrect PB targets may have speculatively written to those registers. Similarly, the *RT* and *PLS* registers are not speculatively written. So they do not need additional hardware support for speculation.

References

[1] 2002. Philips Trimedia Processor Home Page. www.semiconductors.philips.com/trimedia

[2] Analog Devices. 2005. ADSP-TS101 TigerSHARC processor programming reference. https://www.analog.com/media/en/dsp-documentation/processor-manuals/34605284816196ts201_pgr.pdf

[3] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25. doi:10.1145/3085572

[4] Anthony Brandon, Joost Hoozemans, Jeroen van Straten, Arthur Lorenzon, Anderson Sartor, Antonio Carlos Schneider Beck, and Stephan Wong. 2015. A sparse VLIW instruction encoding scheme compatible with generic binaries. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–7. doi:10.1109/ReConFig.2015.7393361

[5] Eui-Young Chung, Cheol Hong Kim, and Sung Woo Chung. 2008. An Accurate and Energy-Efficient Way Determination Technique for Instruction Caches by Early Tab Matching. In *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*. 190–195. doi:10.1109/DELTA.2008.57

[6] John L Hennessy and David A Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc. 192–196 pages.

[7] Yuan-Shin Hwang and Jia-Jhe Li. 2010. On reducing load/store latencies of cache accesses. *Journal of Systems Architecture* 56, 1 (2010), 1–15. doi:10.1016/j.sysarc.2009.10.001

[8] Taisong Jin, Minwook Ahn, Donghoon Yoo, Dongkwan Suh, Yoonseo Choi, Do-Hyung Kim, and Shihwa Lee. 2014. Nop compression scheme for high speed DSPs based on VLIW architecture. In *2014 IEEE International Conference on Consumer Electronics (ICCE)*. 304–305. doi:10.1109/ICCE.2014.6776016

[9] Cheol Hong Kim, Sung Woo Chung, and Chu Shik Jhon. 2006. PP-cache: A partitioned power-aware instruction cache architecture. *Microprocessors and Microsystems* 30, 5 (2006), 268–279. doi:10.1016/j.micpro.2005.12.004

[10] Nam Sung Kim, K. Flautner, D. Blaauw, and T. Mudge. 2002. Drowsy instruction caches. Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* 219–230. doi:10.1109/MICRO.2002.1176252

[11] Haris Lekatsas, Jörg Henkel, and Venkata Jakkula. 2002. Design of an one-cycle decompression hardware for performance increase in embedded systems. In *Proceedings of the 39th annual Design Automation Conference*. 34–39. doi:10.1145/513918.513929

[12] R. S. Nikhil. 1989. Can dataflow subsume von Neumann computing? *SIGARCH Comput. Archit. News* 17, 3 (April 1989), 262–272. doi:10.1145/74926.74955

[13] S. Onder and R. Gupta. 1998. Automatic generation of microarchitecture simulators. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*. 80–89. doi:10.1109/ICCL.1998.674159

[14] Seok-Won Seong and Prabhat Mishra. 2006. A bitmask-based code compression technique for embedded systems. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design* (San Jose, California) *(ICCAD '06)*. Association for Computing Machinery, New York, NY, USA, 251–254. doi:10.1145/1233501.1233551

[15] H. Sharangpani and H. Arora. 2000. Itanium Processor Microarchitecture. *IEEE Micro* 20, 5 (sep 2000), 24–43. doi:10.1109/40.877948

[16] Zheng Shen, Hu He, Xu Yang, Di Jia, and Yihe Sun. 2009. Architecture design of a variable length instruction set VLIW DSP. *Tsinghua Science and Technology* 14, 5 (2009), 561–569. doi:10.1016/S1007-0214(09)70118-X

[17] Texas Instruments. 2010. CPU and instruction set reference guide. *Texas Instruments, Literature Number SPRUGH7* (2010).

[18] Yuan Xie, Wayne Wolf, and Haris Lekatsas. 2002. Code compression for VLIW processors using variable-to-fixed coding. In *Proceedings of the 15th International Symposium on System Synthesis* (Kyoto, Japan) *(ISSS '02)*. Association for Computing Machinery, New York, NY, USA, 138–143. doi:10.1145/581199.581231

[19] Chun-Chang Yu, Yu Hen Hu, Yi-Chang Lu, and Charlie Chung-Ping Chen. 2021. Power Reduction of a Set-Associative Instruction Cache Using a Dynamic Early Tag Lookup. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1799–1802. doi:10.23919/DATE51398.2021.9474191

[20] Hongtao Zhong, K. Fan, S. Mahlke, and M. Schlansker. 2005. A distributed control path architecture for VLIW processors. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 197–206. doi:10.1109/PACT.2005.5

Received 2026-03-19; accepted 2026-05-01