The Load Slice Core Microarchitecture

Trevor E. Carlson[1] Wim Heirman[2] Osman Allam[3] Stefanos Kaxiras[1] Lieven Eeckhout[3]
[1] Uppsala University, Sweden [2] Intel, ExaScience Lab [3] Ghent University, Belgium

Abstract

Driven by the motivation to expose instruction-level parallelism (ILP), microprocessor cores have evolved from simple, in-order pipelines into complex, superscalar out-of-order designs. By extracting ILP, these processors also enable parallel cache and memory operations as a useful side-effect. Today, however, the growing off-chip memory wall and complex cache hierarchies of many-core processors make cache and memory accesses ever more costly. This increases the importance of extracting memory hierarchy parallelism (MHP), while reducing the net impact of more general, yet complex and power-hungry ILP-extraction techniques. In addition, for multi-core processors operating in power- and energy-constrained environments, energy-efficiency has largely replaced single-thread performance as the primary concern.

Based on this observation, we propose a core microarchitecture that is aimed squarely at generating parallel accesses to the memory hierarchy while maximizing energy efficiency. The Load Slice Core extends the efficient in-order, stall-on-use core with a second in-order pipeline that enables memory accesses and address-generating instructions to bypass stalled instructions in the main pipeline. Backward program slices containing address-generating instructions leading up to loads and stores are extracted automatically by the hardware, using a novel iterative algorithm that requires no software support or recompilation. On average, the Load Slice Core improves performance over a baseline in-order processor by 53% with overheads of only 15% in area and 22% in power, leading to an increase in energy efficiency (MIPS/Watt) over in-order and out-of-order designs by 43% and over 4.7×, respectively. In addition, for a power- and area-constrained many-core design, the Load Slice Core outperforms both in-order and out-of-order designs, achieving a 53% and 95% higher performance, respectively, thus providing an alternative direction for future many-core processors.

ISCA '15, June 13-17, 2015, Portland, OR, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3402-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750407

1. Introduction

Processor cores have evolved from simple in-order designs to complex superscalar out-of-order machines. The original design goal was to expose instruction-level parallelism (ILP) in an effort to keep all available execution units occupied and hence improve single-threaded application performance [36, 37]. In more recent years, the discrepancy in processor performance with that of main memory resulted in a phenomenon typically referred to as the memory wall [41]. Additionally, saturating ILP extraction and power limitations led to the emergence of multi- and many-core processors. This move changed the focus from single-thread performance to energy-efficient cores that maximize total chip performance within the power budget [28]. Many-core chips also resulted in increasingly complex on-chip memory hierarchies, driving up memory latency even further through large on-chip networks and the need for coherency transactions. Hiding this memory latency, therefore, has become an ever more important task of the core. Since ILP-extracting techniques automatically expose *memory hierarchy parallelism (MHP)* as well, out-of-order cores are naturally successful in coping with this problem [10]. (Similar to how MLP is defined for off-chip accesses by Chou et al. [10], we define memory hierarchy parallelism, MHP, from the core's viewpoint as the average number of overlapping memory accesses that hit anywhere in the cache hierarchy.) However, this success comes at a large cost in design complexity, chip power and area budget. Driven by the increased focus on energy efficiency, many contemporary many-core machines have reverted back to simpler, in-order cores which are much more energy-efficient but are limited in the amount of ILP and MHP they can extract (e.g., Intel Xeon Phi [11], Tilera [29]).

The ideal many-core building block is therefore an energy-efficient core that can still maximize extraction of memory hierarchy parallelism, a combination that neither traditional in-order or out-of-order processors provide. Other techniques exist to prevent the processor from stalling on pending long-latency loads. Examples include runahead execution to discover and prefetch independent memory accesses [15, 25], and slice processors that — statically or dynamically — extract independent program slices that can be executed out-of-order with respect to the blocked instruction flow. Slice processors make the concept of extracting independent instruction slices, which are clusters of dynamically executed (not necessarily

This work was completed while Trevor E. Carlson and Wim Heirman were at Ghent University.

contiguous) instructions, explicit. One category of related work specifies slices in software [5, 13, 14, 20, 30, 33, 38, 42]. Others identify slices in hardware, either for speculation and re-execution [4, 8, 15, 24, 25], or they cache these slices for later re-execution, thus avoiding the overhead of continuously re-building them [9, 12, 18, 26]. Yet, each of these proposals result in compromises in one or more areas: they add additional complex hardware structures, require the recompilation or modification of existing software, or rely on re-execution of part of the instruction stream — wasting both time and energy. (See Section 7 for a more detailed discussion on related work.)

To address these limitations, we reconsider the design of the processor core. We propose the *Load Slice Core microarchitecture* which is a restricted out-of-order machine aimed squarely at extracting parallelism from the memory hierarchy. By restricting slices to memory operations, we do not try to solve the problem of finding ILP in all forms, but rather focus specifically on the memory hierarchy. Backward slices are constructed, stored and finally recalled in an energy-efficient, hardware-based manner. The Load Slice Core builds on the commonly used superscalar in-order core with a stall-on-use policy. Memory hierarchy parallelism is achieved by executing select instructions out-of-order with respect to the main instruction flow. By placing strict limitations on which instructions can bypass others, and favoring simpler structures (RAMs and FIFOs) over complex ones (CAMs), the microarchitecture can be kept small and efficient, resulting in good power- and area-efficiency while still performing close to fully out-of-order designs. Conceptually, the Load Slice Core is most closely related to the decoupled access/execute architecture (DAE) [33], which provides separate pipelines for memory accesses (including address generation) and computation. The Load Slice Core microarchitecture follows a similar split with two in-order pipelines: a primary pipeline for the instruction stream, and a secondary pipeline that handles loads and address calculations. In contrast to DAE, the Load Slice Core automatically detects address-generating instructions in hardware using a novel, iterative technique, and can therefore be used in combination with unmodified application binaries.

In this work, we present and evaluate the Load Slice Core microarchitecture, making the following contributions:

- We propose *iterative backward dependency analysis*, a low-cost, hardware-based technique to select backward instruction slices from load and store instructions for early execution. This technique iteratively learns the address generating instructions that lead up to memory accesses during application loops that occur naturally in software.
- We propose the Load Slice Core microarchitecture, a restricted out-of-order, decoupled access/execute-style microarchitecture. Scheduling decisions are made early in the front-end of the pipeline, through iterative backward dependency analysis, without the need to modify application binaries. This technique avoids using expensive wake-up and selection logic in the back-end of the processor, resulting in a small amount of additional hardware over a typical stall-on-use in-order processor: just 15% area overhead compared to the ARM Cortex-A7.
- We provide a comprehensive analysis of the Load Slice Core's performance, area and power consumption, and compare it to standard in-order and out-of-order alternatives. The Load Slice Core is 43% and 4.7× more energy-efficient than in-order and out-of-order cores, respectively. In addition, we demonstrate that the Load Slice Core, when used in a power- and area-limited many-core design, outperforms both in-order as well as out-of-order designs, by 53% and 95%, respectively.

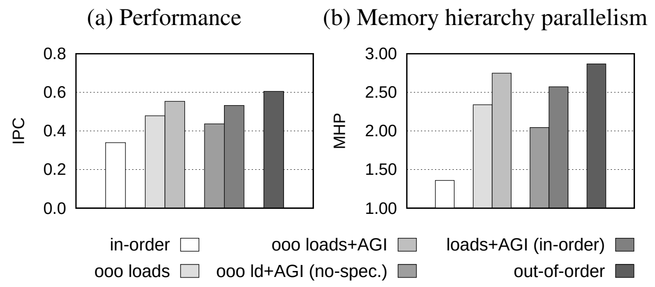

Figure 1: Selective out-of-order execution performance (left) and memory hierarchy parallelism extraction (right).

2. Motivation

To characterize the effect of ILP and MHP on performance, we set up the following experiment. We start from a simulation model for an out-of-order, two-wide superscalar core, and explore a number of different instruction issue rules and assess their effect on performance. We use the SPEC CPU 2006 benchmarks, and a modern memory subsystem including prefetchers; see Section 5 for further methodological details.

Traditional architectures. Figure 1 plots the performance (in instructions per clock cycle, IPC) and capability of extracting memory hierarchy parallelism (MHP, in average overlapping core memory accesses). All architectures are based on a two-wide superscalar pipeline with a 32-entry instruction window. The *out-of-order* variant (right-most bar) schedules instructions out-of-order, and can execute any two instructions provided their operands are ready. (We assume this variant has a perfect bypass network and disambiguation of load and store addresses). The *in-order* variant (left-most bar) can only issue instructions that are at the head of the instruction window, making this effectively an in-order, stall-on-use core. As can be expected, the performance of the out-of-order core, averaged over all SPEC CPU workloads, is almost double that of the in-order design.

Out-of-order loads. To quantify the effect of variations on a hypothetical core that improves memory hierarchy parallelism, we extend the in-order core and allow it to execute select instructions out-of-order. The *out-of-order loads* variant can execute loads once they are ready, i.e., when all operands needed to compute the address are available and no conflicts exist with earlier pending stores (this includes speculation beyond unresolved branches). As in the in-order variant, non-load instructions execute in program order only. In all cases, the maximum number of instructions (of any type) to execute in a given cycle is restricted to two. The *out-of-order loads* architecture exposes additional MHP over the in-order machine, but has the same ILP extraction. Yet, performance improves over in-order execution, as this architecture is able to issue loads earlier. This reduces stall time for instructions that consume the result of the load, and more importantly, enables loads to bypass instructions that block the head of the window while waiting for previous loads to complete, hence allowing more loads to be issued in parallel.

Address-generating instructions. Still, less MHP is extracted than in a fully out-of-order core since load addresses can depend on non-load instructions, which are still executed in-order. The second variant, *ooo loads+AGI*, also enables address-generating instructions to be executed early (as soon as their operands are available). We define an *address-generating instruction (AGI)* as any instruction still in the instruction window for which a dependency chain exists from that instruction to the load address (potentially across control flow). The *ooo loads+AGI* architecture is assumed to have perfect knowledge of which instructions are needed to calculate future load addresses, and enables all of them to execute out-of-order. This in turn generates load addresses earlier, enabling more loads to be executed out-of-order as well, further improving performance up to a level that approaches fully out-of-order execution.

Speculation. The importance of speculating across control flow is illustrated by the *ooo ld+AGI (no-spec.)* variant, which executes both loads and AGIs out-of-order, but not beyond unresolved branches. Its performance is significantly lower than even the variant that considers only loads for early execution (but still enables speculation), showing that a large fraction of the performance improvement observed is in fact because loads can be executed speculatively. An architecture that wants to expose memory hierarchy parallelism therefore has to be able to speculate, and contain provisions for recovery from mispredicted branches.

In-order scheduling. While an implementation of our hypothetical *ooo loads+AGI* architecture would probably be almost as complex as a fully out-of-order design, it turns out that we can make one crucial simplification: we will execute loads and AGIs in-order with respect to each other, but out-of-order with respect to the main instruction sequence. Such a scheduling policy can be implemented efficiently by using two in-order queues, one *bypass queue* for loads and AGIs, and a *main queue* for all other instructions. The performance of this design is shown in Figure 1 as the *ooo ld+AGI (in-order)* variant, and is 53% better than an in-order core and within 11% of a core with full out-of-order execution.

Finally, detecting AGIs can be done in hardware very efficiently if we rely on loops, which naturally occur in software, to build the list of AGIs *iteratively*. Rather than trying to assemble the complete dependency chain for load addresses in a single execution pass, we find one producer at a time, and mark instructions as address-generating one backward step per loop iteration. Only those instructions that were already marked as AGI in a previous iteration are issued to the bypass queue, greatly simplifying the logic needed to dispatch instructions to the right queue.

Key insights. The design of the Load Slice Core microarchitecture relies on three key insights. First, extending an efficient, in-order stall-on-use processor with the ability to execute both loads and address-generating instructions out-of-order, allows these loads to bypass older instructions that are blocked waiting for memory. This exposes additional memory hierarchy parallelism, and can lead to a performance level that is close to full out-of-order scheduling. Second, even though loads and AGIs execute out-of-order with respect to the main instruction flow, they do not need to execute out-of-order with respect to each other. Instruction scheduling can therefore be implemented using two in-order queues, of which only the heads are considered for execution, rather than requiring complex wake-up and selection logic present in fully out-of-order designs. Third, detection of address-generating instructions can be done iteratively, one backwards step at a time, using loop behavior present in applications. This allows AGI detection to be implemented fully in the processor front-end.

To make a working design, two more elements are required. Through-memory dependencies occur when a load overlaps with earlier store instructions. To be able to detect these dependencies, we split store instructions in two parts, one part that calculates the address and another part to collect the data and update memory.[1] The address part of the store uses the bypass queue, while the data part executes from the main queue. This way, stores with an unresolved address automatically block future loads (due to the bypass queue being in-order), while loads that do execute can check their address against that of all pending stores in the store queue and block when required to honor read-after-write dependencies. Finally, by tracking completion in a scoreboard, precise exceptions can be supported just as in stall-on-use in-order processors. In the following sections, we will describe this architecture in more detail.

3. Iterative Backward Dependency Analysis

The Load Slice Core extracts memory hierarchy parallelism by selecting critical instruction slices for early execution. These slices end at a load or store instruction and contain all instructions needed to generate the memory address. Backward dependency analysis [43] is one way to identify candidate slices, with techniques available both for software [14] and

[1]Most out-of-order processors that perform micro-operation cracking do in fact already have separate store-address and store-data micro-ops.

out-of-order hardware [12, 24]. Yet, the Load Slice Core does not need to explicitly generate collections of critical instruction slices; all that is required is to know whether individual instructions are part of an address-generating slice or not.

The goal of *iterative* backward dependency analysis (IBDA) is to identify address-generating instructions in a low-cost, hardware-friendly way. IBDA avoids analyzing long traces of executed or committed instructions in large hardware structures. Instead, by making use of a program's natural loop behavior, we can find the complete backward slice one instruction at a time, in subsequent loop iterations. Each instruction that is found is marked as being part of an address-generating slice, and *its* producers can be marked one loop iteration later.

IBDA is implemented in the processor front-end, inspecting instructions as they are being dispatched into the back-end, and requires two new hardware structures. The *instruction slice table* (IST) contains the addresses of instructions that have been identified as belonging to a backward slice, and is initially empty. By using the data from the IST at dispatch, the Load Slice Core can determine whether this instruction has been marked for bypass: instructions present in the IST are inserted into the bypass queue, while other instructions use the main queue instead. The second component of IBDA is the *register dependency table* (RDT). The RDT contains an entry for each physical register, and maps it to the instruction pointer that last wrote to this register. Starting with a memory access, the RDT will be used to look up the previous instruction(s) that produced the registers necessary for address calculation. These instructions are considered *address-generating instructions* (AGIs), and their instruction addresses are recorded in the IST. For each following loop iteration, the hardware will propagate this information one additional level backwards, by looking up producers for known address-generating instructions and recording them in the IST as well.

The complete IBDA procedure operates on an in-order instruction stream. Since the actual (dynamic) instruction that produced the value has already advanced further through the processor pipeline (or may even have been committed), it is not affected and will not be moved to the bypass queue retroactively. Instead, we rely on the fact that future execution behavior will see the same dependency chain. The next time the producer instruction is executed it will use the bypass queue and in addition, *its* producers will be added to the IST.

By implementing IBDA in hardware, we enable the use of original application binaries and do not require recompilation, application analysis or software hints. The IST itself is organized as a cache and requires only a moderate number of entries: only the address-generating instructions for the current inner loop are required while older data can be evicted, these instructions will be re-discovered during the first few iterations of their relevant inner loop. We found that for our set of benchmarks, a 128-entry IST is sufficient to collect most address-generating instructions relevant to the current loop, and that after just seven loop iterations the IBDA algorithm

Instruction		In-order w/ stall		OOO	Load Slice Core		
		@miss	@use		i1	i2	i3+
(1)	`mov (r9+rax*8), xmm0`	●	●	●	●	●	●
(2)	`mov esi, rax`		●	●	●	●	●
(3)	`add xmm0, xmm0`						
(4)	`mul r8, rax`			●			●
(5)	`add rdx, rax`			●		○	●
(6)	`mul (r9+rax*8), xmm1`			●	○	○	●

● overlapped with long-latency load (1)
○ issued to bypass queue but blocked by dependency in main queue

Figure 2: Example inner loop, marking which instructions can be overlapped with the initial long-latency load in the various architectures.

has been fully trained (over 99.9% of all relevant instructions have been marked). See Section 6.4 for more details on the trade-offs between performance and the area used by the IST. Next, we walk through a detailed example of the operation of IBDA in the Load Slice Core.

Instructive example. To illustrate the potential for extracting memory hierarchy parallelism in the various architectures, we provide an annotated code snippet (the hot loop from `leslie3d`) in Figure 2. Instruction (1) is a long-latency load, which produces a result (written to register `xmm0`) that is consumed by instruction (3). Instructions (2), (4) and (5) calculate the new value of `rax` which will be part of the address calculation for a second long-latency load (6).

When executing this instruction sequence on an in-order, stall-on-miss processor, the pipeline would stall immediately after issuing the initial missing load (1). An in-order, stall-on-*use* processor would be able to perform part of the address calculation (instruction 2) but stalls at (3), and is again not able to expose the second load. An out-of-order processor, on the other hand, can analyze the dependency chain throughout the complete instruction sequence. It is able to conclude that only instruction (3) depends on outstanding load (1), and can therefore execute all address-calculating instructions and the second load without waiting for the first load to complete.

When a Load Slice Core core executes this instruction sequence, initially the IST will be empty. All loads (instructions 1 and 6) will be issued to the bypass queue, while all other instructions are issued to the main queue. As in the stall-on-use in-order processor, instructions (1) and (2) will be executed. Once (3) reaches the head of the main queue this queue will block, preventing (4) and (5), and hence load (6) in the bypass queue, from being executed underneath the initial miss. At the same time, instruction (5) will be detected as being an address generator for load (6), and inserted into the IST. In the second iteration of this instruction sequence, instruction (5) will be found in the IST at dispatch and will thus enter the bypass queue. Instruction (4) is still in the main queue, and is blocked by the initial load's dependent (3), again preventing parallelism in the memory hierarchy from being exposed. But now (4) will be detected as a producer for an instruction already in the IST (instruction 5), and instruction (4) will be inserted into the IST. Finally, from the third iteration onward,

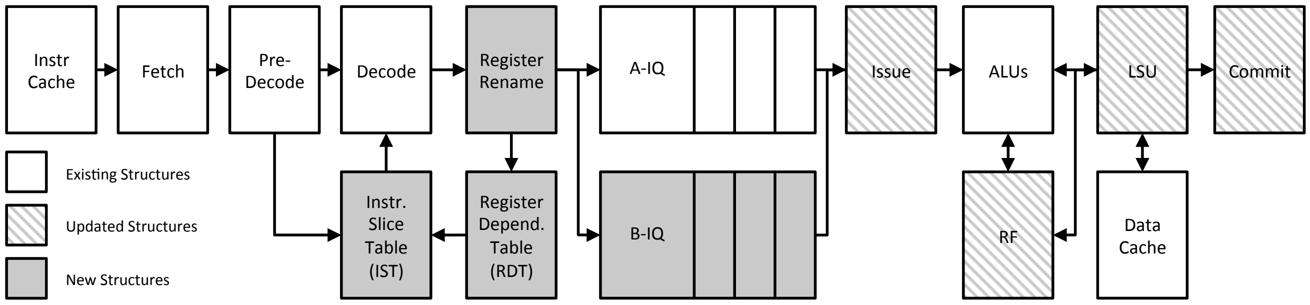

Figure 3: Load Slice Core microarchitecture schematic. The Load Slice Core adds a number of new structures (gray components) while extending some others (dashed components) over an in-order, stall-on-use baseline (white components).

both (4) and (5) will be found in the IST and issued to the bypass queue, and are hence no longer hidden behind (3), which blocks the head of the main queue. Instructions (4) through (6) will issue from the bypass queue instead, enabling both loads to be overlapped. The strength of the Load Slice Core is that IBDA can dynamically identify critical instruction slices, across control flow instructions and basic blocks.

4. The Load Slice Core Microarchitecture

Figure 3 provides a high-level overview of the Load Slice Core microarchitecture. Its design builds on an in-order, stall-on-use microarchitecture by adding to or modifying a number of components. First, a second in-order instruction queue, the *bypass queue* (B-IQ), is added in addition to the *main queue* that exists in in-order processors (A-IQ). The bypass queue is used to enable memory accesses and instructions identified as address-generating to potentially be dispatched for early execution. Instructions in the bypass queue are allowed to execute ahead of those in the original queue. The RDT and IST implement IBDA and enable low-overhead detection of AGIs. To support speculative execution of instructions in the bypass queue, register renaming has been added. Finally, we enlarge the memory support structures such as the load-store queue and MSHRs to enable a larger number of outstanding misses. We now detail the changes that have been made to the in-order, stall-on-use baseline processor, and describe other important implementation aspects of the Load Slice Core microarchitecture.

Front-end pipeline. After instructions are fetched from the instruction cache, the IST is queried to see if they are known address-generating instructions. This generates an IST hit bit, which is set to one for instructions found in the IST, and is passed down the front-end pipeline for use in the dispatch stage. The IST is indexed by the instruction pointer, so for architectures with variable-length instruction encoding this step is performed after the pre-decode stage, which determines instruction lengths and boundaries.

Register renaming. Register renaming serves an additional function in the Load Slice Core in addition to eliminating artificial dependencies. It enables the core to more easily handle interactions between the two instruction queues. The goal of the bypass queue is to continue to make forward progress as long as the true dependencies of the instruction have been satisfied. But, as instructions in the B queue may run ahead of the A queue, keeping track of dependencies and hazards can become difficult without extending the traditional in-order core. By using register renaming, the results from the bypass queue can be computed ahead of time, stored in the register file, and referenced later by either the bypass or main queues.

Register renaming is implemented with a merged register file scheme. A register mapping table translates logical registers into physical registers for each register of an instruction. If the instruction produces a result, the register mapping table is updated with a new register from the free list. The physical registers are used to access the register file, keep track of dependencies and access the RDT. In addition, a recovery log is used to rewind and recover the register mappings in case of a branch misprediction or exception, and is completed within the branch misprediction penalty.

Dependency analysis. A high-level overview of the IBDA algorithm can be found in Section 3. IBDA is implemented using two structures: the instruction slice table (IST) and register dependency table (RDT). The IST is maintained as a cache tag array; we assume a 128-entry, 2-way set-associative design with LRU replacement (see Section 6.4 for a further exploration). The IST stores addresses for all instructions identified as address-generating, and contains no data bits — a hit in the IST means the instruction was previously identified as address-generating, a miss means that either the instruction is not address-generating or is yet to be discovered as such. In either case, instructions that miss in the IST are sent to the A queue. Loads and stores are sent to the B queue automatically, and hence do not have to be stored in the IST. We assume complex instructions are broken up into micro-operations, each of which is either of load, store, or execute type. Load and store micro-ops automatically go to the B queue, so the IST applies to the execute-type micro-ops only.

The RDT is used to identify dependencies between instructions. Each physical register in the RDT contains the instruction address of the last instruction that wrote to this register. As instructions are decoded and renamed, their address and current IST hit bit are written to all RDT entries that the instruction writes to. An instruction's producers can be found by reading the RDT entries corresponding to all registers read by the instruction. If the current instruction is a load, store, or marked address generator, all of its producers are looked up in the RDT,[2] and if the producer's IST bit (which is cached by the RDT) was not already set, the producer's address is inserted into the IST. Updates to the IST are made off the critical path, while the producer instruction itself, even if it is present in the processor further down the pipeline, is not affected — instructions are never moved from the A to the B queue retroactively.

Instruction dispatch. Instructions are dispatched into the appropriate queue, either according to their type (load/store), or by their IST hit bit. Load instructions are always inserted into the B queue. Address-generating instructions will go to the B queue if they were present in the IST at fetch time. Stores are entered into both queues: store address calculations are performed from the B queue such that unresolved store addresses automatically block future loads; while store data is collected from the A queue, enabling the store to proceed to update memory in program order only and after ensuring no exceptions were encountered. All other instructions are dispatched into the main queue.

Issue/execute. The instruction scheduler can select up to two instructions for execution, chosen from the heads of the A and/or B queues. When ready instructions are found in both queues, the oldest instructions in program order are executed first.[3] For simplicity, we assume all execution units are shared between the A and B pipelines. An alternative implementation could further separate the A and B pipelines, giving each its own cluster of execution resources and further simplifying instruction scheduling. Since address-generating instructions usually consist of simple arithmetic operations, this alternative could restrict the execution cluster for the B pipeline to the memory interface and simple ALUs. To prevent complex address-generating instructions from entering the B queue, the front-end would make an extra selection based on opcode and insert them into the A queue even if their IST hit bit was set.

Memory dependencies. In addition to register dependencies, which are known at instruction decode time, there may exist dependencies through memory from stores to loads with an overlapping memory address. While some out-of-order designs speculate on these conflicts, allowing loads to bypass earlier stores and recover from potential misspeculation, the Load Slice Core instead sends store instructions to both the main and bypass queues. Store *data* is collected from the main queue, while all address calculation for both loads and stores is handled through the bypass queue. Because this queue is in-order, all address calculation is performed in-order so stores with an unresolved address automatically prevent future — potentially conflicting — loads from being executed. Once the operands needed to compute the store address are known, the store is allowed to execute from the bypass queue, writing its store address into the store buffer. When the store reaches the head of the main queue, its data is available as well and the store can be written out to memory.

[2]For stores, only operands relevant to compute the address are considered.

[3]Experiments where priority was given to the bypass queue, which could make loads available even earlier, did not see significant performance gains.

Component	Parameters		
	in-order	Load Slice Core	out-of-order
Core		2 GHz, 2-way superscalar	
Reorder logic	none	32-entry IQ-A/B	32-entry ROB, sched.
IST	—	128-entry, 2-way LRU	—
Branch predictor		hybrid local/global predictor	
Branch penalty	7 cycles	9 cycles	9 cycles
Execution units		2 int, 1 fp, 1 branch, 1 load/store	
L1-I		32 KB, 4-way LRU	
L1-D		32 KB, 8-way LRU, 4 cycle, 8 outstanding	
L2 cache		512 KB, 8-way LRU, 8 cycle, 12 outstanding	
Prefetcher		L1, stride-based, 16 independent streams	
Main memory		4 GB/s, 45 ns access latency	
Technology node		28 nm	

Table 1: Simulated single-core microarchitecture details.

Loads that are about to issue can thus be checked against the addresses of outstanding stores in the store buffer in program order. This avoids the need for speculative address disambiguation. Since unknown store addresses now block loads, we want store address computation to be executed on the bypass queue as well. Therefore, the IBDA algorithm considers both load and store addresses as roots for finding backward slices.

Commit. The commit stage checks for exceptions and makes the (previously speculative) state architecturally visible, and releases structures such as store buffer entries and rename registers. Instructions are entered in-order into a scoreboard at dispatch, record their completion out-of-order, and leave the scoreboard in-order. This structure operates similarly to the existing scoreboard in stall-on-use in-order processors which allows them to support out-of-order completion of variable execution latencies, although the Load Slice Core enlarges it to support the larger number of outstanding instructions.

5. Experimental Setup

We use the Sniper multi-core simulator [6] and the cycle-level in-order and out-of-order core models [7] as baselines for this work. In addition, we extended the simulator with a detailed, cycle-level core model for the Load Slice Core microarchitecture. See Table 1 for the main architectural parameters. All cores are two-wide superscalar, with the same execution units and cache hierarchy, and use hardware prefetchers. Second-level cache capacity and main memory bandwidth are representative for each core's fair share in modern many-core processors [11]. Branch misprediction penalties for the Load Slice Core and out-of-order architectures are slightly higher to account for the extra pipeline stages (rename and dispatch) in the front-end. Experiments are evaluated using the SPEC

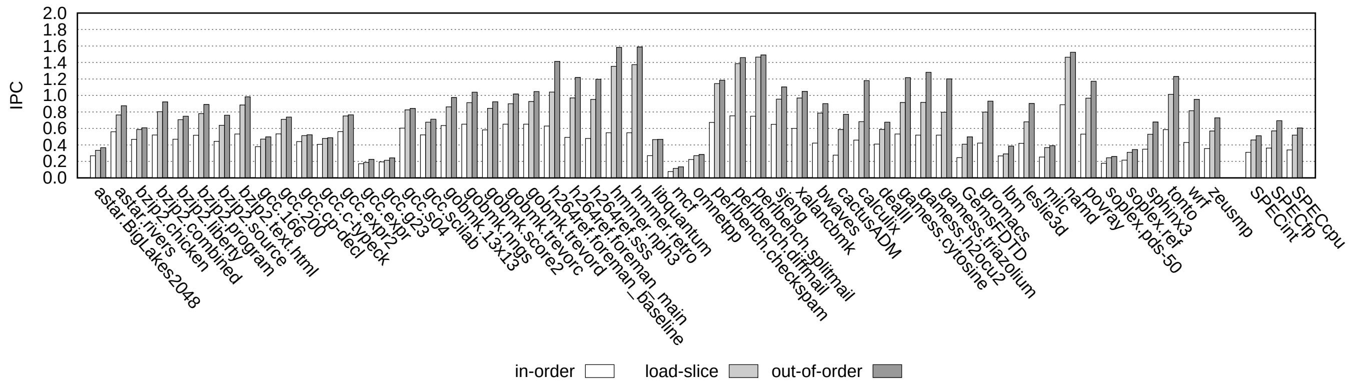
Figure 4: Load Slice Core performance for all SPEC CPU 2006 workloads compared to in-order and out-of-order cores.

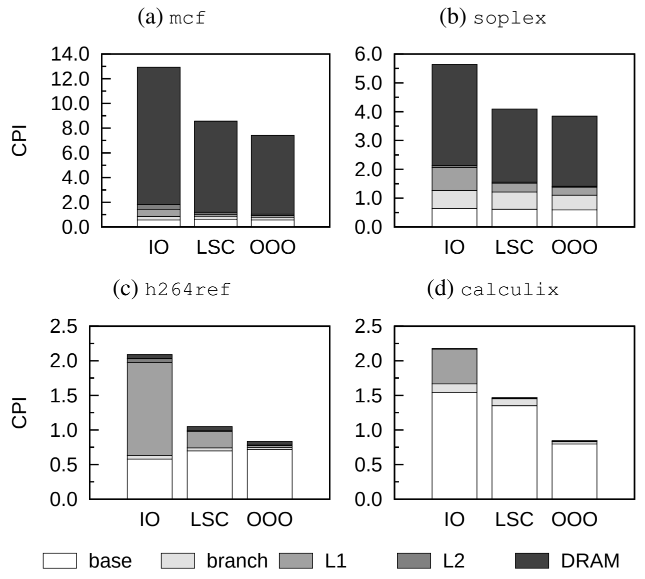
Figure 5: CPI stacks for selected workloads.

CPU2006 benchmark suite (ref input set), where a single, most representative region of 750 million instructions was chosen using the SimPoint methodology [32]. In addition to SPEC CPU2006, we also evaluate multicore scaling with the NAS Parallel Benchmarks (NPB) (A input set) [19] and the SPEC OMP2001 application suites (ref input set) [3]. We simulate representative phases of these parallel applications according to previous guidelines [17]. Area and power estimates were obtained from CACTI 6.5 [23] for the 28 nm technology node. Static power consumption and per-access energy values from CACTI were combined with activity factors obtained from the timing simulation to compute per-structure power and energy.

6. Results and Analysis

6.1. Load Slice Core performance

We simulate the performance of the Load Slice Core architecture, and compare it to in-order and out-of-order baselines. Figure 4 plots performance in instructions per clock cycle (IPC) of all three core types for the SPEC CPU 2006 workloads. Averaged over the complete benchmark suite, the out-of-order processor variant outperforms the in-order baseline processor by 78%. Our proposed Load Slice Core architecture improves performance by 53% over the in-order baseline, covering more than half the difference with the out-of-order core.

Performance breakdown. The Load Slice Core microarchitecture is able to exploit both on-chip cache parallelism as well as off-chip memory-level parallelism and therefore exceeds in-order performance for both cache-fitting and DRAM-limited workloads. A performance breakdown in the form of CPI stacks is provided in Figure 5 for a number of representative SPEC CPU2006 benchmarks.

The `mcf` workload (top left) is bound by off-chip access latency. The in-order core spends over 80% of its execution time stalled on accesses to main memory, while the out-of-order core is able to find independent loads, hence exposing MHP leading to a performance improvement of almost 2×. The Load Slice Core can similarly expose almost the same amount of MHP resulting in performance close to that of the out-of-order core. In `soplex` (top right), dependencies exist that prevent off-chip memory accesses from occurring in parallel (pointer chasing). Neither the out-of-order nor the Load Slice Core are able to expose significant amounts of MHP.

In contrast, `h264ref` is a compute-intensive workload with few cache misses (bottom left). Still, the in-order core sees a significant penalty resulting from stalls caused by L1 hits: even though the L1 access latency is only three cycles, immediate reuse still causes the in-order core to stall. In contrast, as the Load Slice Core uses the bypass queue for all loads including L1 hits, it is able to issue those loads earlier and avoid stalls of the main queue, thereby approaching out-of-order performance on this workload. Finally, for `calculix` (bottom right) the Load Slice Core is able to improve on in-order performance by overlapping L1 access time, while the out-of-order core retains a significant performance advantage as it can expose additional ILP for instructions other than loads and address producers.

Component name	Organization	Ports	Area (μm^2)	Overhead	Power (mW)	Overhead
Instruction queue (A)	32 entries × 22B	2r 2w	7,736	0.74%	5.94	1.88%
Bypass queue (B)	32 entries × 22B	2r 2w	7,736	1.72%	1.02	1.02%
Instruction Slice Table (IST)	128 entries, 2-way set-associative	2r 2w	10,219	2.27%	4.83	4.83%
MSHR	8 entries × 58 bits (CAM)	1 r/w 2s	3,547	0.39%	0.28	0.01%
MSHR: Implicitly Addressed Data	8 entries per cache line	2 r/w	1,711	0.15%	0.12	0.05%
Register Dep. Table (RDT)	64 entries × 8B	6r 2w	20,197	4.49%	7.11	7.11%
Register File (Int)	32 entries × 8B	4r 2w	7,281	0.56%	3.74	0.65%
Register File (FP)	32 entries × 16B	4r 2w	12,232	1.10%	0.27	0.11%
Renaming: Free List	64 entries × 6 bits	6r 2w	3,024	0.67%	1.53	1.53%
Renaming: Rewind Log	32 entries × 11 bits	6r 2w	3,968	0.88%	1.13	1.13%
Renaming: Mapping Table	32 entries × 6 bits	8r 4w	2,936	0.65%	1.55	1.55%
Store Queue	8 entries × 64 bits (CAM)	1 r/w 2s	3,914	0.43%	1.32	0.54%
Scoreboard	32 entries × 10B	2r 4w	8,079	0.67%	4.86	1.26%
Load Slice Core			516,352	14.74%	121.67	21.67%
Cortex-A9			1,150,000	155.56%	1259.70	1159.70%

Table 2: Load Slice Core area and power (in μm^2 and mW) calculated with CACTI 6.5 in 28 nm. Results compared to a Cortex-A7 CPU core of 450,000 μm^2 and 100 mW average power consumption [2]. The 2-wide out-of-order ARM Cortex-A9 is listed for comparison [1]. All numbers are exclusive of the L2 cache.

6.2. Area and Power Overheads

We compare the area of the Load Slice Core against both an in-order and an out-of-order design. Our in-order baseline, the ARM Cortex-A7, is equipped with a 2-wide superscalar in-order pipeline, and occupies approximately 0.45 mm^2 in the 28 nm technology node [2]. As the out-of-order comparison point we selected the ARM Cortex-A9 processor. It has an area of approximately 1.15 mm^2 in 28 nm, or an overhead of 156% over our in-order baseline [1]. In both cases, we include the area for the L1 instruction and data caches, but not for the L2 cache. Both baselines are conservative: the in-order Cortex-A7 is one of the smaller in-order processors available (hence overestimating the relative overhead of the structures added by the Load Slice Core), while the A9 is only partially out-of-order and is therefore smaller than a more aggressive out-of-order design (reducing the area advantage the Load Slice Core has over the out-of-order baseline). The power used by the Cortex-A9 at 28nm was scaled with an aggressive ITRS scaling estimate of 35% per technology node as reported by Esmaeilzadeh et al. [16].

For each major Load Slice Core component, we compute its area and power consumption using CACTI 6.5 [23] assuming a 28 nm technology node. All components are at or below 0.2 ns access time, and therefore support a clock frequency of at least 2 GHz (accounting for logic delays). The absolute area and power values are listed in Table 2. We also list the additional area and power that each structure needs over its corresponding equivalent in the in-order core (if any), expressed as a fraction of the total in-order core area or power consumption. All values listed in Table 2 use activity factors averaged over all SPEC CPU applications. The main instruction queue is increased from 16 entries to 32 entries to allow additional instructions to enter the bypass queue. The scoreboard holds

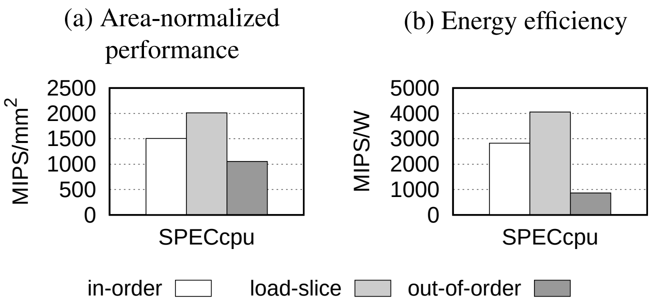

Figure 6: Area-normalized performance and energy efficiency of the Load Slice Core.

32 entries to support a maximum of 32 in-flight instructions. The MSHR and register files are also assumed to exist in partial form in the baseline processor. We extend the MSHRs to support 8 outstanding misses and double the size of the register files to support 32 physical registers each. Ports were sized to allow for the lookup of two instructions per cycle, with a potential of three input registers and one output register per instruction, and for a recovery of up to 4 instructions per cycle. In total, we find the Load Slice Core to increase area of an in-order stall-on-use baseline processor core by just 15%. The resulting power consumption overhead of the Load Slice Core is 21.7% on average, with individual workloads going up to at most 38.3%.

When combining these power and area estimates with each core's simulated performance, we find that the Load Slice Core outperforms traditional in-order and out-of-order designs in both area-normalized performance and in energy efficiency. As shown in Figure 6 (which includes the power and area of the L2 cache), the Load Slice Core achieves an area-normalized performance of 2009 MIPS/mm^2 and an energy efficiency of 4053 MIPS/W. In contrast, the in-order

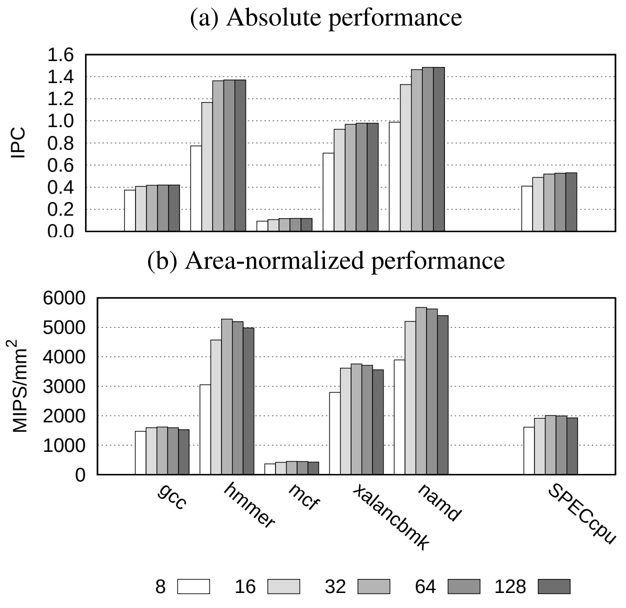

Figure 7: Instruction queue size comparison.

core obtains 1508 MIPS/mm^2 and 2825 MIPS/W because of its (much) lower performance at only slightly lower area and power consumption, while the out-of-order core is much less efficient at just 1052 MIPS/mm^2 and 862 MIPS/W as it needs significantly larger and more power-hungry structures to obtain a limited performance boost over the Load Slice Core.

6.3. Instruction Queue Size

In Figure 7 we explore the size of the instruction queues (we assume both A and B queues and the scoreboard have the same size). We plot absolute performance (top) and area-normalized performance (bottom, MIPS/mm^2) for a selection of interesting workloads, in addition to the harmonic mean over the complete SPEC CPU 2006 suite. For some benchmarks (`gcc`, `mcf`), queue size does not affect performance much, while for other benchmarks (`hmmer`, `xalancbmk`, `namd`), performance saturates at a queue size of 32 to 64 entries. Remember that both queues are executed in-order, but that — since dispatch stalls whenever one of the queues fills up — the queue size determines how far instructions from the bypass queue can execute ahead of those remaining in the main queue. The queue size therefore determines how far (in number of instructions) loads and their address-generating producers can be hoisted up in front of those instructions that consume load data.

When considering area-normalized performance (Figure 7, bottom), we can see that a queue size of 32 entries is the optimum design point: smaller queues degrade performance, while larger queues do not increase performance to an extent that justifies their larger area.

6.4. IST Organization

Thus far we assumed an instruction slice table (IST) of 128 entries with a two-way set-associative organization. Figure 8 explores different IST organizations, including a variant that forgoes an IST, and a design where the IST functionality is integrated into the first-level instruction cache. From top to bottom, the figures plot each option's absolute performance, area-normalized performance, and the fraction of the dynamic instruction stream dispatched to the bypass queue.

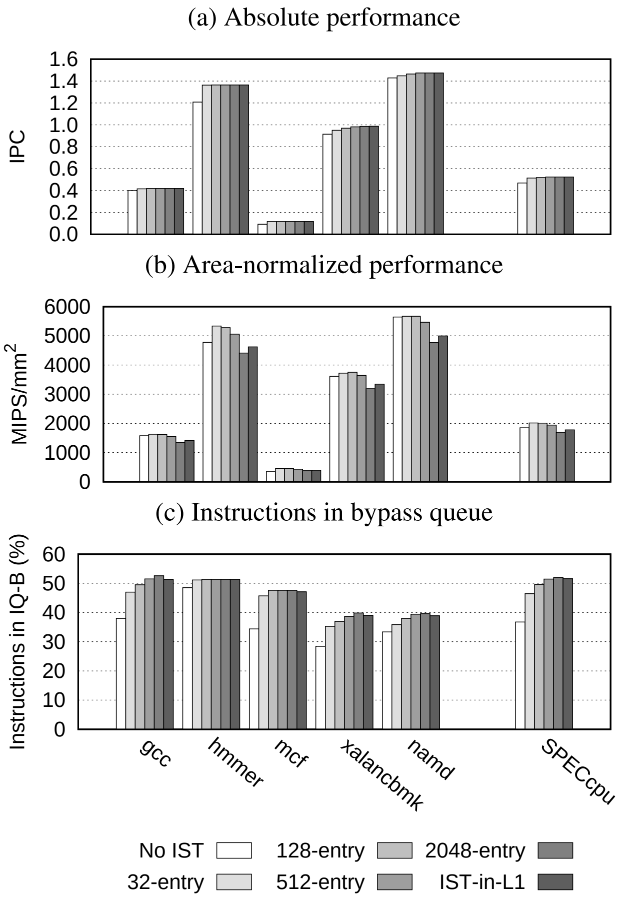

Figure 8: IST organization comparison.

In an architecture that forgoes the IST, only loads and stores use the bypass queue while address-generating instructions remain in the main queue. Larger ISTs capture progressively more address-generating instructions, enabling more loads to be executed out-of-order, but have an additional area cost. A 128-entry IST suffices to capture the most relevant instructions, and provides the highest area-normalized performance.

An alternative design to the stand-alone IST is to integrate its functionality into the instruction cache. In this implementation, the L1 I-cache is extended with one bit per instruction (assuming the worst case of one-byte encodings for each instruction). This *dense* representation is able to accommodate for the widest variety of application patterns, but is not very space-efficient as most instructions are not on backwards slices, or can be recognized as loads or stores from their opcode and should be issued to the bypass queue automatically.

In contrast, a *sparse* representation using a stand-alone IST is more space-efficient, since very few instructions actually

Iteration	1	2	3	4	5	6	7
% Coverage	57.9%	78.4%	88.2%	92.6%	96.9%	98.2%	99.9%

Table 3: Cumulative distribution of the number of address-generating instructions found by subsequent IBDA iterations.

need to be stored at any given time. From the bottom graph of Figure 8, comparing the no-IST case (which only bypasses loads and stores) with the larger IST architectures, we can see that at most around an additional 20% of all (dynamic) instructions are dispatched to the B queue. The instruction pointers for these 20% have to be stored in the IST. Since training the IST is relatively fast (only a few loop iterations are needed), the IST needs to hold just those instructions corresponding to the current inner loop. With an inner loop of at most a few hundred instructions, of which 20% need to be marked in the IST, an IST structure of 128 entries should indeed be large enough.

The number of IBDA iterations needed to find all address-generating instructions is shown in Table 3. This table plots the cumulative number of instructions found at a given distance from the memory operation that starts a backward slice, and hence indicates how many loop iterations are needed before the IBDA algorithm can discover them. Backward slices are typically short: only three iterations are needed to find 88% of all address-generating instructions, while seven iterations cover 99.9%. Note that address producers that have completed execution before the dispatch of dependent instructions are not considered performance-critical, as such we do not consider them to be part of the backward slice.

Other design choices for the stand-alone IST structure include its associativity, and the instruction pointer bits used to address IST sets. We performed an extensive evaluation of these parameters and found that larger associativities were not able to improve on the baseline two-way associative design. To address the IST, we used the least-significant bits of the instruction pointer, which works well given x86's variable length instruction encoding and yields a good distribution of instructions over all sets. Other instruction-set architectures with a fixed encoding length should shift the instruction pointer address bits accordingly to avoid set imbalance.

6.5. Power-limited Many-core Processors

When considering just single-thread performance, the Load Slice Core is outperformed by a fully out-of-order design. However, today nearly all commercial processors contain multiple cores, and the objective is to maximize their aggregate performance within strict power and area constraints. We now consider a power-constrained many-core processor that uses either in-order, out-of-order, or Load Slice Cores, and compare the application performance of the various design alternatives.

In the following experiment, we compare three separate power-limited processors, each with a homogeneous collection of either Load Slice Cores, in-order cores or out-of-order cores. With a power budget of 45 W and a maximum area of 350 mm^2, we can support 105 in-order cores, 98 Load Slice Cores and 32 out-of-order cores. Each core has a private 512 KB L2 cache while the chip has eight memory controllers that each provide 32 GB/s of bandwidth to on-package memory. On-chip communication is supported by a 2-D mesh network that provides 48 GB/s bandwidth per link per direction. Table 4 contains the complete simulated system specifications (core parameters are identical to those in Table 1).

Component	Parameters		
	in-order	Load Slice Core	out-of-order
Core count	105	98	32
On-chip topology	15×7 mesh	14×7 mesh	8×4 mesh
On-chip network	48 GB/s per link per direction		
Coherency protocol	directory-based MESI, distributed tags		
Main memory	8 controllers × 32 GB/s each		
Power consumption	25.5 W	25.3 W	44.0 W
Area	344 mm^2	322 mm^2	140 mm^2

Table 4: Simulated microarchitecture details for the power-limited processor configuration, where different from Table 1. The power consumption was capped at 45 W with a maximum core area of 350 mm^2.

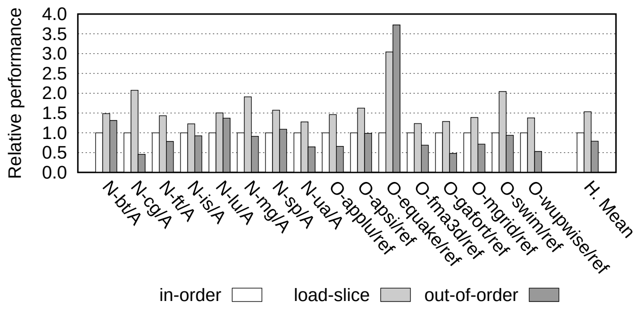

Figure 9: Parallel workload performance on a power-limited many-core processor, by core type. The 98 Load Slice Cores are able to outperform both a collection of 105 in-order and a collection of 32 out-of-order cores.

Figure 9 plots the simulated performance (one over execution time, relative to the in-order platform) of the NAS Parallel Benchmarks and SPEC OMP2001 suites, which are collections of scalable OpenMP workloads. By combining good single-thread performance with maximum energy efficiency, the Load Slice Core-based many-core processor outperforms the design alternatives based on traditional out-of-order or in-order cores: on average, 98 Load Slice Cores achieve 53% higher performance than the 105-core in-order design and are 95% faster than the out-of-order variant which, due to power constraints, can support only 32 cores. Only `equake` performs better on the low core-count out-of-order platform due to its bad scaling behavior to 98 or 105 cores (although undersubscription could potentially recover most of the performance loss [17]). The Load Slice Core therefore represents a strong design choice for future power-constrained many-core processors.

7. Related Work

Many techniques exist to prevent the processor from stalling on pending long-latency loads. In-order stall-on-use processors continue execution as long as the result of the load is not required, which results in a very efficient implementation although their performance benefit (specifically with respect to ILP) can be limited compared to out-of-order cores. Out-of-order processors dynamically construct the full dependency graph of all instructions in a window, and keep executing independent instructions while pending loads block the head of the window. This can be considered the most general solution, but has considerable complexity and low energy efficiency.

Both in-order and out-of-order processors implicitly extract *instruction slices* for execution, which are clusters of dynamically executed (not necessarily contiguous) instructions. Other solutions make the concept of identifying, caching and executing independent instruction slices explicit in order to improve performance or energy efficiency.

Hardware-software solutions. Helper threads [21] and speculative precomputation [13] are software-only techniques that use SMT [39] hardware contexts to execute key instruction slices early. Both spawn helper threads, containing instructions identified by the compiler or manual code additions, to prefetch data in a timely manner.

Other work combines compiler techniques or up-front application processing and analysis with custom hardware architectures. Examples include DAE [33], the braid microarchitecture [38], speculative slice execution [42], OUTRIDER [14] and flea-flicker multi-pass pipelining [5]. These works, in a similar fashion to software-only techniques, rely on decoupling critical, performance-degrading program slices [40] from the rest of the application. Such slices can be the backward slices of long-latency loads and difficult-to-predict branches or small chunks of mostly isolated instructions (braids). The braid microarchitecture modifies software through recompilation or binary translation to identify small, relatively isolated software chunks for parallel execution. Speculative slice execution [42] spawns additional, speculative SMT threads using up-front profiling information. In OUTRIDER, a single application is split into memory and compute slices while explicit thread communication instructions are inserted between threads during a compilation step. Each hardware thread context executes independently, enabling address generation and memory access instructions to potentially execute ahead of compute and control-flow instructions.

Hardware-only solutions. In order to apply these techniques to operate on unmodified application binaries, in addition to executing slices, hardware-based solutions need to identify and potentially cache slices for re-execution. Research in this area has taken two broad directions.

One direction is to start from an out-of-order design, and add additional structures that will allow the processor to perform even better in the presence of stalling events. Examples include runahead execution, which continues to execute past stalling events [22, 25], and designs that attempt to precompute the required data data before it is needed [12, 24, 34, 35]. While these solutions are typically able to obtain high performance, they require most of the hardware structures present in out-of-order processors, and add even more overhead for slice generation and caching. Their energy efficiency is therefore the same, if not worse, than a typical out-of-order processor.

An alternative direction is to build up from an in-order processor with additional hardware structures, such as multiple bound, in-order pipelines, to support execution of all types of slices to improve ILP extraction [27]. Nevertheless, the ILP that is naturally exposed by out-of-order cores tends to be both wide and shallow, limiting the ability of multiple in-order queues from extracting generic ILP in an efficient manner [31]. Out-of-order cores naturally build non-contiguous instruction slices for execution out of their reorder buffer, but this is not necessarily the case for strictly in-order processors. This is because, by the time a long-latency load event has been seen, it is already too late to try and prefetch its result. Therefore, prior works have focused on executing forward slices after a miss event [4, 8, 15, 18, 26], as it is relatively easy to propagate poison bits to squash future instructions or store them for later processing. To support runahead execution, these microarchitectures either re-execute instructions, duplicate the processor pipeline or use large structures to store runahead state. Flea-flicker two-pass pipelining [4] supports runahead through the use of an additional back-end pipeline including functional units, as well as other structures required to merge the state between the two pipelines and maintain memory consistency. Runahead execution [8, 15] continues executing after a cache miss, but uses a checkpoint to restore and re-execute all instructions that run ahead after that miss. In the iCFP microarchitecture [18], slice buffers are used to store both the decoded instruction, as well as all intermediate results computed. SLTP [26], on the other hand, does not re-execute instructions, but does require additional hardware for the slice of unexecuted instructions after the miss event, as well as storage for the intermediate data values and complete register checkpoints. Simultaneous Speculative Threading [9] is similar to SLTP except it uses an extra multithreading context to perform runahead execution.

In contrast, the Load Slice Core focuses only on memory hierarchy parallelism, rather than expending additional complexity to extract general-purpose ILP which is much more difficult to exploit in an energy-efficient way. The iterative identification of backward slices fits well into the in-order execution paradigm and allows for a simple hardware-based implementation that retains the energy efficiency of an in-order processor while significantly improving its performance.

8. Conclusion

We propose the Load Slice Core, a novel processor microarchitecture aimed at extracting memory hierarchy parallelism from unmodified binaries while keeping hardware overheads to a minimum. This is achieved by extracting backwards slices containing address-generating instructions through a novel iterative algorithm that can be implemented efficiently in hardware. These backward slices are then executed on a second in-order pipeline, enabling them to bypass instructions blocked by pending loads. The Load Slice Core design improves on existing work by providing a lightweight, hardware-based method of executing past pending loads while avoiding re-execution.

Based on detailed timing simulations and estimates of area and power, we demonstrate that the Load Slice Core is substantially more area- and energy-efficient than traditional solutions: average performance is 53% higher than an in-order, stall-on-use core, with an area overhead of only 15% and an increase in power consumption of just 22%. This enables a power- and area-constrained many-core design based on the Load Slice Core to outperform both in-order and out-of-order based alternatives, by 53% and 95%, respectively. We therefore believe that for today's context of constrained multi-core processors, the Load Slice Core strikes a good balance between single-thread performance and energy efficiency.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. We would also like to thank Arvind for inspiring this research and Magnus Själander for his feedback to improve this work. This work is supported by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

References

[1] ARM, "2GHz capable Cortex-A9 dual core processor implementation," http://www.arm.com/files/downloads/Osprey_Analyst_Presentation_v2a.pdf, archived at the Internet Archive (http://archive.org).

[2] ARM, "ARM Cortex-A7 processor," http://www.arm.com/products/processors/cortex-a/cortex-a7.php.

[3] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady, "SPEComp: A new benchmark suite for measuring parallel computer performance," in *OpenMP Shared Memory Parallel Programming*, R. Eigenmann and M. Voss, Eds., Jul. 2001, vol. 2104, pp. 1–10.

[4] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. W. Hwu, "Beating in-order stalls with "flea-flicker" two-pass pipelining," in *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, Jan. 2003, pp. 18–33.

[5] R. D. Barnes, S. Ryoo, and W. W. Hwu, ""Flea-flicker" multipass pipelining: An alternative to the high-power out-of-order offense," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Nov. 2005, pp. 319–330.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011, pp. 52:1–52:12.

[7] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014.

[8] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay, "High-performance throughput computing," *Micro, IEEE*, vol. 25, no. 3, pp. 32–45, 2005.

[9] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Simultaneous speculative threading: A novel pipeline architecture implemented in sun's rock processor," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2009, pp. 484–495.

[10] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Jun. 2004, pp. 76–87.

[11] G. Chrysos, "Intel® Xeon Phi coprocessor (codename Knights Corner)," in *Proceedings of the 24th Hot Chips Symposium*, Aug. 2012.

[12] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Dec. 2001, pp. 306–317.

[13] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, May 2001, pp. 14–25.

[14] N. C. Crago and S. J. Patel, "OUTRIDER: Efficient memory latency tolerance with decoupled strands," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2011, pp. 117–128.

[15] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th International Conference on Supercomputing (SC)*, Jul. 1997, pp. 68–75.

[16] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2011, pp. 365–376.

[17] W. Heirman, T. E. Carlson, K. Van Craeynest, I. Hur, A. Jaleel, and L. Eeckhout, "Undersubscribed threading on clustered cache architectures," in *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2014.

[18] A. Hilton, S. Nagarakatte, and A. Roth, "iCFP: Tolerating all-level cache misses in in-order processors," in *15th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2009, pp. 431–442.

[19] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS Parallel Benchmarks and its performance," NASA Ames Research Center, Tech. Rep., Oct. 1999.

[20] D. Kim, S. S. Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, "Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors," in *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2004, p. 27.

[21] D. Kim and D. Yeung, "A study of source-level compiler algorithms for automatic construction of pre-execution code," *ACM Transactions on Computer Systems (TOCS)*, vol. 22, no. 3, pp. 326–379, Aug. 2004.

[22] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, May 2002, pp. 59–70.

[23] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2011, pp. 694–701.

[24] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi, "Slice-processors: An implementation of operation-based prediction," in *Proceedings of the 15th International Conference on Supercomputing (SC)*, Jun. 2001, pp. 321–334.

[25] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2003, pp. 129–140.

[26] S. Nekkalapu, H. Akkary, K. Jothi, R. Retnamma, and X. Song, "A simple latency tolerant processor," in *IEEE International Conference on Computer Design (ICCD)*, Oct. 2008, pp. 384–389.

[27] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 1997, pp. 206–218.

[28] F. J. Pollack, "New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Nov. 1999.

[29] C. Ramey, "TILE-Gx100 manycore processor: Acceleration interfaces and architecture," in *Proceedings of the 23th Hot Chips Symposium*, Aug. 2011.

[30] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 37–48.

[31] P. Salverda and C. Zilles, "Dependence-based scheduling revisited: A tale of two baselines," in *Sixth Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, Jun. 2007.

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.

[33] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the Ninth Annual Symposium on Computer Architecture (ISCA)*, Jun./Jul. 1982, pp. 231–238.

[34] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2004, pp. 107–119.

[35] K. Sundaramoorthy, Z. Purser, and E. Rotenburg, "Slipstream processors: Improving both performance and fault tolerance," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov. 2000, pp. 257–268.

[36] J. E. Thornton, "Parallel operation in the control data 6600," in *Proceedings of the Fall Joint Computer Conference*, Oct. 1964, pp. 33–40.

[37] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, Jan. 1967.

[38] F. Tseng and Y. N. Patt, "Achieving out-of-order performance with almost in-order complexity," in *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, Jun. 2008, pp. 3–12.

[39] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, May 1996, pp. 191–202.

[40] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 352–357, Jul. 1984.

[41] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Computer Architecture News (CAN)*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[42] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th Annual Internationan Symposium on Computer Architecture (ISCA)*, Jun./Jul. 2001, pp. 2–13.

[43] C. B. Zilles and G. S. Sohi, "Understanding the backward slices of performance degrading instructions," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2000, pp. 172–181.