Yoga: A Hybrid Dynamic VLIW/OoO Processor

Carlos Villavieja José A. Joao Rustam Miftakhutdinov Yale N. Patt

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2014-001
March 2014

This page is intentionally left blank.

Yoga: A Hybrid Dynamic VLIW/OoO Processor

Carlos Villavieja José A. Joao Rustam Miftakhutdinov Yale N. Patt

Department of ECE
Univ. of Texas at Austin
{villavieja,joao, rustam,
patt}@hps.utexas.edu

Abstract

Out of order (OoO) processors use complex logic to maximize Instruction Level Parallelism (ILP). However, since the resulting dynamic instruction schedule of many applications seldom changes, it is reasonable to store and reuse the schedule instead of reconstructing it each time. To do this, we propose Yoga, a hybrid VLIW/OoO processor that dynamically stores instruction schedules generated while in OoO mode as enlarged basic blocks for future VLIW execution. The enlarged basic blocks are transformed into VLIW words, where the width of the VLIW words corresponds to the ILP found in each block. Yoga switches between OoO mode and VLIW mode, depending on the costs and benefits at each point in time, saving power by turning off OoO structures and unused VLIW lanes when in VLIW mode. Yoga reduces energy consumption by 10% on average, with a maximum of 18%, with negligible performance loss with respect to a conventional OoO processor.

1. Introduction

Out of Order (OoO) processors require a large amount of power to dynamically schedule instructions [16]. The necessary complex logic, which includes structures such as reservation stations, renaming logic, and the reorder buffer, allows the processor to continue doing useful work in the shadow of long latency instructions and cache misses. However, this complex logic consumes a lot of energy even in the absence of these long latency events.

In contrast, Very Long Instruction Word (VLIW) processors rely on static scheduling of instructions, eliminating the need for the complex scheduling logic of OoO processors.

The power efficiency of static scheduling and the high performance of dynamic scheduling suggest the following: dynamically construct a high performance instruction schedule and amortize the associated energy cost by reusing this schedule multiple times.

The feasibility of reusing instruction schedules is supported by prior work. Palomar et al. [17] have shown experimentally that the OoO execution schedule of micro-ops in a large instruction window rarely changes between different dynamic instances of the same static code. Thus, it is reasonable to store the instruction scheduling information and reuse it rather than reconstruct it each time with the power-hungry OoO scheduling logic.

These observations suggest Yoga. Yoga is a hybrid processor design that dynamically switches between OoO and VLIW execution. Programs start executing in OoO mode. When a sequence of basic blocks connected by easy to predict branches is determined to be frequently executed, Yoga creates an enlarged basic block (which we call a

frame). A fill unit in the front end builds the frame which contains decoded micro-ops. The fill unit encodes the uops in VLIW format based on the dynamic instruction schedule generated during OoO execution, and stores the resulting VLIW frame in a VLIW cache. Subsequent executions of the frame are done in VLIW mode as long as the VLIW performance is within a pre-established margin from the OoO performance. In VLIW mode, frames execute as atomic units. A branch misprediction in VLIW mode rolls back the state of the machine to the beginning of the frame, then execution resumes in OoO mode.

Since the VLIW cache contains decoded micro-ops, significant parts of the processor front end (fetch and rename) are clock gated. Since most of the OoO structures are not needed during VLIW mode, they too are clock gated.

Additional power is saved in VLIW mode by dynamically adjusting the width of each VLIW word in each frame, based on the average ILP in the frame. This optimization makes the VLIW traces more efficient by minimizing the number of empty scheduled slots (no-ops), and saves power by turning off the functional units (lanes) that are not used by the current frame. Additionally, by using enlarged basic blocks, VLIW execution can benefit from the wider functional units (up to 6) and sometimes increase ILP with respect to OoO.

Cycle accurate simulation of all SPEC CPU 2006 benchmarks shows that Yoga saves 10% of the total energy on average, with a maximum of 18% without degrading performance over a traditional 4-wide OoO processor. On average, Yoga executes 20% of all dynamic instructions in VLIW mode.

The contributions of this paper are:

- We introduce Yoga, the first hybrid processor that can run in OoO mode, where it uncovers and exploits ILP to maximize performance, and in VLIW mode, where it reuses OoO scheduling information to save power without significantly degrading performance.
- We design a dynamic variable-length VLIW execution stage that can adapt to the actual ILP of an enlarged basic block. In VLIW mode, this design can save additional power by turning off the functional units that are not used when ILP is low.
- We introduce a simple controller that dynamically decides whether to execute a frame in VLIW mode based on performance feedback. It also decides when mode switching will result in minimal performance loss based on Reorder Buffer occupancy.

2. Background

2.1. VLIW vs OoO

We use the Nvidia Tegra 4 as an example [16] to discuss the energy breakdown of a state-of-the-art mobile processor. Tegra 4 uses an ARM Cortex-A15 OoO processor with energy efficiency as one of the main design goals. The

processor front end [16] (instruction fetch, decode and branch prediction) consumes 40% of the CPU power. By using a loop buffer about half of the front end power is saved. However, a large part of the power budget (15-20%) is still spent in the OoO logic (e.g., instructions must still be renamed, and inserted in Reservation Stations (RS) and in the Reorder Buffer (ROB)).

Applications tend to have repetitive execution patterns. Having this in mind, we observed that the instruction schedules produced by the OoO scheduling logic are similar for multiple dynamic instances of the same code segments. Therefore, it is possible to save and reuse the scheduling information from previous executions to improve power-efficiency.

An ideal candidate architecture to reuse execution information is VLIW. VLIW architectures are very energy efficient because their datapath is simple and there is no need for large RS and ROB. However, VLIW processors rely on the compiler to produce static schedules and handle dependency checking. The compiler can produce enlarged basic blocks to increase Instruction Level Parallelism (ILP).

Considering the significant 40% power reduction that Tegra 4 achieves in the front end, there is significant potential to not only reuse decoded instructions from a loop buffer but also reuse OoO schedules.

2.2. Enlarged basic blocks

As the number of instructions in a block increases more optimization opportunities exist. Several researchers have proposed enlarged basic blocks for optimization purposes [6, 13, 18]. When several basic blocks are linked by easy to predict branches, it is possible to merge the basic blocks and schedule instructions across the entire enlarged basic block. Thus, the bigger the block, the easier it becomes to find independent instructions that can execute in parallel, increasing ILP and potentially reducing total execution time.

To quantify the potential of enlarged basic blocks (BBL) we find the fraction of the total dynamic instructions that are part of potential enlarged blocks of different sizes. Starting from a hard to predict branch, we count dynamic instructions and basic blocks scheduled in the reservation stations until the next hard to predict branch. Those instructions could become an enlarged basic block. In Figure 1, the bar labeled BBL-*N* is the fraction of dynamic instructions that belong to a enlarged basic block formed by merging *N* consecutive basic blocks. While there is significant variability across benchmarks, this experiment shows that enlarged basic blocks of up to three basic blocks could potentially cover, on average, around 70% of the dynamic instruction trace. This shows the coverage that could be achieved by dynamically scheduling instructions over different enlarged basic blocks sizes.

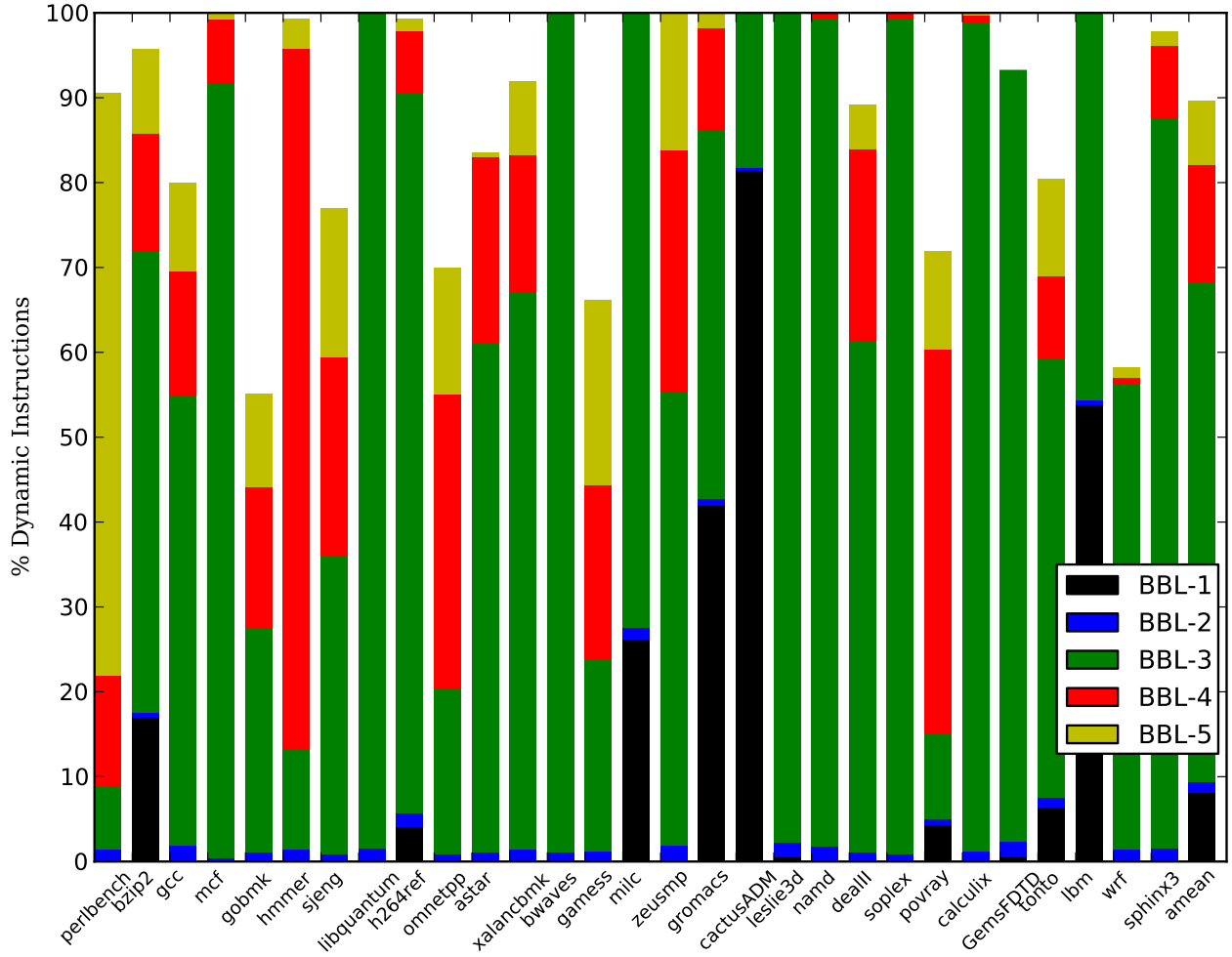


Figure 1. Percentage of instructions in enlarged basic blocks

2.3. VLIW vs In-Order

In-order execution processors consume less power than OoO processors because they require a much simpler microarchitecture[1]. However, a traditional in-order processor still requires hardware to a) keep track of instruction dependencies and stall instruction execution until all operands are ready, and b) commit instructions executed in parallel in different functional units in-order. VLIW processors consume even less power than in-order processors. Instruction dependence checking and scheduling rely on the compiler. Also, all instructions in a VLIW word are independent and execute in parallel and in lockstep. Therefore, a VLIW processor provides a very efficient implementation of in-order execution.

2.4. A hybrid VLIW/OoO dynamic processor

OoO execution provides higher performance (greater ILP) if the code is irregular or dynamic execution latencies can benefit from run-time decisions of instruction scheduling [12]. If scheduling decisions can be made at compile

[1]Since in-order hardware is much simpler than OOO hardware, it switches less capacitance per cycle (and leaks less static energy per cycle) as compared to OOO, and thus consumes less power than OOO.

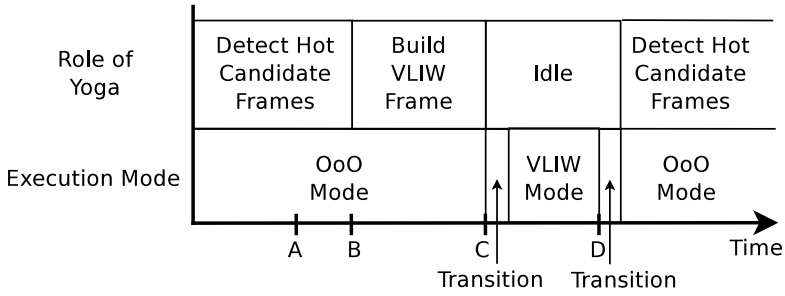
time, VLIW execution provides the benefit of energy efficiency.

Our goal is to design a hybrid VLIW/OoO processor that dynamically combines the advantages of both execution modes: high-performance execution of irregular code and energy-efficient execution of regular code.

3. Mechanism

3.1. Overview

Yoga is designed as a regular OoO processor with additional components to dynamically build VLIW code based on the OoO instruction schedules and then run in either VLIW mode or OoO mode to improve the performance-energy trade-off. Figure 2 shows the timeline of how Yoga[2] builds and executes a VLIW frame. While Yoga executes in OoO mode, the retired instruction stream is monitored to detect *hot candidate frames* that could better run in VLIW mode, i.e., frequently-executed sequences of basic blocks[3] connected by easy to predict branches. When a hot candidate frame is detected, the next time the processor executes it, the OoO scheduled instruction stream is used to build a VLIW frame, which is stored in a VLIW Cache. The next time the frame is about to start executing, the processor switches to VLIW mode and executes instructions from the VLIW Cache instead of from the OoO front end, which is clock-gated. Execution continues in VLIW mode while the actual control flow matches a sequence of one or more frames present in the VLIW Cache. Also, performance for each frame is monitored in both OoO and VLIW modes, so that if the VLIW performance becomes less than 90% of the OoO performance, the frame is identified to be executed in OoO mode.



A. Hot candidate frame identified for VLIW
B. Hot candidate frame begins execution in OoO mode for the first time since identification
C. Branch Target PC and history hit in the Frame Confidence Table
D. Branch Target PC and history miss in the Frame Confidence Table

Figure 2. An example of Yoga timeline execution

Figure 3 shows the high level components of Yoga, with the new blocks required for VLIW mode shaded. The Frame Confidence Table (FCT) receives the retired control instructions in OoO mode, off the critical path of execution,

[2]The name Yoga is inspired by the VLIW mode, which we consider a *relaxed* mode of operation.

[3]For our purposes, a basic block is a sequence of contiguous instructions that starts with the target of a control instruction and ends with the next control instruction.

and tracks the sequences of basic blocks that would be good candidates to become VLIW frames. Once a hot candidate frame is detected and is not present in the VLIW cache, the FCT notifies the I-cache to mark the starting instruction of the frame for VLIW frame creation. The next time the potential frame starts executing, the OoO scheduled instruction stream, i.e., all micro-ops, are sent to the VLIW Fill Unit (VFU), which builds a VLIW frame and stores it in the VLIW Cache (VC). Each ROB entry is augmented with the dispatch cycle to keep the order of instructions to FUs (scheduling order). The retirement stage is responsible for sending this information to the VFU. While running in OoO mode, the FCT is accessed with the predicted target of every control instruction and if there is a VLIW frame in the VC, the processor can switch to VLIW mode. The Yoga Controller (YC) block includes the logic to decide when to switch modes and effectively control power gating and clock gating of different blocks during the mode transitions.

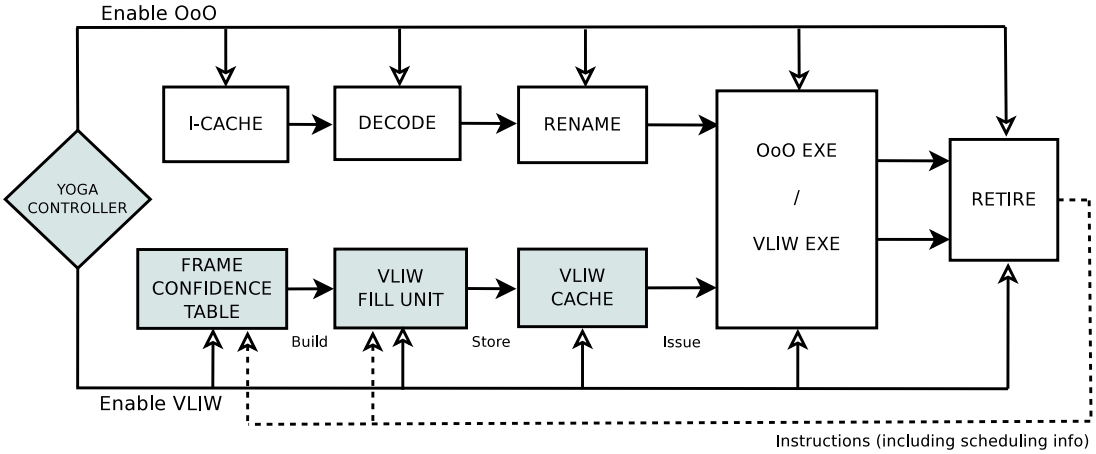


Figure 3. Yoga high level architecture

3.2. Frame identification

A frame that is a hot candidate for VLIW execution is a sequence of basic blocks connected by easy to predict branches, which have been executed more than a minimum number of times.

Easy-to-predict branch detection. Unconditional jumps and calls are considered easy to predict branches. For conditional and indirect branches, we extend the BTB to identify easy to predict branches. We add a 2-bit saturating counter *Easy-to-predict* to each entry. *Easy-to-predict* is incremented when a branch is predicted correctly and reset when it is mispredicted. A branch is considered easy to predict only after *Easy-to-predict* reaches the maximum count (3).

Frame Confidence Table (FCT). The FCT is a cache indexed by a combination of the starting PC and branch history of the first basic block of a frame. Each FCT entry corresponds to a potential frame, and includes the branch direction of the branches. A frame can have up to 16 basic blocks. Additionally, the FCT entry includes all the addresses of the

control instructions in the frame. The FCT also includes an execution counter *ExecCnt* that is incremented when the frame completes execution in OoO mode. When *ExecCnt* reaches a threshold, the frame is identified as a candidate for VLIW execution. The FCT might contain several frames starting with the same address but they would always have different branch history.

3.3. VLIW frame construction

Figure 4 shows a code example, the OoO execution through a dataflow diagram and the steps to construct the corresponding VLIW frame.

OoO schedule recording. When the FCT decides to start building a VLIW frame, it sets a *StartRecording* bit in the I-cache for the first instruction of the first basic block in the hot candidate frame. While executing in OoO mode, the marked instruction and the following instructions in program order are sent to the VLIW Fill Unit in scheduling order. OoO schedule recording is enabled for all instructions in each basic block until the last control instruction of the frame. The address of every control instruction encountered must match the saved addresses in the FCT; if they do not match, recording is cancelled. When an instruction is fetched its *StartRecording* bit is cleared, so that recording only happens once.

VLIW Fill Unit (VFU). The VFU builds VLIW frames based on the information about the stream of micro-ops in the OoO schedule recording (see Figure 4). Micro-op information includes opcode, operands, destination and dispatch cycle. All control micro-ops include their target.

In VLIW execution, Yoga uses a smaller VLIW Register File (VRF). As each recorded micro-op is received from the OoO scheduler, the VFU places it in a VLIW word in a Word Table. Thus, the sources and destination are renamed to avoid the need for renaming at VLIW mode runtime. Branches are replaced by assert micro-ops that test the evaluated target of the branch against the target stored in the micro-op.

Since each micro-op is processed by the VFU in a valid OoO retirement order, all micro-ops producing the source operands for future consumers have already been processed and placed in a VLIW word. Therefore, all micro-ops are added to a VLIW word subsequent to the words producing their source operands.

Since micro-ops have already been renamed for OoO execution, there are no anti-dependence or output dependence violations. Since internal branches become asserts within a frame, control dependencies do not have to be considered and micro-ops are automatically scheduled across basic block boundaries according to data dependencies.

Each VLIW word in a frame is uniquely identified and has a prefix that includes the identifier of the latest VLIW word that it depends on. Thus, we guarantee that all source operands for all ops in the word will be ready before it executes. This simple dependence check mechanism avoids the need for empty VLIW words when there are not

enough useful ops to insert between a producer and consumer word, and also potentially stalls on the first use of the result of a load instruction that misses in the cache.

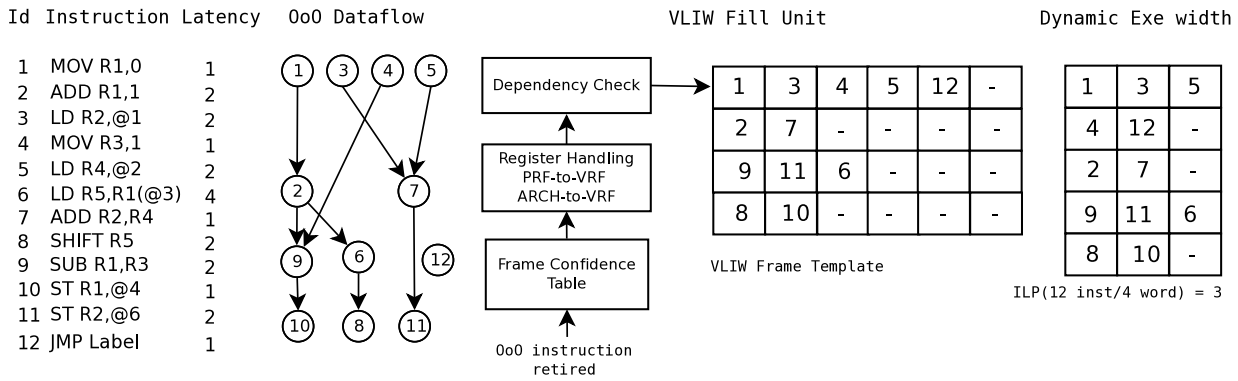


Figure 4. Example code with its OoO execution data-flow and steps to build the VLIW Frame through the FCT and the VFU

Register handling. VLIW frames are atomic units of execution. Live-ins are the values that are used but not produced inside the frame and have to be read from the initial architectural state. To simplify register handling, live-ins are specified by their architectural register IDs. Live-outs are the last values written to any architectural register inside the frame, which have to be written back to the architectural state when the VLIW frame commits. Any value is created and destroyed (i.e., its architectural register is overwritten) inside the frame is treated as a temporary register and statically assigned to a physical register in the VRF. The VRF is split into two blocks: one block *Arch-out* that holds live-outs and directly maps to the architectural registers and one block *VRF-temp* that holds temporary registers. Note that the micro-ops sources and destinations are specified with both architectural and PRF IDs when sent to the VFU. To detect live-ins and live-outs, and to map PRF registers to VRF registers during frame creation, the VFU uses two simple tables: *PRF-to-VRF* and *ARCH-to-VRF*.

The *PRF-to-VRF* table is indexed by PRF ID and contains the VRF register ID VRF_ID, the $VLIW\_word$ (i.e., cycle within the frame) that produces the value, the architectural register ID $Arch$ corresponding to the PRF entry, the $Latency$ of the micro-op producing the value, and a $Valid$ bit which is reset at the beginning of the construction of a new frame. A micro-op that sources a PRF entry whose ID has not been seen yet (invalid entry in *PRF-to-VRF*) is actually reading a live-in, for which a new valid entry in the *PRF-to-VRF* table is set with $VLIW\_word = 0$ and the $Arch$ ID of the register, necessary to read the value from the initial architectural state. A micro-op that writes to a PRF entry updates the corresponding entry with a new unused VRF_ID in the temporary register block, and the $Latency$ of the operation (computed as the difference between retirement cycle and dispatch cycle). A micro-op that reads from a PRF entry that already has a valid entry in the *PRF-to-VRF* can determine the ready time of the operand

as $VLIW\_word + Latency$. To avoid stalls, a micro-op should be placed in a VLIW word that will execute when all its source operands are ready. Therefore, a micro-op with source operands S_i produced by micro-ops with latencies L_i already placed in words W_i should be placed no earlier than VLIW word $max(W_i + L_i)$.

The *ARCH-to-VRF* table is used to find live-outs and is indexed by architectural register ID. Each entry in this table contains VRF_ID, the $VLIW\_word$ that last wrote to this architectural register, the micro-op program order $Prog\_order$ and a $Valid$ bit. An entry is updated for each value produced by any micro-op with a later $Prog\_order$ than the existing one. After placing all micro-ops in the VLIW frame, the *ARCH-to-VRF* table contains the VRF registers that should be treated as live-outs instead of temporary registers. For that purpose, VLIW words are scanned starting at $VLIW\_word$ and all references (i.e., reads and writes) to VRF_ID are replaced with accesses to an $Arch\_ID$ register in the *Arch-out* VRF block that maps to architectural state.

Dynamic execution width. After all micro-ops in a VLIW frame are placed, VLIW words may have different numbers of micro-ops. To increase energy-efficiency, the VFU determines the average width, rounds it up, and makes a pass over all VLIW words to adjust the maximum width to that new width. To avoid complicating dependence checking and scheduling, words that are longer than the new width are simply split as necessary to fit in the new effective width. The effective width of the VLIW frame is also stored in the FCT entry (*FU_Cnt*) to be able to dynamically power gate the functional units that will not be used for the execution of the VLIW frame.

Note that in the example code in Figure 4, the ops with higher latency are kept in the first of the two words that resulted from splitting word 1 (op 5 latency is greater than op 1). This ensures that all values are produced as early as possible, which reduces the likelihood of pipeline bubbles/stalls.

VLIW Cache (VC). The VLIW Fill Unit uses a VLIW frame template to construct the VLIW words (see Figure 4). When a VLIW frame's construction is complete, it is installed in the VLIW Cache. Each VLIW word contains a variable number of ops and includes a header that stores the word width. The FCT entry is updated with a *VC-present bit* when the VLIW frame is installed. The FCT entry also contains the VLIW word width, a *Live-ins* mask with one bit set for each architectural register that is a live-in, and a pointer to the first VC line containing the VLIW frame. If a frame requires multiple cache lines, they are connected as a simple linked list. The FCT is also responsible for managing the VLIW Cache evictions.

3.4. OoO to VLIW transition

When the OoO front end is about to start fetching from the target of a control instruction (which comes from the BTB), the target PC and branch history are used to index the FCT. If there is a hit in the FCT, the *VC-present* bit is set, and the Yoga Controller (explained in Section 3.7) agrees, Yoga starts a transition to VLIW mode. The processor stops

fetching from the I-cache and starts fetching and executing from the VLIW Cache. In case of a branch misprediction inside the frame, the VLIW transition is aborted. The processor invalidates the VLIW instructions in-flight, and resumes executing in OoO mode from the target address, which is simple given that no changes to the RAT or PRF were made in VLIW mode.

The *Live-ins* mask in the FCT entry allows the processor to immediately start populating the *Arch-in* table with the values of the live-in architectural registers, read from the PRF through the RAT (step 1 in Figure 5). Pending OoO instructions write their results to the PRF and if necessary, also to the *Arch-in* table, making the value ready. A VLIW word executes as soon as all of its live-in operands are ready. Meanwhile, the functional units may be executing both OoO ops and VLIW ops during the transition from OoO to VLIW mode.

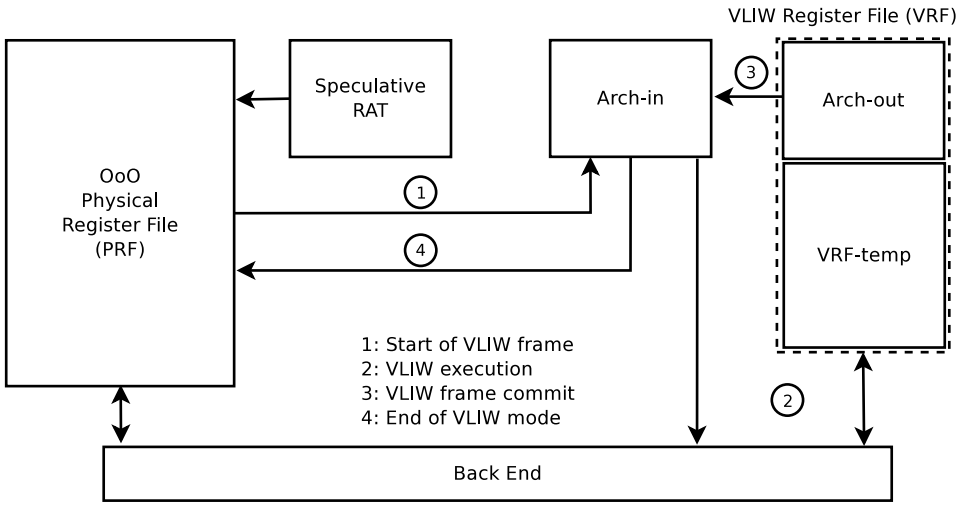


Figure 5. Register handling in VLIW mode and transitions

3.5. VLIW execution

Dynamic execution width. The VLIW width specified in the FCT entry is used to determine how many functional units are needed to execute the frame, while keeping the rest of the functional units power gated. To hide as much of the power gating latency as possible, an increase in VLIW width is started as soon as the new VLIW width is known. However, a decrease in VLIW width is delayed until the last VLIW word in the previous VLIW frame commits.

Register read and write. VLIW ops reading live-ins specify an architectural register, whose value is read from the *Arch-in* table after the value is available (step 2, Figure 5). VLIW ops reading or writing VRF registers access the VLIW Register File. In particular, VLIW ops writing or reading live-outs were set to access the VRF block that directly maps to architectural state *Arch-out*. To reduce the stall time for live-outs from the previous VLIW frame before that frame commits, *Arch-out* is accessed and any valid value found there takes precedence over the register

value in *Arch-in*. Meanwhile, any op writing to a valid VRF register stalls until the previous frame commits and clears the VRF.

Stores are kept in the Load-Store Queue (LSQ) until the VLIW frame either commits or rolls back. Meanwhile, the LSQ is used for Store-to-Load forwarding within the VLIW frame.

Memory disambiguation. The OoO schedule that was used to build the frame was based on memory dependencies for the memory addresses generated in the recorded instance of that frame. Thus, non-conflicting loads and stores may have executed out of program order. When the VLIW frame is executing, the actual memory addresses may generate different memory dependencies. To ensure correct execution, each load and store in a VLIW frame includes a memory sequence order field. Thus, the LSQ handles memory disambiguation as in OoO mode, and violations cause the VLIW frame to be rolled back.

VLIW frame commit is atomic by simply copying any entries from the architectural VRF block *Arch-out* to the *Arch-in* table holding the architectural state, which is easy because there is a 1-to-1 correspondence between the two structures (step 3, Figure 5). Additionally, any stores in the LSQ that are part of the committing frame are made visible to the memory system.

Determining the next VLIW frame. The FCT entry contains the address of the branch that ends the VLIW frame. Therefore that branch can be predicted before the frame finishes. The branch history at the beginning of the frame is updated with the branch history for the frame's branches (contained in the FCT). The new branch history plus target PC are used to look up for a possible next frame in the FCT. In case of a hit, the processor remains in VLIW mode and continues executing the next VLIW frame. Otherwise, it transitions to OoO mode when the current frame finishes.

3.6. VLIW to OoO transition

Yoga can go back to OoO mode in two cases: when the predicted target address after the current VLIW frame is not another VLIW frame present in the VLIW cache and enabled for VLIW execution by the Yoga Controller, or when the current frame cannot commit. A frame cannot be committed in three cases: a control flow mismatch, a memory disambiguation conflict and exceptions. Rolling back the first VLIW frame executed in VLIW mode is easy because the RAT and PRF still contain the final state of the last OoO episode. If any VLIW frame has been committed, the last valid architectural state is in the *Arch-in* table. Therefore, transitioning into OoO mode requires merging any modified entries in the *Arch-in* table back into the PRF, going through the RAT that still holds the renaming state from the last OoO to VLIW mode transition (step 4, Figure 5). Meanwhile, any OoO instruction that sources or writes to an architectural register that has not been copied has to stall at rename time. In case of a rollback due to an exception, the exception is treated precisely when re-executing the exception causing instruction in OoO mode.

Additionally, all pending stores after the beginning of the rolled back frame are invalidated in the LSQ and fetching resumes in OoO mode from the starting address of the VLIW frame.

As soon as the decision to switch to OoO mode is made, the number of functional units that are powered on is set to be at least the width of the OoO configuration. When the last VLIW word retires, any unneeded functional units are power gated.

3.7. Yoga Controller

The Yoga Controller (YC) shown in Figure 3 is responsible for deciding whether to switch the execution mode from OoO to VLIW or vice versa due to performance trade-offs. Its objective is to skip VLIW execution of frames where there is a significant drop in performance. A VLIW frame is set as *VLIW-friendly* in the FCT entry if the frame does not lose more than 10% of IPC between OoO execution and VLIW execution. Then, only VLIW-friendly frames are allowed to run in VLIW mode.

When the YC finds out that a frame's performance is below the maximum performance loss limit, or is rolled back to OoO, the frame's entry in the FCT is reset: the *VLIW friendly* bit is cleared, the *ExecCnt* is reset and the threshold number of accesses to execute in VLIW is doubled.

The transition time is a key factor for VLIW frame performance. Thus, the controller checks the ROB occupancy before switching to VLIW. If the ROB occupancy is more than 50%, the controller does not allow a transition from OoO to VLIW. The insight behind this is that when more than 50% of the ROB is occupied, the transition takes longer. This is because there can be inflight (or previous) misses. Therefore, the number of instructions to commit is large enough to delay the transition significantly. Besides cache misses, a phase of long dependency chains, which would delay the transition, may also be detected by checking the ROB occupancy.

Additionally, we set a minimum number of instructions *MinFrameSize* for a VLIW frame to start a transition from OoO to VLIW. Once Yoga starts executing in VLIW, frames with fewer number of instructions than *MinFrameSize* are allowed to execute in VLIW. The reason behind this restriction is to amortize the cost of the transition. *MinFrameSize* is set to 64 in our evaluation.

3.8. Hardware cost

Table 1 shows the cost of the Yoga hardware components. The FCT is a cache to hold active frames. Each FCT line contains the following fields: start address(8B), number of words in the frame(1B), list of branches(max 16x8B), number of branches(4b), branch history(4B), OoO IPC(1B), VLIW IPC(1B), ExecCnt(1B), ExecCnt threshold(1B), Live-in mask(8B), VC Present(1b), VLIW Friendly(1b), FU_Cnt(3b) and VC pointer(7b). Total 155 bytes per cache line. The VLIW Frame Cache contains the decoded VLIW Frame. Each cache line is 128B. The VLIW Fill Unit is

Structure	Parameters	Cost
FCT	128 entries, 4-way, 2-cycle	19.4 KB
VC	128B lines, 2-cycle	32 KB
VFU	256 ops, 6-wide	3 KB
ROB	12b per entry (dispatch cycle)	1.3 KB
Arch-in	32 entries	260 B
VRF	64 entries	512 B
	Total storage cost	56.5 KB

Table 1. Hardware structures for Yoga and their storage cost

Front end		OOO Core (4/6)		VLIW mode		All Caches		ICache	DCache	L2
Uops/cycle	4/6 Uops/cycle		4/6 Uops/cycle		1-6 Line size		64 B Size	32 KB	32 KB	1 MB
Branches/cycle	2 Pipe depth		14/14 Pipe depth		8 MSHRs		32 Assoc.	4	4	8
BTB entries	4K ROB size		128/142 ROB size		- Repl.		LRU Cycles	3	3	18
Predictor	TAGE [23] RS size		48/72 RS size		-		Ports	1R/1W	2R/1W	1
	Func.Units		4/6 Func.Units		6					
	RF		128/128 RF		64					

DRAM Controller		Bus		DDR3 SDRAM [14]			Stream prefetcher [24]			
Policy	FR-FCFS [22]	Freq.	800 MHz	Chips	8×256 MB	Row size	8 KB	Streams	64	Distance	64
Window	32 requests	Width	8 B	Banks	8	CAS[b]	13.75 ns	Queue	128	Degree	4

Table 2. Simulated processor configurations for OoO and VLIW mode

mainly composed of logic to pack the VLIW frame and 2.5K of temporary storage for the VLIW frame construction. The total storage cost of Yoga is only 56.5 KB.

4. Methodology

Benchmarks: We use all of the SPEC CPU2006 benchmark suite for our simulations. We run each benchmark with the reference input set for 100M instructions selected using PinPoints [20].

Simulator: We use an in-house cycle-accurate and execution driven x86 simulator for our evaluation. We faithfully model all the processor pipeline stages and its structures, port contention, queuing effects, bank conflicts, and other memory system constraints.

Processor Backend: Table 2 shows the configuration of the baseline OoO processor. The extensions required by Yoga are described in Section 3.8.

Power: We use a modified version of McPAT 0.8 [10] to model chip power. We extended McPAT to support the new components (Frame Confidence Table, VLIW Fill Unit, VLIW Cache and VLIW Register File) used in Yoga. During the cycles Yoga runs in VLIW mode, the OoO stages/structures Fetch, Decode, Rename, ROB and OoO Register File are clock gated. During VLIW execution, we adjust the number of active functional units to the VLIW frame width,

thus, we power gate the functional units that are not needed. During OoO execution, two functional units are also power gated; the core acts as a regular 4-wide OoO processor.

5. Experimental Evaluation

5.1. VLIW mode coverage

VLIW coverage is the fraction of instructions Yoga executes in VLIW mode. To execute a frame in VLIW, several conditions must be satisfied. As described in Section 3.7, a frame must include a minimum number of instructions and a performance ratio (VLIW vs OoO IPC) to be considered for VLIW execution.

Figure 6 shows the VLIW coverage in percentage of instructions in VLIW for SPEC2006. Two different configurations are shown. The left bar of each application shows a relaxed configuration. This configuration does not discard frames unless they contain fewer than 16 instructions and no performance monitoring is done by the controller. The rightmost bar for each application shows a limited configuration, named as Yoga. In this configuration, each frame requires at least 64 instructions and can only lose 10% performance with respect to the OoO execution.

The relaxed configuration shows the potential time in VLIW mode for each benchmark. This configuration shows that the percentage of dynamic instructions that belong to basic blocks linked by easy to predict branches is 34% on average. Note that some applications can run in VLIW mode almost 100% for some applications.

Yoga configuration (restricted) shows an average of around 20% of the dynamic instructions where applications can run in VLIW mode without losing more than the established 10% performance. Benchmarks such as *milc*, *GemsFDTD* and *lbm* execute almost all the code in VLIW. The number of frames for these benchmarks is small compared to others, and there are a small number of cache misses, therefore, VLIW frames in these benchmarks obtain a similar performance to their OoO execution. Note that *GemsFDTD* has more VLIW coverage in the restrictive configuration (Yoga) than in the relaxed. The reason for this is that the relaxed configuration generates more small frames that do not fit in the FCT. However, the Yoga configuration generates larger frames producing fewer transitions, more consecutive frames in VLIW and therefore more VLIW coverage.

There are important differences in the coverage for some benchmarks. First, integer benchmarks (on the left side of Figure 6) have fewer instructions per basic block than floating point benchmarks (on the right side of the Figure 6). As an example, *gcc* requires at least 16 branches to build 64 instruction frames.

Second, the branch behavior is also harder to predict for integer benchmarks with the exception of perlbench. An alternative to branch predictability in these benchmarks would be to predicate conditional branches inside a frame. However, since the frames are constructed dynamically, this would increase the hardware complexity of the VLIW fill unit.

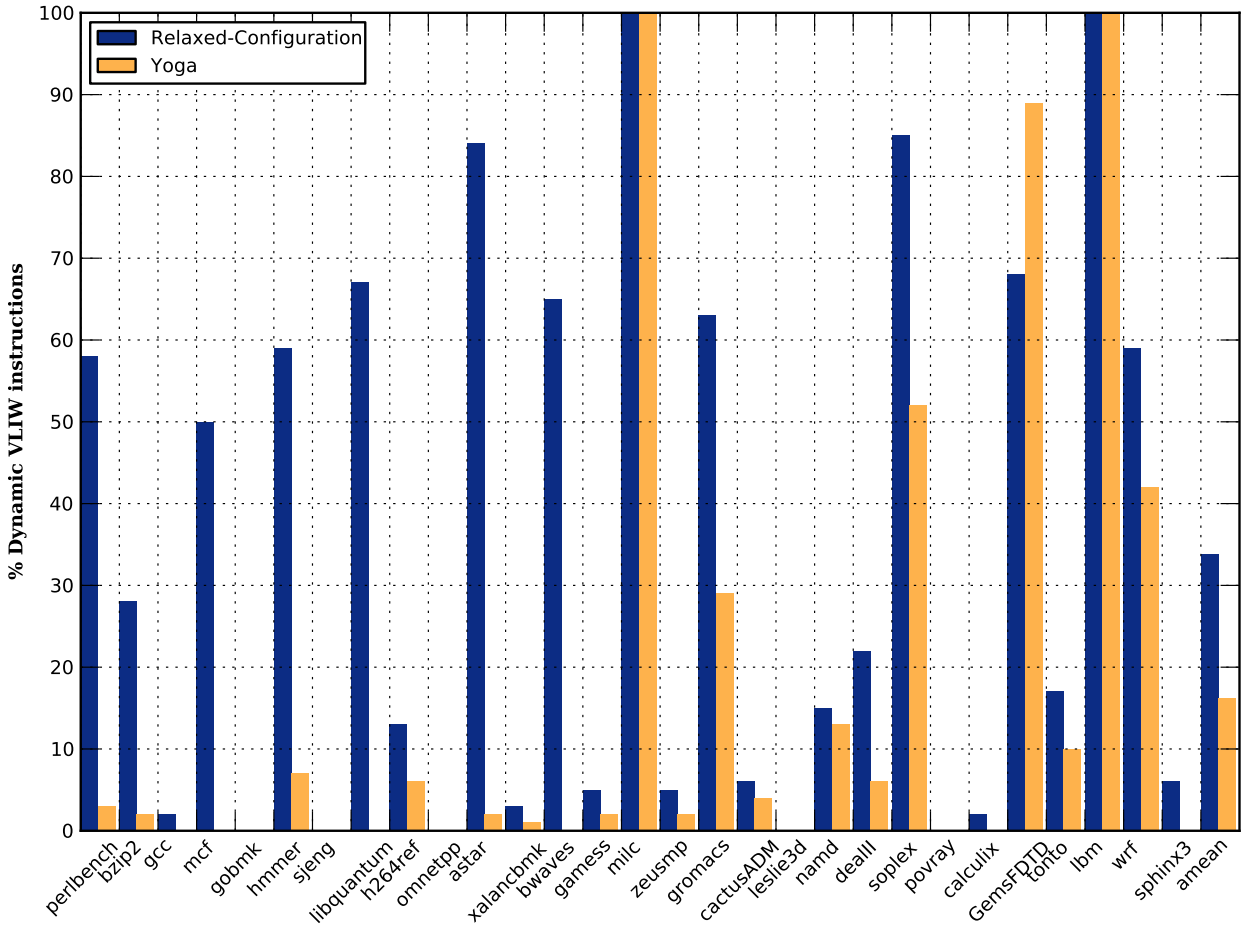


Figure 6. Percentage of dynamic instructions in SPEC2006 that are executed in VLIW mode using a relaxed and a limited configuration of the controller.

5.2. Performance comparison

Figure 7 shows the performance of Yoga compared to a 4-wide and 6-wide OoO processor. The OoO-4 acts as our baseline since the main goal of Yoga is to achieve the same performance at a lower power budget. Additionally, we include a 6-wide OoO configuration as a performance comparison point. Table 2 includes detailed information of both configurations.

When Yoga runs in VLIW mode, only the functional unit (FU) lanes that are needed are powered on. Since Yoga might consume less power in VLIW mode than the baseline 4-wide OoO, we allow the fill unit to build VLIW frames that use up to 6 FUs. However, since the average ILP of VLIW frames is usually lower, in VLIW mode Yoga tends to turn off functional units and require less than 4, reducing the static power of the FUs with respect to OoO-4 (see Table 3).

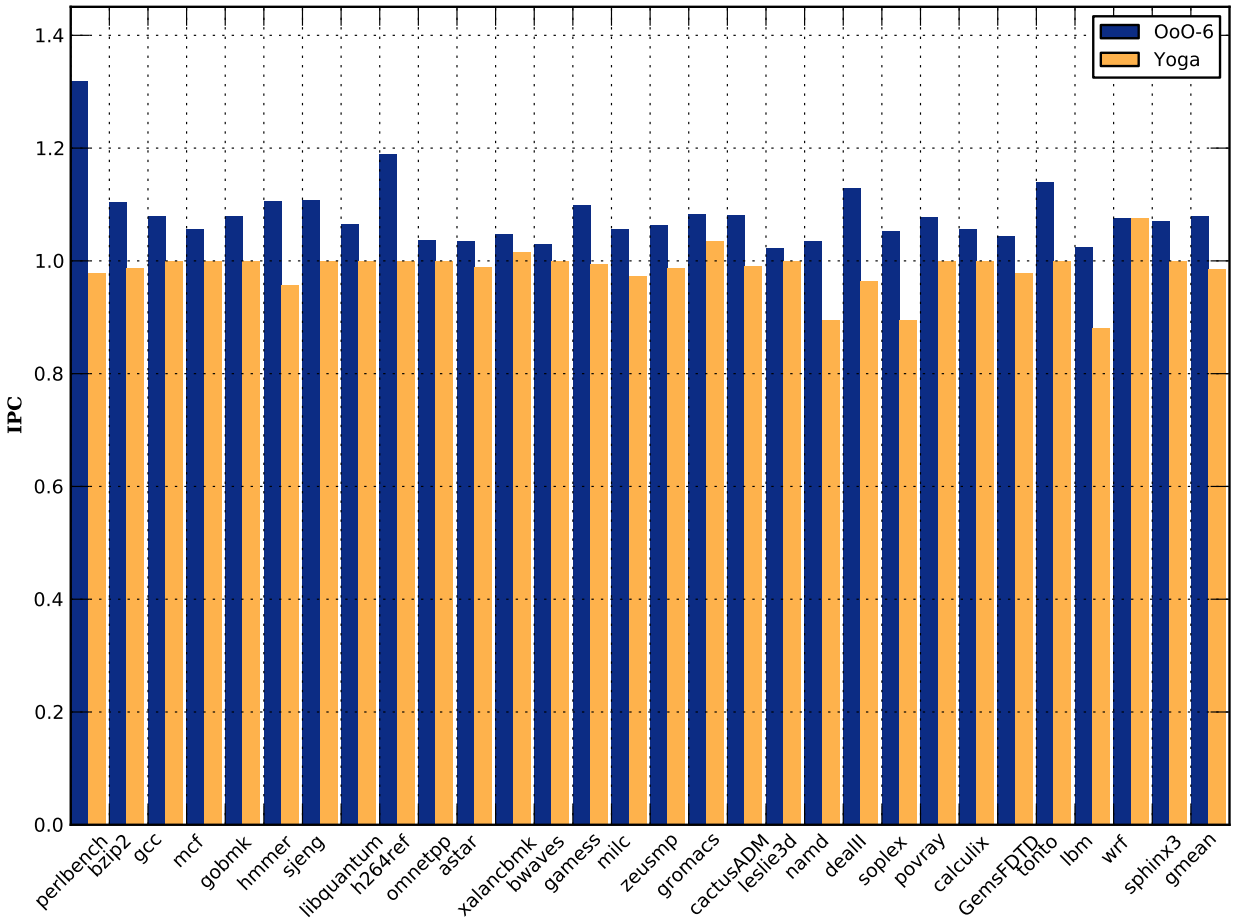


Figure 7. Performance of Yoga normalized to a regular 4-wide OoO and compared to an 6-wide OoO configuration

In Figure 7 we can observe that several benchmarks barely change their performance because they do not have more than 5% of VLIW coverage, see Figure 6. Benchmarks with less branch prediction accuracy and smaller basic blocks (mainly integer benchmarks) belong this category. *Xalancbmk*, *wrf*, *tonto* and *gromacs* achieve a small performance improvement with respect to the OoO-4 baseline, because some instructions are scheduled earlier in the VLIW frame achieving more ILP. Table 3 shows that *tonto* and *gromacs* utilize the extra functional units in some VLIW frames when executing in VLIW mode. Even the VLIW fill unit uses the OoO schedule, it has a complete view of all instructions in the frame. Thus, it can schedule independent instructions earlier. Note that the average number of instructions per frame is also large for these two benchmarks (75.72 and 97.08 respectively), much larger than the OoO RS size (48).

Namd, *soplex* and *lbm* suffer performance degradation in VLIW mode. Yoga captures and builds a large number of VLIW frames for *namd*. Therefore, more untrained/new frames are executed in VLIW mode before they are discarded

by the controller because of their low performance compared to OoO. *soplex* achieves less ILP in VLIW and thus its VLIW performance is low. *lbm* builds a small number of frames but it has high VLIW coverage. However, most of the frames achieved lower performance than OoO without reaching the 10% performance loss threshold to be discarded.

The configuration of the controller is set to allow a 10% IPC performance difference between the execution of OoO and VLIW frames. The controller is software programmable and it can be configured to discard frames within a more restrictive margin, however, the VLIW coverage might also be reduced significantly below the current 20%.

In general, by monitoring the IPC difference between modes, the controller discards VLIW frames that suffer additional cache misses in VLIW mode, which that might cost stalls in VLIW mode.

5.3. Energy Consumption

Figure 8, 9 show the static and dynamic energy consumption respectively of Yoga compared to the baseline 4-wide OoO processor.

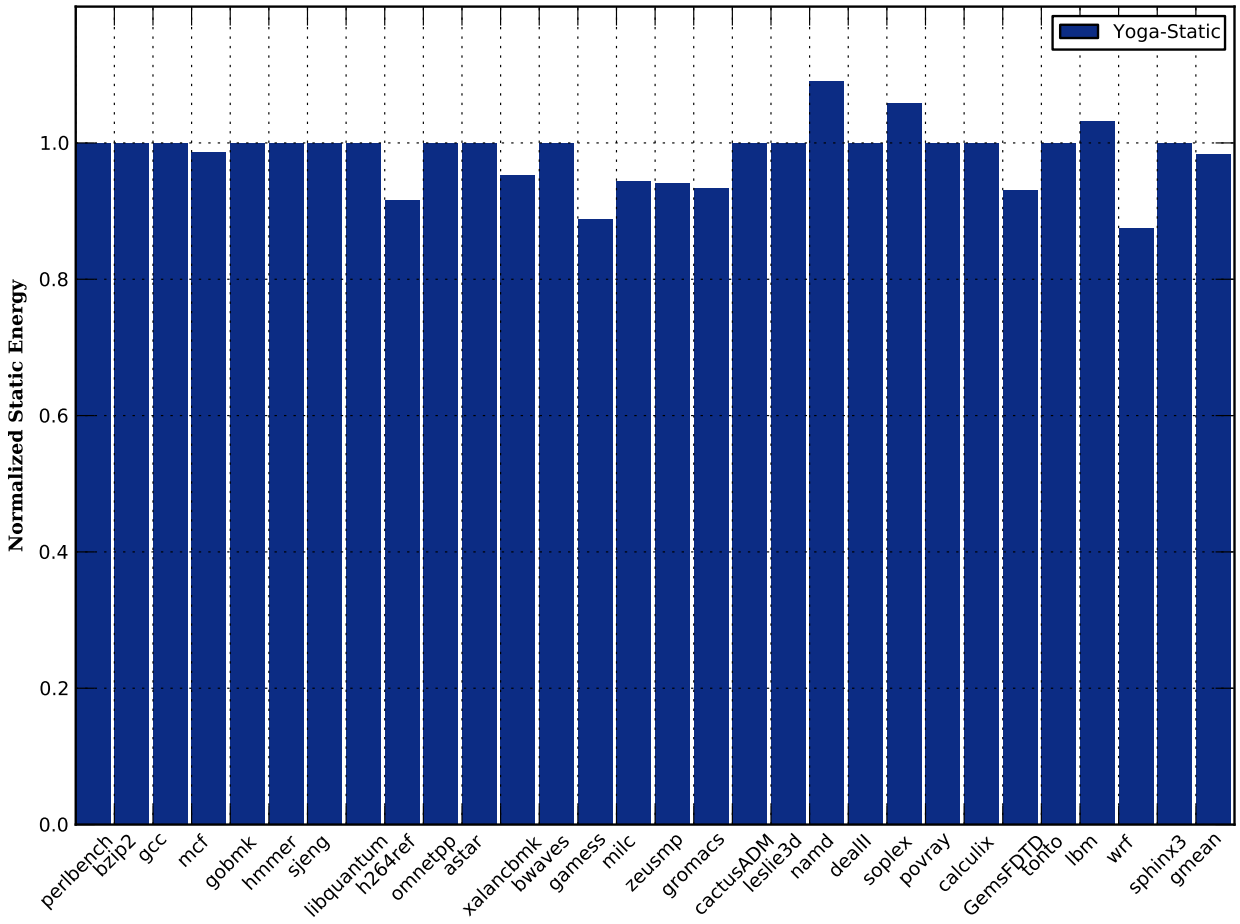


Figure 8. Static energy consumption of Yoga vs an 4-wide OoO processor

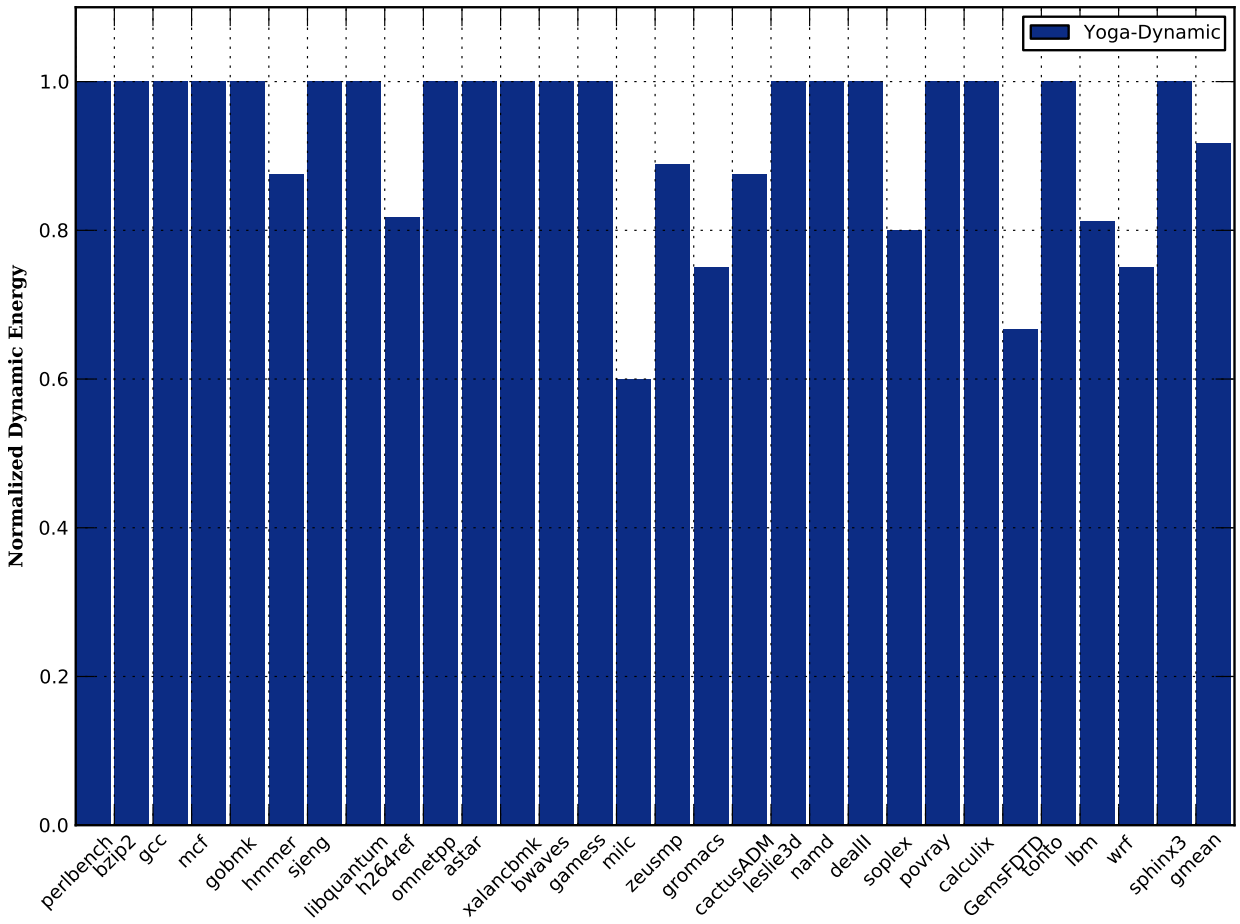


Figure 9. Dynamic energy consumption of Yoga vs an 4-wide OoO processor

The average static energy consumption in Figure 8 is relatively similar to the normalized OoO-4. Benchmarks with small coverage (*perlbench*, *gcc*, *gamess* and *astar*) do not show extra energy consumption because the VLIW cache remains turned off when there are no VLIW frames stored and the power consumption of the Frame Confidence Table (FCT) is relatively small. *Soplex* and *lbm* suffer an increase of static consumption because they are the benchmarks that perform worst in VLIW mode, increasing their execution time. *Milc* achieves a particularly beneficial static energy reduction because it executes 99% of the time in VLIW mode. During VLIW mode, the decoder and any unused functional units are power gated. The I-cache, ROB, RS and the PRF are clock gated.

Figure 9 shows the dynamic energy consumption. Yoga dynamic energy consumption is 10% less than that of OoO configuration. The reason for this is that since performance is similar in both configurations, when a benchmark runs

Stat Name	perlbench	bzip2	gcc	mcf	gobmk	hmmer	sjeng	libquantum	h264	omnetpp	astar	xalancbmk	bwaves	gamess	milc
OPS_FRAME	55.51	47.05	21.93	19.94	22.55	30.33	54.92	0.00	19.59	49.67	17.61	30.98	64.65	58.48	70.07
TRANS_CYC	15.18	4.37	2.02	0.43	1.74	2.58	5.41	0.00	122.64	1.67	4.77	0.10	1.02	14.89	0.00
WORD_ILP	2.49	2.58	1.77	1.42	1.70	1.81	2.45	0.00	1.46	2.44	1.47	1.47	2.98	2.35	2.48
UNTRAINED	2.00	3785.00	488.00	0.00	0.00	951.00	11.00	0.00	42.00	23.00	163.00	58.00	3.00	492.00	2.00
ROB_OK	95.06	2.61	0.69	22.06	1.13	96.90	3.37	0.01	0.13	0.15	95.36	0.15	0.14	0.35	0.94
SWITCHES	14285	5694	1039	301	9	29380	505	0	3496	48	12978	881	1	5901	1
OPS_FU_0	40.22	38.83	56.51	70.30	58.80	55.24	40.83	0.00	68.39	41.02	67.80	68.19	33.57	42.47	40.27
OPS_FU_1	21.47	25.51	22.99	29.69	26.87	26.63	25.23	0.00	26.61	21.97	24.40	30.43	22.82	24.37	23.96
OPS_FU_2	20.16	17.29	13.71	0.00	13.31	15.26	20.77	0.00	2.58	18.70	6.91	0.77	21.81	19.22	20.18
OPS_FU_3	18.15	9.75	6.23	0.00	1.02	2.85	13.15	0.00	1.26	18.26	0.89	0.61	10.90	13.89	14.17
OPS_FU_4	0.00	4.88	0.55	0.00	0.00	0.02	0.02	0.00	1.15	0.05	0.00	0.00	10.90	0.05	1.43
OPS_FU_5	0.00	3.73	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00

Stat Name	zeusmp	gromacs	cactusADM	leslie3d	namd	dealII	soplex	povray	calculix	GemsFDTD	tonto	lbm	wrf	sphinx3	amean
OPS_FRAME	65.47	97.08	79.00	53.88	45.99	43.54	14.99	74.75	26.02	92.74	75.72	59.44	87.18	43.77	49.1
TRANS_CYC	8.19	259.59	46.40	11.87	0.11	1.86	0.01	12.97	1.33	43.69	88.43	0.00	0.71	1.98	22.6
WORD_ILP	2.03	2.08	3.37	2.59	1.44	1.99	1.32	1.99	1.67	2.13	2.28	2.51	2.23	2.04	2.0
UNTRAINED	2515.00	290.00	160.00	4.00	1504.00	1034.00	61.00	180.00	18.00	203.00	379.00	5.00	11907.00	360.00	849.7
ROB_OK	0.05	1.77	0.08	0.03	0.04	3.31	2.24	0.10	26.48	0.66	1.35	0.19	1.01	17.59	12.9
SWITCHES	2747	10691	774	7	2507	3289	197	445	62	2159	35419	18	7828	416	4864.8
OPS_FU_0	49.24	48.09	29.72	38.62	69.29	50.20	75.53	50.23	59.89	47.02	43.90	39.88	44.80	48.91	48.9
OPS_FU_1	27.61	28.56	24.21	23.27	30.57	25.56	18.37	28.35	19.85	28.15	25.54	25.20	26.32	27.35	24.5
OPS_FU_2	16.59	13.67	13.90	20.49	0.09	18.54	6.10	15.34	12.42	17.42	19.46	18.08	18.29	17.78	13.8
OPS_FU_3	6.51	7.65	11.24	17.63	0.03	5.19	0.00	5.77	7.84	7.38	10.23	13.86	10.58	5.96	7.6
OPS_FU_4	0.05	2.02	10.51	0.00	0.02	0.51	0.00	0.31	0.00	0.03	0.70	2.97	0.00	0.00	1.2
OPS_FU_5	0.00	0.01	10.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.17	0.00	0.00	0.00	0.5

Table 3. Extended Yoga statistics of VLIW mode performance. The average is across all 29 benchmarks.

in VLIW mode, several structures (RS, ROB, Rename logic) are clock gated and therefore do not consume dynamic power. Benchmarks such as *milc*, *soplex*, *GemsFDTD*, *wrf* and *lbm* obtain a significant dynamic energy reduction because they spend more than 50% (Figure 6) of their execution in VLIW mode.

In summary, a 10% of total energy reduction is achieved for SPEC CPU 2006 and a maximum of 18% energy reduction is obtained for milc which runs 99% of the time in VLIW mode.

5.4. VLIW performance

Table 3 shows the main statistics of the performance modeling of Yoga to better understand the performance/energy results. *OPS_FRAME* is the average number of instructions in a VLIW frame. *WORD_ILP* shows the average ILP of words in each benchmark. These two statistics show that for benchmarks with large frames, the ILP tends to be higher. *UNTRAINED* shows the number of frames that execute in VLIW mode for the first time without performance monitoring from the controller. *ROB_OK* shows the percentage of potential transitions from OoO to VLIW mode when the ROB occupancy is below 50%. Note that *libquantum* never executes in VLIW mode. The ROB occupancy is less than 50% for only 0.01% of the potential transitions. However, even for those few transitions, frames never reach the *MinFrameSize* requirement. *Milc* shows a very low *ROB_OK* even though it executes almost 100% of the time in VLIW mode because before switching to VLIW mode once, several transitions did not pass the ROB occupancy requirement. *TRANS_CYC* shows the average cycles it takes to transition from OoO to VLIW. *H264*, *gromacs* and

tonto are the benchmarks suffering longer transitions. This is because they suffer more long latency misses during a transition. This happens even using ROB occupancy to discard frames in the controller. The transition cycles are also taken into consideration when computing the VLIW IPC. Thus, when a frame transition becomes a limiter to performance the controller later discards it when comparing the frame IPCs. *SWITCHES* shows the number of transitions to VLIW. It also represents how many VLIW frames Yoga executes. As we can observe, benchmark *milc* contains only one mode switch to VLIW and remains in VLIW mode for the remainder of execution. Therefore, its coverage is almost 100%.

Finally, Table 3 shows the *OPS_FU_X* which represent the percentage of operations executed in VLIW by each functional unit. As we can observe, the additional 2 functional units are rarely used except for high IPC benchmarks such as *cactusADM*. Yoga can power these extra FUs while remaining within the power budget because the total power is reduced in VLIW mode. Also, Yoga power gates all unused FUs in advance (we know the frame ILP before starting its execution), making the back end more energy-efficient.

6. Discussion

Frame construction. The threshold to consider a branch easy to predict is small (2 bit) compared to previous works that use 32 [19]. However, since the controller requires a minimal number of times to execute the frame in OoO before switching to VLIW, the times a branch is easy-to-predict becomes effectively higher.

Yoga performance. The performance of Yoga VLIW mode depends significantly on two conditions. The transition from OoO to VLIW and the performance of VLIW frames. The transition stops the processor from fetching instructions from the I-cache, and switches to fetching from the VLIW cache. Once the VLIW execution unit receives the first VLIW word to execute, it cannot start its execution if any of the ops in the word contains a live-in dependency with ops being executed in the OoO mode. Therefore, the VLIW mode stalls its pipeline until the live-in is produced. This can produce significant stalls if the live-in is produced by a long latency operation. To mitigate this bottleneck, the Yoga controller checks the ROB occupancy. The performance of VLIW frames depends on how good the OoO schedule is when used in VLIW during future executions of the frame. The VLIW IPC is a good metric for this. A bad performance frame is usually caused by transition cycles and cache misses. In general, unexpected cache misses in VLIW stall VLIW words and increase execution time. Experimental evaluation shows that using 64 as *MinFrameSize* allows the VLIW Fill Unit to build VLIW schedules introducing more ILP and therefore better mitigate the unexpected cache misses. However, in OoO mode, the processor may not always needs the large ROB and RS. Some phases of benchmarks only use a limited number of entries in the RS, thus, the ILP of those schedules is limited. We foresee potential for smaller frames by more effectively detecting the behavior of the OoO scheduling as it is scheduling

instructions in OoO mode. This would increase the VLIW coverage and potential VLIW coverage.

VLIW roll back. Once Yoga switches to VLIW execution there are several reasons why Yoga might have to roll back to OoO mode in the middle of the frame execution. One is a branch misprediction. In this case, the selected frame does not contain the correct path and therefore all ops will be re-executed in OoO mode. A possible solution would be to add predication and allow the VLIW Fill Unit to build predicated frames containing both paths for conditional branches. However, this solution makes the VLIW Fill Unit more complex and reduces the potential of ILP by executing both paths of a branch. Also, in order to dynamically build predicated frames, both execution paths should be in the Frame Confidence Table to merge both instruction schedules. This potentially reduces the coverage of the FCT.

Memory disambiguation misprediction is another reason to roll back to OoO. Rolling back to OoO solves the problem because the stores in VLIW mode do not leave the LSQ until the frame completes its execution. However, memory disambiguation misprediction rarely happens in VLIW mode because the regularity of a frame execution captures frames where this tends to not happen.

VLIW frame ExecCnt threshold. Besides all the branches in a frame being easy to predict, Yoga requires the frame to be executed a number of iterations before it becomes available in VLIW format. However, some small frames tend to change their behavior quickly and produce performance degradation because of cache and branch misses. For this reason, Yoga doubles the *ExecCnt* threshold once a frame underperforms in VLIW, reducing VLIW coverage for benchmarks with poor VLIW performance.

7. Related Work

We first present related work on power efficient architectures and follow with work on dynamic optimizations.

7.1. Power Efficient Architectures

Power efficiency has become a major issue in processor design. One common approach is reconfigurable processors which use different execution modes to deliver high performance and energy savings.

Kim et al. [9] and Ipek et al. [7] introduce techniques to compose several cores into a larger one. However, building the core datapath with fused cores makes it less energy efficient because extra latency is added to the pipeline stages in order for the cores to communicate. Khubaib et al. [8] introduce MorphCore, a hybrid processor that can run a single OoO thread or several in-order threads. These microarchitectural proposals adapt the hardware to the workload's number of independent threads, while Yoga improves energy efficiency of a single thread.

Lukefahr et al. [11] introduce composite cores. In composite cores, two different back ends are integrated in the same processor and a controller switches between OoO or in-order back ends depending on the performance/power ratio. This proposal shares with Yoga the idea of having different modes of execution for single threaded execution.

However, it does not re-use microarchitectural execution information from one mode to run in another mode. Yoga uses the OoO instruction schedule and packs the instructions in VLIW format reusing the OoO schedule in a more energy efficient way.

Palomar et al. [17] propose the ReLaSch processor to reuse schedules from an OoO processor and improve performance by caching the instruction schedule and removing the OoO scheduling logic from the critical path. We take a similar approach by reusing the previous OoO instruction schedule. However, Yoga continuously checks the performance of the OoO and VLIW modes; this allows Yoga to choose the mode that is best suited for the particular frame. Also, the targeted performance-power ratio of Yoga can be chosen in the controller. In addition, Yoga improves energy efficiency by turning off OoO structures in VLIW mode, while ReLaSch is not able to turn off any structures.

Nair et al. [15] introduce Dynamic Instruction Formatting. DIF is a parallel engine in addition to an in-order processor. The engine executes portions of the program by capturing the dynamic instruction trace from the main processor and dynamically schedules, renames and builds VLIW-like instructions. DIF is designed with the goal of accelerating portions of the application. It only supports common ops, and it requires a complete additional back end. In contrast, Yoga benefits from the reuse of OoO instruction schedules, does the mapping/renaming of a frame only once, and does not require the extra logic of an additional back end. Yoga supports all types of operations and is designed with the goal of saving energy without sacrificing performance.

Petrica et al. [21] introduce Flicker, a dynamically adaptive architecture that uses an optimization algorithm to find the lanes that can be powered off without losing performance in an OoO processor. Unlike Yoga, Flicker always operates in OoO mode.

7.2. Dynamic Optimizations

A large number of dynamic optimization frameworks have been proposed. Here, we compare those similar to Yoga.

Transmeta processors [3] introduce the code morphing system, a binary translation mechanism in firmware that fetches x86 code and dynamically converts it to the chip's ISA in VLIW format. The processor back end works only in VLIW. Since Transmeta has a single mode of execution, it can not benefit from OoO scheduling to tolerate long latency operations like Yoga does. Transmeta executes in VLIW with runtime system software performing the scheduling logic.

RePLay [18] is an optimization framework built on top of previous trace cache systems. Using branch promotion and dynamic optimizations, RePLay builds a frame or contiguous set of instructions and optimizes at the micro-op level. Unlike Yoga, RePLay does not utilize an adaptive back end.

Structuring code in larger pieces has already been used in previous work such as traces [4], frames [18] and su-

perblocks [6]. However, none of the previous proposals use runtime execution information for further optimizations.

The idea of enlarged basic blocks has been proposed by several researchers. Melvin et al. [13] proposed the concept of the block structured ISA allowing larger units of work by replacing branches with asserts and therefore increasing the issue bandwidth of the machine. Burger et al. [2] propose the TRIPS architecture which also relies on enlarged basic blocks and a simple power efficient core architecture. Both of these proposals rely on the compiler to extract more ILP and their performance benefit is therefore limited. On the other hand, Yoga builds the blocks at runtime and therefore adapts dynamically to any application input set.

Gupta et al. [5] introduced BERET. They identify program hot-spots in a general purpose processor and run them in a special purpose co-processor. By off-loading execution to this co-processor, they reduce energy consumption by eliminating the need for front end, decode and branch prediction logic. However, BERET requires the compiler to break down programs into a bundle of micro-code instructions, while we propose a runtime solution that only switches to the more efficient VLIW mode when an energy benefit is predicted.

Finally, Brandon and Wong [1] propose a single generic binary format for VLIW processors that includes different widths of the VLIW words in the executable image. Their solution requires multiple instances of the same code to support a variable number of lanes. Yoga also adapts the width of the VLIW words to the ILP found in OoO but improves upon it by using enlarged basic blocks. Also, Yoga uses a single version of the executable image, thereby reducing the instruction footprint as compared to [1].

8. Conclusions

We propose Yoga, a hybrid dynamic VLIW/OoO processor that saves energy when applications have regular behavior (branch predictability and cache misses) by running in VLIW mode and maintaining the performance of a regular OoO processor. Our proposal can benefit from the OoO logic to detect when regular code executes. Yoga captures the behavior of portions of the code with easy to predict branches and stores the OoO instruction schedule. When the code becomes a hotspot, it switches the back end execution mode to VLIW, saving power from the OoO structures not used and by turning off the functional units not required. Yoga executes an average of 20% of dynamic instructions for SPEC CPU 2006 in VLIW mode with a 10% average energy reduction and a maximum energy reduction of 18% with negligible average performance loss.

References

[1] A. Brandon and S. Wong. Support for dynamic issue width in VLIW processors using generic binaries. In *DATE*, pages 827–832, 2013.

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

[3] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing® software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO*, pages 15–24,

2003.

[4] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: dynamic optimizations for trace cache microprocessors. In *MICRO*, pages 173–181, 1998.

[5] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO*, pages 12–23, 2011.

[6] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, May 1993.

[7] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ISCA*, pages 186–197, 2007.

[8] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *MICRO*, pages 305–316, 2012.

[9] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *MICRO*, pages 381–394, 2007.

[10] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, pages 469–480, 2009.

[11] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. In *MICRO*, pages 317–328, 2012.

[12] D. S. McFarlin, C. Tucker, and C. Zilles. Discerning the dominant out-of-order performance advantage: is it speculation or dynamism? In *ASPLOS*, pages 241–252, 2013.

[13] S. Melvin and Y. Patt. Enhancing instruction scheduling with a block-structured ISA. *Int. J. Parallel Program.*, 23(3):221–243, June 1995.

[14] Micron Technology, Inc. *MT41J512M4 DDR3 SDRAM Datasheet Rev. K*, Apr. 2010. `http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf`.

[15] R. Nair and M. E. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *ISCA*, pages 13–25, 1997.

[16] Nvidia. NVIDIA Tegra 4 family CPU architecture. In *Nvidia White Paper*. Nvidia, 2012.

[17] O. Palomar, T. Juan, and J. J. Navarro. Reusing cached schedules in an out-of-order processor with in-order issue logic. In *ICCD*, pages 246–253, 2009.

[18] S. J. Patel and S. S. Lumetta. Replay: A hardware framework for dynamic optimization. *IEEE Transaction on Computers*, 2001.

[19] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *MICRO*, pages 303–313, 2000.

[20] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO*, pages 81–92, 2004.

[21] P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and C. A. Shoemaker. Flicker: a dynamically adaptive architecture for power limited multicore systems. In *ISCA*, pages 13–23, 2013.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA*, pages 128–138, 2000.

[23] A. Seznec. A new case for the TAGE branch predictor. In *MICRO*, pages 117–127, 2011.

[24] J. Tendler, J. S. Dodson, J. S. Fields Jr., L. Hung, and B. Sinharoy. POWER4 system microarchitecture. *IBM J. of Research and Develop.*, 46:5–25, Oct. 2001.