BOLT: Energy-Efficient Out-of-Order Latency-Tolerant Execution

Andrew Hilton and Amir Roth
Department of Computer and Information Science, University of Pennsylvania
{adhilton, amir}@cis.upenn.edu

Abstract

LT (latency tolerant) execution is an attractive candidate technique for future out-of-order cores. LT defers the forward slices of LLC (last-level cache) misses to a slice buffer and re-executes them when the misses return. An LT core increases ILP without physically scaling the issue queue and register file and increases MLP without additional software threads that can reduce cache performance. Unfortunately, proposed LT designs are not energy efficient. They require too many additional structures and they defer and re-execute too many instructions to justify their performance gains.

In this paper, we address these inefficiencies. We introduce a microarchitecture called BOLT (Better Out-of-Order Latency-Tolerance) that implements LT as an alternative use of SMT (Simultaneous Multi-Threading). We also present a new slice buffer organization and traversal scheme that increases performance and reduces overhead by pruning instances of useless and redundant LT. Collectively, these modifications turn out-of-order LT into a technique that improves performance in an energy-efficient way.

1. Introduction

Future multi-core processors are likely to include a large number of simple in-order cores as well as a few high-performance out-of-order cores. The in-order cores will execute parallel code. The out-of-order cores will execute sequential codes, the sequential portions of parallel codes, and critical sections [8]. These out-of-order cores will have to deliver single-thread performance in an energy-efficient way, because any additional energy they consume will reduce the number of simple cores that can be activated for parallel sections.

LT (latency-tolerant) execution promises to improve single-thread performance in a way that is energy-efficient and that is effective exactly when DVFS (Dynamic Voltage and Frequency Scaling) based overclocking is not [13]. LT attacks one of the primary sources of low performance in conventional out-of-order cores, long-latency LLC (last-level cache) misses. In a conventional core, an LLC miss stalls retirement, clogging the window, and halting execution. LT allows execution—and retirement—to continue in these situations and it does so without physically scaling the issue queue and register file. LT defers LLC misses and their dependent instructions, removing them from the window and saving them in a slice buffer. Deferred instructions release their issue queue and physical register resources, allowing younger instructions to enter the window and execute. When the miss returns, LT re-executes the load and its dependent instructions from the slice buffer. LT increases ILP (instruction level parallelism) by overlapping the execution of additional miss-independent instructions with the miss. If these instructions launch parallel LLC misses, LT also increases single-thread MLP (memory-level parallelism) [7]. Single-thread MLP is preferred to MLP exposed using multiple threads because it lacks the undesirable side effects of reduced cache performance and increased memory traffic. Avoiding the re-fetch and re-execution of miss-independent instructions gives LT higher performance and lower execution overhead than RA (runahead) [5, 15, 25], a related technique that unblocks the window in the presence of an LLC miss, but then re-executes all younger-than-miss instructions when the miss returns.

Despite conceptual simplicity and intuitive appeal, proposed designs have failed to deliver on LT's energy-efficiency promise [4, 12, 16, 17, 22]. They require too many additional structures, increasing both design complexity and energy consumption. And despite their overhead advantages over RA, they still defer and re-execute more instructions than their performance gains justify. This is primarily because they lack the mechanisms to recognize and prune instances of useless and redundant LT execution. On LLC-miss intensive programs from SPEC2000, CFP—a proposed LT design—improves performance by 12%, but executes 65% more

instructions to do so. Using a simple energy model (Section 5 has details), we calculate that CFP increases ED^2 by 27% over non-LT execution. In other words, it is less efficient than DVFS which preserves ED^2 [13]. Surprisingly, RA reduces ED^2 by 3% for the same programs. RA's efficiency derives from its ability to avoid useless and redundant RA episodes—it improves performance by 9% but executes only 27% more instructions.

Our work attacks LT's inefficiencies. We introduce an LT microarchitecture called *BOLT (Better Out-of-order Latency-Tolerance)* that re-renames deferred slices using the additional map tables available in an SMT (Simultaneous Multi-Threading) core [6, 24]. This reuse is possible because BOLT successfully marries a program order slice buffer [12, 17] with a unified physical register file that supports aggressive reclamation [1, 22]. BOLT also includes indexed load and store queues that scale to large sizes to support LT execution efficiently. By reusing existing structures and adding only simple RAMs, BOLT minimizes LT's implementation cost and static energy overhead—important for programs that do not benefit from LT—and provides a low-energy substrate for LT execution.

The exploitation of ILP and a non-episodic nature make it difficult for LT to reuse RA's mechanisms for recognizing and avoiding useless and redundant deferral and re-execution. We introduce a dependence-tracking scheme that recognizes instances of pointer-chasing LLC misses—the pathological useless LT idiom—and suppresses LT in response. We also describe a slice buffer traversal scheme that reduces redundant LT by limiting re-dispatch to the set of deferred instructions that depend on LLC miss(es) that just returned and on no other pending misses.

Together, BOLT and the useless and redundant LT pruning optimizations turn out-of-order LT execution into an energy-efficient performance technique. On the same LLC miss intensive programs, they improve performance by 15% while re-executing only 23% more instructions, reducing ED^2 by 8% relative to baseline.

2. Background: RA and LT

We begin by reviewing RA and LT.

RA (Runahead). An out-of-order processor uses RA to expose MLP in the presence of LLC misses [15]. When a pending LLC miss reaches the head of the ROB, RA checkpoints register state, poisons the destination register of the load, removes the load from the window, and begins RA execution. RA execution propagates the poison bit via data dependences to identify the load's forward slice and allow the instructions that comprise it to execute and release their issue queue slots. RA retirement processes executed instructions in program order, removes them from the ROB and frees their physical registers, but does not commit them to architected state—specifically, RA stores do not write the data cache. When the miss that triggered the RA episode returns, the processor restores the register checkpoint and fetches and executes all instructions younger than the miss. Re-fetch and re-execution of instructions that already executed in RA mode is accelerated because RA execution initiated parallel LLC misses and warmed up the caches.

LT (Latency Tolerance). LT improves on RA by re-executing *only* instructions that depend on a returning LLC miss, not all instructions younger than the miss. LT achieves this efficiency by buffering miss-dependent instructions in a slice buffer during the initial execution pass. Miss-dependent instructions have at least one miss-dependent input but may also have miss-independent "side" inputs. A key step of LT is to buffer these miss-independent inputs in the slice buffer during initial deferral. When an LLC miss returns, the side inputs of its dependent instructions may have been overwritten and released. By saving them in the slice buffer, LT decouples the slice from the rest of the program and allows it to re-execute in a self-contained manner at an arbitrary point in the future. When all deferred instructions have re-executed and the slice buffer is empty, LT releases the checkpoint, begins writing younger-than-checkpoint stores to the data cache, and continues normal mode execution. If slice re-execution fails for any reason—*e.g.*, because a miss-dependent instruction triggers a page fault—LT loses the ability to reuse any of the miss-independent instructions. Here, it restores the checkpoint, empties the slice buffer and restarts execution at the miss a la RA.

High-performance LT designs implement not only LT execution, but also LT re-execution. If slice re-execution triggers a dependent LLC miss, the miss and its forward sub-slice re-defer to the slice buffer rather than re-clogging the window. An LT processor with LT re-execution makes multiple passes over the slice buffer with each pass executing some subset of the still unexecuted instructions.

WIB, KILO, CFP, and D-KIP. WIB (Waiting Instruction Buffer) [12] and KILO instruction processor [4] are "pseudo" LT designs. They un-block the issue queue but not the register file so they need a large register file to be effective. WIB uses the ROB as a large sparse slice buffer. KILO routes miss-dependent instructions to a low complexity "slow lane" issue queue. Both

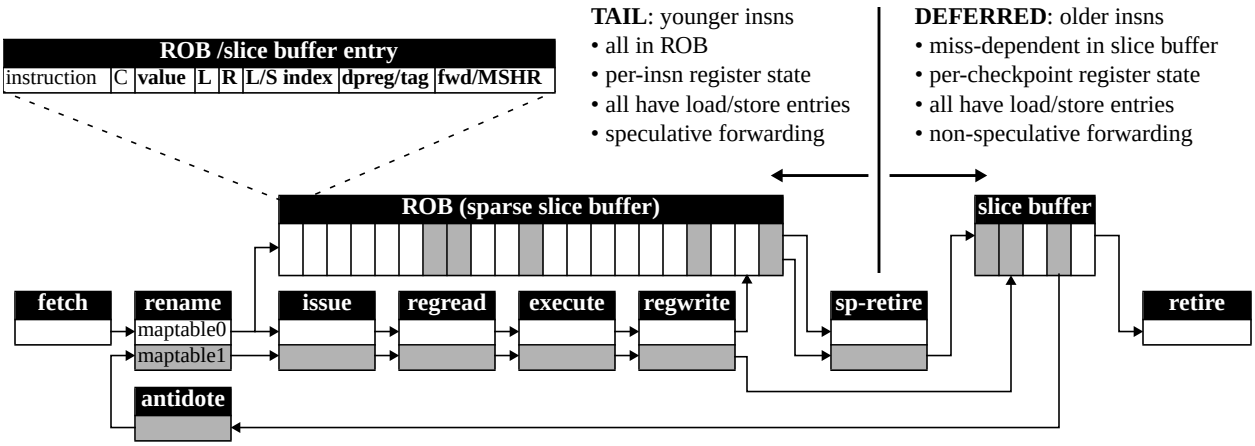


Figure 1. BOLT structure and pipeline diagram

support LT re-execution.

CFP (Continual Flow Pipelines) [22] is a true LT design with a unified non-blocking register file managed using the CPR (Checkpoint Processing and Recovery) algorithm [1]. CPR's execution-order register reclamation and lack of a ROB force CFP to use an execution order slice buffer in which register dependences are represented using physical register numbers. In turn, this forces CFP to re-rename deferred slices using physical-to-physical renaming. Physical-to-physical renaming is a non-traditional algorithm that requires a new map table. More significantly, it is register inefficient because a slice buffer pass does not clobber and incrementally release registers. CFP supports LT re-execution.

D-KIP (Decoupled KILO instruction processor) [17] and FMC (Flexible Multi-Core) [16] are chained dual-core LT designs. A ROB-based out-of-order core executes miss-independent instructions and uses retirement to defer misses and their dependents in program order. Deferred slices re-execute on one or more in-order cores. The dual-core setup means that D-KIP must copy every register value from the out-of-order core to the in-order core. It also leads to unnecessary deferrals when an instruction in the out-of-order core needs a value that is only available in the in-order core. Our experiments show that a D-KIP style design increases deferrals by up to 7%. D-KIP does not support LT re-execution while FMC uses multiple cores to support it partially.

3. BOLT

BOLT is an out-of-order LT core with LT re-execution that closely resembles CFP. Like CFP, BOLT re-executes deferred instructions in the out-of-order core—in-order re-execution is often considerably slower especially in floating-point programs where deferred dataflow graphs can have high ILP. Also like CFP, BOLT uses a single physical register file to both rename new instructions at the tail of the program and re-rename deferred instructions. BOLT differs from CFP in that it re-renames deferred slices using the additional map tables available on an SMT (Simultaneous Multi-Threading) processor [6, 24]. The re-purposing of SMT follows in-order LT designs like Rock [3] and iCFP [10] that use the additional register files available in a multi-threaded in-order processor to re-execute slices.

3.1 Structure

Renaming deferred slices on a conventional map table requires that slice register dependences be represented using logical register numbers. In turn, this requires that the slice buffer track deferred instructions in program order. Combining a program order slice buffer with a unified physical register file that supports aggressive reclamation of miss-independent registers is a challenge. BOLT meets this challenge by essentially hybridizing WIB, CFP, and D-KIP.

Overview. Figure 1 shows a high-level diagram of BOLT. The figure shows a ROB, a slice buffer, and the major pipeline stages. Shading distinguishes miss-independent instructions (white) from miss-dependent deferred instructions (gray). Re-executing deferred instructions pass through a BOLT-specific *antidote* stage (whose function is explained in Section 4.2) prior to re-renaming.

BOLT follows D-KIP and explicitly divides the dynamic instruction window into two regions. The *tail* region contains the younger instructions and BOLT's implementation of this region resembles WIB. In the tail, BOLT manages instructions using a ROB, does not aggressively reclaim registers, and supports instruction-granularity mis-prediction recovery. BOLT uses out-

of-order execution to create a sparse but program order slice buffer in the ROB—although poison instructions defer into the ROB in execution order, they defer into slots that were assigned to them in program order. The *deferred* region contains the older instructions and BOLT's implementation of this region resembles CFP. Here, deferred instructions are explicitly held in a dense slice buffer, but all miss independent instructions are represented implicitly using checkpoints. In the deferred region, BOLT aggressively reclaims registers and supports only checkpoint-granularity precise state and misprediction recovery. Instructions pass from the tail region to the deferred region in program order via a D-KIP style speculative retirement stage. This stage creates deferred region checkpoints, reclaims overwritten physical registers that do not appear in checkpoints, and copies deferred instructions from the sparse ROB to the dense slice buffer, essentially compacting the deferred instruction stream.

ROB/slice buffer format. In BOLT, the ROB and slice buffer comprise a single, logically contiguous structure. Because tail instructions initially defer into the ROB, BOLT may need to re-dispatch instructions out of the ROB when LLC misses return. BOLT slice buffer passes traverse first the slice buffer and then the ROB.

Figure 1 also shows the format of a BOLT ROB/slice buffer entry. An entry contains an instruction, a completion bit (*C*), a value, and left and right input poison bits (*L* and *R*). An instruction can be poisoned via both, one, or neither of its inputs. An instruction can be poisoned via neither input because BOLT uses the ROB as a sparse slice buffer and does not compact the (dense) slice buffer after each pass, leaving "holes." If a two-input instruction is poisoned via only one input, the value of the second input is captured in the value field. BOLT treats loads as two-input instructions. The left input is the address register. The right input is "data"—either from memory or a forwarding store. An LLC miss is poisoned via its right input as is a load that forwards from a data-poisoned store.

In addition to these fields, a ROB/slice buffer entry also includes a load or store queue index, a physical register number and tag, and a field that contains the store queue index of the forwarding store for forwarding loads and the MSHR number for loads that miss in the LLC. The functions of these fields are explained later in the paper. The total size of each entry is 167 bits.

Load and store queues. LT requires a more complex in-flight data memory system than RA. An RA processor does not commit any RA mode instructions. During RA execution, it can use best-effort store-load forwarding and does not need to disambiguate loads with respect to external stores from other threads. In contrast, LT requires loads to forward non-speculatively from speculatively retired stores to guarantee correct deferral. It also requires re-executing loads to execute non-speculatively in the presence of younger stores to the same address. Finally, it must continue to disambiguate all younger-than-checkpoint loads against external stores until all slices have re-executed. BOLT implements these functions using scalable, indexed load and store queues that hold all loads and stores, respectively, in the tail and deferred regions.

3.2. Pipeline

BOLT is an LT processor with LT re-execution and so dynamic instructions that depend on LLC misses may defer and re-execute multiple times, passing through some portion of the pipeline each time. However, BOLT imposes some high-level invariants on top of this multi-pass execution model. Every dynamic instruction is (i) fetched, (ii) dispatched into the tail region and allocated ROB and load/store queue entries, (iii) speculatively retired from the tail region to the deferred region, and (iv) permanently retired from the deferred region. Instructions pass through these stages in program order and every dynamic instruction passes through each of these stages exactly once. Loads and stores are assigned load and store queue slots during initial tail dispatch and keep these slots through every re-execution pass. Similarly, instructions are assigned ROB slots during initial tail dispatch and deferred instructions are assigned slice buffer slots during speculative retirement. They stay in these slots for as long as they remain in the tail and deferred regions, respectively [22].

This section describes several aspects of the BOLT pipeline in detail.

Poison tracking. BOLT tracks poison on a per physical register and store queue entry basis. The output poison of an instruction is the logical OR of the poison of its inputs. Poison travels alongside values through the bypass network. Register poison is read at the register read stage and written at the register write stage Store poison is cleared at dispatch and written (by stores) and read (by forwarding loads) at the execute stage. Register poison is also logically cleared at dispatch but it does not have to be physically cleared because it is only read during execution. BOLT reads store poison to determine which loads to re-dispatch when a given miss returns (Section 4.2 describes this process). Executing (and re-executing) instructions write their poison into their ROB/slice buffer entries. If a two-input instruction has one poison input and one present non-poison input,

the present input is copied into the ROB/slice buffer as well.

Speculative retirement, load verification, and store-load forwarding. In addition to creating checkpoints, freeing registers, and copying deferred instructions from the ROB to the slice buffer, BOLT's speculative retirement stage also verifies load speculation relative to same-thread stores. An important aspect of BOLT is that only tail region loads are speculative relative to same-thread stores. Deferred region loads—both miss-dependent and miss-independent are *non-speculative* relative to same-thread stores. This allows BOLT to avoid re-processing speculatively retired stores for verification purposes.

BOLT achieves this invariant as follows. Forwarding from tail region stores is speculative and performed using store queue index prediction [21]. Forwarding from deferred region stores is non-speculative and performed using address-hash chaining [10]. BOLT uses the in-order speculative retirement stage to overlay an address-based hash table on top of the store queue portion that holds deferred region stores. Loads forward from speculatively-retired stores by following a chain of links starting at the appropriate entry in a "root table." During initial tail dispatch, a load obtains a store queue index for its predicted forwarding store. During execution, the load reads the store queue at this predicted index while reading the root table in parallel. If the load's address matches that of the predicted tail store, the load forwards from that store. If the predicted tail store's address does not match *and* the root table indicates a potential match from a deferred store, the load begins a process in which it reads the store queue iteratively following the address-hash links. Chained access does add latency and does require replaying the load's dependents as if the load had missed in the cache. However, it is also rare—the average number of hops through the deferred region of the store queue is less than 2 per 100 loads. At speculative retirement, BOLT verifies speculative forwarding from tail stores using SVW-filtered load re-execution [18]. BOLT actually unifies SVW with address hash chaining, using chaining's root table as SVW's Bloom filter. Like speculative retirement itself, this verification takes place once per instruction.

By construction, deferred region loads are older than all tail region stores and so re-executing deferred region loads forward using address-hash chaining only. Address-hash chaining allows these loads to execute non-speculatively in the presence of younger stores to the same addresses. A re-executing load knows its age relative to stores in the store queue and ignores all logically younger stores as it follows the chain. Because forwarding from speculatively-retired stores is non-speculative, re-executing deferred region loads do not have to be re-verified.

Retirement, load verification, and store completion. If there are no deferred instructions and deferred region checkpoints, BOLT retires at instruction granularity like a conventional ROB processor. If there are deferred instructions, BOLT retires at a checkpoint granularity once the slice buffer portion corresponding to the oldest checkpoint is empty. Retirement involves writing the stores to the data cache and releasing the checkpoint. Writing stores to the cache at checkpoint granularity is complicated in a multi-processor which defines the memory consistency model at instruction-granularity. This is because deferred region loads are vulnerable to external stores and resolving conflicts require squashing entire checkpoints. BOLT sidesteps this problem using DSC/SDR (Decoupled Store Completion/Silent Deterministic Replay) [11] which allows stores to complete incrementally within a checkpoint. On a conflict, BOLT squashes the checkpoint but then replays silently up to the youngest completed store. BOLT detects conflicts with external stores using per-checkpoint signatures [2, 10].

Slice buffer passes. At the beginning of each slice buffer pass, BOLT initializes a fresh map table. This map table is initially empty because a deferred slice has no external register inputs—all register values are either read from the slice buffer and treated as immediate inputs during re-execution or produced by instructions in the slice. When a pass completes, BOLT releases this map table. A given slice buffer pass may start at any point in the ROB/slice buffer (*e.g.*, at the miss that just returned) but must traverse the ROB/slice buffer from that point to its end. If a given pass re-dispatches instruction X, all instructions that depend on X must re-dispatch during the same slice buffer pass. This is the only way to ensure that they can pick up values produced by older instructions in the slice.

3.3. Register Management

BOLT uses two algorithms to manage a single register file. In the tail, it uses ROB-style register management where physical registers are logically held by (*i.e.*, allocated to) the architectural map table and by the destinations of in-flight tail instructions. In the deferred region, it uses CPR-style register management where physical registers are held by checkpoints, by the map table allocated to the current slice buffer pass, and by the sources and destinations of deferred instructions which have re-dispatched to the window but have not executed.

Register reference counting. BOLT unifies these two register management schemes using physical register reference counting [1]. Register reference counting is most naturally implemented as a matrix in which each column corresponds to a physical register and each row corresponds to an entity that can hold a physical register, *e.g.*, an instruction or a register checkpoint. ORing the bits in each column creates a bitvector-style freelist from which free registers are allocated using encoders [19]. BOLT's hybrid implementation begins with CPR's reference counting structures—one bitvector for the map table, a RAM with one row per checkpoint, and a second RAM with one row per issue queue entry. To these, BOLT adds two structures that formulate ROB-style management as reference counting—a bitvector for the architectural map table, and a second bitvector for the destinations of all instructions in the ROB.

BOLT's register management scheme resembles CPROB [9]. CPROB hybridizes CPR and ROB in the same window region to provide opportunistic minimal recovery on branch mis-predictions. BOLT implements CPR and ROB in separate window regions.

Tracking live destinations. Using a single register file to hold all register values minimizes deferrals by allowing values computed by re-executing deferred instructions to immediately appear in the tail region. When BOLT re-renames an instruction whose destination register is still alive in the tail region, it reuses the physical register originally allocated to the instruction rather than allocating a new physical register. BOLT tracks tail liveness using a table that maps each physical register to a unique number, *e.g.*, number of the immediately older checkpoint concatenated with number of instructions since that checkpoint. This table is read and updated by every instruction at rename. An instruction records its destination physical register and that register's tag in its ROB/slice buffer entry—this is the purpose of the *dpreg/tag* fields. A deferred instruction uses its saved destination physical register to re-check the table at re-rename. If the tag in the table matches the tag in its slice buffer entry, the originally allocated physical register is still alive. Note, the destinations of deferred instructions that re-dispatch out of the ROB are always alive. These instructions are always re-assigned the same physical registers and, in effect, can skip re-renaming altogether.

Tracking destination register liveness at the granularity of a single instruction is important because it allows tail renaming and slice re-renaming to proceed in parallel, essentially exploiting TLP (thread level parallelism).

4. Energy-Efficient Slice Processing

BOLT provides an energy-efficient substrate for LT, but energy-efficient LT also requires deferral and re-execution policies that maximize performance while minimizing execution overhead. RA has two such policies [14]. First, it remembers loads whose RA episodes expose no additional MLP and thus represent pure overhead, and suppresses RA during future encounters with these loads. Second, when an RA episode executes X instructions, future episodes are suppressed until X instructions commit. The first policy limits *useless* RA. The second limits *redundant* RA, *i.e.*, it limits every dynamic instruction to executing in RA mode at most once. In an additional advance over previous LT designs, we replicate these policies in BOLT.

4.1. Avoiding Useless LT

Recognizing and avoiding useless activity in LT is more difficult than doing the same in RA. For one thing, the defintion of utility is less clear—LT execution can be useful if it exposes ILP and no additional MLP. For another, LT's non-episodic nature makes it difficult to identify dynamic instruction ranges over which utility should be measured. Because measuring utility directly and correlating it with particular loads is difficult, BOLT takes a different approach. Specifically, it recognizes idioms that produce useless LT and suppresses LT when it detects them. The classic idiom that produces useless LT is pointer-chasing in which multiple loads in the chain miss in LLC. Pointer-chasing exposes no MLP because the misses in the chain are taken serially. It produces little to no ILP because pointer-chasing loads usually form an inductive chain and many instructions in the ensuing loop and function bodies depend on them. When a load in the chain misses, all of these instructions will defer leaving few to retire speculatively. Post-chain MLP and ILP are also difficult to exploit as these are typically masked by a mis-predicted loop control branch (*e.g.*, `while (node != NULL)`). Multiple misses in the chain will cause large sub-slices to re-defer and re-dispatch repeatedly. In the worst case, if every load in a chain of N loads misses, LT will collectively defer and re-execute the loads $N^2/2$ times.

Recognizing and avoiding pointer-chasing. To recognize pointer-chasing, BOLT attaches a *dependent-load bit* to each physical register and store queue entry. The dependent-load bit is propagated along with the poison bit through data dependences, and like poison it is re-calculated at each execution. The dependent-load bit of an instruction and its destination physical register is the OR of the dependent-load bits of its inputs. A dependent-load bit is initially set to 1 when BOLT exe-

cutes a load that is poisoned via its address-register input. When BOLT encounters a load that is poisoned via its address-register input *and* that input already has the dependent-load bit set, it recognizes pointer-chasing. It responds by preventing the load from issuing and deferring to the ROB and stalling tail dispatch. Tail dispatch resumes when the miss the load depends on returns and the load executes.

BOLT's single-bit dependent-load scheme identifies any two-deep chain of loads that depends on a pending LLC miss as pointer chasing. Concretely, in the chain of loads A→B→C where A is an LLC miss, B sets the initial dependent-load bit, and C signals pointer-chasing. The rationale behind choosing this definition is that LT is inefficient under pointer-chasing but not under *all* dependent-load idioms. For instance, sparse matrix codes typically feature dependent loads. However, these dependence chains are both short (*i.e.*, A→B) and do not influence control flow, a combination that permits BOLT to exploit post-chain ILP and MLP. BOLT's threshold of two-deep chains disables useless LT on expensive (*i.e.*, long chain) instances of true pointer-chasing while enabling LT on most "benign" dependent-load idioms.

4.2. Avoiding Redundant LT

BOLT's pointer-chasing optimization avoids useless LT but permits redundant LT—multiple deferral and re-execution for the same dynamic instruction—in the case of dependent LLC misses. However, it limits deferral and re-execution to a maximum of two times per affected dynamic instruction.

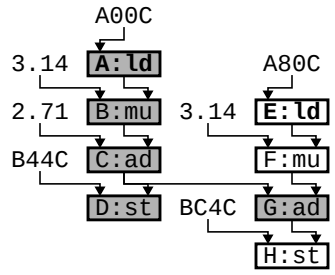
Avoiding redundant LT in the more common case of independent LLC misses is somewhat of a misnomer. It is easy to avoid redundant LT, and CFP largely does. When an LLC miss returns, CFP re-dispatches the entire contents of the slice buffer to the window starting at that miss. This strategy means that most deferred instructions defer and re-execute once. The exception are instructions that depend on multiple LLC misses that return out of program order. Instructions that depend on multiple LLC misses re-dispatch when the first miss returns but immediately re-defer because they have a dangling register input that belongs to a logically older but still-pending miss. However, misses generally return in program order and so this phenomenon is limited to a few programs. The bigger problem with the CFP approach is that it re-dispatches instructions that depend on misses that will not return in the near future. This reduces performance by increasing issue queue occupancy and potentially delaying either dispatch of new instructions at the tail or re-dispatch of instructions that depend on the returning miss. It is possible to relieve this contention and improve performance by immediately re-defering instructions that depend on still pending LLC misses. However, this results in high overhead in the form of redundant LT. The challenge is really to avoid both delaying slice execution and introducing contention between the tail and the slice in a way that does not involve excessive re-deferral.

Re-dispatch throttling. One option is to throttle slice buffer passes and stall them on any LLC miss that has not returned. Tail dispatch proceeds while slice re-dispatch stalls, although we stall tail dispatch also if issue queue or physical registers are low in order to prevent resource inversions that require squashing tail instructions to resolve. In practice, such throttling reduces performance while increasing execution overhead. Performance degrades because miss-dependent instructions that are younger than still-pending misses do not dispatch and re-execute quickly enough. Overhead increases because the tail continues executing and defers miss-dependent instructions it would otherwise be able to execute itself had slices executed faster.

Miss pruning. A more effective strategy is to combine throttling with *miss pruning*. Here, a given slice buffer pass stalls at a miss that has not returned only if that miss is expected to return in the near future. If a miss is not expected to return shortly, the slice buffer pass skips it and does not re-dispatch instructions that depend on it. Predicting when misses will return is accomplished by attaching timestamps to MSHRs. Miss pruning uses a global per-logical-register *antidote* (*i.e.*, anti-poison) bitvector that tracks registers that depend on returning misses. A deferred instruction re-dispatches only if it is poisoned along one of its inputs *and* the antidote bit is set for the corresponding register.

Figure 2 demonstrates miss pruning via antidotes—the actions shown take place at the *antidote* stage in Figure 1. Figure 2a shows a deferred dataflow graph consisting of eight instructions A–H. Loads A and E are LLC misses. Figure 2b shows these eight instructions as they initially appear in the slice buffer. Each slice buffer entry contains the instruction, a value (*Val*), and poison bits for the left and right operands (*L* and *R*). Instruction G is poisoned via both inputs. All other instructions are poisoned via one input and have captured their other input in the slice buffer's value field. Again, the present left inputs of misses A and E are their addresses, the poisoned right inputs represent memory.

Figure 2c shows the slice buffer pass corresponding to the return of load A's LLC miss. Instructions that re-dispatch are shaded. This figure shows the modified contents of the slice buffer—only the captured values



(a) deferred dataflow graph

	Instruction	Val	L	R
A	ld [r10] -> f1	A00C	0	1
B	mu f0,f1 -> f2	3.14	0	1
C	ad f3,f2 -> f3	2.71	0	1
D	st f3 -> [r11]	B44C	0	1
E	ld [r10] -> f1	A80C	0	1
F	mu f0,f1 -> f2	3.14	0	1
G	ad f3,f2 -> f3		1	1
H	st f3 -> [r11]	BC4C	0	1

(b) initial slice buffer contents
(instruction & value/poison-bits)

Val	L	R	0	1	2	3	re-dispatch?	antidote
~~A00C~~	0	**0**	0	**1**	0	0	yes	set f1
~~3.14~~	0	**0**	0	1	**1**	0	yes	set f2
~~2.17~~	0	**0**	0	1	1	**1**	yes	set f3
~~B44C~~	0	**0**	0	1	1	1	yes	–
A80C	0	1	0	**0**	1	1	no	clear f1
3.14	0	1	0	0	**0**	1	no	clear f2
6.62	**0**	1	0	0	0	**0**	yes	clear f3
BC4C	0	1	0	0	0	0	no	–

(c) slice buffer pass for returning miss A
(antidote bitvector & value/poison-bits)

Figure 2. Minimizing re-dispatch of deferred instructions using miss and join pruning.

and poison bits are modified—and the contents of the antidote bitvector for registers f0–f3. At the beginning of a slice buffer pass, the antidote vector is clear. Returning load A re-dispatches, clears its memory poison bit (*R*) and sets the antidote bit for its destination register, f1. B re-dispatches because it is poisoned along its f1 input and the antidote bit for f1 is set. B clears its own poison bit and propagates the antidote to its own destination register f2. C and D follow.

Arriving at load E, BOLT determines that the miss will not return soon and decides not to re-dispatch E. Poisoned instructions that do not re-dispatch during the current pass clear the antidote bit corresponding to their destination registers so E clears f1's antidote bit. Because the f1 antidote bit is clear, F does not re-dispatch as well and clears the antidote bit for its destination f2.

Join pruning. Instructions G and H illustrate another slice processing optimization. G joins the forward slices of misses A and E. G and its forward slice (here only H) belong to the forward slices of both loads. However, if only miss A returns, then only G should re-dispatch. H should not re-dispatch despite technically belonging to A's slice. G, the join instruction, needs to re-dispatch so that it can pick up its left input f3 and store it in its slice buffer entry while it waits for its right input f2 to become available. But H can neither execute nor pick up any values until G actually executes and produces one.

The antidote scheme performs this *join pruning* optimization. An instruction with two poison inputs re-dispatches if the current slice buffer pass supplies the antidote for *any* input. However, the antidote propagates to the instruction's destination register only if *all* poison inputs have antidotes. Here, G re-dispatches because f3's antidote bit is set. But because f2's antidote bit is clear, G clears the antidote bit for f3. This prevents H from re-dispatching during the current pass.

Handling store-load dependences. To prune as many instructions as possible during a given slice buffer pass, BOLT must track antidote status through store-load dependences. BOLT determines whether to re-dispatch a forwarding load by reading the current poison status of its forwarding store from the store queue—the forwarding store's store queue index is saved in the load's slice buffer entry. A clear poison bit means the store re-dispatched during the current pass and is interpreted as the store having set the antidote bit on its own memory "destination." In this case, the load re-dispatches and propagates the antidote to its own destination register. A set poison bit on the store means that the store either did not re-dispatch during the current pass or that it re-dispatched and re-deferred. In either case, the load does not re-dispatch and clears its destination's antidote bit.

Pipelined sparse slice buffer traversal. BOLT does not compact the slice buffer during each pass and the ROB portion of the slice buffer is sparse to begin with. Antidote pruning only has to inspect poison instructions and un-executed instructions that may execute and become poison. It can skip over "holes" in the slice buffer and ROB. The ability to traverse a sparse ROB/slice buffer efficiently accelerates passes and reduces the impact of LLC misses that return out of program order. BOLT splits off the slice buffer's poison and completion information, organizes it for wide access (*e.g.*, 16 instructions at a time), and pipelines its access with the access to the rest of the slice buffer's "payload." BOLT first accesses the poison and completion bits to determine which instructions need to participate in the antidote calculations. It then performs the antidote calculations for four instructions at a time in the next cycle. Antidote calculations require a register dependence cross-check and so even though they do not read the map table, they are only performed at a moderate bandwidth.

Bpred	48 Kbyte 3-table tagged PPM direction predictor. 2K-entry 4-way set-associative target buffer.
Memory	32 Kbyte, 8-way set-associative, 64 byte line, 3-cycle access instruction and data caches, with 8-entry victim buffers. 2 Mbyte, 16-way set-associative, 128-byte line, 10-cycle access L2 with 8-entry victim buffer. 8 8-entry stream buffers. 400 cycle memory latency to the first 16 bytes, 4 cycles to each additional 16 bytes, 32 cycles to fill an L2 miss. 16 outstanding misses.
Pipeline	14 stages: 3 fetch, 2 decode, 1 rename, 1 dispatch, 1 issue, 2 regread, 1 execute, 1 regwrite, 1 SVW, 1 retire.
OOO	160/160 int/FP physical registers, 32/32 int/FP issue queue entries, 4-way supercalar issue with up to 4 integer, 2 FP, 2 loads, 1 store, and 1 branch per cycle. 3 Kbyte, distance-based memory-dependence predictor.
ROB	128-entry ROB. 64/64-entry indexed load/store queue .
RA	128-entry ROB. 64/64-entry indexed load/store queues. 1 checkpoint. 256-entry forwarding cache.
CFP/BOLT	128-entry ROB. 512/256-entry indexed load/store queues. 8 checkpoints. 256-entry slice buffer.

Table 1. Simulated processor configurations

5. Evaluation

We use cycle-level simulation to compare the performance and energy-efficiency of BOLT to those of three other architectures—a non-LT ROB baseline, RA, and CFP. Table 1 describes each of these configurations. There are two important notes here. First, all architectures use scalable indexed load and store queues. Factoring out load and store queue design facilitates the energy comparison and isolates the effects of the primary LT mechanisms for deferral and re-execution. Second, CFP is modeled as BOLT—ROB and all—but with physical-to-physical slice re-renaming. This setup eliminates the extraneous negative effects of CFP's checkpoint-only substrate, specifically its performance degradation on programs with poor branch prediction.

Benchmarks. BOLT targets single-thread performance and energy-efficiency so our evaluation uses the SPEC2000 benchmarks. The benchmarks are compiled for the Alpha AXP ISA at optimization level -O4. They execute to completion on their training inputs using 2% periodic sampling with 4 million instructions of warmup and 1 million instructions sampled per period. We exclude benchmarks *fma3d* and *sixtrack* because our infrastructure cannot handle them. Table 2 characterizes the benchmarks using L2 misses per 1000 committed instructions (MPKI), MLP, and IPC on the ROB baseline. It also prints speedups for an idealized ROB (I-ROB) and for idealized memory (I-MEM). Our evaluation focuses on those benchmarks (in bold) with more than one L2 miss per 1000 instructions. We report averages over these programs as well as over all of SPEC.

Energy and ED^2 methodology. Our energy-efficiency metric is ED^2 which is invariant under DVFS [13]. We consider a technique to be energy-efficient if it reduces ED^2. We note that a performance technique that increases ED^2 still has value in an energy conscious environment, only in fewer scenarios [8].

Because we are interested in *relative* ED^2, we only need to measure *relative* energy consumption. This is fortunate because measuring absolute energy consumption using bottom-up techniques is difficult. We approximate relative energy using relative area. We justify this approximation by observing that the four architectures we are comparing are similar—all execute instructions in the same out-of-order core—and that the differences between them manifest as a few additional structures, most of which are RAMs.

Assuming 20% of baseline ROB energy is static, we compute the relative static energy for architecture X as $(1+(MA_X + MA_{X-LT})/A_{ROB})\times(T_X/T_{ROB})\times 0.2$. A_{ROB} is ROB's area. MA_X and MA_{X-LT} are the areas of the additional structures of architecture X—MA_X for structures that are always active (*e.g.*, larger load

		ROB		I-ROB	I-MEM
Bench	MPKI	MLP	IPC	%Spd	%Spd
ammp	5.7	1.8	0.53	40	177
applu	5.4	5.4	1.44	52	44
apsi	0.2	7.0	1.80	9	2
art	43.7	9.9	0.44	17	17
equake	3.6	1.6	0.91	115	111
facerec	3.6	8.4	1.47	25	22
galgel	0.1	3.4	3.05	9	3
lucas	0.0	1.9	2.57	10	0
mesa	0.6	6.8	3.01	-2	5
mgrid	0.7	5.0	2.85	9	8
swim	9.5	6.1	1.54	23	20
wupwise	1.4	2.5	2.27	32	36
bzip2	1.0	3.4	2.07	4	35
crafty	0.1	1.2	2.32	11	5
eon	0.0	1.3	2.66	4	0
gap	1.1	3.5	1.42	8	27
gcc	0.9	3.8	1.67	5	18
gzip	0.2	11.7	1.95	1	0
mcf	47.9	2.2	0.09	9	330
parser	1.1	1.2	1.05	8	70
perl	0.1	1.6	1.85	4	4
twolf	0.0	1.4	1.53	5	0
vortex	0.3	1.5	2.31	22	19
vpr	3.9	1.6	0.66	52	100

Table 2. Benchmark characterization

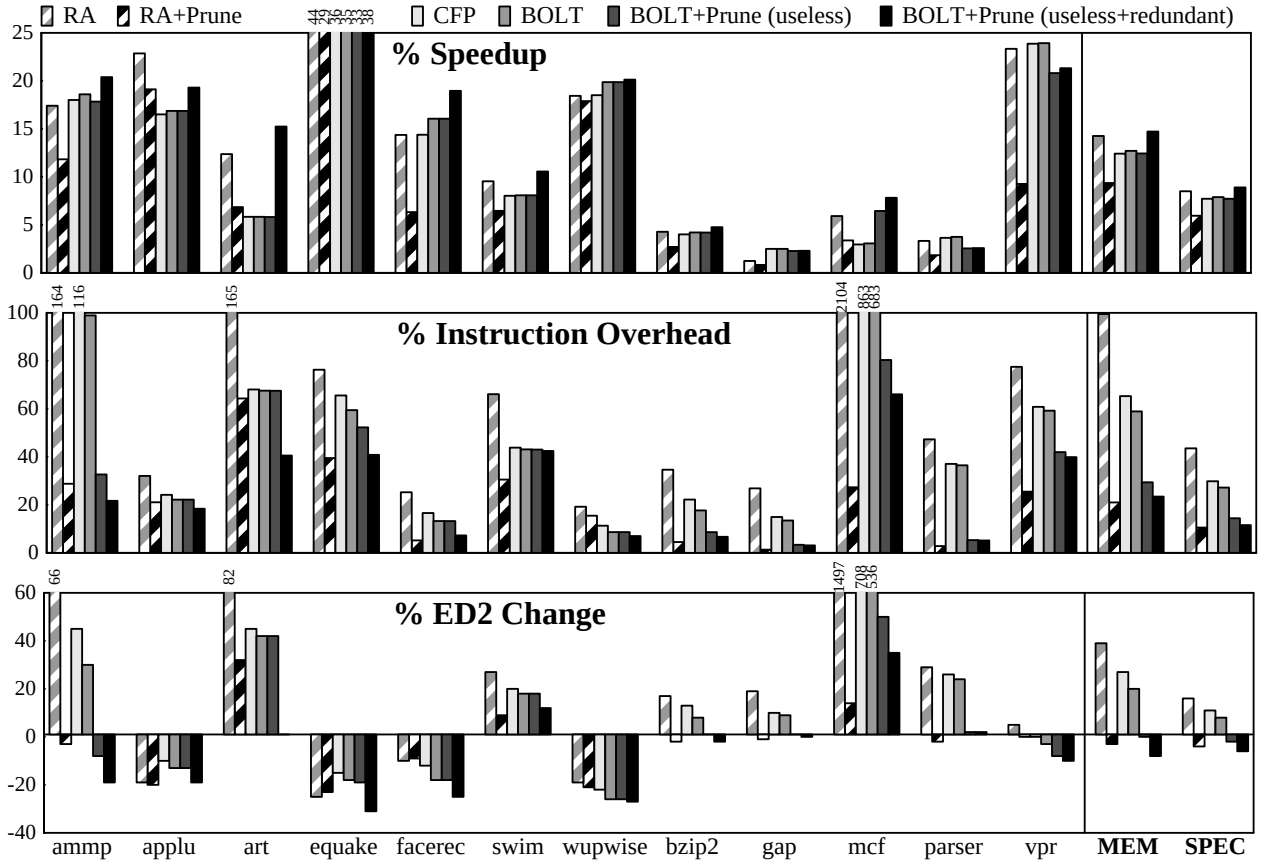


Figure 3. Relative-to-ROB speedup, instruction overhead, and ED^2 for RA, CFP, and BOLT.

and store queues), MA_{X-LT} for structures that activate only during LT execution (*e.g.*, slice buffer). T_X and T_{ROB} are the execution cycle counts on X and ROB, respectively. This calculation assumes that the additional structures leak at the same rate as the average existing structure. The relative dynamic energy calculation is $(1+2\times(MA_X/A_{ROB}))\times(I_X/I_{ROB})\times 0.8 + (2\times(MA_{X-LT}/A_{ROB}))\times(I_X/I_{ROB}-1)\times 0.8$. Here I_X and I_{ROB} are instruction execution counts. For dynamic energy we assume that additional structures consume *twice* as much energy per executed instruction as the average existing structure.

Energy models. We use CACTI-4.1 [23] to estimate structure areas and compute area overheads relative to the area of a Core 2 processor (25 mm^2 in 45nm).

RA's area overhead, register checkpoint machinery (0.03 mm^2) and forwarding cache (0.11 mm^2) occupies a total of 0.14 mm^2. CFP needs more complicated checkpointing machinery (0.06 mm^2), larger load and store queues (0.26 mm^2), a 320-entry physical-to-physical map-table (0.53 mm^2), a 320-entry sequence number table (0.24 mm^2), and a slice buffer/extended ROB (0.21 mm^2). Total area overhead is 1.30 mm^2. The slice buffer is active only during deferral and re-execution and the physical-to-physical map table is active only during re-execution. BOLT resembles CFP but lacks the physical-to-physical map table. Its area overhead is 0.77 mm^2.

5.1. BOLT vs. RA and CFP

The graphs in Figure 3 shows IPC speedups (top), instruction execution overheads (middle), and change in ED^2 (bottom) relative to the ROB baseline. Results for all of SPEC have the same shape as results for the miss intensive benchmarks (*MEM*) but smaller magnitudes because they include many programs with few LLC misses on which RA and LT never activate.

RA and RA+Prune. The two striped bars on the left show un-pruned RA [15] and pruned RA [14], respectively. On MEM, un-pruned RA improves performance by 14% but executes 99% more instructions in the process, increasing ED^2 by 39%. Pruning useless and redundant episodes improves energy-efficiency significantly. Speedup drops to 9%, but overhead drops to 21%, resulting in an ED^2 that is 3% lower than ROB's.

Both raw and pruned, RA is more effective on FP programs, which have higher levels of MLP.

CFP. The four solid bars correspond to LT designs and the first of these is CFP [22]. On MEM, CFP improves performance by an average of 12%, increases instruction execution by 65%, and increases ED^2 by 27%. As expected, (un-pruned) CFP is more energy efficient than un-pruned RA, achieving nearly the same performance while re-executing 34% fewer instructions. RA's performance advantages are due to the fact that it can expose more MLP than CFP. On *equake*, this happens because CFP stalls when the slice buffer fills—it should be possible to default to RA mode execution at this point, but our model does not do this. On *art* and *swim*, CFP's re-execution scheme—which re-dispatches all deferred instructions younger than a returning miss—fills the issue queue and prevents the tail from making progress and launching more misses. CFP's lower overhead reflects its ability to retire rather than re-execute miss-independent instructions.

CFP is significantly less energy-efficient than pruned RA. One component of this is that CFP's performance gains must overcome a higher static energy overhead than RA's—5% rather than 0.5%. The more significant component is increased dynamic energy consumption from useless and redundant LT. On the pointer-chasing program *mcf*, CFP (863% overhead) is more efficient than un-pruned RA (2104%) but far less efficient than pruned RA (27%) which suppresses this behavior.

BOLT. The second solid bar corresponds to the BOLT substrate sans pruning optimizations. Comparing this configuration to CFP isolates the effects of the register management and renaming mechanisms. The switch from execution-order slice buffer and physical-to-physical renaming to program-order slice buffer and logical-to-physical renaming improves performance slightly (by about 0.5% overall and by 2% on some programs), but reduces execution overhead from 65% to 59% and ED^2 increase from 27% to 20%. Overhead reductions come from more efficient register usage during slice re-renaming. With physical-to-physical re-renaming, a slice-buffer pass does not clobber any registers, holding on to all registers it allocates until it completes. In fact, completing a pass sometimes requires squashing some tail instructions to free registers—this problem is magnified on ROB-free CFP where tail instructions can only be squashed at checkpoint granularity. With logical-to-physical re-renaming, a slice buffer pass clobbers registers allowing incremental reclamation and avoiding squashes. This register efficiency—along with avoiding the need to implement a complex new algorithm like physical-to-physical renaming—is the biggest advantage of the BOLT substrate, outweighing the 2% area and static energy overhead saved by removing the physical-to-physical map table itself.

BOLT+Prune. The final two solid bars augment bolt with pruning. Pruning useless LT effectively eliminates CFP's pathological behavior in the presence of pointer-chasing. It reduces execution overhead from 59% to 29% with most of these reductions coming on pointer-chasing programs *ammp* (99% to 32%), *mcf* (683% to 80%), and *parser* (36% to 5%). These dramatic overhead reductions do compromise speedups slightly—*e.g.*, *vpr* speedup decreases from 24% to 21%—because the definition of pointer-chasing as a two-deep load dependence chain does disable some useful LT. A more conservative definition virtually eliminates performance loss, but also admits more overhead on real pointer-chasing. Conversely a more aggressive heuristic reduces overhead but also further dampens speedups, *e.g.*, reducing *equake* speedup from 32% down to 5%. Pruning useless LT reduces the ED^2 penalty to 0% on average, making BOLT as energy-efficient as non-LT execution in the average case.

The final bar adds redundant LT pruning using antidotes. Here, a slice buffer pass pauses at a pending LLC miss if its timestamp indicates a return within 32 cycles. 32 cycles is our L2 fill latency so this is a prediction that the miss will return as soon as the current fill ends. If the miss is not predicted to return quickly, the slice buffer pass prunes it and all instructions that depend on it. The addition of miss and join pruning reduce overhead from 29% to 23% while raising speedups from 12% to 15%. The biggst improvements are seen on *art* where speedup increases from 6% to 15% and overhead decreases from 68% to 41%. In contrast with useless LT pruning, here overhead reductions and performance improvements go hand in hand. Performance improves because slice buffer passes re-dispatch only instructions that will issue quickly, reducing contention with tail dispatch and increasing ILP and MLP. Overhead is reduced because slices re-execute more quickly allowing new instructions that depend on them to execute during their initial pass through the pipeline and avoid deferral.

The combination of the BOLT substrate and the pruning techniques produces an ED^2 that is 8% lower than non-LT ROB execution for the MEM programs and 6% lower across all of SPEC. BOLT has ED^2 that is equal to or lower than that of ROB on 19 of 24 programs, and is at least 10% less efficient than ROB on only two programs, *mcf* (35%) and *swim* (12%). In contrast, CFP is 10% less efficient than ROB on seven programs.

6. Conclusion

LT (latency-tolerant) execution has been proposed as a way of exposing additional ILP and MLP using an out-of-order core without physically scaling critical window structures like the issue queue and register file.

In this paper, we observe that previously proposed LT designs are not energy efficient, requiring too many additional structures and re-executing more instructions than their performance gains justify. To correct the first deficiency, we introduce BOLT, an out-of-order LT design that reuses SMT hardware to re-rename deferred slices. To correct the second deficiency, we introduce a set of mechanisms that recognize and avoid instances of useless and redundant LT execution. Avoiding useless LT is done by explicitly recognizing pointer-chasing on LLC misses. Avoiding redundant LT is accomplished using a new slice buffer traversal method. For an LLC miss intensive subset of SPEC2000, these changes improve LT's performance from 12% to 15%, reduce its execution overhead from 65% to 23% and turn LT from an energy-inefficient technique that increases ED^2 by 27% to an energy-efficient one that reduces ED^2 by 8%. The ED^2 reduction property is significant because it makes LT more generally applicable in energy-constrained settings, *e.g.*, to single-threaded programs.

The slice buffer organization and traversal techniques we introduced are not specific to BOLT, out-of-order LT, or even LT. For instance, they may be useful in TLS (thread level speculation) processors where they can be used to incrementally repair speculative tasks as in Re-Slice [20]. In the immediate future, we plan to investigate their use in in-order LT processors [3, 10] which apply LT to both LLC and data cache misses and whose slice buffers often contain the slices of many independent misses that return out of program order.

7. Acknowledgments

We thank the reviewers for their comments. This work was supported by NSF grant CCF-0541292.

References

[1] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. 36th Intl. Symp. on Microarchitecture*, Dec. 2003.

[2] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proc. 33rd Intl. Symp. on Computer Architecture*, Jun. 2006.

[3] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous Speculative Threading: A Novel Architecture for Chip Multiprocessors. In *Proc. 36th Intl. Symp. on Computer Architecture*, Jun. 2009.

[4] A. Cristal, O. Santana, M. Valero, and J. Martinez. Toward KILO-Instruction Processors. *ACM Trans. on Architecture and Code Optimization*, 1(4):389–417, Dec. 2004.

[5] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proc. 1997 Intl. Conf. on Supercomputing*, Jun. 1997.

[6] J. Emer. Simultaneous Multithreading: Multiplying Alpha's Performance. Microprocessor Forum, Oct. 1999.

[7] A. Glew. MLP Yes! ILP No! 8th Conf. on Architectural Support for Programming Languages and Operating Systems, Wild and Crazy Ideas Session, Oct. 1998.

[8] M. Hill and M. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, Jul. 2008.

[9] A. Hilton, N. Eswaran, and A. Roth. CPROB: Checkpoint Processing with Opportunistic Minimal Recovery. In *Proc. 18th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep. 2009.

[10] A. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating All-Level Cache Misses in In-Order Pipelines. In *Proc. 15th Intl. Symp. on High Performance Computer Architecture*, Feb. 2009.

[11] A. Hilton and A. Roth. Decoupled Store Completion and Silent Deterministic Replay: Enabling Scalable Memory Systems for CPR/CFP Processors. In *Proc. 36th Intl. Symp. on Computer Architecture*, Jun. 2009.

[12] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proc. 29th Intl. Symp. on Computer Architecture*, May 2002.

[13] A. Martin, M. Nystroem, and P. Penzes. ET2: A Metric for Time and Energy Efficiency of Computation. Technical Report CSTR:2001.007, CalTech, 2001.

[14] O. Mutlu, H. Kim, and Y. Patt. Techniques for Efficient Processing in Runahead Execution Engines. In *Proc. 32nd Intl. Symp. on Computer Architecture*, pages 370–381, Jun. 2005.

[15] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *Proc. 9th Intl. Symp. on High Performance Computer Architecture*, pages 129–140, Feb. 2003.

[16] M. Pericas, A. Cristal, F. Cazorla, R. Gonzalez, D. Jimenez, and M. Valero. A Flexible Heterogeneous Multi-Core Architecture. In *Proc. 12th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2007.

[17] M. Pericas, R. Gonzalez, D. Jimenez, and M. Valero. A Decoupled KILO-Instruction Processor. In *Proc. 12th Intl. Symp. on High Performance Computer Architecture*, Feb. 2006.

[18] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proc. 32nd Intl. Symp. on Computer Architecture*, Jun. 2005.

[19] A. Roth. Physical Register Reference Counting. *Computer Architecture Letters*, 7(1), Jan. 2008.

[20] S. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. Re-Slice: Selective Re-Execution of Long-Retired Misspeculated Instructions using Forward Slicing. In *Proc. 38th International Symp. on Microarchitecture*, Dec. 2005.

[21] T. Sha, M. Martin, and A. Roth. Scalable Store-Load Forwarding via Store Queue Index Prediction. In *Proc. 38th Intl. Symp. on Microarchitecture*, Nov. 2005.

[22] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proc. 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[23] D. Tarjan, S. Thoziyoor, and N. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett-Packard Labs Technical Report, Jun. 2006.

[24] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. 22nd Intl. Symp. on Computer Architecture*, Jun. 1995.

[25] H. Zhou. Dual-Core Execution: Building a Highly Scalable Single Thread Instruction Window. In *Proc. 2005 Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2005.