Appears in the Proceedings of the 47th Int'l Symp. on Microarchitecture (MICRO-47), December 2014

Architectural Specialization for Inter-Iteration Loop Dependence Patterns

Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{ss2783,bi45,mt453,gl387,zhiruz,cbatten}@cornell.edu

***Abstract*—Hardware specialization is an increasingly common technique to enable improved performance and energy efficiency in spite of the diminished benefits of technology scaling. This paper proposes a new approach called explicit loop specialization (XLOOPS) based on the idea of elegantly encoding inter-iteration loop dependence patterns in the instruction set. XLOOPS supports a variety of inter-iteration data- and control-dependence patterns for both single and nested loops. The XLOOPS hardware/software abstraction requires only lightweight changes to a general-purpose compiler to generate XLOOPS binaries and enables executing these binaries on: (1) traditional microarchitectures with minimal performance impact, (2) specialized microarchitectures to improve performance and/or energy efficiency, and (3) adaptive microarchitectures that can seamlessly migrate loops between traditional and specialized execution to dynamically trade-off performance vs. energy efficiency. We evaluate XLOOPS using a vertically integrated research methodology and show compelling performance and energy efficiency improvements compared to both simple and complex general-purpose processors.**

I. Introduction

Serious physical design issues are breaking down traditional abstractions in computer architecture and motivating an increasing emphasis on hardware specialization. At the same time, computer architects have long realized the importance of focusing on the key loops that often dominate application performance. These two trends have led to a diverse array of specialized hardware for exploiting intra- and/or inter-iteration loop dependence patterns.

Hardware specialization to exploit *intra-iteration* loop dependence patterns usually involves custom instructions and/or small reprogrammable functional units well-suited to accelerating common sequences of operations *within* an iteration. Examples include application-specific instruction-set processors [1, 6] and techniques for subgraph execution [4, 11]. Hardware specialization to exploit *inter-iteration* loop dependence patterns focuses at a higher level on how *different* loop iterations interact. Examples include data-parallel accelerators which exploit loops with no inter-iteration dependences [8, 17, 34] and thread-level speculation which exploit loops with infrequent inter-iteration dependences [19, 30, 31]. Coarse-grained reconfigurable arrays [10, 13] and weakly programmable application-specific accelerators [33] target both intra- and inter-iteration loop dependence patterns.

All of these proposals must carefully navigate the tension between less efficient general architectures and more efficient specialized architectures. Some argue for exposing as much of the specialized microarchitecture as possible to enable flexible software configuration while maintaining efficiency [7, 12]. Unfortunately, this comes at the expense of a clean hardware/software abstraction; highly configurable specialized architectures are often tightly coupled to a specific microarchitectural implementation. A key research challenge involves creating clean hardware/software abstractions that are highly flexible, yet still enable efficient execution on both traditional and specialized microarchitectures.

To address this challenge, we focus on architectural specialization for inter-iteration loop dependence patterns. *Inter-iteration data-dependence patterns* include loops with no inter-iteration dependences and loops with inter-iteration dependences encoded through registers and/or memory. An interesting data-dependence pattern often found in graph algorithms involves iterations that manipulate a shared data structure such that the iterations can be executed in any order as long as their updates to memory appear atomic to the other iterations. *Inter-iteration control-dependence patterns* include loops that terminate based on comparing an induction variable to a loop-invariant fixed bound, or loops that terminate based on a data-dependent-exit condition. An interesting control-dependence pattern found in more irregular worklist-based algorithms involves a loop induction variable compared to a dynamic bound that is monotonically increased during the loop execution. The inter-iteration dependence pattern for a given loop will be a combination of a specific data- and control-dependence pattern, and nested loops can be captured using the composition of multiple loop patterns.

In this paper, we explore *explicit loop specialization* (XLOOPS) which is based on the idea of explicitly encoding inter-iteration loop dependence patterns in the instruction set to enable exploiting fine-grain loop-level parallelism. Section II describes our approach for designing XLOOPS instruction sets, compilers, and microarchitectures. Our XLOOPS instruction set can encode: data-dependence patterns where the loops can appear to execute in any order both concurrently or atomically; data-dependence patterns where the loops must preserve ordering constraints expressed through either register or memory dependences; and control-dependence patterns based on fixed and dynamic bounds. Our XLOOPS compiler uses programmer annotations to automatically generate an efficient XLOOPS binary. The XLOOPS abstraction enables XLOOPS binaries to execute on either traditional microarchitectures with minimal performance impact or on specialized microarchitectures that exploit fine-grain loop-level parallelism to improve performance and energy efficiency. This abstraction also enables adaptive execution where a loop is seamlessly migrated by hardware between traditional and specialized microarchitectures in order to find the optimal performance/efficiency trade-off.

To make the case for XLOOPS, we use a vertically integrated evaluation methodology. Section III describes the application kernels we use for evaluation and modifications to an LLVM-based compiler to support XLOOPS. Section IV

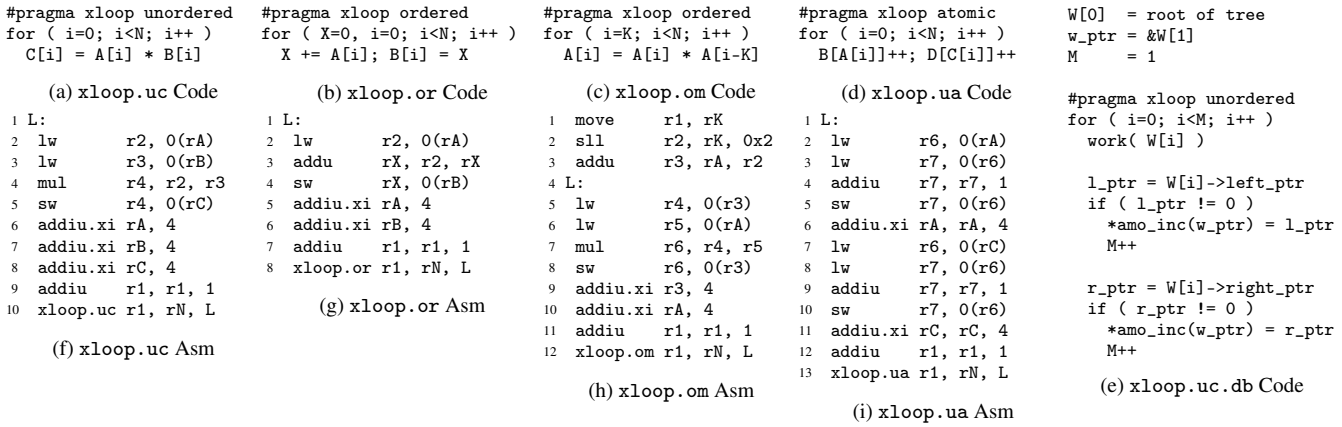

Figure 1. XLOOPS Instruction Set Examples – Unless otherwise specified, a fixed-bound control-dependence pattern is assumed. (a,f) `xloop.uc` encodes an unordered-concurrent data-dependence pattern, `addiu.xi` encodes a simple associative loop-carried dependence; (b,g) `xloop.or` encodes an ordered-through-registers data-dependence pattern, line 3 captures the loop-carried dependence through `rX`; (c,h) `xloop.om` encodes an ordered-through-memory data-dependence pattern, line 6 depends on an earlier instance of line 8; (d,i) `xloop.ua` encodes an unordered-atomic data-dependence pattern; (e) `xloop.uc.db` encodes an unordered-concurrent data-dependence with a dynamic-bound control-dependence pattern, `amo_inc()` uses an atomic memory operation to increment the tail pointer of the worklist by four.

describes the cycle-level modeling of XLOOPS microarchitectures that support traditional, specialized, and adaptive execution. Section V describes the register-transfer-level (RTL) implementation of a simple XLOOPS microarchitecture capable of specialized execution and area, energy, and timing analysis using a commercial ASIC CAD toolflow.

Using specialized execution, XLOOPS is able to achieve 2.5× or higher speedup at similar or better energy efficiency for most application kernels compared to a simple single-issue in-order processor with only 40% area overhead. Compared to aggressive two- and four-way out-of-order processors, XLOOPS is able to achieve 1.5–3× improvement in energy efficiency while having speedups in the range of 1.25–2.5× on most application kernels. Adaptive execution enables applications that perform worse with specialized execution to automatically migrate to the general-purpose processor for increased performance at reduced energy efficiency.

The primary contributions of this work are: (1) we propose an elegant new XLOOPS hardware/software abstraction that explicitly encodes inter-iteration loop dependence patterns; (2) we describe an XLOOPS compiler based on lightweight modifications to a general-purpose compiler; (3) we propose new XLOOPS microarchitectures that enable traditional, specialized, and adaptive execution; and (4) we evaluate XLOOPS using a vertically integrated methodology.

II. XLOOPS: Explicit Loop Specialization

In this section, we describe the instruction set and compiler modifications required for XLOOPS, and we propose various XLOOPS microarchitectures to enable traditional, specialized, and adaptive execution.

A. XLOOPS Instruction Set

The XLOOPS instruction set is carefully designed to enable efficient execution on *both* traditional general-purpose processors (serial execution) and specialized microarchitectures (parallel execution). The XLOOPS instruction set is formed by extending a general-purpose instruction set with the instructions shown in Table I. The key idea is to express inherent loop-level parallelism by encoding inter-iteration data- and control-dependence patterns using variants of the `xloop` instruction. All `xloop` instructions encode the notion of a *parallel loop body* which is defined as the static instruction sequence between a given label L and the address of the `xloop` instruction. It is undefined for the label L to point to an address greater than or equal to the address of the `xloop` instruction. Figure 1 uses short pseudocode and assembly examples to illustrate how these instructions are used in practice. The suffixes for the `xloop` instruction indicate the data- and control-dependence patterns. An `xloop` can contain arbitrary instructions including: arithmetic operations, memory operations, atomic memory operations (AMOs), memory fences, control flow, nested `xloop`s, and system calls (although this is not recommended). Currently, the `xloop` instruction only supports fixed- and dynamic-bound control-dependence patterns; we leave exploring data-dependent-exit control-dependence patterns to future work. An `xloop` cannot write live-in registers and all live-out register values are undefined once the loop is finished executing, meaning an `xloop` must store results in memory.

Table I. XLOOPS Instruction Set Extensions

`xloop.{d}.{c}`	`rI, rN, L`	goto L if R[rI] ≠ R[rN]
`addiu.xi`	`rX, imm`	R[rX] ← R[rX] + imm
`addu.xi`	`rX, rT`	R[rX] ← R[rX] + R[rT]

Loop body is defined as the static sequence of instructions between `L` and the `xloop` instruction. `{d}` suffix indicates data-dependence pattern: `uc` = unordered concurrent, `or` = ordered through registers, `om` = ordered through memory, `orm` = ordered through registers and memory, `ua` = unordered-atomic. `{c}` suffix indicates control-dependence pattern: no suffix implies fixed bound, `db` = dynamic bound. R[`rT`] must be a loop-invariant value.

`xi` **Instruction –** Mutual induction variables (MIVs) are variables that can be computed as a linear function of a loop induction variable. Modern compilers include a *loop-strength*

reduction pass that transforms expensive MIV computations into cheap iterative operations. Naively using such optimizations can impose extra, potentially unnecessary inter-iteration dependences, but avoiding such optimizations can introduce non-trivial address computation overhead, especially when working with multi-dimensional arrays. The cross-iteration instructions (denoted with an `xi` suffix) explicitly encode MIVs to allow hardware to handle MIVs either iteratively or in parallel using specialized logic. Note that the register operand R[`rT`] in an `addu.xi` instruction must be a loop-invariant value. The instructions on lines 6–8 in Figure 1(f) illustrate the use of the `xi` instruction.

`xloop.uc` **Instruction –** An `xloop.uc` encodes an unordered-concurrent data-dependence pattern. The iterations can appear to execute concurrently and in any order. Data races are possible, but atomic memory operations can provide efficient synchronization if required. Figure 1(a,f) illustrates using an `xloop.uc` for element-wise vector multiplication. The XLOOPS ISA specifies that an `addiu` writing the loop induction variable (e.g., line 9) does not impose an ordering constraint.

`xloop.or` **Instruction –** An `xloop.or` encodes an ordered-through-registers data-dependence pattern. We term registers that impose ordering constraints as *cross-iteration registers* (CIRs). The value in a CIR for a given iteration must be the same as if the `xloop` was executed serially. Any general-purpose register can act as a CIR. The CIRs must be read at least once and can be written zero or more times. As an exception to the restriction on `xloop` register live-outs, each CIR is guaranteed to have the same value as a serial execution when the loop is finished. As with an `xloop.uc`, there are no ordering constraints with respect to memory, so memory races are possible. Figure 1(b,g) illustrates using an `xloop.or` to implement parallel-prefix summation with `rX` as a CIR.

`xloop.om` **Instruction –** An `xloop.om` encodes an ordered-through-memory data-dependence pattern. Values read and written to memory must be the same as if the loop was executed serially. Since an `xloop.om` guarantees a specific order with respect to memory, there can be no race conditions. For example, if each iteration updates different portions of a shared data structure, then iterations may occasionally conflict in which case the updates are guaranteed to occur in the same order as if the loop was executed serially. Figure 1(c,h) illustrates using an `xloop.om` to implement a simple loop where the load instruction on line 6 in iteration `i` depends on the store instruction on line 8 in iteration `i-K`. An `xloop.orm` encodes a pattern that combines ordering through registers and memory.

`xloop.ua` **Instruction –** An `xloop.ua` encodes an unordered-atomic data-dependence pattern. The iterations can appear to execute in any order, but their memory updates must appear to execute atomically. While race conditions are not possible, the results can be non-deterministic since the hardware is free to reorder iterations. This data-dependence pattern is often found in graph algorithms that manipulate a shared data structure where the iterations can execute in any order given that iterations update memory atomically. Figure 1(d,i) illustrates using an `xloop.ua` to modify two histograms with a single atomic update.

`xloop.*.db` **Instruction –** The above data-dependence patterns assume a fixed-bound control-dependence pattern. An `xloop.*.db` encodes a different inter-iteration control-dependence pattern where iterations are allowed to monotonically increase the loop bound. Figure 1(e) illustrates using an `xloop.uc.db` to perform work on a binary tree using a worklist-based implementation. Each iteration uses an AMO to reserve space at the tail of the worklist before adding new nodes and incrementing the loop bound. This example could also be encoded as an outer for loop with an inner `xloop.uc` to iterate over the nodes in a given level of the tree, but using an `xloop.uc.db` results in a more natural mapping and can enable more efficient specialized execution.

The XLOOPS instruction set provides precise exceptions at the instruction level within an `xloop` iteration. This means exceptions within a loop iteration are guaranteed to occur in order with respect to the other instructions *in that loop iteration*. Exceptions in *different* iterations of an `xloop.{uc,ua,or}` can occur in any order; exceptions in *different* iterations of an `xloop.{om,orm}` are guaranteed to occur in the same order as a serial execution.

The XLOOPS ISA is a clean hardware/software abstraction that provides significant freedom when designing XLOOPS compilers and XLOOPS microarchitectures. Any given loop can usually be encoded in multiple ways. For example, any valid `xloop.uc` is also a valid `xloop.or`, any valid `xloop.ua` is also a valid `xloop.om`, and any fixed-bound `xloop` is a valid `xloop.orm`. Software should choose the "least restrictive" inter-iteration dependence pattern to enable execution on simpler specialized microarchitectures and to give hardware the most freedom in choosing how to execute the `xloop`. Specialized execution of `xloop.om` is more complex than `xloop.or` which in turn is more complex than an `xloop.uc`, so an architect can choose to only support specialized execution for an `xloop.uc` and use traditional execution for the remaining patterns. Similarly, the maximum number of instructions in an `xloop` is not part of the instruction set; while software should target fine-grain loops, it is perfectly fine to generate a relatively large loop body (e.g., 200 instructions). A specific microarchitecture can always fall back to a traditional execution if the `xloop` is too large. Finally, the XLOOPS instruction set enables cleanly nesting `xloop`s. Software can provide hints to the hardware to indicate which `xloop` might be best for specialized execution, or the hardware might adaptively explore specialized executions for different `xloop`s.

B. XLOOPS Compiler

The XLOOPS compiler currently uses programmer inserted annotations to determine which loops to encode using the XLOOPS instruction set. Figure 1(a–e) illustrates using `#pragma` directives and the keywords `unordered`, `ordered`, and `atomic` to convey the data-dependence patterns. Figure 2 illustrates annotating nested loops in the Floyd-Warshall

```
for ( int k = 0; k < n; k++ )
  #pragma xloops ordered
  for ( int i = 0; i < n; i++ )
    #pragma xloops unordered
    for ( int j = 0; j < n; j++ )
      path[i][j] = min( path[i][j], path[i][k] + path[k][j] );
```

Figure 2. C Code for *war* Application Kernel – Kernel from Polybench suite implementing Floyd-Warshall shortest path algorithm. XLOOPS compiler maps inner loop to `xloop.uc` and uses dependence analysis to map outer loop to `xloop.om`.

```
#pragma xloops ordered
for ( int i = 0; i < num_edges; i++ ) {
  int v = edges[i].v; int u = edges[i].u;
  if ( vertices[v] < 0 && vertices[u] < 0 ) {
    vertices[v] = u; vertices[u] = v; out[k++] = i;
} }
```

Figure 3. C Code for *mm* Application Kernel – Kernel from Problem-Based Benchmark Suite implementing greedy algorithm for maximal matching on undirected graph. XLOOPS compiler uses dependence analysis to map the loop to `xloop.orm`.

shortest path algorithm from the Polybench Suite [27], and Figure 3 illustrates annotating an ordered loop in the maximal matching application kernel present from the Problem-Based Benchmark Suite [29].

The XLOOPS compiler is implemented with lightweight changes to an existing general-purpose compiler. The XLOOPS approach does not interfere with existing compiler algorithms for mid-level optimization passes, and back-end algorithms for instruction scheduling, register allocation, and code generation. The XLOOPS compiler modifies the loop-strength reduction pass to directly generate appropriate `xi` instructions to encode the MIVs. Loops annotated with the `unordered` keyword are usually encoded using `xloop.uc`. Loops annotated with the `atomic` keyword are encoded using `xloop.ua`. Programmers use the `ordered` keyword to annotate loops that must preserve inter-iteration data-dependences. The programmers need not specify whether this data-dependence is through registers or memory or both. The XLOOPS compiler includes analysis passes to determine how the data-dependence is communicated and encodes the dependence patterns using `xloop.{or/om/orm}`. Register dependence testing is implemented by analyzing the use-definition chains through the PHI nodes and identifying CIRs. Memory dependence testing is implemented using well known dependence analysis techniques such as the zero-, single-, and multiple-index variable tests [9]. Additionally, the XLOOPS compiler includes a pass to detect updates to the loop bound to encode such loops using `xloop.*.db`.

Although these lightweight changes to a general-purpose compiler should produce a reasonable XLOOPS compiler, there are opportunities for additional optimizations. For example, the performance of executing an `xloop.or` is limited by the *inter-iteration critical path* for each CIR. The inter-iteration critical path is the distance between the first dynamic instruction in the loop body that reads the CIR and the last dynamic instruction in that same loop body that updates the CIR. Compiler optimizations to reduce the inter-iteration critical path by modifying the instruction scheduling within the loop body could improve the ability of XLOOPS microarchitectures to overlap independent work from different iterations. We explore the potential of such an optimization by manually scheduling instructions in Section IV-G.

C. XLOOPS Traditional Execution

XLOOPS binaries can be executed efficiently on a general-purpose processor (GPP) with minimal changes to the decoder logic. An `xloop` instruction is executed as a conditional branch instruction, and an `xi` instruction is executed as a simple addition. Efficient traditional execution is important for two reasons: (1) to enable gradual adoption of XLOOPS without any penalty when using XLOOPS binaries on GPPs; and (2) to enable adaptive execution to migrate an `xloop` to a GPP if it is determined that specialized execution is not resulting in any performance benefit.

D. XLOOPS Specialized Execution

Figure 4 shows a novel XLOOPS microarchitecture that augments a GPP with a loop-pattern specialization unit (LPSU). The GPP can either be a simple in-order or a complex out-of-order processor. The LPSU contains four decoupled lanes and a lane management unit (LMU). In our current design, each lane in the LPSU is similar to a simple in-order processor, but it is certainly possible to use more aggressive superscalar or out-of-order lane microarchitectures to better exploit intra-iteration instruction-level parallelism. Each lane includes a loop instruction buffer to store instructions, an index queue (IDQ) to store loop indices waiting for execution, a 2r2w-port physical register file, and functional units for simple arithmetic, address generation, and control flow. The GPP and LPSU dynamically arbitrate for the data memory port and the long-latency functional unit (LLFU). The LLFU provides support for integer multiplication, integer division, and floating-point arithmetic. Specialized execution occurs in two phases: a *scan phase* initiated by the GPP and a *specialized execution phase* that occurs on the LPSU.

Scan Phase – The GPP starts the scan phase when it reaches an `xloop` instruction. In this phase, the instructions and live-in register values within the loop body are incrementally written to the instruction buffers and register files in the LPSU. To reduce the required amount of physical register storage in the LPSU, the LMU renames architectural register specifiers and updates the instruction encoding as it writes instructions into the instruction buffers. Since the registers are renamed once during the scan phase, the energy consumed for register renaming is amortized over all iterations. A complex out-of-order GPP can overlap the scan phase with the execution of instructions that are before the `xloop` body. The specialized execution phase does not start until all previous instructions are retired, all instructions in the `xloop` body have been scanned, and the `xloop` instruction reaches the head of the reorder buffer. Once the scan phase is complete, the GPP stalls until the LPSU has finished the specialized execution phase.

Specialized Execution Phase – In this phase, the LMU enqueues iteration indices into the IDQs as free IDQ entries

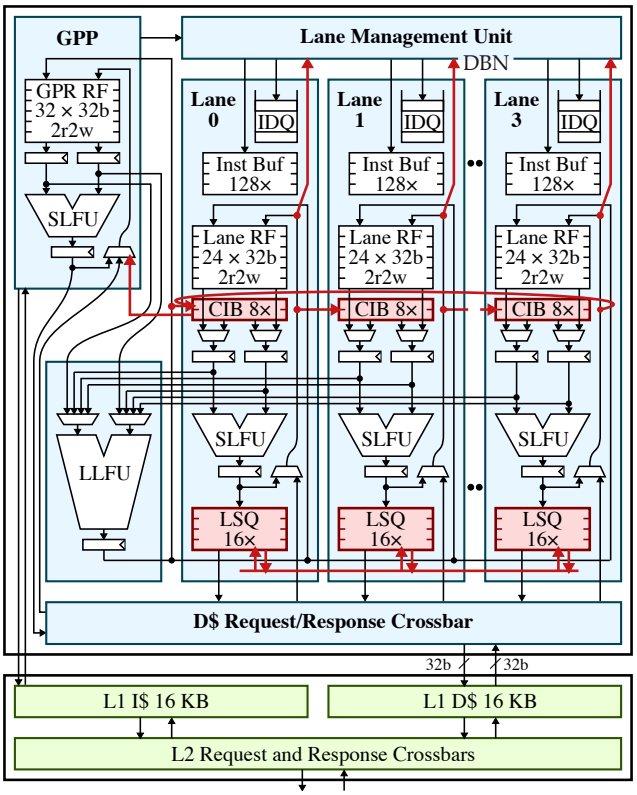

Figure 4. XLOOPS Microarchitecture – GPP and L1 memory system augmented with four-lane loop-pattern specialization unit (LPSU). Mechanisms required to support `xloop.{or,om,orm,ua}` beyond basic `xloop.uc` support are shown highlighted in red. GPP = general-purpose processor (either in-order or out-of-order); GPR = GPP regs; RF = regfile; IDQ = index queue; Inst Buf = instruction buffer; DBN = dynamic-bound notification; CIB = cross-iteration buffer; SLFU = short-latency functional unit; LLFU = long-latency functional unit; LSQ = load/store queue.

become available. For `xloop.uc`, IDQ entries can become available in any order enabling simple dynamic load balancing, while for other inter-iteration dependence patterns, IDQ entries naturally become available in iteration order. Each lane dequeues an iteration index and executes the corresponding iteration. Since the XLOOPS ISA guarantees live-in registers are not written in the `xloop` body, there is no need to restore state before the execution of each iteration. When the execution of the entire `xloop` is finished and all memory updates are complete, the LMU notifies the GPP that the specialized execution phase has ended.

`xi` **Execution –** The LPSU uses specialized logic to execute `xi` instructions. In the scan phase, the LMU uses a mutual induction variable table (MIVT) to track the register specifier for the MIV and the loop-invariant increment value (i.e., either `imm` for `addiu.xi` or R[`rT`] for `addu.xi`). In the specialized execution phase, the lanes check the read register specifiers of a decoded instruction and compare it to the specifiers stored in the MIVT using a bit vector. If the register specifier matches an entry present in the MIVT, then the lane computes the value of the mutual induction variable using: the value present in the register file, the difference in the loop indices, and the loop-invariant increment value from the MIVT as shown:

$$R[\texttt{rX}] \leftarrow R[\texttt{rX}] + (\text{increment} \times (1 + i_{current} - i_{previous}))$$

Since the difference in inter-iteration loop indices is small (usually close to the number of lanes), the lanes can use an inexpensive, narrow multiplier. When the lane executes the actual `xi` instruction, the result of the above computation is stored in the register file and used by the next iteration executed on the lane.

`xloop.uc` **Execution –** Supporting `xloop.uc` requires just the mechanisms described above. Figure 4 illustrates these mechanisms and highlights the additional mechanisms required to support the more sophisticated inter-iteration dependence patterns described in the rest of this section.

`xloop.or` **Execution –** The cross-iteration buffers (CIBs) between neighboring lanes are small associative buffers that are used to communicate inter-iteration register dependences when executing an `xloop.or`. The LMU needs to identify each CIR and the last instruction that updates each CIR. During the scan phase, the LMU uses two bit-vectors to track register reads and writes. Registers that are first read and then written are identified as CIRs. In scan phase, the LMU also tracks the largest PC of an instruction that updates a CIR and sets a special *last CIR write bit* in the instruction buffer for this instruction. When a lane executes an instruction, it checks if a source register is a CIR and stalls if the CIR value is not available in the CIB connected to the previous lane. If the value is available, it writes this value to the lane register file and uses this value for the execution of the instruction. The lane also checks the last CIR write bit when executing an instruction. If this bit is set, then the lane writes the instruction result to the CIB connected to the next lane. Updates to a CIR can be conditional depending on the dynamic control flow. If an instruction with the last CIR write bit set was skipped, then at the end of the iteration the lane will copy the corresponding CIR value to the CIB.

`xloop.om` **Execution –** Efficient parallel execution of `xloop.om` requires hardware memory disambiguation support to determine when speculative iterations violate the serial memory ordering constraint. Each lane includes a small 2r1w load-store queue (LSQ) to track memory accesses across iterations. Memory dependence ordering is enforced by the LMU based on the loop iteration index. A lane with the lowest iteration index is considered as non-speculative, whereas lanes with higher iteration indices are considered speculative. Loads and stores issued by a non-speculative lane are allowed to bypass the LSQ and access memory immediately. A store issued by a speculative lane is buffered in the speculative lane's LSQ and does not update memory. A load issued by a speculative lane first checks for a matching store address in the speculative lane's LSQ for store-load forwarding. If there is no match in the speculative lane's LSQ, the load is issued to the memory system. More aggressive implementations can additionally allow a load to check the LSQs across lanes for inter-iteration store-load forwarding. To detect memory dependence violations, the address for each store issued by

the non-speculative lane is broadcast to the speculative lanes when the store executes. Each speculative lane compares this broadcasted store address to the entries present in the speculative lane's LSQ. A memory dependence violation occurs if a speculative lane has already issued a load request to the same address as a store issued by a non-speculative lane. When a speculative lane detects a memory dependence violation, the lane restarts the execution of the corresponding iteration. Squashing iterations is fast since an `xloop` cannot write live-in registers; the lane simply flushes the pipeline (including the LSQ) and restarts execution from the first instruction in the `xloop` body. Speculative lanes stall execution if they fill up their corresponding LSQ. When the LMU promotes a speculative lane to be non-speculative, the lane drains its LSQ, broadcasts store requests to other lanes, and updates the memory. Supporting `xloop.orm` involves combining the mechanisms required for supporting both `xloop.or` and `xloop.om`.

`xloop.ua` **Execution –** Similar to `xloop.om`, efficient parallel execution of `xloop.ua` requires hardware memory disambiguation support. However, `xloop.ua` does not enforce sequential ordering of the loop iterations. Currently, we execute `xloop.ua` using the same mechanisms as `xloop.om`. Future work could explore microarchitectures that are less restrictive in terms of iteration index ordering and take better advantage of the `xloop.ua` data-dependence pattern.

`xloop.*.db` **Execution –** Execution of loops with a dynamic-bound control-dependence pattern is similar to loops with a fixed-bound dependence pattern with minor changes to the LMU and lane control logic. Each lane checks for instructions that update the register containing the loop bound and communicate the value of the updated loop bound to the LMU. The LMU updates the maximum bound for the loop execution and generates additional iteration indices which are enqueued in the IDQs as space becomes available. Mechanisms to execute any data-dependence pattern can be combined with the mechanism to execute the dynamic-bound control-dependence pattern, although in this work we focus on `xloop.uc.db`.

E. XLOOPS Adaptive Execution

For certain applications with significant intra-iteration instruction-level parallelism and limited inter-iteration parallelism, traditional execution on complex out-of-order GPPs can achieve better performance than specialized execution on the LPSU's simple in-order lanes. The XLOOPS hardware/software abstraction enables microarchitectures to adaptively migrate an `xloop` between traditional execution on the GPP and specialized execution on the LPSU.

Adaptive execution adds two new phases for profiling. When the GPP first executes an `xloop` it begins a *GPP profiling phase* to determine the performance of traditional execution. After profiling for a set number of iterations or cycles, the scan phase takes place as described in Section II-D. At the end of the scan phase, the GPP sends the number of profiled loop iterations and recorded cycles to the LPSU. The LPSU then begins an *LPSU profiling phase* to determine the performance of specialized execution. After the LPSU has executed the same number of iterations used in the GPP profiling phase, the LPSU compares the relative performance of traditional and specialized execution. If specialized execution is slower than traditional execution, the LPSU simply instructs the GPP to finish executing the remaining iterations. For `xloop.or`, CIR values for the last iteration executed on the LPSU are copied back to the GPP.

Migrating an `xloop` between the GPP and LPSU only occurs at loop iteration boundaries and involves transferring very little state. This makes `xloop` migration significantly more efficient compared to thread migration across cores with private caches. Since the profiling phase itself is a valid execution of the `xloop`, adaptive execution is an efficient mechanism that increases the performance of loops that struggle with specialized execution.

The GPP includes an adaptive profiling table (APT) to record the profiling progress for a small number of recently seen `xloop` instructions. The APT is indexed by the PC of the `xloop` instruction and contains an iteration count and, if profiling is complete, the decision on whether to use traditional or specialized execution for future dynamic instances of the `xloop`. When the GPP executes an `xloop` instruction, it checks the APT to see if it should continue profiling or immediately choose traditional or specialized execution. The APT enables the profiling phases to stretch across multiple dynamic instances of the `xloop` which is especially important for `xloop`s with small iteration counts. Our current implementation of adaptive execution does not reconsider the profiling results once a decision has been made, although this is an interesting direction for future work.

III. XLOOPS Application Kernels

We explored a diverse set of application kernels that capture multiple inter-iteration data- and control-dependence patterns for both single and nested loops. We include both numeric and non-numeric kernels with regular and irregular data and control flow. Table II shows the list of application kernels and corresponding inter-iteration dependence patterns for each loop in the kernel. Our application kernels are drawn from MiBench [14], PolyBench [27], PBBS [29], and our own custom kernels. The suffix for an application name indicates the inter-iteration dependence pattern that dominates the execution time. All of the kernels were parallelized by adding programmer annotations with minimal modifications to the original serial kernel. For select kernels, we also explored manual loop-transformations and hand-coded assembly implementations as described in Section IV-G.

We briefly describe the custom kernels. *rgb2cmyk-uc* performs color space conversion on a test image. *sgemm-uc* implements a single-precision matrix multiplication for square matrices using standard triple-nested loops. *ssearch-uc* implements the Knuth-Morris-Pratt algorithm to search a collection of byte streams for the presence of substrings. *viterbi-uc* decodes frames of convolutionally encoded data using the Viterbi algorithm. *dither-or* generates a black-and-white image from a gray-scale image using Floyd-Steinberg dithering. *kmeans-or* implements the *k*-means clustering algorithm.

TABLE II. XLOOPS APPLICATION KERNELS AND CYCLE-LEVEL RESULTS

		Loop Characteristics			Dynamic Insns			io Speedups		ooo/2 Speedups			ooo/4 Speedups		
Name	**Suite**	**Type**	**Num Insns**	**Num Iters**	**GPI**	**XLI**	**X/G**	**T**	**S**	**T**	**S**	**A**	**T**	**S**	**A**
rgb2cmyk-uc	C	uc	32	80	209K	209K	1.00	1.00	3.13	1.00	2.24	2.18	1.00	1.22	1.21
sgemm-uc	C	uc	27	32	340K	340K	1.00	1.00	4.03	1.06	2.29	2.03	1.00	1.17	1.10
ssearch-uc	C	uc	37–58	57	2.3M	2.3M	1.00	1.00	3.93	1.07	2.65	2.56	0.99	1.52	1.51
symm-uc	Po	uc	43	32	267K	266K	1.00	1.01	3.38	1.00	1.97	1.95	1.03	1.08	1.08
viterbi-uc	C	uc	31–34	1–2K	2.5M	2.3M	0.92	1.07	2.57	1.14	2.10	2.10	1.13	1.15	1.13
war-uc	Po	uc	21	32	438K	438K	1.00	1.00	3.33	1.00	1.91	1.90	1.00	1.85	1.84
adpcm-or	M	or	52	20K	932K	992K	1.06	0.97	1.16	0.94	0.82	0.94	0.94	0.55	0.94
covar-or	Po	or	8–17	32	177K	161K	0.91	1.05	2.58	1.00	1.38	1.35	1.03	0.85	1.05
dither-or	C	or	36	256	2.3M	2.3M	1.00	1.12	1.49	1.07	0.90	1.07	0.95	0.58	0.95
kmeans-or	C	or,ua,uc	7–41	1–100	430K	428K	1.00	1.00	3.20	0.99	1.58	1.60	1.01	0.95	1.02
sha-or	M	or,uc	6–24	20–64	53K	51K	0.96	1.03	1.17	1.03	0.82	0.97	0.98	0.55	0.88
symm-or	Po	or	16	1–30	267K	268K	1.00	1.00	2.40	1.01	1.60	1.59	1.02	0.93	0.93
dynprog-om	Po	om	26	1–79	794K	795K	1.00	1.00	1.26	1.00	0.71	0.99	1.01	0.36	1.00
knn-om	P	om,uc	26–54	1–14	791K	750K	0.95	1.00	1.44	1.05	1.36	1.35	1.05	1.12	1.12
ksack-sm-om	C	om	21	99	50K	62K	1.23	0.77	2.72	0.56	1.71	1.64	0.36	1.05	1.03
ksack-lg-om	C	om	21	99	35K	39K	1.12	0.87	3.46	0.69	1.92	1.78	0.53	1.31	1.28
war-om	Po	om	21	32	438K	438K	1.00	1.00	1.09	1.00	0.63	0.99	1.00	0.60	0.99
mm-orm	P	orm,uc	7–22	256–2K	31K	31K	0.99	1.01	3.13	1.01	2.76	2.47	0.99	2.33	2.21
stencil-orm	Po	orm	20	126	639K	639K	1.00	1.00	1.02	1.00	0.66	1.00	1.00	0.66	1.00
btree-ua	C	ua,uc	11–14	1K	101K	101K	1.00	1.00	1.52	0.99	1.07	1.04	1.00	1.06	1.02
hsort-ua	C	ua	42–46	512–1K	274K	278K	1.01	0.99	1.34	0.96	0.88	1.00	1.10	0.71	1.13
huffman-ua	C	ua,uc	6–48	256–14K	290K	292K	1.01	0.96	1.57	0.97	1.09	1.18	0.99	0.74	0.96
rsort-ua	C	ua	12	1K	202K	218K	1.08	0.89	2.46	0.92	1.58	1.56	0.89	0.84	0.88
bfs-uc-db	C	uc.db	36	DYN	62K	64K	1.04	0.97	2.96	0.53	2.11	1.83	0.41	1.54	1.35
qsort-uc-db	C	uc.db	70	DYN	146K	136K	0.93	1.07	2.94	1.10	2.69	2.61	1.02	2.17	2.18

Suite shows the benchmark suites: Po = PolyBench; M = MiBench; P = PBBS; C = Custom. Loop characteristics shows: Type = the dependence pattern type (multiple entries means different `xloop`s); Num Insns = range for static instruction counts for each `xloop` body; Num Iters = range for number of `xloop` iterations; Dynamic Insns = dynamic instruction counts for the timing critical loop; GPI = general-purpose ISA; XLI = XLOOPS ISA; X/G = normalized XLOOPS ISA dynamic instruction count compared to general-purpose ISA; *io* = in-order speedups; *ooo/2* = 2-way out-of-order speedups; *ooo/4* = 4-way out-of-order speedups; T = traditional execution; S = specialized execution; A = adaptive execution. Speedups are normalized to a standard serial implementation compiled for the general-purpose ISA and executed on the corresponding baseline GPP. For example, the **io:T** column shows the speedup of an XLOOPS binary using traditional execution on an in-order GPP relative to a serial implementation of the application kernels compiled for the general-purpose ISA executing on the same in-order GPP.

Assignment of objects to clusters is a dominant loop with inter-iteration register dependences. *ksack-*-om* solves the unbounded knapsack dynamic programming problem. For this problem, we have two variants, *ksack-sm-om* and *ksack-lg-om*, which have datasets of small (< 10) and large (> 10) weights respectively. *btree-ua* constructs a binary tree from a random set of integer inputs. *hsort-ua* implements the heap-sort computation given a set of integer inputs. *huffman-ua* implements the Huffman entropy coding compression algorithm. *rsort-ua* performs an incremental radix sort on an array of integers. Each iteration updates histograms of digit lookups using a `xloop.ua` and computes prefix-sum updates for the next stage of sorting. *bfs-uc-db* uses a dynamically growing worklist to traverse an input graph in a breadth-first order and computes the distance given a source node to every other node. *qsort-uc-db* implements the quicksort algorithm using a dynamically growing worklist of partitions to be sorted.

We used LLVM-3.1 [24] for preprocessing, optimizing, and compiling, and GNU binutils for assembling and linking. We added a custom target machine backend for a 32-bit RISC ISA that does not support a branch delay-slot and uses a unified register file for integer and floating-point instructions. We implemented a preprocessing script to replace the `#pragma` annotations with external function calls to tag the parallel loops for analysis within LLVM, and modified the `LoopRotation` and `LoopStrengthReduction` passes to include register and memory dependence analysis to compile XLOOPS kernels.

IV. XLOOPS CYCLE-LEVEL EVALUATION

In this section, we describe our cycle-level modeling methodology and results comparing XLOOPS to three baseline GPPs: a simple single-issue in-order processor, a moderate two-way out-of-order superscalar processor, and an aggressive four-way out-of-order superscalar processor.

A. Cycle-Level Methodology

For our cycle-level studies, we modified the GPP models within the gem5 simulation framework [2], and we implemented a model of the LPSU using PyMTL, a Python-based hardware modeling framework [25]. Our changes to gem5 included: modifying the in-order and out-of-order GPP models to support AMOs and traditional execution; modifying the in-order and out-of-order GPP models to support co-

TABLE III. CYCLE-LEVEL SYSTEM CONFIGURATION

	io	ooo/2	ooo/4	Per lane
Issue Width	1	2	4	1
Phys Regs	32	64	128	24
Int ALU	1	2	4	1
AGU/Br Pred		2/1	2/1	
IQ Entries		16	32	
ROB Entries		48	96	
Ld/St Queue Entries		16/16	32/32	8/8
Inst Buffer Entries				128
Int Mul/Div Latency	4/10 cycles			
FP Mul/Div Latency	6/6 cycles			
FP Add/Sub Latency	4/4 cycles			
L1I$/L1D$/L2$/L3$	16KB/16KB/1MB/16MB			
Out-of-Order Features	Tournament Branch Pred Store-Set-Based Memory Dep Pred			

simulation with the PyMTL-based LPSU model; and implementing mechanisms to migrate loop execution between the GPP and LPSU models to support adaptive execution.

We used McPAT-1.0 to estimate the energy of the in-order and out-of-order GPPs in a 45 nm process technology [22]. The energy for the lanes in the LPSU was modeled by adapting McPAT's models for simple in-order GPPs. We configured McPAT to model properly sized instruction buffers in each lane. We included an additional energy overhead of 5% to model the energy of the LMU, index queues, and arbiters based on the results from our detailed VLSI implementation (see Section V). We conservatively accounted for the energy of `xi` instructions as 32-bit multiply operations, and accounted for the energy of inter-iteration register dependence communication with additional register-file read and write events. Lastly, we used the energy of an out-of-order load-store queue to conservatively model the energy of the LSQs in the LPSU.

Table III shows the configurations for the cycle-level models of the baseline GPPs and the LPSU lanes. We used three baseline GPPs: a single-issue in-order GPP (*io*), a two-way out-of-order superscalar GPP (*ooo/2*), and a four-way out-of-order superscalar GPP (*ooo/4*). These baseline designs enable us to quantitatively explore the performance and energy of XLOOPS compared to both simple, low-energy processors as well as complex, high-performance processors. We augmented each baseline GPP with an LPSU to create three XLOOPS configurations: *io+x*, *ooo/2+x*, and *ooo/4+x*. Each of these configurations supports traditional, specialized, and adaptive execution. Integrating the LPSU into all three baseline GPPs enabled understanding the subtle interactions between out-of-order and specialized execution (e.g., out-of-order scan phase, memory fences before and after specialized execution), and also enabled exploring adaptive execution in various contexts.

B. Traditional Execution

Table II shows the results for traditional execution of XLOOPS binaries. Each **T** column shows the speedup for each kernel compiled for the XLOOPS ISA using traditional execution on one of the GPPs relative to the kernel compiled for the general-purpose ISA executing on the same GPP. The goal for traditional execution is for this speedup to be as close to 1× as possible. In other words, for traditional execution, we simply wish to reduce the performance overhead of using the XLOOPS ISA compared to the general-purpose ISA when executing on traditional general-purpose microarchitectures. We observe that the performance overhead of traditional execution is minimal and is within 5% of the general-purpose ISA for all processors with the exception of *ksack-*-om* and *rsort-ua*. The dynamic instruction counts suggest that compiler optimizations could potentially close the gap for these kernels by reducing the number of extra instructions generated when using the XLOOPS ISA. In addition, we occasionally required additional AMOs in the XLOOPS binary compared to the general-purpose binary. Our current implementation of AMOs on the out-of-order GPPs is rather conservative, and this partly accounts for the discrepancy in traditional execution on these out-of-order GPPs (i.e., speedups <1 in **T** columns). These results are encouraging and make a case for gradual adoption of the XLOOPS abstraction in GPPs without significant overhead. In addition, efficient traditional execution will be a key enabler for adaptive execution.

C. Specialized Execution

Table II shows the results for specialized execution of XLOOPS binaries. Each **S** column shows the speedup for each kernel compiled for the XLOOPS ISA using specialized execution on a GPP+LPSU relative to the kernel compiled for the general-purpose ISA executing on the corresponding GPP. We observe that specialized execution always benefits the in-order processor. For a total of 25 application kernels, specialized execution performs better for 18 kernels compared to *ooo/2*, and performs better for 12 kernels compared to *ooo/4*.

Figure 5 summarizes the results comparing the baseline GPPs and the XLOOPS configurations. All speedups are normalized to each kernel compiled for the general-purpose ISA executing on *io*. The figure shows the speedup for each kernel compiled for the general-purpose ISA executing on *ooo/2* and *ooo/4*, and also shows the speedup for each kernel compiled for the XLOOPS ISA using specialized execution on *ooo/2+x*. Results for *io+x* and *ooo/4+x* are similar to *ooo/2+x* and are omitted for simplicity. Figure 6 shows the breakdown of stall and squash cycles for specialized execution.

Specialized execution for kernels dominated by `xloop.uc` shows speedups in the range of 2.7–4× compared to *io*. Performance of *sgemm-uc*, *war-uc*, and *symm-uc* are limited by intra-iteration RAW dependencies. *rgb2cmyk-uc* and *viterbi-uc* are constrained by stalls due to contention for the shared memory port. Figure 6 shows that sharing the LLFU does not significantly hurt the performance of any of the `xloop.uc` kernels. Sharing the LLFU drastically reduces the area overhead of XLOOPS (see Section V). Our results show that for `xloop.uc`, specialized execution is superior to *io* and complexity-effective compared to the more complicated out-of-order GPPs.

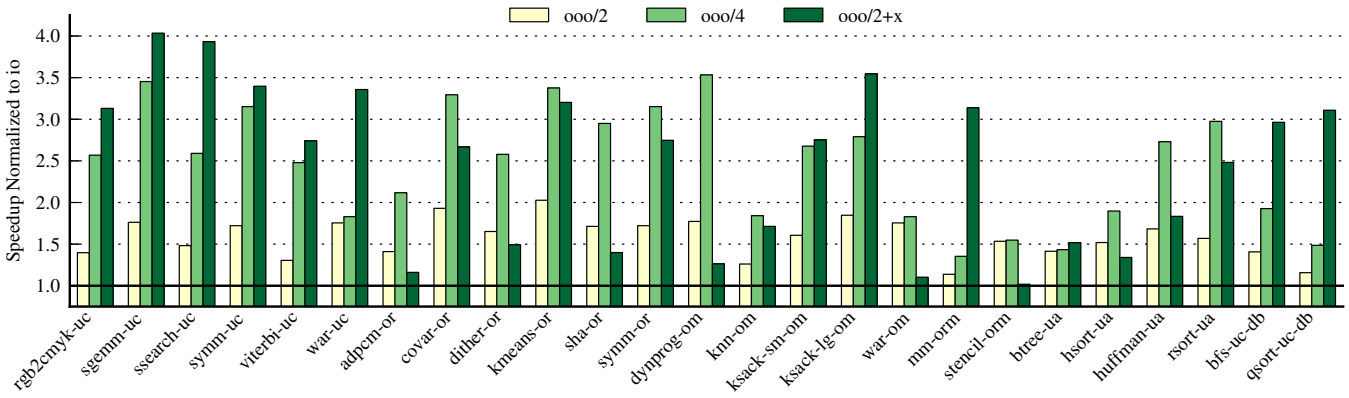

Figure 5. XLOOPS Cycle-Level Speedups – Each bar shows the speedup normalized to in-order (io) processor baseline kernels. ooo/2 = 2-way out-of-order processor; ooo/4 = 4-way out-of-order processor; ooo/2+x = ooo/2 augmented with LPSU.

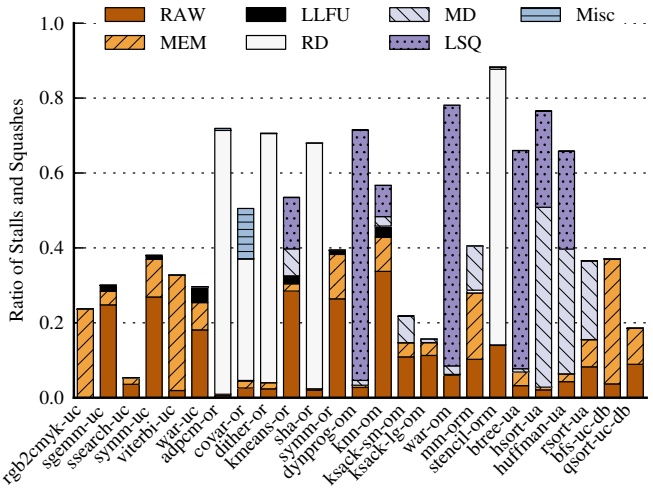

Figure 6. Stall and Squash Breakdown – Breakdown of average stall and squash cycles normalized to the number of cycles when the LPSU is active. RAW = read-after-write stalls; MEM = stalls due to data memory port access contention; LLFU = stalls due to LLFU access contention; RD = stalls due to inter-iteration register dependences; MD = squashes due to inter-iteration memory dependence violations; LSQ = stalls due to LSQ structural hazards; Misc = stalls due to write-after-write register-file port contention for LLFU operations and other structural hazards.

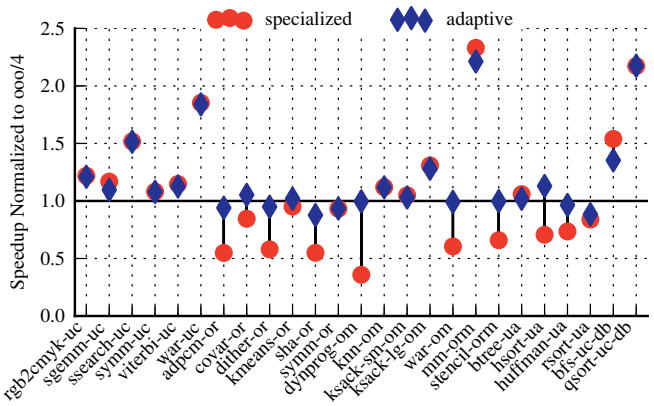

Figure 7. Adaptive Execution Speedups – Results for specialized execution and adaptive execution of kernels encoded with XLOOPS ISA on ooo/4+x relative to kernels encoded with general-purpose ISA on ooo/4.

Specialized execution for kernels dominated by `xloop.or` is usually limited by the inter-iteration critical path. For *kmeans-or* and *symm-or*, this critical path is a single instruction, resulting in improved performance compared to *ooo/2*. Most of the other `xloop.or` kernels have much longer inter-iteration register dependences. For these kernels, the out-of-order GPPs perform better than specialized execution due to their ability to exploit intra-iteration instruction-level parallelism. Future work could explore superscalar and out-of-order lane microarchitectures.

Specialized execution for kernels dominated by `xloop.{om,orm,ua}` is usually limited by LSQ structural hazards and squashing speculative work due to memory dependence violations. *btree-ua*, *dynprog-om*, *war-om*, *mm-orm*, and *knn-om* are all limited by LSQ structural hazards. *hsort-ua*, *huffman-ua*, and *rsort-ua* kernels are all limited by squashing speculative work. Even with these limitations, specialized execution is still competitive with *ooo/2* on many of these kernels and even out-perform *ooo/4* on *mm-orm* and *btree-ua*. Note that the number of squashes can depend on the dataset. For example, *ksack-sm-om* has an input dataset of small weights that results in nearby iterations accessing the same memory addresses. This increases the number of memory dependence violations. *ksack-lg-om* has an input dataset of large weights that results in fewer memory dependence violations. Static compiler analysis would have difficulty predicting these data-dependent performance results.

Specialized execution for kernels dominated by `xloop.uc.db` significantly out-perform both *ooo/2* and *ooo/4*. This is because the worklist-based implementation allows the LPSU to exploit significant inter-iteration instruction- and memory-level parallelism compared to the out-of-order processors. `xloop.uc.db` kernels illustrate the potential for encoding more sophisticated inter-iteration dependence patterns in the instruction set.

D. Adaptive Execution

Adaptive execution bridges the performance gap between aggressive out-of-order GPPs and specialized execution. Figure 7 shows the results comparing the performance of spe-

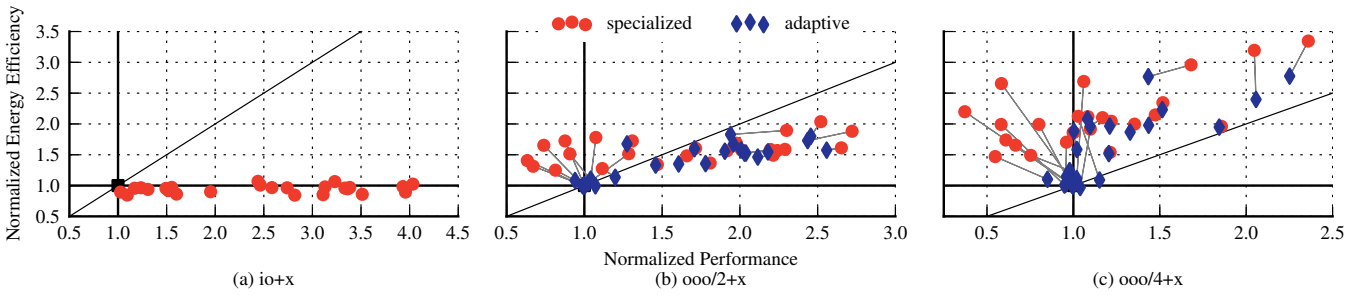

Figure 8. Energy Efficiency vs. Performance – Cycle-level performance and McPAT energy results of specialized and adaptive execution for (a) *io+x* normalized to *io*, (b) *ooo/2+x* normalized to *ooo/2*, (c) *ooo/4+x* normalized to *ooo/4*. Diagonal lines are iso-power contours.

cialized and adaptive execution on *ooo/4+x*. Based on preliminary experiments, we use 256 iterations and 2000 cycles as thresholds for the profiling phases. For kernels where traditional execution performs better than specialized execution, adaptive execution is able to automatically choose to migrate the execution from the LPSU back to the GPP. For kernels where specialized execution performs better than traditional execution, our results show that the overhead of profiling and work migration cause only minimal performance degradation. Table II also includes results for adaptive execution with *ooo/2+x*. Adaptive execution makes a compelling case for the flexibility provided by the elegant XLOOPS hardware/software abstraction.

E. Energy Efficiency vs. Performance

Figure 8 shows the dynamic energy efficiency and performance for specialized and adaptive execution on *io+x*, *ooo/2+x*, and *ooo/4+x*. The *io+x* results are normalized to kernels compiled for the general-purpose ISA executing on *io*, the *ooo/2+x* results are normalized to kernels compiled for the general-purpose ISA executing on *ooo/2*, and so on. The diagonal lines represent iso-power contours. Specialized execution adds minimal energy overhead and results in increased performance compared to *io* across all applications. For *ooo/2+x* and *ooo/4+x*, specialized execution is more energy efficient across all applications. The performance trends are as explained in Sections IV-C and IV-D. Specialized execution on *io+x* consumes more dynamic power for all applications, while specialized execution on *ooo/2+x* is power-efficient for 10 applications. Compared to *ooo/4*, specialized execution is not only more energy efficient but also consumes less power. Figure 8(b,c) shows that the performance benefit of adaptive execution comes at the cost of reduced energy efficiency. Overall, the results suggest that a combination of specialized and adaptive execution offers a complexity-effective design point compared to more traditional GPPs.

F. Microarchitectural Design Space Exploration

The XLOOPS hardware/software abstraction enables a rich microarchitectural design space with a variety of different potential microarchitectural optimizations. In this section, we explore some of this design space. We evaluate these features using select kernels that are representative of various inter-iteration dependence patterns.

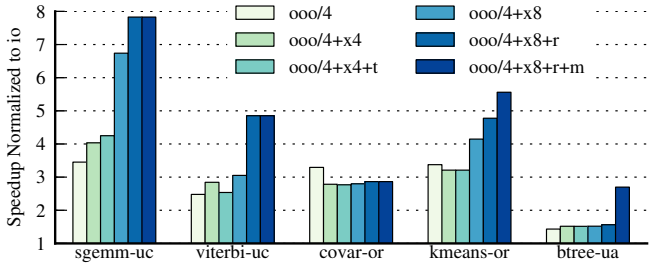

Figure 9. Microarchitectural Design Space Exploration Speedups – Normalized to running baseline kernel on *io*. ooo/4 = 4-way out-of-order processor; x4/x8 = LPSU design with 4 and 8 lanes, respectively; r = additional LLFUs and data memory ports; t = 2-way multithreading; m = additional LSQs.

We first consider adding limited vertical multi-threading to the lanes. Application kernels such as *sgemm-uc* that are limited by read-after-write stalls benefit from two-way multi-threading (see *ooo/4+x4+t* in Figure 9). However, multi-threading does not benefit all kernels, and we see reduced performance for *viterbi-uc* due to increased contention for the shared memory ports. We disable multi-threading for `xloop.{or,om,orm}` as it slows the execution of the inter-iteration critical path and/or the non-speculative lane.

Doubling the number of lanes to eight (*ooo/4+x8*) improves the performance for *sgemm-uc* by 68% and *kmeans-or* by 28% as neither of these applications are limited by the shared LLFU and memory port. *viterbi-uc* only sees moderate improvement as it is limited by memory port contention. Kernels limited by inter-iteration critical paths (e.g., *covar-or*) or by LSQ structural hazards (e.g., *btree-ua*) do not benefit from increased lanes.

We also consider doubling the shared LLFUs and memory ports (*ooo/4+x8+r*). This improves the performance of *viterbi-uc* by reducing memory port contention and the performance of *sgemm-uc* by reducing LLFU contention. *kmeans-or* benefits from both increased memory and LLFU resources. Finally, we explore increasing the size of the LSQs to 16+16 entries (*ooo/4+x8+r+m*) and find that the performance of *btree-ua* improves by 80%. None of the improvements in the final aggressive LPSU design reduce stalls due to inter-iteration register dependence, so we see no significant improvement in the performance of *covar-or*.

Overall, our final highly optimized LPSU design is able to significantly increase performance compared to the baseline

TABLE IV. CASE STUDY RESULTS

Name	Loop Type	io+x	ooo/2+x	ooo/4+x
adpcm-or-opt	or	1.86	1.32	0.88
dither-or-opt	or	2.48	1.51	0.97
sha-or-opt	or	1.55	1.13	0.82
bfs-uc	uc	2.73	1.96	1.50
dither-uc	uc	2.49	1.54	1.00
kmeans-uc	uc	3.60	1.79	1.08
qsort-uc	uc	2.35	2.15	1.62
rsort-uc	uc	1.85	1.23	0.68

Speedups normalized to kernel compiled for general-purpose ISA.

LPSU design, but of course these optimizations also increase area and design complexity.

G. Application Case Studies

In this section we consider hand-optimized `xloop.or` kernels and manual loop transformations. Results are summarized in Table IV.

Hand-Optimized `xloop.or` – We observed that out-of-order GPPs perform better than the LPSU designs for several `xloop.or` kernels because of their ability to extract ILP when the LPSU lanes stall due to inter-iteration register dependences. We compare the benefits of reducing the cross-iteration critical path for each CIR, by hand-scheduling compiler generated code, for a few select kernels. Table IV shows that *adpcm-or-opt*, *dither-or-opt*, and *sha-or-opt* boost the performance of specialized execution by 50–70%. With these scheduling optimizations, specialized execution of `xloop.or` kernels is competitive with *ooo/2* and *ooo/4*. Future work can improve the XLOOPS compiler to schedule instructions more optimally.

Loop Transformations – We explored alternative loop parallelization strategies including: general parallel programming techniques such as privatize-and-reduce; using split worklists as opposed to unified worklists; and atomic data-structure updates to parallelize loops with register and memory dependences. Our results from Tables II and IV suggest that transforming `xloop.or` and `xloop.om` into `xloop.uc` does not always result in improved performance. Kernels such as *bfs-uc*, *kmeans-or*, *rsort-ua*, and *qsort-uc* outperform their loop transformed counterparts. Only *dither-uc* benefits from these transformations. Because simply annotating serial versions of the kernels often performs better than code with significant transformations, XLOOPS allows ease-of-programmability without sacrificing performance.

V. VLSI EVALUATION

In this section we present a register-transfer-level (RTL) model for a basic LPSU which supports `xloop.uc` instructions. We synthesize and place-and-route this implementation using a commercial ASIC CAD toolflow and present results for area, energy, and timing.

A. VLSI Methodology

Our RTL baseline design is a five-stage in-order GPP that executes 32-bit RISC instructions. The GPP uses a 16KB instruction and a 16KB data cache. We implemented a variety of detailed cycle-accurate LPSU configurations capable of supporting `xloop.uc` using parameterized Verilog RTL models to evaluate the area, energy, and timing. Note that our current RTL implementation does not support `xi` instructions. To compile the applications, we modified the `LoopStrengthReduction` pass in LLVM to disable the generation of `xi` instructions.

We target a 40 nm TSMC process using a Synopsys ASIC CAD toolflow: VCS for RTL simulation, DesignCompiler for synthesis, IC Compiler for place-and-route, and PrimeTime for power analysis. We did not have access to a memory compiler for our target process, so we model cache tag/data SRAMs and the LPSU instruction buffer SRAM using CACTI [26]. The datasets were tailored to fit in the L1 cache.

TABLE V. VLSI AREA, CYCLE TIME RESULTS FOR LPSU

				Percentage Area Breakdown								
Name	CT	AA	AO	SP	I$	D$	MD	FP	IB	LN	IQ	MI
scalar	1.95	0.25		8	33	37	11	10				
lpsu+i096+ln4	2.16	0.35	42	6	24	26	9	8	6	19	~0	2
lpsu+i128+ln4	2.14	0.36	44	5	23	26	9	8	6	19	~0	2
lpsu+i160+ln4	2.12	0.36	45	5	23	26	9	8	6	20	~0	2
lpsu+i192+ln4	2.20	0.37	48	5	23	25	8	8	10	18	~0	2
lpsu+i128+ln2	1.98	0.31	24	6	27	30	10	9	4	12	~0	1
lpsu+i128+ln6	2.28	0.41	65	5	20	23	8	7	8	26	1	2
lpsu+i128+ln8	2.54	0.44	77	4	19	20	8	7	10	29	1	2

CT = cycle time in nanoseconds; AA = absolute area in mm^2; AO = percent area overhead compared to scalar baseline; SP = scalar processor; I$ = instruction cache; D$ = data cache; MD = integer multiply-divide unit; FP = floating-point unit; IB = LPSU instruction buffers; LN = LPSU lanes; IQ = index queues; MI = arbiters for data-memory and LLFUs, and other miscellanous control logic; Percents rounded to nearest tens

B. VLSI Area Results

Table V presents area results based on post-place-and-route area estimates. We compare the area of the baseline GPP and the LPSU designs with four lanes by varying the capacity of instruction buffer (96–192 entries) and by varying the number of lanes (2–8) with a fixed instruction buffer size of 128 entries. Each configuration name begins with `lpsu` and a suffix with `i` to denote the instruction buffer size and `ln` to denote the number of lanes.

Total area of the primary LPSU design (lpsu+i128+ln4) is 0.36 mm^2 which is only 43% larger than the GPP (0.25 mm^2). Sharing the LLFU and memory port is a key design decision that results in minimal area overheads. Varying the instruction buffer size (96–192) with four lanes shows that area overheads range between 41–48% compared to GPP suggesting that larger instruction buffers are reasonable. Varying the number of lanes (2–8) for a fixed instruction buffer size of 128 shows that area overheads range between 24–77%. These results confirm that the area overhead of a given LPSU design increases roughly linearly with the number of lanes.

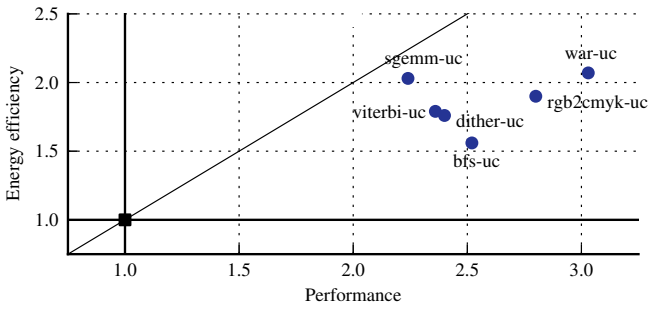

Figure 10. VLSI Energy Efficiency vs. Performance – Compares the in-order GPP augmented with the LPSU normalized to baseline the in-order processor. Diagonal line is the iso-power contour.

C. VLSI Energy Efficiency vs. Performance Results

Figure 10 compares the energy efficiency and performance of specialized execution relative to the kernels compiled for the general-purpose ISA executing on the GPP. Specialized execution improves performance by 2.4–4× (*ssearch-uc* gets a speedup of 4×; not shown in Figure 10). Specialized execution on the LPSU consumes more power compared to the GPP since the LPSU executes several instructions in parallel. Performance results roughly correspond to the trends seen using our cycle-level models (see Section IV). *sgemm-uc* has worse performance compared to the cycle-level evaluation due to an increase in dynamic instruction count caused by the lack of `xi` instructions. Our ASIC CAD toolflow reports that the energy for an access to an LPSU instruction buffer is cheaper by a factor of ten compared to an access to the instruction cache. Since most of the application time is spent in executing `xloops`, LPSU designs result in significant energy savings due to reduced instruction cache accesses. Improvements in energy efficiency are in the range of 1.6–2.1×. These results suggest that our McPAT results are relatively conservative.

VI. Related Work

Most of the previous work on loop-level specialization including data parallel accelerators (DPAs), speculative parallelization, hardware task scheduling, transactional memory, and accelerators, tightly couple the abstraction and microarchitecture. XLOOPS is an elegant approach that unifies many of these proposals with a novel abstraction that can be mapped on to traditional, specialized, and adaptive microarchitectures.

ASIPs integrate specialized circuits into a traditional processor pipeline which benefits a specific domain of applications and are limited in generality [6]. Architectures such as CCA [4], DySER [11], and BERET [13] provide reconfigurable datapaths to accelerate critical subgraphs of computation within a loop iteration. Our current work focuses more on inter-iteration loop dependence patterns.

`xloop.uc` – Many commercial DSPs [5, 32] support zero-overhead loops in the form of a special loop or repeat instruction. These architectures allow the execution of loops with no ordering constraints and require no hardware support for control speculation. DPAs are examples of architectures that exploit inter-iteration data-level parallelism. Streaming SIMD extensions, Advanced Vector Extensions (AVX), and vector ISA extensions [8, 17, 34] amortize the overheads of instruction processing and increase performance by executing parallel operations. These architectures suffer when executing code with intra-iteration control-flow, loop-carried register-dependences, and divergent memory accesses [12]. Furthermore, they rely heavily on vectorizing compilers which is an active area of research. Mainstream GPUs [23, 35] and Maven [21] alleviate the problems of traditional vector processors but require more radical changes across ISA, compiler and microarchitecture compared to XLOOPS.

`xloop.ua` – Transactional memory (TM) systems [15] coordinate the execution of parallel computations by committing non-conflicting memory updates. In [36], authors modify traditional architectures to include a hardware TM system and expose transactions to software through instruction-set extensions to exploit loop-level parallelism. Our XLOOPS abstraction allows for a variety of microarchitectures that can take advantage of the `xloop.ua` data-dependence pattern.

`xloop.or` – Multiscalar [30], vector-like proposals [16, 18], and others [19,36] propose register bypass networks similar to the CIBs to handle inter-iteration register dependences. HELIX-RC [3] proposes a ring-cache architecture to communicate register dependences. XLOOPS is potentially more elegant as it avoids requiring ISA extensions to specify the dependence communication unlike previous proposals.

`xloop.om` – Multiscalar and TLS proposals [19, 31] are speculative parallelization techniques that provide hardware memory-dependence speculation to exploit loop-level parallelism. XLOOPS proposes per-lane LSQs and a store broadcast network to support memory-dependence speculation in hardware. Previous speculative parallelization techniques show promise but demand dramatic changes in the microarchitecture, compiler, and/or ISA. HELIX-RC [3] takes an alternative approach of decoupling memory dependence communication without employing speculation but relies on an aggressive parallelizing compiler. The XLOOPS ISA could be extended to include instructions for lane synchronization to benefit compiler optimizations as in HELIX-RC.

`xloop.*.db` – Carbon [20] and Asynchronous Data Messages (ADM) [28] are two proposals that exploit fine-grain loop-level parallelism through hardware-only and hybrid hardware-software work distribution queues. The XLOOPS dynamic-bound construct is similar in spirit by allowing mapping loops with dynamic work generation.

VII. Conclusions

In this paper, we have introduced XLOOPS, a new hardware specialization approach for exploiting inter-iteration loop dependence patterns. The XLOOPS instruction set provides an elegant hardware/software abstraction that serves as an effective compiler target and enables a variety of microarchitectures supporting traditional, specialized, and adaptive execution. We have used a vertically integrated evaluation methodology spanning applications, compilers, cycle-level modeling, RTL modeling, and VLSI implementation to make a compelling case for augmenting both in-order and

out-of-order general-purpose processors with a loop-pattern specialization unit. Future work might explore additional inter-iteration loop dependence patterns (e.g., data-dependent exit conditions) and microarchitectures (e.g., more directly exploiting nested loop patterns).

Acknowledgments

This work was supported in part by NSF CAREER Award #1149464, NSF XPS Award #1337240, a DARPA Young Faculty Award, and donations from Intel Corporation, Synopsys, Inc, and Xilinx, Inc. The authors acknowledge and thank Pol Rosello and Paul Jackson for their help in writing XLOOPS application kernels, Christopher Torng for his help bringing up the gem5 simulation framework, Aadeetya Shreedhar for his help in bringing up traditional execution on gem5, and Derek Lockhart for developing the Python hardware modeling framework used in the cycle-level model of the LPSU.

References

[1] K. Atasu, L. Pozzi, and P. Ienne. Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints. *Design Automation Conf.*, Jun 2003.

[2] N. Binkert et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, May 2011.

[3] S. Campanoni et al. HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs. *Int'l Symp. on Computer Architecture*, Jun 2014.

[4] N. Clark et al. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. *Int'l Symp. on Microarchitecture*, Dec 2004.

[5] L. Codrescu et al. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *IEEE Micro*, 34(2):34–43, Mar/Apr 2014.

[6] J. Cong et al. Application-Specific Instruction Generation for Configurable Processor Architectures. *Int'l Symp. on Field Programmable Gate Arrays*, Feb 2004.

[7] W. J. Dally et al. Efficient Embedded Computing. *IEEE Computer*, 47(7):27–32, Jul 2008.

[8] R. Espasa, M. Valero, and J. E. Smith. Vector Architectures: Past, Present, and Future. *Int'l Symp. on Supercomputing*, Jul 1998.

[9] G. Goff, K. Kennedey, and C.-W. Tseng. Practical Dependence Testing. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Jun 1991.

[10] V. Govindaraju et al. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro*, 32(5):38–51, Sep/Oct 2012.

[11] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy-Efficient Computing. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2011.

[12] V. Govindaraju, T. Nowatzki, and K. Sankaralingam. Breaking SIMD Shackles with an Exposed Flexible Microarchitecture and the Access Execute PDG. *Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sep 2013.

[13] S. Gupta et al. Bundled Execution of Recurring Traces for Energy-efficient General Purpose Processing. *Int'l Symp. on Microarchitecture*, Dec 2011.

[14] M. R. Guthaus et al. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *IEEE Annual Workshop on Workload Characterization*, Dec 2001.

[15] T. Harris, J. Larus, and R. Rajwar. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[16] C. Jesshope. Implementing an Efficient Vector Instruction Set in a Chip Multiprocessor Using Micro-Threaded Pipelines. *Australia Computer Science Communications*, 23(4):80–88, 2001.

[17] C. Kozyrakis and D. Patterson. Scalable Vector Processors for Embedded Systems. *IEEE Micro*, 23(6):36–45, Nov 2003.

[18] R. Krashinsky et al. The Vector-Thread Architecture. *Int'l Symp. on Computer Architecture*, Jun 2004.

[19] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Computer*, 48(9):866–880, Sep 1999.

[20] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. *Int'l Symp. on Computer Architecture*, Jun 2007.

[21] Y. Lee et al. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerator Cores. *Int'l Symp. on Computer Architecture*, Jun 2011.

[22] S. Li et al. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. *Int'l Symp. on Microarchitecture*, Dec 2009.

[23] E. Lindholm et al. NVIDIA Tesla: A Unified Graphics and Computer Architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.

[24] The LLVM Compiler Infrastructure Project. Online Webpage, 2011 (accessed February, 2011). `http://www.llvm.org`.

[25] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture*, Dec 2014.

[26] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. *HP Technical Report HPL-2009-85*, 2009.

[27] Polyhedral Benchmark Suite. Online Webpage, 2014 (accessed May, 2014). `http://www.cse.ohio-state.edu/~pouchet/software/polybench`.

[28] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar 2010.

[29] J. Shun et al. Brief Announcement: The Problem Based Benchmark Suite. *Symp. on Parallel Algorithms and Architectures*, Jun 2012.

[30] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. *Int'l Symp. on Computer Architecture*, Jun 1995.

[31] J. G. Steffan et al. A Scalable Approach to Thread-Level Speculation. *Int'l Symp. on Computer Architecture*, May 2000.

[32] TMS320C28x Floating Point Unit and Instruction Set. Reference Guide, 2008. `http://www.ti.com/lit/ug/sprueo2a/sprueo2a.pdf`.

[33] G. Venkatesh et al. QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores. *Int'l Symp. on Microarchitecture*, 2011.

[34] J. Wawrzynek et al. Spert-II: A Vector Microprocessor System. *IEEE Computer*, 29(3):79–86, Mar 1996.

[35] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31(2):50–59, Mar/Apr 2011.

[36] H. Zhong et al. Uncovering Hidden Loop Level Parallelism in Sequential Applications. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2008.