Approaching Overhead-Free Execution on FPGA Soft-Processors

Charles Eric LaForest Jason Anderson J. Gregory Steffan
Department of Electrical and Computer Engineering
University of Toronto, Canada
{laforest, janders}@eecg.toronto.edu

***Abstract*—Implementing systems on FPGA soft-processors, rather than as custom hardware, eases and accelerates the development process, but at the cost of a great reduction in performance. Orthogonal to limitations in parallelism or clock frequency, this reduction in performance primarily originates in the intrinsic addressing and flow-control overheads of scalar microprocessors, which expend a considerable number of cycles interleaving address calculations and branch decisions within the actual useful work. We present an improved FPGA soft-processor architecture which statically overlaps "overhead" computations and executes them in parallel with the "useful" computations, significantly reducing the number of processor cycles needed to execute sequential programs, while reducing maximum clock frequency to 0.939x of its original value. In addition to eliminating almost all overhead computations, the proposed soft-processor can operate at 500 MHz on the Altera Stratix IV FPGA – 0.909x of the absolute maximum rating. Combined, the high speed and execution efficiency increase the range of FPGA designs amenable to soft-processors rather than custom hardware. We evaluate our cycle count improvements with multiple benchmarks, achieving speedups ranging from 1.07x for control-heavy code, to 1.92x for looping code, never performing worse than the original sequential code, and always performing better than a totally unrolled loop.**

I. Introduction

Implementing computations in hardware on FPGAs can offer a significant speedup relative to computations in software running on a standard processor. Speedups of an order of magnitude have been reported (e.g.: [1]), despite the FPGA hardware running at a considerably lower clock speed (F_{max}) than the processor. Much of this speedup arises from exploiting spatial parallelism in the FPGA hardware, but a significant portion comes from removing addressing and flow-control overhead ("support instructions" [2]–[6]). Hardware implementations remove such overhead by computing it in parallel with the actual work using application-specific Finite-State Machines (FSM) to accept/reject input, count loops, compute memory addresses, and perform the equivalent of flow-control by driving multiplexers. However, implementing a given algorithm in hardware remains a difficult and laborious process, and hardware skills are comparatively rare vs. software skills. Hardware design traditionally involves the use of hardware description languages (HDLs), which require specification at the bit level and explicit coordination of computation and communication, making design and debug challenging. Even High-Level Synthesis (HLS) systems such as BlueSpec, LegUp, or compiling from OpenCL to hardware kernels, ultimately generate an HDL circuit description, which must then be synthesized, placed, and routed, taking hours or days for the largest designs.

As an alternative process, a designer can implement a "soft-processor" on the FPGA, benefiting from the flexibility of the FPGA, while retaining the programmability of software. Unfortunately, soft-processors pay a large performance penalty relative to what the underlying FPGA hardware can achieve, and also relative to conventional "hard" CPUs. Furthermore, soft-processors carry the same addressing and flow-control overheads as any other processor. In most cases, unrolling loops and vectorizing code can eliminate addressing and flow-control overheads, but unrolling bloats code and increases the required instruction memory bandwidth, while vectorizing [7], [8] can prove challenging to use effectively and cannot improve control code. Pipelining and multi-threading [9], [10] will increase raw speed, but still do not eliminate overhead.

While we can extend FPGA soft-processors with application-specific custom instructions and/or accelerators to avoid overhead, this again involves more difficult hardware design. This paper proposes an alternative: a soft-processor architecture which can eliminate control-flow and addressing overheads, increasing the range of applications amenable to soft-processors, and retaining the ease-of-use of software. Our key contribution extends the Octavo soft-processor [10] to remove addressing and flow-control overheads:

1) We locate the addressing and flow-control overheads in sub-graphs interleaved within a sequential program, which we can extract and execute in parallel.
2) We introduce the Branch Trigger Module (BTM) as a means of folding *multiple* branches in parallel with an ALU instruction, based off arbitrary conditions.
3) We introduce the Address Offset Module (AOM), which enables per-thread private data for shared code, implements indirect addressing, and optionally automatically post-increments indirect addresses after use.
4) We show, using several micro-benchmarks, that the BTM and AOM *always reduce cycle count*, always more so than loop unrolling, and without significantly affecting cycle time.
5) We also show an increase in the ratio of useful work done, *often approaching the maximum provided by total loop unrolling*, without having to unroll loops.

II. Related Work

Previous branch folding approaches dynamically combined together an instruction and a following branch in the processor's instruction cache [11], [12] and speculatively executed conditional branches, flushing the pipeline on a misprediction. In

contrast, Octavo's multi-threaded design never stalls or flushes its pipeline, and avoids the need to speculate on branches. Also, the BTM allows us to hoist the definition of a branch outside of loops – a critical feature for Octavo's un-cached, tightly-coupled instruction memory, without spare time for pre-fetching both branch paths [13]–[15]. Davidson and Whalley [16] describe a sophisticated system of branch registers similar to the BTM. However, their approach does not support folding, multi-way, or cancelling branches as the BTM does.

For vector processors, decoupling the scalar and vector components can provide the *appearance* of zero-overhead loops [17], but only if the scalar processor has time to execute loop control operations, while the vector processor operates. Truly eliminating loop overhead still requires loop counting and operand addressing hardware [18]–[20].

Digital Signal Processors (DSPs) support zero-overhead loops with a variety of mechanisms: Analog Devices' TigerSharc [21] has a pair of loop counters and matching branch conditions, while their Blackfin [22] extends this approach with two sets of Loop Top and Loop Bottom registers, but only for simple counted loops. Finally, Texas Instruments' C64x+ [23] uses a loop buffer and some counter registers to execute compact software-pipelined loops. In contrast, the BTM provides a simpler and more general mechanism (though we have not yet implemented loop counters), and its support for cancelling branches enables useful branch folding in any flow-control code, not just loop branches.

Prior work on multi-way branches focused on improving ILP in VLIW processors [24], [25]. Our work does not require the same complex Control-Data Flow Graph (CDFG) analysis to merge branches, nor the generation of duplicate code. Our multi-way branches reduce the number of consecutive tests on the same data by using parallel "fast compares" [26], [27].

III. Motivation and Overview

Before delving into the details our proposed processor, we briefly review the Octavo soft-processor [10] as it forms the basis of our work. The intent of Octavo was to create a processor that could operate at the maximum allowable speed of the underlying FPGA fabric which, in Stratix IV devices, is limited to 550MHz by the M9K Block RAMs. To reach this limit, Octavo has 10 pipeline stages and 36-bit instructions, comprising a 4-bit opcode and three 10-bit addresses: two source, one destination, with 2 (of 36) bits unused. The original Octavo has no register file; rather, it reads/writes 36-bit integer operands directly from Block RAM memory. It supports only *direct* addressing, which with 10-bit operands implies 1024 words of addressable memory space per operand. A key implication of the lack of indirect addressing is that, to implement pointers, Octavo requires self-modifying code. For example, to implement an Octavo program that walks through the elements of an array, the program itself must overwrite the locations of operands specified in the instructions. Finally, Octavo *multi-threads* its pipeline with 8 independent threads, with thread instructions issued in fixed round-robin order to avoid any RAW hazards or branch penalties within each thread. In effect, each thread instruction completely finishes before the next one begins.

While our work improves the Octavo soft-processor, it also alters its programming model sufficiently that we can no longer compare code on both versions. One of our improvements effectively implements memory indirection, eliminating the need for self-modifying code, making comparison difficult and clouding our results. Thus, we must compare our work against an idealized model which supersedes the original Octavo.

A. Baseline: A "Perfect" MIPS-like CPU

We can use the multi-threaded nature of Octavo to create a cycle-accurate emulation of an ideal "perfect" MIPS-like processor on our improved Octavo processor and then implement a micro-benchmark on both this emulated ideal model and natively on the improved Octavo. We do not need to compare against the original Octavo since the ideal model will always perform better than any actual scalar processor.

We can thus focus on the intrinsic overheads found in a general-purpose scalar processor, separate from implementation issues such as pipelining, and architectural issues such as hazards. We can show that, despite the absence of stalls and hazards in the ideal case, significant control-flow and addressing overheads remain relative to the actual desired computation. We then show how to extract these overheads as separate *parallel* sub-programs, and overlap their execution with that of the actual work-producing code on our improved Octavo processor.

Our baseline ideal MIPS-like CPU has these properties:

- No memory access latency.
- Single-cycle instruction and data memory access.
- No load or branch delay slots.
- Branch conditions known at the start of pipeline.
- Result forwarding across pipeline stages.
- No structural hazards across instructions.
- Single-cycle instruction execution.

While we cannot build such an ideal CPU with high performance, each individual Octavo thread closely approaches this ideal, allowing us to execute a cycle-accurate emulation of the ideal model within a thread.

B. Benchmark: Hailstone Numbers

To overview our improvements, we calculate hailstone numbers as a simple, manually tractable example which nonetheless exhibits flow-control and addressing overheads. The hailstone benchmark iteratively computes a series from a positive seed number (if n is even: $n = n/2$, else $n = 3n + 1$), presenting many basic forms of computation (addition, shifting, multiplication, bit-masking, branching, looping). We apply this calculation to an array of 100 positive integers, terminated by -1, to introduce addressing overhead and average the time spent in the even/odd cases. We also output each new value as we compute it. Listing 1 shows the pseudo-code.

The pseudo-code of Listing 1 directly translates into MIPS-like assembly in Listing 2, with the instruction format convention of `OP dest, src1, src2`. This code, running on the emulated ideal MIPS-like processor, averages 970 cycles per pass over 100 seed values with an execution efficiency of 0.655. Unrolling the same code results in an average of 769 cycles per pass and an efficiency of 0.824, a 1.26x improvement in both cycle count and efficiency.

Listing 1. Hailstone Pseudo-Code

```
1  outer: seed_ptr = ptr_init
2  inner: temp = MEM[seed_ptr]
3         if (temp < 0):
4             goto outer
5         temp2 = temp & 1
6         if (temp2 == 1):
7             temp = temp * 3
8             temp = temp + 1
9         else:
10            temp = temp / 2
11        MEM[seed_ptr] = temp
12        seed_ptr += 1
13        OUTPUT = temp
14        goto inner
```

Listing 2. Hailstone MIPS-like Assembly Code

```
1  outer:  ADD  seed_ptr, ptr_init, 0
2  inner:  LW   temp,  seed_ptr
3          BLTZ outer, temp
4          AND  temp2, temp,  1
5          BEQZ even,  temp2
6          MUL  temp,  temp,  3
7          ADD  temp,  temp,  1
8          JMP  output
9  even:   SRA  temp,  temp,  1
10 output: SW   temp,  seed_ptr
11         ADD  seed_ptr, seed_ptr, 1
12         SW   temp,  OUTPUT
13         JMP  inner
```

Listing 3. Hailstone Octavo Assembly Code

```
1  outer:  ADD seed_ptr, ptr_init, 0
2  inner:  LW  temp, seed_ptr
3          MUL temp, temp, 3 ; BEVNn even
4          (continued)       ; BLTZn outer
5          ADD temp, temp, 1 ; JMP output
6  even:   SRA temp, temp, 1
7  output: SW  temp, seed_ptr
8          SW  temp, OUTPUT  ; JMP inner
```

We define *execution efficiency*, the ratio of "useful" to "not useful" instructions executed, as follows: Useful instructions are those which use the ALU and remain after total loop unrolling. Thus, loads, stores, and ALU operations count as useful. Pointer initialization and incrementing also count as useful if loop unrolling cannot eliminate them. An unrolled loop will always approach an efficiency of 1.00, minus obligatory non-loop branches. Branches never count as useful by themselves since the ALU does no work during their execution.

We can then use our Octavo improvements to optimize the Listing 2 code into Listing 3, which shows the resulting optimized Octavo assembly code with the folded branches placed next to their concurrent ALU instruction. First, we add support for post-incrementing `seed_ptr` (eliminating line 11 of the MIPS code), add a "fast compare" [26], [27] to the result of the previous instruction to create a Branch on Even (`BEVN`), eliminating the Boolean masking on MIPS line 4. We also fold together both branches to `outer:` and `even:` (MIPS lines 3 and 5) into the start of the odd-number case (MIPS line 6) while also setting a "Predict Not Taken" bit (denoted by a suffix `n` on the branch instruction) to implement a multi-way cancelling branch which cancels the `MUL` instruction if we do not fall through into the odd-number case. Finally, we fold the unconditional `JMP`s into their preceding instructions (eliminating MIPS lines 8 and 13).

We later show in the results section that this optimized code, running on the improved Octavo processor, averages 504 cycles per pass over 100 seed values with an execution efficiency of 0.863, giving a 1.92x cycle-count speedup and 1.32x execution efficiency increase over the ideal MIPS-like CPU, exceeding the benefits of loop unrolling, while only reducing Octavo's clock frequency to 0.939x of its original value.

IV. Extracting Control and Data Flow Sub-Graphs

We can think of a single sequential program as containing three separate "sub-programs", each responsible for different tasks:

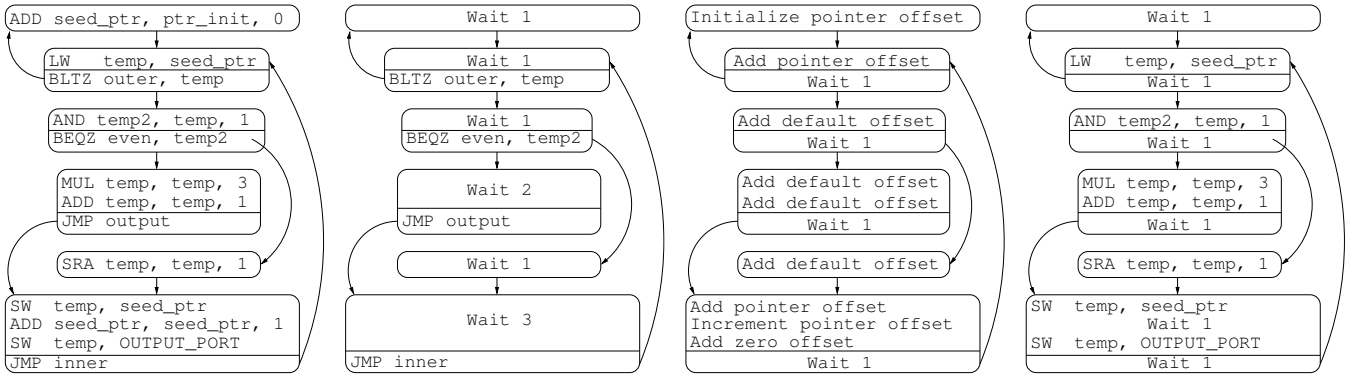
1) The actual computational work.
2) The flow-control to realize repetition and decision.
3) Computing memory addresses.

We cannot practically eliminate the flow-control and addressing sub-programs, else we would have to scale the actual work program along with its data by explicitly encoding each memory location into each instruction and duplicating the code for every repeated or conditional computation, at enormous cost in performance and code size [28]. Interleaving these three sub-programs imposes a sequential ordering to their operations. The core concept underlying our architectural enhancements to Octavo recognizes that portions of these sub-programs are independent from one another and can therefore be executed in *parallel* with each other.

Fig. 1 illustrates how we can express all three sub-programs as sub-graphs of the original Control-Data Flow Graph (CDFG) of the Hailstone benchmark. Figure 1(a) shows the original MIPS-like code broken into basic blocks. If we keep only the branch and jump instructions and replace the other instructions with a `Wait` instruction having an argument to represent the length of the body of the basic block, we end up with Figure 1(b), which describes the flow-control sub-program. Similarly, keeping only instructions which relate to addressing gives us the addressing sub-program in Figure 1(c), where we either add a fixed offset to any regular memory address (one per thread), a zero offset to any shared absolutely-addressed memory (e.g.: memory-mapped hardware), or a pointer offset to any memory location used as a pointer. We also keep instructions which alter the state of the addressing sub-program by initializing or incrementing offsets. Finally, removing any flow-control or state-altering addressing instructions from the initial CDFG leaves us with the actual useful work program in Figure 1(d). We can now visually find opportunities for executing flow-control and addressing in parallel with useful work, manifesting as instructions horizontally overlapping with `Wait` statements across sub-graphs, so long as no sequential dependency exists with the previous instruction.

Several approaches exist to execute parallel sub-programs: superscalar processing, Very-Long Instruction Word (VLIW) computers, sub-units executing horizontal microcode, or multiple processors executing separate threads. However, these approaches require complex instruction scheduling hardware, larger instruction words and memory bandwidth, or require synchronization of multiple threads of execution.

Instead, we observe that the Program Counter (PC) suffices to represent the current location within *any* of the three sub-graphs, and that the flow-control and addressing sub-programs contain very little information: the flow-control sub-program spends most of its time waiting for a branching point, while the addressing sub-program simply adds a constant offset unless a pointer or I/O access happens. Viewed another way: even nested loops have only a few repeatedly reached branches at any one time, and only the most memory-bound code performs loads/stores more often than internal computations.



(a) Control-Data Flow Graph (b) Flow-Control Sub-Graph (c) Addressing Sub-Graph (d) Useful Work Sub-Graph

Fig. 1. **We can extract multiple concurrent and coordinated sub-graphs from the Control-Data Flow Graph (CDFG) of the Hailstone benchmark. Opportunities for folding addressing and flow-control with useful work manifest as instructions horizontally overlapping "`Wait`" statements in other sub-graphs, so long as no sequential dependency exists with the previous instruction.**

Furthermore, from the PC value, it is apparent at any point in the overall program which branching and addressing operation comes next. It suffices to provide this information to some machinery ahead of time (and ideally outside of busy loops), and let the PC and instruction operands indicate when special flow-control or addressing operations must happen.

Motivated by the discussion above, in the Octavo processor, we reduce the execution of the flow-control and addressing sub-programs to pattern-matching on the set of conditions needed to generate the branches and address offsets required at the current point in the actual work program. We achieve this by using a small memory to encode the patterns to look for, along with simple selection and comparison matching logic.

Pattern-matching allows the designer to vary the number of entries to easily trade off area and speed against the need to reload pattern-matching entries as the program executes. At the limit case of a single entry, the mechanism returns to the original case of sequentially interleaved sub-programs: each cycle saved by performing a flow-control or addressing operation in parallel with an ALU instruction returns as a cycle spent loading the entry for the next branch or address. Ideally, the designer only needs to include enough entries to fully parallelize the branching and addressing operations inside the most critical sections of the program. Even matching a partial set of these operations will still improve performance some amount.

V. Implementation

Rather than executing the flow-control and addressing sub-programs on their own sub-processors, or as Instruction-Level Parallelism (ILP) in a superscalar, VLIW, or micro-coded processor (any of which would represent a departure from Octavo's scalar, multi-threaded architecture), *we use the FPGA's capacity for fine-grained parallelism to check all sub-program conditions concurrently with the instruction fetch.* We use our knowledge of the current PC, the addresses in the instruction operands, and the result of the previous instruction, to select the desired effect on the flow-control and addressing of the current instruction as it flows through the pipeline.

A. Address Offset Module

The Address Offset Module (AOM) executes the addressing sub-program. We need three instances of the AOM, one for each instruction operand, as any one may access a pointer or other special memory at any time. The AOM adds a selected offset to the address contained in its associated instruction operand. The offset added may be 1) constant, 2) changing dynamically as the program executes (e.g.: automatically incremented), or 3) zero. The ability to add a constant offset to addresses is useful for threads that share program code but read/write from different regions of memory. Dynamically changing offsets are useful for operations such as walking through an array. Zero offsets are useful for addresses that are shared across all threads, for example addresses of memory-mapped hardware in the system.

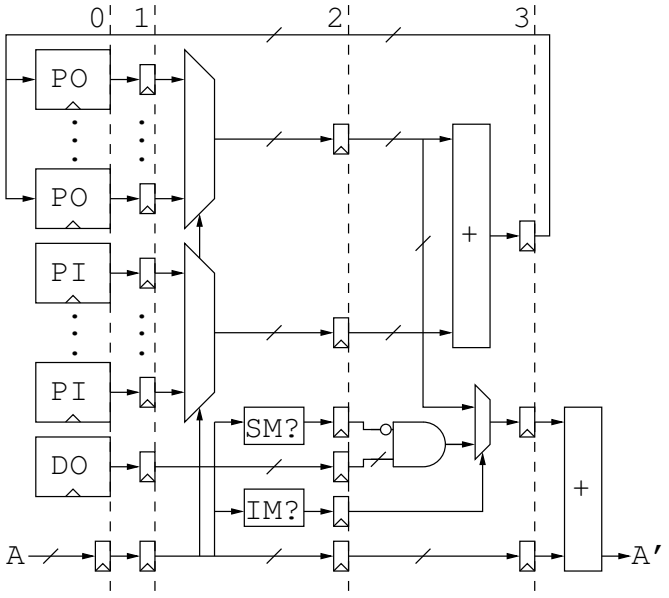


Fig. 2. **Address Offset Module implementation.**

Fig. 2 shows the block diagram of the AOM hardware. The numbers at the top denote the Octavo pipeline stage numbers. The instruction memory read happens at stage 0, and the data memory read happens at stage 4. Thus the AOM must do all its work in stages 0–4. Each thread running on the processor sees its own private AOM instances; specifically, the AOM memories are multiplexed using the thread index (0-7) (not shown).

In stage 0, the AOM reads a number of Programmed Offset (`PO`) and Programmed Increment (`PI`) memories, as well as

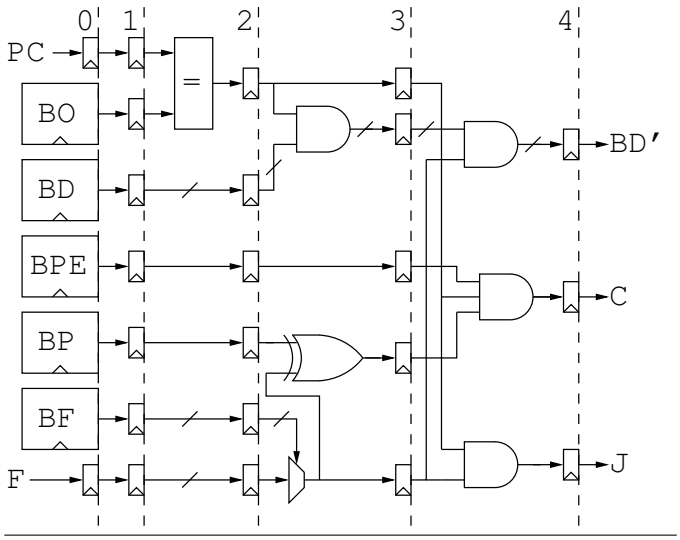


Fig. 3. **Branch Trigger Module implementation.**

a single Default Offset (DO) memory, all implemented using MLAB Block RAMs. Stage 1 serves to pipeline away the latency of the MLABs, and to receive the address (A) from the instruction operand. We cannot use A to address a single, deeper PO and PI memory to select the desired PO and PI values, as A is not available yet, and moving the PO and PI memories forward in the pipeline leaves too few stages to maintain a high F_{max}. Thus we must read multiple memories and select the desired values in the next stage.

In stage 2, we use the least-significant bits of A to select one of the PO and PI values. We also combinationally decode A to determine if it refers to a Shared Memory location (SM?) and/or an Indirect Memory location (IM?). The SM? and IM? logic modules simply decode part of the processor's memory map, set at design time, defining fixed ranges of memory locations which either act as absolutely-addressed shared resources (i.e.: I/O ports) or which act as pointers. Since pointers exist at fixed addresses, simply adding a Programmed Offset (PO) can make them point to any other address. Running different programs does not require resynthesis of the AOM, only agreement on the memory map of pointers and shared resources.

In stage 3, if A refers to a Shared Memory location, the AOM zeroes-out the DO. However, if A refers to an Indirect Memory location, the AOM discards the DO and instead propagates the previously selected PO. In parallel, the AOM increments the PO with the PI and stores it back into its memory. Finally, at the beginning of stage 4, just before the read from data memory, we add the final offset to A, yielding the final address A′. Programmed and Default Offsets have the same width as the instruction operand they modify: 10 or 12 bits. Programmed Increments currently use only 1 bit, for values of $+1$ or 0.

To use the AOM, the program must load DO with the offset for its own thread's memory region, and the PO and PI entries with the offset and post-increment values for the pre-defined memory locations (set at design-time by the IM? decoder) which act as pointers.

B. Branch Trigger Module

The Branch Trigger Module (BTM) executes the flow-control sub-program. We need one BTM instance for each branch we wish to execute in parallel. The BTM monitors the Program Counter (PC) and some flags based on the result of the previous instruction: if the PC matches the location of a branch, and the flags match the branch condition, the BTM generates a destination address and signals the PC controller (not shown) to replace the current thread's next PC value with the destination address, performing a jump. The BTM optionally also outputs a signal to cancel the current instruction if the branch does not go as statically predicted, which allows us to always place a useful instruction in parallel with the branch. Also, if either the PC or the flags do not match, the BTM outputs all zeroes, which allows us to more efficiently take the bit-wise OR of the output of multiple BTMs, rather than using multiplexers. Finally, having multiple BTMs operate in parallel incidentally enables multi-way branches for free.

Fig. 3 shows the block diagram of the main BTM hardware. The numbers at the top denote the Octavo pipeline stage numbers. The instruction memory read happens at stage 0, and any instruction cancelling happens between stages 3 and 4, so the BTM must do its work between stages 0 and 4. However, merging the outputs of all the BTM instances into a single branching decision can happen later in the pipeline. Each thread running on the processor sees its own private BTM instances; specifically, the BTM memories are multiplexed using the thread index (0-7) (not shown).

In stage 0, the BTM reads a number of memories describing a branch operation: the Branch Origin (BO) contains the memory address of the branch, the Branch Destination (BD) contains the branch target address, the Branch Predict Enable (BPE) bit controls whether static branch prediction occurs or if the parallel ALU instruction always executes, the Branch Predict (BP) bit selects between "Predict Taken" and "Predict Not Taken" instruction cancelling behavior, and the Branch Flag (BF) selects one of the flags (F) derived from the result of the previous instruction as the condition for the branch. Stage 1 pipelines away the latency of these MLAB Block RAMs.

In stage 2, the BTM compares the PC of the current instruction with BO and generates a match signal. In stage 3, we use this match signal to mask BD. We also select one of the branch flags (F) and compare it to BP. In stage 4, we use the selected flag to mask BD again, and the BO match signal to mask the selected flag, resulting in the final branch destination BD′ and jump signal J. If either the branch flag or the branch origin do not match, both BD′ and J are zero. Finally, also in stage 4, if we enabled branch prediction for this branch (BPE set), and the selected flag and the branch prediction disagree, and the branch origin matches, we generate a signal (C) to cancel the current instruction in parallel with the branch, converting it to a no-op. Branch Origin and Destination memories have the same width as the Program Counter (10 bits), while the Branch Predict Enable, Branch Predict, and Branch Flag memories have 1, 1, and 3 bits respectively.

To use the BTM, the program must load all the memories with the values describing an upcoming branch. We automatically compute the branch flags (F) from the result of the previous instruction, using separate dedicated logic instead of the main ALU, in the manner of Katevenis' "fast compare" [26], [27]. Currently, we support 8 flags total: Negative, Positive, Zero, Non-Zero, Always (for JMPs), and Even, with 2 flags still left unused. Since all BTM instances function concurrently and

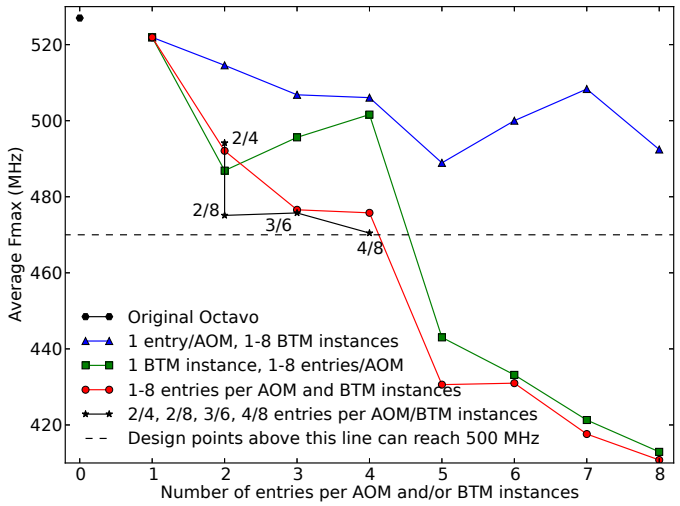


Fig. 4. **The average F_{max} of configurations varying the number of entries per AOM instance and the number of BTM instances. We always require 3 AOM instances, multiplying the actual total number of implemented AOM entries. We ran our benchmarks on point 2/4: 2 entries per AOM and 4 BTM instances.**

eventually merge their decision output, we can easily support multi-way branches so long as all branches with matching origins have mutually exclusive branch conditions.

VI. Experimental Methodology

For each Octavo instance considered in this study, we instantiate it inside a synthesis test harness that registers all inputs and outputs to ensure an accurate timing analysis. We use Altera's Quartus 13.1 to target a Stratix IV `EP4SE230F29C2` FPGA device of the highest available speed grade. We configure the synthesis process to strongly favor speed over area and enable all relevant optimizations. To confirm the intrinsic performance of a design without interference from optimizations which can blur together the netlists of the design and of the test harness, we constrain the whole design under test to its own design partition, excluding the test harness.

Place and Route (P&R) are configured for maximum effort with two constraints: (i) to avoid using I/O pin registers to prevent artificially long paths that would affect the clock frequency, and (ii) to set the target clock frequency to 550MHz, which is the maximum clock frequency specified for M9K BRAMs. Setting a target frequency higher than 550MHz does not improve results. We report the unrestricted maximum operating frequency (F_{max}) by averaging the results of 10 place and route runs, each starting with a different random seed, on the slow process corner at 85°C and 900mV. We compute the number of cycles needed for a program's execution by using ModelSim 10.1d to simulate the processor's HDL implementation.

VII. Evaluation, Benchmarks, and Results

Octavo has, as one of its major features, the capacity to approach the maximum possible clock frequency supported by the underlying FPGA. Thus, we evaluate how many Address Offset Module (AOM) entries and Branch Trigger Modules (BTM) instances we can include, and in what ratios, before the raw clock frequency begins to suffer too much. We consider an F_{max} of 500 MHz as a realistic goal, still at 0.909x of the limiting 550 MHz absolute maximum rating for simple dual port M9K Block RAMs in Stratix IV devices [29], and more representative of the actual rated limit of most such devices.

Our implementation aimed for flexibility and generality for design space exploration, describing each individual memory in the AOM and BTM as separate MLABs, regardless of depth or width. We have not yet analyzed the synthesis results to determine if Quartus can automatically merge separate but logically contiguous MLABs. Therefore, we do not know if the area results will be representative or comparable across design points, and cannot report detailed area results at this time.

Fig. 4 outlines the major features of the design space, charting the F_{max} of Octavo versions with 0 to 8 AOM entries and/or BTM instances, and a few relevant combinations thereof. All values represent the average achievable clock frequency. All points above the horizontal dotted line (470 MHz) denote designs which can reach or exceed 500 MHz after place-and-route (P&R). Meaning that for such designs, across 10 P&R runs, at least one produced an implementation whose F_{max} reached or exceeded 500 MHz.

The single hexagon in the top-left corner represents the original Octavo at 527 MHz (averaged across 10 P&R runs). The triangle markers show the improved Octavo with a minimal set of 3 single-entry AOMs and 1 to 8 BTMs. Since the BTM modules avoid multiplexing and sit in a long pipeline ending at the PC controller, the average F_{max} scales quite well, staying above 490 MHz at up to 8 instances, and staying above the dotted line at up to 16 instances (not shown).

Conversely, the square markers show an improved Octavo with 1 BTM instance, and AOM instances with 1 to 8 entries each. Note that since each addressing path to memory requires an AOM instance, we must always have three instances: one to support each instruction operand. Because of these multiple instances, their use of multiplexing, and the shorter pipeline between instruction fetch and data read, F_{max} scales poorly as the number of entries increases, falling off quickly for case with more than 4 entries per AOM.

The circle markers show the line through the design space with equal numbers of BTM instances and entries per AOM, 1 through 8, suggesting that the number of AOM entries tends to dominate the scaling of the system as a whole.

Finally, given the previous lines through the design space, the star markers point out some useful configurations: 2 AOM entries with 4 BTM instances (2/4), 2 AOM entries with 8 BTM instances (2/8), 3 AOM entries with 6 BTM instances (3/6), and 4 AOM entries with 8 BTM instances (4/8), which likely represents the largest useful configuration which can still reach 500 MHz after place-and-route.

For the remaining results in this section, we used the first point, 2 AOM entries with 4 BTM instances (2/4), as our benchmarking configuration since some benchmarks will require more than these resources, demonstrating benefits even without complete AOM/BTM support and at 0.939x of the original average F_{max} (495 MHz, down from 527 MHz). The peak F_{max} of the 2/4 configuration still reaches 510 MHz – 0.927x of the absolute maximum 550 MHz rating.

TABLE I. BENCHMARK CYCLE COUNT SPEEDUP AND EFFICIENCY IMPROVEMENTS. ROWS SHOW THE IMPACT OF AOMS AND BTMS. COLUMNS SHOW THE IMPACT OF LOOP UNROLLING.

Benchmark	Cycles per Pass			Execution Efficiency		
Hailstone	MIPS	Octavo	Speedup	MIPS	Octavo	Increase
Looping	970	504	1.92x	0.655	0.863	1.32x
Unrolled	769	701	1.10x	0.824	0.899	1.09x
Speed./Incr.	1.26x	0.72x	—	1.26x	1.04x	—
Increment	MIPS	Octavo	Speedup	MIPS	Octavo	Increase
Looping	716	376	1.90x	0.631	0.907	1.44x
Unrolled	431	331	1.39x	1.000	1.00	1.00x
Speed./Incr.	1.66x	1.14x	—	1.58x	1.10x	—
Reverse	MIPS	Octavo	Speedup	MIPS	Octavo	Increase
Looping	404	354	1.14x	0.748	0.856	1.14x
Unrolled	309	309	1.00x	1.000	1.000	1.00x
Speed./Incr.	1.31x	1.15x	—	1.34x	1.17x	—
FIR	MIPS	Octavo	Speedup	MIPS	Octavo	Increase
Looping	2902	2502	1.16x	0.897	0.960	1.07x
Unrolled	2614	2406	1.09x	0.996	0.998	1.00x
Speed./Incr.	1.11x	1.04x	—	1.11x	1.04x	—
FSM	MIPS	Octavo	Speedup	MIPS	Octavo	Increase
—	807	753	1.07x	0.564	0.467	0.83x

A. Benchmarks

Since no compiler currently supports AOM/BTM functions, we wrote our own set of micro-benchmarks in assembly. These benchmarks represent various points in the general structure of sequential programs, including simple loops, complex branching, and numerical processing, and sometimes show the behaviour of our improvements under non-ideal conditions where we cannot eliminate all overhead. All benchmarks run under an outermost infinite loop, and we base our measurements on the number of completed passes over 200,000 simulation cycles.

We wrote the benchmarks in a strict load-compute-store form, emulating a MIPS-like processor. Normally, Octavo's register-less, flat memory model combines loads and stores with ALU operations, which would make the comparison unfair. Specifically, we would be unable to isolate the speedup and efficiency improvement that arise solely from incorporating the BTM and AOM, as some additional speedup/efficiency would originate from the elimination of loads/stores to/from registers. Thus, for fairness, we report *conservative* results, avoiding Octavo-specific advantages and measured against the emulated ideal MIPS-like CPU, affected only by our use of the BTM and AOM. Furthermore, all processing happens at Octavo's native word width of 36 bits.

Table I summarizes the cycle count speedups and execution efficiency improvement of our benchmarks. We measure efficiency using the notion of "useful" instructions introduced in Section III. We show both looping and unrolled versions since unrolling reveals which instructions are fundamentally useful to a benchmark. However, unrolling itself is impractical, as the code size increases to nearly fill Octavo's instruction memory. The "MIPS" entries refer to benchmarks run on the emulated ideal MIPS-like CPU, and the "Octavo" entries refer to optimized benchmarks using AOMs and BTMs. Reading across columns shows the impact of the AOMs and BTMs, while reading down columns shows the impact of loop unrolling.

Hailstone We previously described the hailstone benchmark, which exercises various computing and branching operations, in Section III. When using the AOMs/BTMs, we see a 1.92x speedup and 1.32x efficiency increase. Using AOMs/BTMs on unrolled code yields worse results since we must keep reloading the BTM with branch data for each unrolled loop.

Array Increment We increment an array of 10 elements by 1, repeated 10 times, then output the entire array, showing a simple iterated calculation with a separate output loop. This benchmark requires five branches, forcing us to periodically reload one of the four BTM entries to support it. However, we can hoist that overhead outside of the loop, and still obtain a significant speedup. Using AOMs/BTMs grants a speedup of 1.90x and an efficiency improvement of 1.44x. The efficiency suffers due to the overhead of reloading the BTM with data for the fifth branch, associated with the inner loop.

Array Reverse We traverse an array of 100 elements using two pointers, top-to-middle and bottom-to-middle, loading and storing to swap their values without any computation or I/O. This memory-bound benchmark forces the bottom-to-middle pointer to decrement, which our current AOM does not support. Furthermore, due to the write-only nature of the AOM, we have to keep a copy of the bottom-to-middle pointer, add -1 to it, then update its AOM entry, all within the main loop. Nonetheless, AOMs/BTMs give us a speedup and efficiency increase of 1.14x.

8-Tap FIR Filter We sequentially read a 100-entry input buffer, applying an 8-tap FIR filter at each step, and output the filtered values to a 100-entry output buffer. We keep all 8 taps and 8 coefficients in registers, shifting values down the taps then performing the multiplications and additions, both as unrolled inner loops. Despite the sequential bulk of the buffering and convolution code, AOMs/BTMs speed up the FIR filter by 1.16x and improve its efficiency by 1.07x. Note that available memory limited us to unroll 25 times instead of 100, thus some loop overhead remained in all cases.

Floating-Point Number FSM We parse a stream of 100 characters using a Finite State Machine, looking for simple space-delimited floating point numbers such as 5.5, $-3.$, $+.8$, etc…, and raise either an Accept or Reject signal. The 100 input characters contain 25 valid numbers and one invalid one, forcing the FSM to walk through all its paths. We implement the FSM directly in the program structure, alternating tests and branches to determine state and action. FSM contains no real loops, no hoistable computations, no significant addressing, no basic block longer than 2 or 3 instructions, and has 34 unique branches, greatly exceeding the capacity of the four-entry BTM and forcing us to continually reload a BTM entry with the data for the next upcoming branch. Nonetheless, we can use the BTM to cache three other branches to save cycles on the most commonly traversed edges, granting a 1.07x speedup and 0.83x efficiency improvement. We cannot unroll FSM due to its complex and variable execution, and efficiency suffers due to some BTM re-loads not folding into branches.

In summary, the use of BTMs and AOMs *always* speeds up sequential code, conservatively measured relative to a "perfect" MIPS-like CPU, and even taking into account the 0.939x change in F_{max} from their implementation. Using AOMs/BTMs also improves execution efficiency of our benchmarks to a minimum of 0.856 of the maximum (1.00) achievable via

loop unrolling, up to a 1.05x ($0.863/0.824$) improvement for Hailstone, without the associated code size increase.

VIII. Conclusions and Future Work

In this work, we located intrinsic overheads in sequential programs as separate addressing and flow-control sub-graphs interleaved with the actual desired computations. We extracted these sub-graphs as data describing sets of conditions the processor can pattern-match against, implemented as Address Offset Modules (AOMs) and Branch Trigger Modules (BTMs) operating in parallel with the instruction fetch.

We found AOMs and BTMs to scale to useful levels, supporting up to 4 active pointers per instruction operand and up to 8 active branches before F_{max} dropped below 500 MHz on a modified Octavo soft-processor on a Stratix IV FPGA. Furthermore, benchmarking revealed that using BTMs and AOMs *always* reduced the cycle count, speeding up execution by 1.07x for branch-heavy control code, up to 1.92x for looping code. In all but one case, which had incomplete AOM support, the speed-up of ordinary looping code *always* exceeded that granted by total loop unrolling, since AOMs/BTMs can additionally eliminate pointer increments and fold branches with useful work. Using AOMs and BTMs also improved execution efficiency, defined as the ratio of executed instructions which perform the actual desired computations. While total loop unrolling often reaches an execution efficiency ratio of 1.00, our benchmarks improved their efficiency to a minimum of 0.856, reaching up to 1.05x the efficiency of unrolled code, all without the associated code size increase. Only control-heavy code, impossible to unroll, suffered a decrease in efficiency from 0.564 to 0.467 (0.828x).

For future work, we plan to add support for additional branch conditions (e.g.: loop counters) in the BTM, to extend the AOM's post-incrementing to negative numbers to avoid the overhead seen in the Array Reverse benchmark, as well as extend the increment range and optionally mask it to enable strided and modulo addressing. Finally, we plan to investigate if our implementation uses MLABs effectively, and thus can provide meaningful area results.

Acknowledgements

The authors gratefully acknowledge funding from the Walter C. Sumner Foundation, Altera, and NSERC. The authors also thank Altera for generously providing Quartus licenses.

References

[1] J. Cong and Y. Zou, "FPGA-Based Hardware Acceleration of Lithographic Aerial Image Simulation," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 3, pp. 1–29, 2009.

[2] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors," *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 162–170, 2004.

[3] S. Sirowy and A. Forin, "Where's the beef? Why FPGAs are so fast," Microsoft Research, Tech. Rep. September, 2008.

[4] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," *International Symposium on Computer Architecture (ISCA)*, pp. 37–47, 2010.

[5] A. Dehon, "Advantage of Configurable Computing," *IEEE Computer*, pp. 41–49, April 2000.

[6] M. C. Herbordt, Y. Gu, T. Vancourt, J. Model, B. Sukhwani, and M. Chiu, "Computing Models for FPGA-Based Accelerators." *Computing in Science & Engineering*, vol. 10, no. 6, pp. 35–45, Oct. 2008.

[7] A. Severance, J. Edwards, H. Omidian, and G. Lemieux, "Soft Vector Processors with Streaming Pipelines," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2014, pp. 117–126.

[8] P. Yiannacouras, J. G. Steffan, and J. Rose, "Portable, Flexible, and Scalable Soft Vector Processors," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 20, no. 8, pp. 1429–1442, Aug. 2012.

[9] M. Labrecque and J. Steffan, "Improving Pipelined Soft Processors with Multithreading," in *International Conference on Field-Programmable Logic and Applications (FPL)*, Aug 2007, pp. 210–215.

[10] C. E. LaForest and J. G. Steffan, "OCTAVO: An FPGA-Centric Processor Family," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, Feb. 2012, pp. 219–228.

[11] D. R. Ditzel and H. R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," in *International Symposium on Computer Architecture (ISCA)*, 1987, pp. 2–8.

[12] L. H. Lee, J. Scott, B. Moyer, and J. Arends, "Low-cost branch folding for embedded applications with small tight loops," in *International Symposium on Microarchitecture (MICRO-32)*, 1999, pp. 103–111.

[13] M. J. Knieser and C. A. Papachristou, "Y-Pipe: A Conditional Branching Scheme Without Pipeline Delays," in *International Symposium on Microarchitecture (MICRO-25)*, 1992, pp. 125–128.

[14] A. Gonzalez and J. Llaberia, "Reducing branch delay to zero in pipelined processors," *IEEE Transactions on Computers*, vol. 42, no. 3, pp. 363–371, Mar 1993.

[15] S. Wang and J. Provence, "Branch-skipped pipelined microprocessor," *Electronics Letters*, vol. 30, no. 14, pp. 1122–1123, Jul 1994.

[16] J. W. Davidson and D. B. Whalley, "Reducing the Cost of Branches by Using Registers," in *International Symposium on Computer Architecture (ISCA)*, 1990, pp. 182–191.

[17] P. Yiannacouras, J. Steffan, and J. Rose, "Data parallel FPGA workloads: Software versus hardware," in *International Conference on Field-Programmable Logic and Applications (FPL)*, Aug 2009, pp. 51–58.

[18] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux, "VEGAS: soft vector processor with scratchpad memory," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2011, pp. 15–24.

[19] A. Severance and G. Lemieux, "VENICE: A compact vector processor for FPGA applications," in *International Conference on Field-Programmable Technology (FPT)*, Dec 2012, pp. 261–268.

[20] ——, "Embedded supercomputing in FPGAs with the VectorBlox MXP Matrix Processor," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sept 2013, pp. 1–10.

[21] Analog Devices, "ADSP-TS101 TigerSHARC DSP Programming Reference," 2005.

[22] ——, "Blackfin Processor Programming Reference," no. 82, 2008.

[23] Texas Instruments, *TMS320C64x/C64x+ DSP CPU and Instruction Set*, 2010, no. July, No. SPRU732J.

[24] C.-M. Chen, Y.-Y. Chen, and C.-T. King, "Branch Merging for Effective Exploitation of Instruction-level Parallelism," in *International Symposium on Microarchitecture (MICRO-25)*, 1992, pp. 37–40.

[25] S.-M. Moon, S. D. Carson, and A. K. Agrawala, "Hardware Implementation of a General Multi-way Jump Mechanism," in *Workshop and Symposium on Microprogramming and Microarchitecture (MICRO-23)*, 1990, pp. 38–45.

[26] M. G. H. Katevenis, *Reduced Instruction Set: Computer Architectures for VLSI*. MIT Press, 1985, ACM doctoral dissertation awards.

[27] S. McFarling and J. Hennesey, "Reducing the Cost of Branches," in *International Symposium on Computer Architecture (ISCA)*, 1986, pp. 396–403.

[28] R. Rojas, "How to Make Zuse's Z3 a Universal Computer," *IEEE Ann. Hist. Comput.*, vol. 20, no. 3, pp. 51–54, Jul. 1998.

[29] Altera, "DC and Switching Characteristics for Stratix IV Devices," Mar 2014. [Online]. Available: http://www.altera.com/literature/hb/stratix-iv/stx4_siv54001.pdf