Long Term Parking (LTP): Criticality-aware Resource Allocation in OOO Processors

Andreas Sembrant, Trevor Carlson, Erik Hagersten, David Black-Shaffer, Arthur Perais, André Seznec, Pierre Michaud

▶ To cite this version:

Andreas Sembrant, Trevor Carlson, Erik Hagersten, David Black-Shaffer, Arthur Perais, et al.. Long Term Parking (LTP): Criticality-aware Resource Allocation in OOO Processors. International Symposium on Microarchitecture, Micro 2015, Dec 2015, Honolulu, United States. pp.11. hal-01225019

HAL Id: hal-01225019
https://inria.hal.science/hal-01225019v1

Submitted on 5 Nov 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Long Term Parking (LTP): Criticality-aware Resource Allocation in OOO Processors

Andreas Sembrant[1], Trevor Carlson[1], Erik Hagersten[1], David Black-Shaffer[1], Arthur Perais[2], André Seznec[2], and Pierre Michaud[2]

[1]Uppsala University
{andreas.sembrant, trevor.carlson, erik.hagersten, david.black-schaffer}@it.uu.se

[2]IRISA/INRIA
{arthur.perais, andre.seznec, pierre.michaud}@inria.fr

ABSTRACT

Modern processors employ large structures (IQ, LSQ, register file, etc.) to expose instruction-level parallelism (ILP) and memory-level parallelism (MLP). These resources are typically allocated to instructions in program order. This wastes resources by allocating resources to instructions that are not yet ready to be executed and by eagerly allocating resources to instructions that are not part of the application's critical path.

This work explores the possibility of allocating pipeline resources only when needed to expose MLP, and thereby enabling a processor design with significantly smaller structures, without sacrificing performance. First we identify the classes of instructions that should not reserve resources in program order and evaluate the potential performance gains we could achieve by delaying their allocations. We then use this information to "park" such instructions in a simpler, and therefore more efficient, Long Term Parking (LTP) structure. The LTP stores instructions until they are ready to execute, without allocating pipeline resources, and thereby keeps the pipeline available for instructions that can generate further MLP.

LTP can accurately and rapidly identify which instructions to park, park them before they execute, wake them when needed to preserve performance, and do so using a simple queue instead of a complex IQ. We show that even a very simple queue-based LTP design allows us to significantly reduce IQ (64 → 32) and register file (128 → 96) sizes while retaining MLP performance and improving energy efficiency.

.

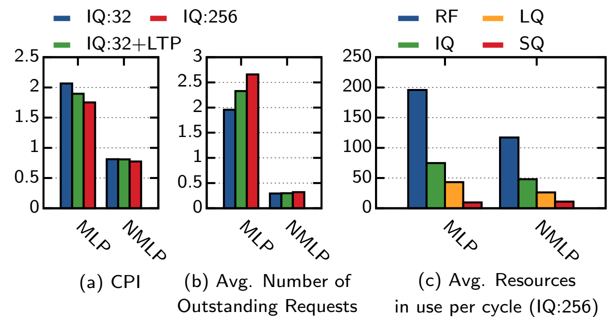

Figure 1: Impact of IQ size on MLP-sensitive and MLP-insensitive execution. (With infinite RF, LQ, SQ, MSHRs, and prefetcher enabled.)

1. INTRODUCTION

Today's out-of-order processors rely on large instruction queues (IQ), registers files (RF), load queues (LQ) and store queues (SQ) to expose significant ILP and MLP. Unfortunately, these resources are some of the most critical and energy-expensive structures in the processor [1, 2, 3, 4], requiring numerous read and write ports and content-addressable lookups. Typically the sizes of these structures are balanced to accommodate the number of instructions the processor must have inflight to uncover sufficient parallelism to keep the execution units and memory system busy. However, entries in these resources are generally allocated in program order, effectively giving all instructions equal priority.

In this work, we investigate the potential of using a simple approach to park instructions that do not contribute to MLP early in the pipeline, and thereby make more room for those that will. This allows us to reduce the size of pipeline resources without sacrificing the ability to expose MLP, and by parking the instructions early in the pipeline, rather than extracting them once they stall, we simplify both the design and integration with existing architectures.

Intuitively, the size of the instruction queue (IQ) should have a first-order impact on the availability of MLP and

ILP in an out-of-order processor. Figure 1 explores this relationship by comparing MLP sensitive and MLP insensitive applications from SPEC2006[1]. In Figure 1a we see that as we increase the IQ size from 32 (left, blue) to 256 (right, red) the MLP-sensitive applications speed up by 18%, while the MLP-insensitive ones hardly change. The speedup of the MLP-sensitive applications is not surprising given that such a dramatic increase in IQ size delivers a 35% increase in the average number of outstanding memory requests (Figure 1b). However, we do not see a corresponding increase for the MLP-*in*sensitive applications, even though we have significantly increased the potential to uncover ILP.

When we examine the resource usage for the MLP-insensitive applications at IQ size 256 (Figure 1c) we see that those applications are unable to take advantage of the additional resources. On average, the MLP-insensitive applications use only 48 of the IQ slots, while the MLP-sensitive ones use 75. The results are similar for registers (117 vs. 196) and the load queue (26. vs 43). The insight from this is that an IQ of size 32 is sufficient to extract nearly all of the ILP performance, and it is primarily MLP-sensitive applications that benefit from significantly larger resources.

This drives our question: Can we reduce the size of expensive pipeline resources without hurting MLP performance?

Our approach accomplishes this by identifying the instructions that do not contribute to MLP and "parking" them early in the pipeline in a much simpler, and hence cheaper, structure, the Long Term Parking (LTP). These parked instructions are woken up such that they can issue without unduly wasting pipeline resources or hurting performance. This allows us to make better use of pipeline resources by allowing critical long-latency loads to execute earlier, which in turn allows us to reduce the size of these resources without hurting MLP performance. By parking instructions in the front-end of the processor, we avoid the cost of re-execution and minimize the complexity of integrating our design with existing processors.

This paper makes the following contributions:

- A dynamic categorization of instructions that allows us to assess which instructions can be delayed in the pipeline without affecting performance and when to wake them.

- A limit study on the potential of parking instructions to enable a reduction in the size of costly pipeline resources (instruction queue, register file, load queue, and store queue).

- An analysis of the types of instructions that are worth parking and the complexity of parking and waking them.

- A evaluation of a simple, queue-based LTP that reduces the IQ size by 50% ($64 \rightarrow 32$), RF size by 25% ($128 \rightarrow 96$), and improves ED^2P for a large out-of-order processor, without requiring significant changes to the backend pipeline or microarchitecture.

[1]Simulation and selection details are described in Section 4.1. Prefetcher enabled.

To investigate LTP, we begin by first describing how to characterize instructions in order to identify those that should be "parked" to expose more useful MLP (Section 2). However, parking instructions is only part of the problem, as we also need to know when to wake them up to avoid a performance penalty from delaying them. To determine the wakeup criteria for different categories of instructions, we investigate how they use the pipeline resources and where the processor is likely to stall (Section 3). With this background, we then undertake a limit study into the potential benefits from the LTP approach, looking particularly at the scaling of the IQ, RF, LQ, and SQ (Section 4). Finally, we use the results of the limit study to apply LTP to a large out-of-order processor and evaluate implementation issues, performance, and energy (Section 5).

2. CHARACTERIZING INSTRUCTIONS

To identify instructions to be parked, we first categorize them in terms of two independent characteristics: their *readiness* to execute and the *urgency* with which they need to be executed to enable other long-latency instructions. Readiness is a function of whether an instruction depends on results from a long-latency instruction, such as an LLC cache miss, division, or square root. If an instruction is *Ready* (R) then it does not depend on any such long-latency instructions and will execute quickly once it is placed in the pipeline. *Non-Ready* (NR) instructions will wait in the pipeline for a long-latency instruction to complete before they can themselves execute. Whether an instruction is *Ready* (R) or *Non-Ready* (NR) is a function of whether the instruction is a *descendent* of a long-latency instruction or not. Conversely, urgency is a function of whether the instruction is an *ancestor* to a long-latency instruction. That is, an instruction is *Urgent* (U) if it has a long-latency instruction that depends upon it and *Non-Urgent* (NU) if it has none.

These definitions give us an intuitive understanding of how we should treat different classes of instructions to avoid consuming pipeline resources for instructions that do not benefit MLP:

- *Urgent + Ready* (U+R) instructions should be placed into the pipeline to execute as quickly as possible since other long-latency instructions depend on them and they will themselves execute rapidly. Address generation generally falls into this category if the dependent load is a cache misses.

- *Urgent + Non-Ready* (U+NR) instructions need to be executed as soon as their long-latency parent instructions are done, but there is no point in placing them into the pipeline immediately as we know they can not execute. Pointer chasing loads that miss in the cache fall into this category.

Loop Code

```
for (i=0; i<10,000; i++) {
   d = B[A[j--]];
   C[i] = d + 5;
}
```

Classification

instruction	class	hit/miss
loop:		
A addrA = baseA + j	U+R	
B t1 = load addrA	U+R	hit
C addrB = baseB + t1	U+R	
D d = load addrB	U+R	**miss**
E j = j – 1	U+R	
F d = d + 5	NU+NR	
G addrC = baseC + j	NU+R	
H store d -> addrC	NU+NR	hit
I i = i + 1	NU+R	
J t2 = i – 10000	NU+R	
K bltz, t2, loop	NU+R	

Figure 2: Example loop and LTP classification of its instructions.

- *Non-Urgent + Ready* (NU+R) instructions will execute rapidly once they are placed in the pipeline, but they do not enable any other long-latency operations. If *Non-Urgent* instructions are issued too early, they will tie up pipeline resources until they commit. Loop counters (that are not used as indices by *Urgent* instructions) and predictable branches tend to fall into this category.

- *Non-Urgent + Non-Ready* (NU+NR) instructions depend on a long-latency instruction but do not enable any other long-latency instructions. Store instructions often fall into this category as they have no directly dependent instructions.

To categorize instructions we use the property that both *urgency* and *readiness* come from an instruction's ancestor/descendent relationship to long-latency instructions. To learn these relationships, we propagate dependency information forwards (*Non-Ready*) or backwards at each iteration (*Urgent*), starting with long-latency instructions. This allows us to efficiently mark the instruction trees (slices) that are ancestors to long-latency instructions (*Urgent*) and descendants (*Non-Ready*) by recording the status of an instruction, and then propagating this status to the instructions that generate its sources or consume its result in the next iteration.

With this characterization, we expect that MLP-sensitive applications will be able to "park" a significant fraction of their non-MLP-generating instructions in the cheaper Long Term Parking, which will allow us to reduce the size of the expensive pipeline structures. This should not hurt performance as long as the parked instructions can be woken up and executed in a reasonably timely manner, since they would have been stalled waiting on the long-latency operations regardless.

For MLP-insensitive applications we expect that there will be few *Urgent* instructions if the application has few long-latency memory accesses, resulting in nearly

Traditional IQ (4 entries)

iter.	instruction	
i1	F	d = d + 5
i1	H	store d -> addrC
i2	F	d = d + 5
i2	H	store d -> addrC

(a) After two loop iterations have completed, the IQ is filled with instructions that are waiting on the result of long-latency loads, causing the processor to stall.

LTP IQ (4 entries)

iter.	instruction	
i3	A	addrA = baseA + j
i3	B	t1 = load addrA
i3	C	addrB = baseB + t1
i3	**D**	**d = load addrB**

(b) After two loop iterations have completed, the IQ will be empty, as the instructions that would have waited on the long-latency loads are in the LTP (below). As a result, the third iteration can begin to issue its urgent instructions to the IQ to begin execution.

LTP (12+ entries)

iter.	instruction		class
i1	F	d = d + 5	NU+NR
i1	G	addrC = baseC + j	NU+R
i1	H	store d -> addrC	NU+NR
i1	I	i = i + 1	NU+R
i1	J	t2 = i – 10000	NU+R
i1	K	bltz, t2, loop	NU+R
i2	F	d = d + 5	NU+NR
i2	G	addrC = baseC + j	NU+R
i2	H	store d -> addrC	NU+NR
i2	I	i = i + 1	NU+R
i2	J	t2 = i – 10000	NU+R
i2	K	bltz, t2, loop	NU+R
i3	…		

(c) After two loop iterations have completed, the LTP will contain the non-urgent instructions from the first two loop iterations, thereby keeping the IQ free to execute more urgent instructions.

Figure 3: Example IQ usage comparing a standard IQ and LTP.

all instructions being characterized as *Non-Urgent* and *Ready*. In that case, the application will park nearly all instructions. To avoid the overhead of parking instructions to no benefit, we can easily detect this scenario and turn off LTP.

The overall effect of the LTP approach is shown in middle (green) bars of Figure 1a and b. Adding LTP to a 32-entry IQ increases MLP by 19% and is able to achieve half of the MLP-benefit of a 256-entry IQ, but without dramatically enlarging the critical pipeline structures or noticeably impacting the MLP-insensitive applications.

2.1 LTP Example

A more detailed example of these effects is shown in Figure 2. Here we start with a simple program that does a series of indirect access ($B[A[j]]$) in a loop and stores them to the array $C[i]$. The indirect accesses to array $B[]$ are assumed to miss in the cache, while those to $A[]$ and $C[]$ are hits due to their prefetch-friendly access patterns.

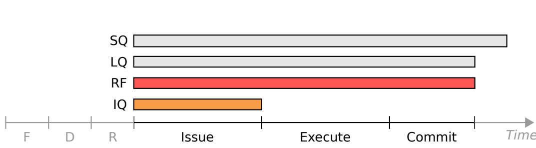

Figure 4: Resource allocation and deallocation in a traditional pipeline.

The classification of the instructions is shown in Figure 2. Here instruction **D** (the load from array $B[]$) is the only long-latency instruction since it is a miss. As a result, instructions **A**, **B**, and **C** will be characterized as *Urgent* because the long-latency load (**D**) depends on them to execute. Instruction **E** will also be characterized as *Urgent* because the long-latency load in the next iteration of the loop depends on it. These five instructions (**A**-**E**) will further be characterized *Ready* since they do not depend on any other long-latency loads.

Instructions **F** and **H** depend on, and must therefore wait for, the long-latency load from array $B[]$, making them *Non-Ready*. The results of these instructions are not used by any other long-latency loads, and are in fact consumed by the store to array $C[]$, which is a hit. As a result these instructions are classified as *Non-Urgent + Non-Ready*. Instructions **G** and **I**-**K** are neither descendants of, nor ancestors to, any long-latency instructions. They will therefore be classified as *Non-Urgent + Ready*.

To see the value of this classification, consider the state of a traditional IQ after two iterations of the loop have been issued, as shown in Figure 3a. Here the IQ is filled with the instructions from the first two iterations that depend on long-latency loads, since these instructions are not ready to execute. This fills the IQ and stalls the processor.

With LTP we improve this situation by only placing instructions that expose more MLP in the IQ. Figure 3b shows the state of the IQ with LTP after two iterations have issued. In this case we see the first instructions from the third iteration have space to issue into the IQ. The reason for this is that the IQ will actually be empty after the second iteration has issued, since the *Non-Ready* instructions that would have blocked it have been parked in LTP (Figure 3c) and only *Ready* instructions went to the IQ. As a result, all the instructions in the IQ executed quickly, freeing up slots for the third iteration, thereby increasing MLP. In this example, the performance benefit of LTP is significant: for every additional 6 slots in the LTP we can issue the *Urgent* load instructions for another iteration and increase MLP. In this case, an 18-entry LTP would provide a 2x speed-up, by delivering an MLP of 4 vs. 2 for the IQ-only design.

3. WAKING UP INSTRUCTIONS

We park non-MLP-generating instructions in LTP to avoid consuming pipeline resources (IQ, RF, LQ, SQ) that could otherwise be used by critical instructions to generate more MLP. While the parked instructions have not been placed in the IQ, they have been allocated an entry in the ROB to ensure in-order commit when they do eventually execute. This means that we need to move the instructions from the LTP to the pipeline (wake them up) early enough that they are done executing by the time they reach the head of the ROB, to avoid stalling. Precisely how early, and how we can trigger this wakeup, is a function of the class of instruction. To understand this, we will now look at the pipeline resources that LTP tackles, and how parking *Non-Ready* and *Non-Urgent* instructions affects allocations and deallocations of these resources.

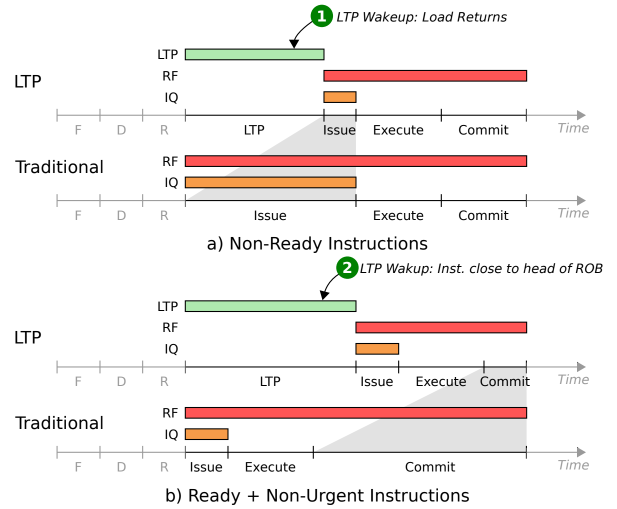

Figure 5: IQ and Register lifetimes of Non-Ready instructions and Ready + Non-Urgent instructions.

3.1 Pipeline Resources

Instruction Queue (IQ). In a traditional pipeline, instructions are dispatched to the IQ after rename in program order. Instructions then wait in the IQ until all of their source operands are marked as ready, after which the scheduler is free to choose them for execution. In this traditional approach, shown in Figure 4, the IQ entry is allocated after rename and freed when the scheduler issues the instruction to execute. As the scheduler may choose instructions to execute out-of-order, this means that while IQ entries are allocated in program order, they are likely to be deallocated out-of-order.

As having *Non-Ready* instructions waiting in the IQ essentially wastes IQ entries, we want to wake them up and move them from LTP to the IQ such that they arrive in the IQ just before the long-latency instruction they are waiting upon finishes. If this is timed correctly, the scheduler will be able to immediately pick and execute them, thereby keeping the time they occupy an IQ entry to a minimum.

The effect of this is shown in Figure 5a. Here the *Non-Ready* instructions stay in LTP until they are woken by the long-latency instruction completing ❶, at which

point they are moved to the IQ. However, since they are now ready to execute, they will quickly leave the IQ, thereby occupying an IQ entry for a much shorter time than in the traditional pipeline.

From the IQ's point of view alone, we should *not* park *Ready* but *Non-Urgent* (R+NU) instructions (Figure 5b) as they will execute quickly, and therefore will only require an entry in the IQ for a short period of time.

Register File (RF). A physical register is allocated for each instruction with a destination at rename when the instruction's architectural destination register is assigned to a physical entry in the register file[2]. However, the register remains unused until the instruction has been scheduled, executed, and finally writes its result to the register. The register is typically freed when the next instruction renaming the same architectural register is committed (to guarantee correctness on squashes).

Unlike the case with IQ entries, instructions that are *Ready* but *Non-Urgent* (R+NU) *should* be parked to avoid occupying RF entries for too long. In Figure 5b we see that because RF entries are not deallocated until much later, parking a *Ready* but *Non-Urgent* instruction will avoid allocating a RF entry for it, and thereby keep more RF entries free for other instructions.

Load and Store Queues (LQ/SQ). As with the register file, load and store queue entries are allocated early in the pipeline and deallocated when the instruction commits[3]. However, as the address may remain unknown until the address generating instructions complete, a *Non-Ready* memory instruction will not be able to execute. To minimize our use of these resources, we should park *Urgent*, but *Non-Ready* memory operations in LTP, and wake them up such that they arrive at the LQ/SQ just when their address is known. While implementing an out-of-order LQ/SQ is decidedly non-trivial, we explore the potential benefits of it in our limit study as the LQ and SQ play a critical role in exposing MLP.

3.2 Wakeup Policy

Non-ready Instructions. We want to wakeup *Non-Ready* instructions from LTP just before the data they are waiting for returns, so that they can be immediately picked by the scheduler. (Figure 5a ❶) Typically, these instructions are waiting on cache misses or long-latency operations such as division or square root. In the case of a cache miss, we can take advantage of the phased L2 and L3 caches to get an early signal to wake up the dependent instruction on a tag hit. This allows us to move *Non-Ready* instructions from LTP to the IQ in time for them to execute without delay upon the arrival of their source data. Similar approaches can be used with the DRAM controller. For operations such as divide and square root, the latency is approximately known and a wakeup signal can be sent at the appropriate time.

[2]Depending on the architecture, instructions may have multiple destination registers allocated at this point.

[3]Stores deallocate their SQ entry after the data has been written back, which typically happen shortly after they commit.

Frequency	3.4 GHz
Width: F / D / R / I / W / C	8 / 8 / 8 / 6 / 8 / 8
ROB / IQ / LQ / SQ	256 / 64 / 64 / 32
Int. / FP Registers	128 / 128
L1 Instruction / Data Caches	32kB, 64B, 8-way, LRU, 4c
L2 Unified Cache	256kB, 64B, 8-way, LRU, 12c
– L2 Prefetcher	Stride prefetcher, degree 4
L3 Shared Cache	1MB, 64B, 16-way, LRU, 36c
DRAM	DDR3-1600 11-11-11

Table 1: Baseline processor configuration. Our proposal reduces the IQ to 32, RF to 96, and adds a 128-entry queue-based LTP.

Non-Urgent Instructions. We want to wakeup *Non-Urgent* instructions as late as possible, but not so late that they end up stalling the ROB. This will minimize their RF and LQ/SQ resource occupancy. To do so, we track long latency instructions in the ROB and use their proximity to the head of the ROB to determine when to wake them. (Figure 5b ❷)

When the ROB stalls, it typically stalls on a long-latency operation, and while it is stalled it can execute many of the instructions after the stalling instruction. When the stalling instruction completes, the ROB can rapidly retire all the entries in the ROB between the first (oldest) stalling instruction and the next stalling instruction in the ROB. This means that we need to wake up *Non-Urgent* instructions early enough that when the long-latency operation blocking the ROB completes, all the instructions between it and the next stalling instruction are ready to be committed.

As a result of this bursty behavior, we wake all *Non-Urgent* instructions backwards from the head of the ROB (oldest first) until the next long-latency instruction. This helps ensure that the ROB will be able to quickly retire as many instructions as it could in a traditional pipeline after the blocking instruction finishes.

4. LIMIT STUDY

To understand the full potential of LTP for reducing the size of key pipeline resources (IQ, RF, LQ, and SQ), we conducted a limit study. For this study we set LTP, cache MSHRs, and all but one of the pipeline resources to an effectively unlimited size and swept the resource in question. This study tells us how each of these resources affect performance in a large processor and the potential of LTP to reduce the size of these resources without hurting performance.

4.1 Methodology

We use the gem5 x86 full-system simulator [5] to evaluate performance. The simulator is configured to be similar to a contemporary high-performance, large-core processor as shown in Table 1. The simulated processor runs at 3.4 GHz, has a 3-level cache hierarchy, and an L2 stride prefetcher.

To evaluate LTP, we use SPECCPU-2006 [6] benchmarks with reference inputs. We use 10 uniformly dis-

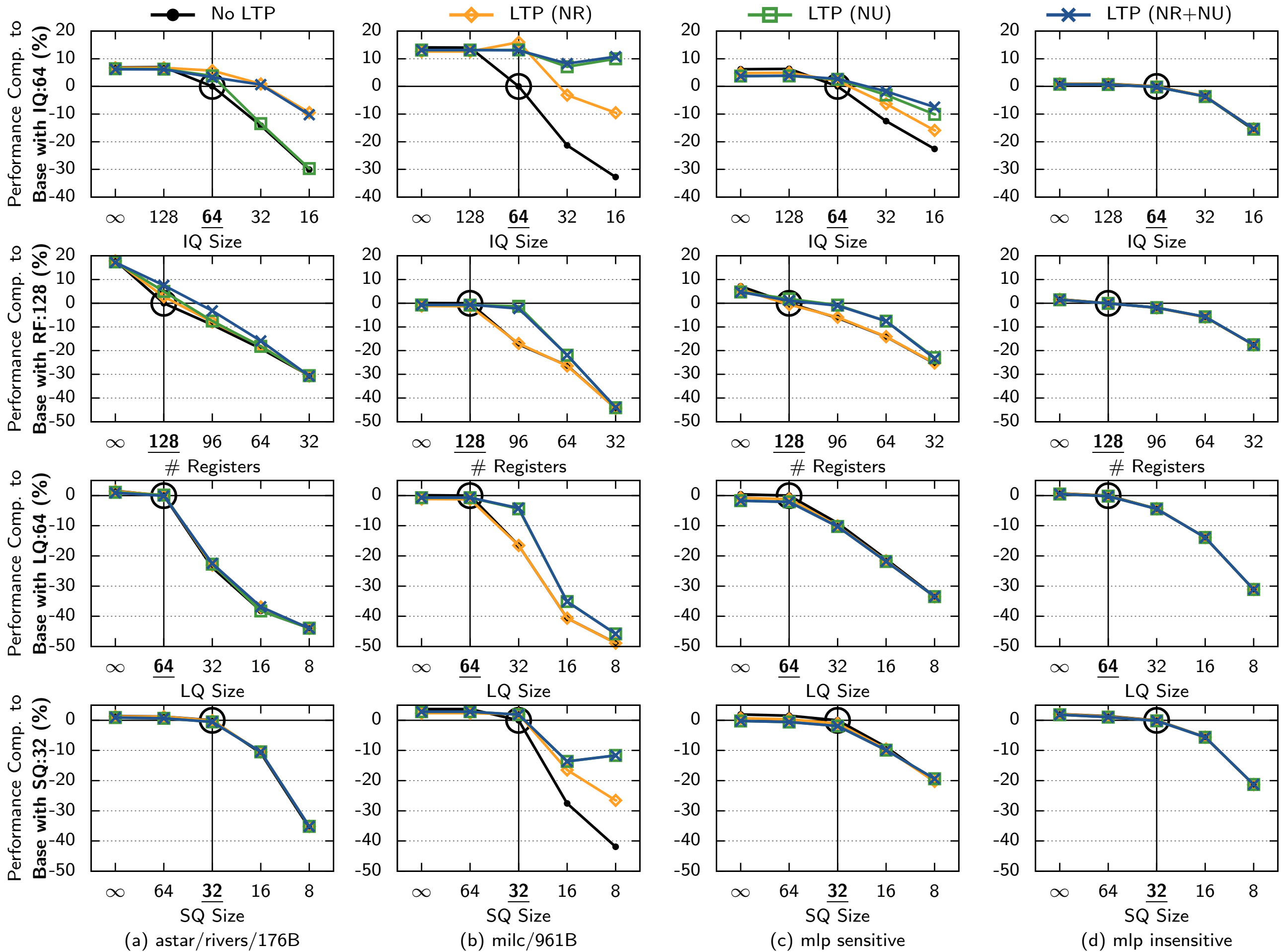

Figure 6: Limit study: Impact of IQ, RF, LQ, and SQ resource sizes with and without LTP. The baseline configuration for each resource is underlined on the x-axis and indicated with a circle. (Prefetcher enabled, unlimited MSHRs.)

tributed simulation points per benchmark and input, for a total of 550 simulation points. For each simulation point the caches are warmed for 250 M instructions, followed by a 100 k instructions of detailed pipeline warming, and then a detailed simulation of 10 M instructions.

To isolate the performance impact of those parts of the applications that are sensitive to MLP, we divide the simulation points into two groups: MLP-sensitive and MLP-insensitive. To identify the sensitive simulation points, we compared the speedup, average cache latency, and number of outstanding memory request per cycle when run on a processor with a 32-entry IQ vs. a processor with a 256-entry IQ. Simulation points that had an average cache latency greater than the L2 latency (mostly L3 and DRAM accesses), and showed more than 5% speedup (benefited from a larger IQ) and 10% more outstanding memory requests with the larger IQ (MLP increased with IQ size) were categorized as MLP-sensitive. These criteria find that 14 of the benchmarks have MLP sensitive phases, spanning more than 11% of the total executed instructions, which, due to the low IPC of MLP-sensitive phases, is around a quarter of the execution time. Note that in both cases the prefetcher is enabled, so applications with regular access patterns are unlikely to be classified as MLP-sensitive.

In the data presented here we show averages for the MLP-sensitive and MLP-insensitive, as well as two individual checkpoints, astar/rivers [cpt:176B] (checkpoint at 176B instructions) and milc [cpt:961B] that demonstrate interesting behavior. Note that for these experiments LTP uses a timer-based DRAM monitor [4] to turn itself off when there are few long-latency loads present, as described in Section 5.2, and an oracle to predict long-latency instructions.

4.2 Limit Study Analysis

Figure 6 shows the performance impact as we sweep each of the resources LTP addresses (IQ, RF, LQ, SQ) while keeping the others unlimited for (top-to-bottom) the IQ (first row), RF (second row), LQ (third row), and SQ (fourth row). The data is normalized to the baseline large processor configuration (Table 1). We do not change the ROB size or width of the processor as we are seeking to evaluate how much we can reduce the struc-

tures LTP can address without hurting performance. For each resource we look at (left-to-right) two simulation points from the benchmarks astar/rivers and milc, as well as the average behavior of the MLP-sensitive and MLP-insensitive benchmarks. For each of these we plot the baseline performance with no LTP, LTP parking only the *Non-Ready* (NR) instructions, LTP parking only the *Non-Urgent* (NU) instructions, and LTP parking both *Non-Ready* and *Non-Urgent* (NR+NU) instructions. For the limit study we model an infinite-sized LTP with perfect instruction classification.

Instruction Queue (Row 1). Decreasing the baseline IQ size from 64 to 32 hurts performance by 13% (MLP-sensitive) and 4% (MLP-insensitive), confirming that an IQ of size 32 is sufficient to expose most ILP. With LTP, the performance at IQ size 32 is only 1.8% lower than the baseline at IQ size 64.

Astar has several instructions that are classified as both *Non-Ready* and *Urgent*. We therefore see a larger performance benefit when LTP parks NR and NR+NU than only NU, since NU will fill the IQ with *Urgent* but *Non-Ready* instructions causing more IQ stalls. With LTP, astar is as fast as the baseline 64-entry IQ.

In contrast to astar, LTP helps milc more by parking *Non-Urgent* instructions than *Non-Ready* instructions. In milc, most *Non-Ready* instructions are also *Non-Urgent*, meaning that parking *Non-Urgent* instructions also parks *Non-Ready* instructions. More instructions are therefore safely parked using *Non-Urgent* analysis since it parks both *Non-Urgent* and *Non-Ready* instructions resulting in a dramatic benefit: with an IQ of only 16-entries, LTP is able to park an average of 6 (NR), 31 (NU) and 39 (NR+NU) instructions, which allows milc to achieve the same performance as the 64-entry IQ baseline.

From these results we can see that parking both *Non-Ready* and *Non-Urgent* instructions is only marginally better than parking *Non-Urgent* alone. The reason for this is that with a *Non-Urgent* only LTP, the only instructions that will not be correctly parked are *Non-Ready + Urgent* instructions. As these are typical only for pointer-chasing code, where LTP can deliver little benefit against the full DRAM latency, we see little gain from handling them. For the MLP-sensitive applications, a 32-entry IQ with LTP (NR+NU) has about the same performance as the baseline with a 64-entry IQ, indicating that an ideal LTP would allow us to cut the IQ in half without hurting performance.

Register File (Row 2)[4]. For the baseline processor design, halving the number of registers from 128 to 64 hurts performance by 14% (MLP-sensitive) and 6% (MLP-insensitive). LTP roughly halves this performance loss at 64 registers, and is able to achieve nearly the same performance at 96 registers as the baseline at 128. LTP has no impact on the MLP-insensitive applications as there are no useful instructions to put into the LTP to reduce register file pressure.

[4]The graphs show the number of available registers, while the physical register file size is typically the sum of the available registers and the number of architectural registers due to how registers are released. For these studies we scale integer and floating point registers in the same manner. We do not investigate status (flag) registers, but it is likely that LTP could see a benefit from them in branch-heavy code.

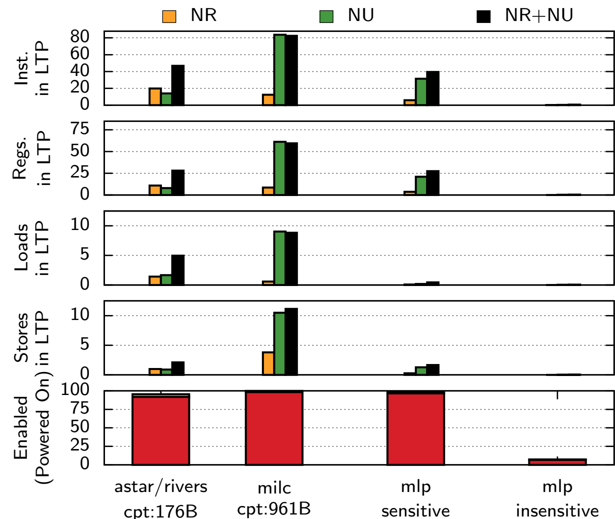

Figure 7: Top: LTP utilization by resource type. Bottom: LTP state (on/off).

MLP-sensitive applications and astar show a smaller benefit to parking *Non-Ready* instructions compared to *Non-Urgent* instructions. There are two reason for this: first *Non-Ready* does not park *Ready* but *Non-Urgent* instructions which steal registers, and, second, *Non-Urgent* but *Non-Ready* instructions are woken up when their data arrives, which can happen before they are close to the head of ROB. As a result, allocating registers to *Non-Urgent* instructions wastes those registers as it may take a long time until they are released (at commit).

Load Queue (Row 3). MLP sensitive and non-sensitive applications see roughly the same impact with LQ scaling, with both needing 64 registers to avoid significant performance loss. On average, most loads are classified as *Urgent* and need to be executed quickly in order to expose MLP, reducing the potential to keep them out of the LQ with LTP. We therefore see only minor benefit with LTP across most MLP-sensitive applications. Milc is an exception, with an average of 8.5 loads parked in LTP, resulting in nearly the same performance at 32 LQ entries as the baseline's 64.

Store Queue (Row 4). As with the LQ, both MLP-sensitive and insensitive applications see roughly the same impact when scaling the SQ, and require 32 entries for good performance. Milc shows a significant benefit of LTP with a very small SQs due to its having an average of 10 stores in LTP, but LTP is unable to match the baseline's performance. On average, there are typically not enough stores parked in LTP to have a significant impact.

LTP Resource Utilization. Figure 7 shows the average number of instructions in an unlimited LTP for a processor with a 32-entry IQ and 96 integer and floating

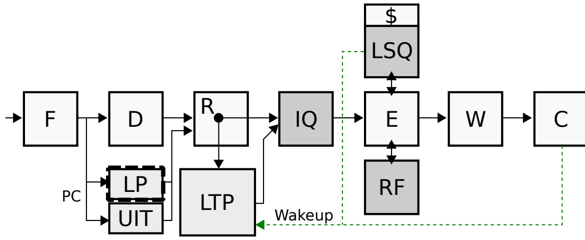

Figure 8: LTP Overview. The Urgent Instruction Table (UIT) (and Long-latency Predictor (LP) for Non-Ready instructions) are accessed early in the pipeline to decide whether instructions should be parked in LTP at the rename stage. Wakeup signals come from completing instructions later in the pipeline.

point registers. We immediately see the reason for the limited impact of LQ and SQ size on LTP: the MLP-sensitive applications tend to have very few parked loads and stores due to most of them being *Urgent*. For the MLP-sensitive applications we see that overall we have an average of 40 instructions accounting for over 25 registers in the LTP (NR+NU), with the *Non-Urgent* instructions playing a far larger role than the *Non-Ready* ones.

Runtime Management. The bottom row in Figure 7 shows the amount of time LTP is enabled based on the presence of long-latency loads (Section 5.2). On average, LTP is enabled for 95% of the execution time in MLP-sensitive applications and only 7% for MLP-insensitive. This indicates that the simple detection mechanism we are using works quite effectively.

4.3 Limit Study Conclusions

The limit study has shown that the simple approach of parking instructions early in the pipeline, before they execute, has significant potential to allow us to reduce the size of expensive pipeline resources, without hurting performance. Specifically, we have shown that:

- An ideal LTP can deliver the same performance as our baseline large processor with an IQ of half the size ($64 \rightarrow 32$) and a 25% smaller register file ($128 \rightarrow 96$).
- LTP does not park enough loads and stores to have a significant impact on the sizes of the LQ and SQ.
- Parking only *Non-Urgent* instructions is nearly as effective as parking both *Non-Urgent* and *Non-Ready* instructions. This is because most *Non-Ready* instructions are also identified as *Non-Urgent*, and the ones that are not are typically pointer-chasing, where we can do little to hide the full DRAM latency.

As a result of this study, we propose a *Non-Urgent*-only design with a 32-entry IQ and 96 registers, and do not attempt to delay allocation for the LQ or SQ. The choice to only address *Non-Urgent* instructions allows us to make the very significant simplification of using a standard queue for the LTP storage, as instructions are only issued from LTP in-order (see Section 5.2). The additional complexity required to handle *Non-Ready* instructions is discussed in the Appendix.

5. A SIMPLE LTP IMPLEMENTATION

To evaluate the effectiveness of our LTP design (simple queue-based LTP, *Non-Urgent* only, IQ 32/RF 96) vs. the baseline (IQ 64/RF 128), we now look at: 1) where to park instructions in the pipeline, 2) how to detect, park, and wakeup *Non-Urgent* instructions, 3) the impact of a finite parking structure and realistic instruction classification, and 4) how these choices come together in terms of performance and energy.

5.1 Where to park instructions

We make the decision as to whether to park instructions at the Rename stage. This allows us to park instructions before resources are allocated and avoid re-execution. To do so, we extend the Register Allocation Table (RAT) to track long latency loads' ancestors (U). The parking decision is then based upon the PC, which is used to query a Urgent Instruction Table (UIT) during decode. (Figure 8.) We send instructions directly to the IQ if they are classified as *Urgent* or to LTP if they are classified as *Non-Urgent*[5]. In both cases, the instruction is recorded in the ROB in-order.

5.2 Parking Non-Urgent Instructions

Detection. To detect *Non-Urgent* instructions we learn which instructions are *Urgent* through Iterative Backward Dependency Analysis [7]. There are two main components: the Urgent Instruction Table (UIT), which indicates whether a particular instruction (PC) is *Urgent*, and an extension to the RAT to propagate urgency backwards to parent instructions:

1) When a long latency load (instruction **C** in Figure 9) is committed, its PC is added to the Urgent Instruction Table (UIT) to indicate that the load is *Urgent*.

2) The urgent status is then propagated backwards to its ancestors by extending the RAT with the PC of the instructions that rename its destination registers. In Figure 9, instruction **B** renames architectural register a1 to physical register p1. The RAT entry for a1 is therefore marked with its PC. When the following instruction **C** looks up the physical register for a1, it also reads the producer's PC **B**. Instruction **B**'s PC is then added to the UIT to indicate that it is *Urgent* since its result is used by the *Urgent* instruction **C**.

3) This process propagates the urgent status further backwards on each execution iteration, and is able to identify 93% of *Urgent* instructions after 4 iterations and >99% after 7 iterations on SPEC [7]. A *Non-Urgent* instruction is simply one that is not present in the UIT.

[5]To handle *Non-Ready* instructions, we need to track the descendants of long latency loads (NR) and then send instructions to the IQ if they are (U and R) and to LTP if they are (NR or NU). See Appendix.

Note that in the absence of long latency loads (i.e., compute-bound) all instructions are classified as *Non-Urgent* since they all miss in the UIT. This waste energy by parking all instructions. To handle this, we use a timer-based DRAM monitor [4] that turns off (power gates) LTP when there are no long latency loads to DRAM. On a demand access that miss in L3, a timer (set to the DRAM latency) is started or restarted, and LTP is enabled. If the timer expires, LTP is turned off.

Parking. When a *Non-Urgent* instruction is detected it is sent to LTP without allocating physical registers. However, the UIT can have conflicts and it takes time to learn all urgent instructions. To avoid deadlocks wherein the IQ fills with instructions that are waiting on others in LTP, we need to make sure that all of the *Non-Urgent* instruction's descendants are sent to LTP. To do so, we extend the RAT with a Parked (P) bit, and park instructions if the UIT determines they are *Non-Urgent* or its source registers have their Parked bit set[6].

Wakeup. *Non-Urgent* instructions are woken up based on their position in the ROB. We wake all *Non-Urgent* instructions between the head (typically the oldest long-latency instruction) and the second long-latency instruction in the ROB. These instructions will typically finish before the blocking long-latency instruction at the head of the ROB and can then be quickly committed when it finishes, thereby only occupying pipeline resources for a short time, but still enabling a high commit throughput. Register allocation happens as instructions leave the LTP, requiring a second RAT for instructions that did not have RAT mappings when they were placed into LTP. The second RAT has the same functionality as a tradictional RAT but renames instructions leaving LTP. (Note that the second RAT requires fewer ports since it only has to match number of LTP ports).

This in-order wakeup has the very important result that the LTP can be implemented as a standard queue. This allows us to park *Non-Urgent* instructions enormously more efficiently than in a standard IQ.

5.3 Memory Dependencies

Modern processors typically have hardware to avoid squashes due to memory order violations by forcing loads to wait until after stores to the same address complete. With LTP we can miss *Urgent* instruction dependency chains that happen through stores and loads to the same address, causing additional squashes. To avoid this, we use the memory dependency unit to classify violating stores as *Urgent* and propagate the parked bit to loads that the dependency unit predicts will be dependent. This forces the dependent load to be parked in the LTP, if its generating store was, and to be woken up when the store completes.

5.4 Avoiding Deadlocks

Instructions deallocate registers, LQ entries, etc., when they commit. If the pipeline runs out of resources that

[6]We need to detect and break artificial dependencies (e.g., zeroing xor in X86) at rename to avoid parking urgent instructions that depend on a false Parked-bit.

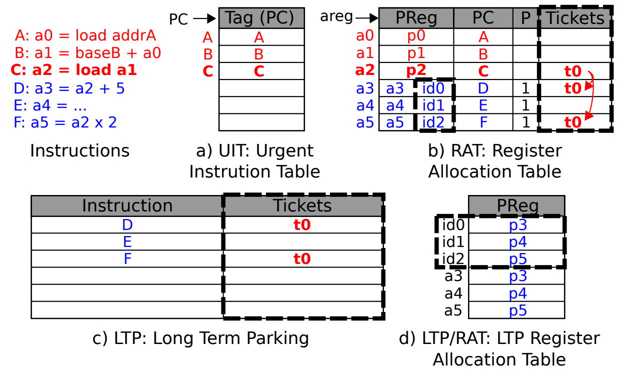

Figure 9: LTP Structures for tracking Non-Urgent and Non-Ready (dark dashed boxes, see Appendix) instructions.

are freed upon commit while the instruction blocking the ROB is still parked in LTP we have a deadlock. To avoid this, we first prioritize renaming instructions from LTP over new instructions. Second, we reserve a set of registers, LQ and SQ entries for parked instructions. Whenever, we start to run out of pipeline resources, we always pick an instructions from LTP (i.e., the oldest instruction since LTP is a queue), since that will free up resources when it is committed.

5.5 Pipeline Resource Costs

Instruction Queue. On an N-way superscalar processor, the IQ must be able to deliver up to N ready-to-execute instructions to the functional units, and manage the readiness of the operands for the instructions. The IQ must wake-up instructions by broadcasting the readiness of operands to all entries on each cycle and select up to N instructions for execution on the next cycle. This selection is an associative search with resource constraints based on the instruction and functional unit mix. Achieving both wake-up of dependent instructions and selection for back-to-back execution is a challenge. As a consequence, the wake-up and selection logic in the IQ is often one of the critical paths determining the cycle time of the processor.

Wake-up delay increases significantly (potentially quadratically [8]) with the number of entries in the IQ. The selection logic delay also increases with the number of entries in the IQ, with a logarithmic increase reported in [8]. Clearly, limiting the number of entries in the IQ is of prime importance to the cycle time.

From a power and energy perspective, the IQ is one of the most power hungry structures of the processors, consuming 18% of the energy according to [9]. As a first order model, the power consumption of the IQ is proportional to the number of comparators in the IQ (entries × issue width), and therefore limiting the number of entries will have a linear impact on its energy consumption. Our proposed LTP replaces a 64-entry IQ with 8 write ports, 6 read ports, and 8 search ports, with a 32-entry IQ with the same width, backed by a 128-entry 4-port queue-based LTP.

Register File. The register file in superscalar processors are typically very expensive due to the need for many ports to deliver operands to multiple simultaneously executing instructions while also receiving results. The cost of the register file scales with the number of entries. For this work we replace a 128-entry register file with 16 read ports and 8 write ports with a 96-entry one.

5.6 Implementation Tradeoffs

The ideal LTP evaluated in the limit study differs from a practical implementation in three key ares: 1) a finite UIT, 2) a limited number of LTP entries, and 3) a limited number of LTP read- and write-ports. Figure 10 shows the performance (top) and energy (bottom) impact of varying the LTP size and number of ports compared to the baseline, for both MLP-sensitive (left) and insensitive (right) applications.

The UIT keeps track of the *Urgent* instructions, and when we shrink it we start to misclassify *Urgent* instructions. This hurts performance since we incorrectly park them, thereby increasing the *Urgent* instruction latency. We found that a UIT of size 256 performed well, with 128 giving up 4 percentage points in performance, and an unlimited UIT only performing 2 percentage points better.

Figure 10 shows that choosing 128 entries for the LTP and 4 ports results in a design that is only 1% slower than the baseline, but has an IQ/RF ED^2P that is nearly 40% lower[7] for MLP-sensitive, and 3% slower with a 38% lower IQ/RF ED^2P for MLP-insensitive.

The red line in Figure 10 shows the effect of removing the LTP (e.g., just IQ 32/RF 96). In this case the performance is noticeably worse for MLP-sensitive applications (nearly as bad as a single-port LTP), resulting in a significantly worse ED^2P savings. However, for the MLP-insensitive applications, while the performance is similar, the ED^2P savings are better than the LTP design as there is no overhead from having the LTP structures. (See Section 5.2 for a description of how we turn off LTP for MLP-insensitive applications.)

5.7 Conclusions: Implementing a Simple LTP

The Limit Study showed the potential of a simple, queue-based, *Non-Urgent*-only LTP to reduce the size of the IQ and RF in a large processor without hurting performance. In this evaluation we have shown that a practical LTP built with a 128-entry, 4-port queue is highly effective (significantly improved efficiency and performance over just reducing the IQ size) for MLP-sensitive applications. The improvement comes from the combination of a much more energy-efficient structure (the queue-based LTP vs. a complex IQ) and the performance gain from LTP's intelligent parking vs. simply reducing the IQ size. For applications that are not MLP-sensitive, the LTP design results in a 3% performance loss, and the added overhead of the LTP support structures results in a lower efficiency improvement than simply reducing the IQ size. This loss is due to our simple DRAM-timer based approach not disabling LTP for 7% of the time during MLP-insensitive phases (Figure 7). Overall LTP achieves its goal of providing a simple approach to enable us to reduce the size of costly pipeline structures on high-performance processors.

[7]Energy has been calculated by using the McPAT/Cacti [10, 11] models for the baseline RF and IQ, scaling them for the LTP design. Results include the overhead of the LTP support structures.

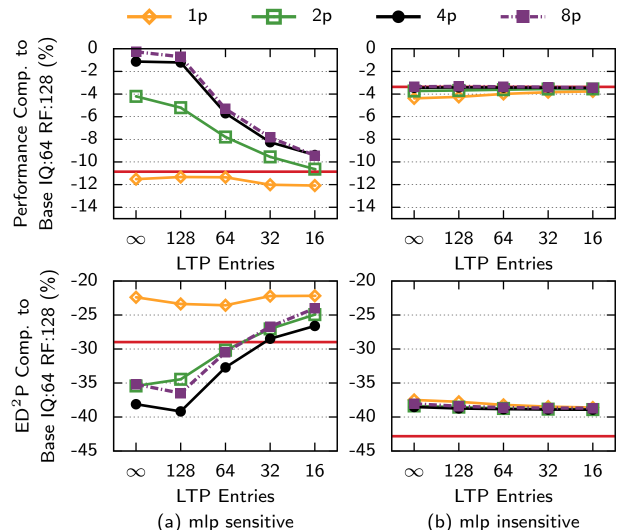

Figure 10: Performance and Energy (IQ/RF) impact of varying LTP size and number of ports for an LTP/IQ 32/RF 96 design relative to the IQ 64/RF 128 baseline. The redline shows the results for IQ 32/RF 96 without LTP.

6. RELATED WORK

There are two main approaches for addressing the memory wall problem in hardware: either decrease the number of long-latency cache misses or make the execution tolerant to the miss latency.

Related solutions for decreasing the number of misses include execution-based prefetching for applications whose memory access streams are not fully prefetchable with conventional hardware prefetchers [12, 13, 14]. Runahead execution is a typical example of such methods [12, 13]. In runahead mode, instructions depending on a miss simply flow through the processor, making room for independent instructions that can execute early and hopefully generate useful prefetches.

Latency tolerance is another way to attack the memory wall problem, generally by increasing the instruction window to exploit more MLP. However scaling up the instruction window increases energy consumption and may impact the clock cycle. Most propositions for latency tolerance exploit the fact that the instruction window enlargement is specialized for memory-bound applications.

Muthler et al. propose a frontend architecture that delays non-critical instructions [15]. There are two very

important differences with LTP: First, they rely on binary annotation from profiling to determine if instructions may be deferred, while LTP works at runtime with no code modifications. Second, they focus on increasing the effective fetch width by fetching deferred instructions as soon as a hole in the fetch stream is detected. LTP issues *Non-Urgent* instructions depending on their position in the ROB to minimize the time they hold pipeline resources, which allows LTP to save pipeline resources to issue more instructions.

Morancho et al. augment the IQ with a Recovery Buffer [16]. Upon an L1 data cache miss, *speculatively issued* instructions depending on the miss are drained from the IQ and are inserted into the recovery buffer for re-execution after the data returns. The same authors subsequently described a scheme wherein load instructions access a hit/miss predictor in the pipeline front end [3] that sends instructions directly to the Recovery Buffer. The authors do not observe any performance gain but significant energy savings. Lebeck et al. introduced the Waiting Instruction Buffer (WIB) [1]. All the instructions dependent on L1 data cache miss are drained from the IQ and inserted into the large WIB where they wait until the miss completes. These works address the IQ scaling problem, whereas LTP targets both RF and IQ based on dynamic *Non-Urgent* (and *Non-Ready*) classification.

There is a large body of latency tolerant proposals that take a holistic approach and redesign significant portions of the backend pipeline. The kilo-instruction processor of Cristal et al. assumes a WIB for instruction scheduling and uses late allocation [17, 18] and early release [19] for physical registers [20]. The Continual Flow Pipelines (CFP) architecture of Srinivasan et al. contains a slice buffer holding instructions depending on L2 cache misses [21]. CFP uses an aggressive register renaming method which allows early recycling of physical registers [19]. Hilton and Roth's BOLT proposal addressed some inefficiencies of the original CFP [22], but retain the slice buffer for re-execution. In particular, they detect pointer chasing situations and disable the slice buffer in this case to prevent needless instruction cycling. These works replace traditional processor pipelines backend with new micro-architectures, whereas LTP augments existing backend architectures with a pre-scheduler in the frontend. It avoids re-executing instructions and can be more simply integrated with existing architectures.

Carlson et al. propose the Load Slice Core (LSC) [7] to improve MLP-sensitive applications' performance on in-order processors. LSC identifies memory instructions and address computations and places them in a separate bypass queue. Loads are therefore able to execute ahead of time. In LSC, resources are allocated for all instructions, whereas LTP identifies non-MLP-generating instructions and parks them to avoid early allocation. Furthermore, LTP wakes up *Non-Urgent* instructions based on their position in the ROB, whereas LSC execute instructions as soon as ready since they do not delay resource allocation.

Kora et al. describe a latency-tolerant IQ that can grow or shrink dynamically [4] based on misses. The degree of IQ pipelining is adjusted depending on the IQ size, the idea being that when L2 misses are frequent, back-to-back execution of dependent instructions is superfluous.

Many papers have proposed ways to make load/store queues larger or less complex. Only few of them specifically target latency-tolerant architectures [23, 24]. Sethumadhavan et al. describe a method allocating for late load/store allocation at issue [25], which uses an *age* CAM is used to recover program order information [25].

There are many ways to define and use instruction criticality. For instance, criticality-aware caching and instruction scheduling [26, 27, 28, 15]. To the best of our knowledge, LTP is the first proposition that uses dynamic instruction urgency for efficient prescheduling and late allocation of instruction window resources.

Since LTP is a frontend improvement, many of the existing latency-tolerance proposals could be combined with LTP to boost backend performance further, but with the cost of additional complexity. The possibility of combining techniques that address different aspects of the problem (e.g., dynamic sizing [4] or larger LQ/SQ [23, 24]) with LTP is an area for future work.

7. CONCLUSIONS

In this work we have explored the potential to reduce the size of the expensive pipeline structures needed to expose MLP without hurting performance on MLP-sensitive code. We have done so by parking *Non-Urgent* instructions in a separate, cheaper Long Term Parking queue until they can be issued without wasting resources or hurting performance. This allows us to only allocate pipeline resources when they are actually needed, and thereby reduce the size of the IQ and RF, thus improving efficiency. Moreover, by parking instructions in the front-end of the processor we avoid the cost of re-execution and minimize the complexity of integrating our design with existing processors.

The main insights from this work are that: 1) Processors only need very limited OoO resources (IQ size 32) to extract ILP, but significantly larger structures for MLP. This results in a very expensive design for the majority of application phases (89% of dynamic instructions) that are not MLP-sensitive. 2) *Non-Urgent* instructions (ones that do not provide results to a long-latency instruction) are the most important ones to park. While performance is slightly better when parking *Non-Ready* ones as well, the *Non-Urgent* analysis covers the majority of the benefit. 3) As *Non-Urgent* instructions are woken from the LTP in program order, we can implement parking storage with a very efficient queue. This allows us to create quite a large LTP structure at a far lower cost than a comparable IQ. 4) Implementing a simple queue-based LTP for *Non-Urgent* instructions allows us to reduce the size of the IQ and RF in a large processor core without hurting performance or efficiency.

Combined, these demonstrate that parking *Non-Urgent*

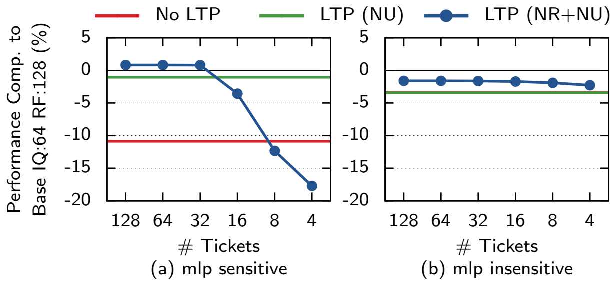

Figure 11: Performance impact of varying number of tickets for an LTP design with both *Non-Urgent* and *Non-Ready*. The red line shows the results for IQ 32/RF 96 without LTP, and the green line shows the results of NU-only with a 128-entry LTP with 4 ports.

instructions in a simple Long Term Parking queue, and waking them when they approach the head of the ROB, can deliver most of the benefit of a large IQ and RF for MLP-sensitive applications without the cost of the larger structures.

APPENDIX

A. PARKING NON-READY INSTRUCTIONS

While our limit study indicates that parking *Non-Ready* instructions is unlikely to contribute enough to be worth the overhead compared to a queue-based LTP that handles just *Non-Urgent* instructions, we implemented a *Non-Ready* LTP to verify these results and investigate what it would take to park and wake them.

There are two key differences between *Non-Urgent* and *Non-Ready* instructions that influence implementation. First, *Non-Ready* instructions are woken up out-of-order when the long-latency instruction they are waiting for completes, while *Non-Urgent* instructions wake up in program order. Second, *Non-Ready* instructions are descendants of a long latency instruction, which means we can identify them as we execute through the RAT, and do not need a separate table for backwards analysis.

Detection. *Non-Ready* instructions are descendants of long-latency instructions, so we must first predict [29, 3] if an instruction is long-latency or not. For variable-latency instructions (e.g., loads) we use a two-level hit/miss predictor that accesses a history table with the last four outcomes of the PC and then hashes these bits with the PC to access the prediction table[8].

After we determine if the instruction is a long latency instruction, a *ticket* is allocated to that instruction. Tickets are tracked through the RAT, and all descendent instructions inherit that ticket and wakeup when the instruction clears that ticket, as seen in the highlighted portion of Figure 9. If a instruction's source registers contain any tickets, we classify that instruction as *Non-Ready*. Note that the Tickets field is a vector of tickets containing all the tickets that the instruction needs to wait for since an instruction can depend on several long latency instructions.

Parking. When a *Non-Ready* instruction is detected, it is sent to LTP without allocating a physical register, but in contrast to *Non-Urgent* instructions where we can keep the architectural register, we need to allocate an internal LTP register ID. These IDs have no physical backing, but are required to make sure that we lookup the right physical register when instructions leave LTP out-of-order. However, we need roughly as many internal LTP register IDs as there are instructions in LTP, which is typically greater than number of architectural registers, thereby requiring a larger $\mathrm{RAT_{LTP}}$. When instructions leave LTP, the $\mathrm{RAT_{LTP}}$ replaces their internal LTP register IDs with physical registers.

Wakeup. Long latency instructions wake up their descendent instructions by sending their ticket to LTP when they are about to finish executing. LTP then clears that ticket for all instructions in LTP. For long latency loads, we take advantage of sequential tag/data cache accesses in the L2/L3 to have the load/store unit clear a load's tickets early on a tag hit. When a *Non-Ready* but *Urgent* instruction's tickets are all cleared it will issue to the IQ and its internal LTP register IDs will be replaced with physical registers using the second RAT. For *Non-Ready* and *Non-Urgent* instructions, we additionally apply the ROB-position criteria as discussed earlier.

Tracking tickets in the LTP requires a CAM or bit matrix [30] that can clear the 1-bit ticket signals and select ready instructions for issuing. While this is more costly than the simple queue for *Non-Urgent* instructions, it is far cheaper than the more flexible structure required for a full IQ. The performance impact of the number of tickets available is shown in Figure 11.

Note that there are at least two options for supporting both *Non-Ready* and *Non-Urgent* instructions: tracking both in the same structure by assigning tickets to *Non-Urgent* instructions and their ancestors, or having a separate queue for *Non-Urgent* and a ticket-based structure for *Non-Ready*.

Summary. Handling *Non-Ready* instructions is much more complex than *Non-Urgent*. First, *Non-Ready* instructions are woken up out-of-order when the long-latency instruction they are waiting for completes, meaning that we can not implement LTP as a standard queue. Second), we need virtual registers to make sure the instructions lookup the right register when they leave LTP out-of-order, which is not needed with a queue-based LTP. Third, the RAT needs to be extended with more meta-data to track which long-latency instructions the register depends on. Finally, the cache tag-arrays need to be extended to send wakeup signals when the data is about to return.

Since, the performance improvement for adding *Non-Ready* instructions does not justify the increase in $\mathrm{ED^2P}$, we recommend a *Non-Urgent*-only solution.

[8]The impact of the predictor vs. an oracle is less than 2 percentage points in performance.

B. REFERENCES

[1] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, “A Large, Fast Instruction Window for Tolerating Cache Misses,” in *Proc. International Symposium on Computer Architecture (ISCA)*, 2002.

[2] D. Ernst, A. Hamel, and T. Austin, “Cyclone: A Broadcast-free Dynamic Instruction Scheduler with Selective Replay,” in *Proc. International Symposium on Computer Architecture (ISCA)*, 2003.

[3] E. Morancho, J. M. Llabería, and A. Olivé, “On Reducing Energy-consumption by Late-inserting Instructions into the Issue Queue,” in *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, 2007.

[4] Y. Kora, K. Yamaguchi, and H. Ando, “MLP-aware Dynamic Instruction Window Resizing for Adaptively Exploiting Both ILP and MLP,” in *Proc. International Symposium on Microarchitecture (MICRO)*, 2013.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 Simulator,” *SIGARCH Comput. Archit. News*, 2011.

[6] J. L. Henning, “SPEC CPU2006 Benchmark Descriptions,” *SIGARCH Comput. Archit. News*, 2006.

[7] T. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, “The Load Slice Core Microarchitecture,” in *Proc. International Symposium on Computer Architecture (ISCA)*, 2015.

[8] S. Palacharla, N. P. Jouppi, and J. E. Smith, “Complexity-effective Superscalar Processors,” in *Proc. International Symposium on Computer Architecture (ISCA)*, 1997.

[9] M. K. Gowan, L. L. Biro, and D. B. Jackson, “Power Considerations in the Design of the Alpha 21264 Microprocessor,” in *Proc. Design Automation Conference (DAC)*, 1998.

[10] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures,” in *Proc. International Symposium on Microarchitecture (MICRO)*, 2009.

[11] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “CACTI 6.0: A Tool to Model Large Caches,” tech. rep., Hewlett Packard Labs, 2009.

[12] J. Dundas and T. Mudge, “Improving Data Cache Performance by Pre-executing Instructions under a Cache Miss,” in *International Conference on Supercomputing (ICS)*, 1997.

[13] 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 *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[14] H. Zhou, “Dual-core Execution: Building a Highly Scalable Single-thread Instruction Window,” in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005.

[15] G. A. Muthler, D. Crowe, S. J. Patel, and S. S. Lumetta, “Instruction Fetch Deferral Using Static Slack,” in *Proc. International Symposium on Microarchitecture (MICRO)*, 2002.

[16] E. Morancho, J. M. Llabería, and A. Olivé, “Recovery Mechanism for Latency Misprediction,” in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[17] S. Wallace and N. Bagherzadeh, “A Scalable Register File Architecture for Dynamically Scheduled Processors,” in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1996.

[18] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Vinals, “Delaying Physical Register Allocation through Virtual-Physical Registers,” in *Proc. International Symposium on Microarchitecture (MICRO)*, 1999.

[19] M. Moudgill, K. Pingali, and S. Vassiliadis, “Register Renaming and Dynamic Speculation: An Alternative Approach,” in *Proc. International Symposium on Microarchitecture (MICRO)*, 1993.

[20] A. Cristal, O. J. Santana, and M. Valero, “Toward Kilo-instruction Processors,” *ACM Transactions on Architecture and Code Optimization*, vol. 1, pp. 368–396, Dec. 2004.

[21] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, “Continual Flow Pipelines,” in *Proc. Internationl Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[22] A. Hilton and A. Roth, “BOLT: Energy-efficient Out-Of-Order Latency-tolerant Execution,” in *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

[23] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai, “Scalable Load and Store Processing in Latency Tolerant Processors,” in *Proc. International Symposium on Computer Architecture (ISCA)*, 2005.

[24] A. Hilton and A. Roth, “Decoupled Store Completion/Silent Deterministic Replay: Enabling Scalable Data Memory for CPR/CFP Processors,” in *Proc. International Symposium on Computer Architecture (ISCA)*, 2009.

[25] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler, “Late-Binding: Enabling Unordered Load-Store Queues,” in *Proc. International Symposium on Computer Architecture (ISCA)*, 2007.

[26] B. R. Fisk and R. I. Bahar, “The Non-Critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency,” in *International Conference on Computer Design (ICCD)*, 1999.

[27] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, “Dynamic Prediction of Critical Path Instructions,” in *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

[28] B. Fields, S. Rubin, and R. Bodík, “Focusing Processor Policies via Critical-path Prediction,” in *Proc. International Symposium on Computer Architecture (ISCA)*, 2001.

[29] B. Calder, D. Grunwald, and J. Emer, “Predictive Sequential Associative Cache,” in *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 1996.

[30] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S.-i. Mori, “A High-speed Dynamic Instruction Scheduling Scheme for Superscalar Processors,” in *Proc. International Symposium on Microarchitecture (MICRO)*, 2001.