R3-DLA (Reduce, Reuse, Recycle): A More Efficient Approach to Decoupled Look-Ahead Architectures

Sushant Kondguli and Michael Huang

Department of Electrical and Computer Engineering
University of Rochester, Rochester, NY
{sushant.kondguli, michael.huang}@rochester.edu

Abstract

Modern societies have developed insatiable demands for more computation capabilities. Exploiting implicit parallelism to provide automatic performance improvement remains a central goal in engineering future general-purpose computing systems. One approach is to use a separate thread context to perform continuous look-ahead to improve the data and instruction supply to the main pipeline. Such a decoupled look-ahead (DLA) architecture can be quite effective in accelerating a broad range of applications in a relatively straightforward implementation. It also has broad design flexibility as the look-ahead agent need not be concerned with correctness constraints. In this paper, we explore a number of optimizations that make the look-ahead agent more efficient and yet extract more utility from it. With these optimizations, a DLA architecture can achieve an average speedup of 1.4 over a state-of-the-art microarchitecture for a broad set of benchmark suites, making it a powerful tool to enhance single-thread performance.

***Keywords*-single thread performance; decoupled lookahead architecture;**

I. Introduction

Modern societies have developed insatiable demands for more computation capabilities. While in certain segments, delivering higher performance via intense human labor (manual parallelization and performance tuning) is justifiable, in many other situations, such effort is not necessarily effective nor is it particularly efficient when the extra resources (energy and loss of productivity) are properly accounted for. Automatic performance improvement remains a central goal in engineering future general-purpose computing systems [13]. After all, the systems have always been designed to increase automation and productivity and to free human from drudgery (including debugging a parallel code).

The two traditional drivers for single-thread performance (faster cycles and advancements in microarchitecture) have all but stopped in recent years – and for good reasons, since further gains from these approaches will come at significant costs. However, the level of implicit parallelism is quite high even in non-numerical codes. The real question is whether we can realize the potential without undue costs. The typical monolithic out-of-order microarchitecture appears to have significant challenges exploiting this level of parallelism. In particular, the instruction and data supply subsystem shoulders a significant responsibility. Fig. 1 illustrates this point. When data and instruction supply subsystem is idealized, the average level of implicit parallelism is significantly higher (about 5x) than when it is not, suggesting the subsystem as a target for improvements.

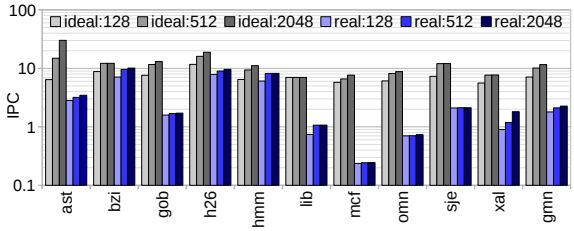


Figure 1. Implicit parallelism of integer applications. The three bars on the left indicate the amount of available parallelism (instructions per cycle) in the application when inspected with a moving window of 128, 512, or 2048 instructions. The three bars on the right repeat the same experiments only this time under a realistic branch misprediction and cache miss situation that further constrains the available parallelism to be exploited. Note that the vertical axis is in log scale.

One possible way of strengthening this subsystem is to use a decoupled look-ahead (DLA) architecture. In DLA, a self-sufficient thread guides the look-ahead activities largely independent of the actual program execution. Simply having a thread for continuous look-ahead is not enough. Many elements have to function properly and efficiently to create a system that can offer sustained, deep, and high-quality look-ahead that will provide significant benefits at acceptable costs. In this paper, we discuss four optimizations on top of a basic DLA architecture. Their effects are mainly to increase the efficiency of the look-ahead thread and simultaneously extract more utility out of it. The spirit of techniques are

This work is support in part by NSF under grants 1514433 and 1722847, and by a gift from Huawei. Due to space constraints, some details are left out and can be found in the extended version of this paper [22].

somewhat analogous to the "reduce, reuse, and recycle" mantra in waste reduction, hence the name R3-DLA.

The rest of the paper is organized as follows: Sec. II discusses variants of DLAs and the related concepts of helper threads; Sec. III explains the design details of the proposed optimizations; Sec. IV performs experimental analysis on these optimizations; and finally Sec. V summarizes the findings.

II. Background and Related Works

A basic on-demand caching system is only part of the solution to a high-performance instruction and data supply subsystem. Anticipating needed data and prefetching them is an indispensable component. Various types of prefetchers have been proposed over the years targeting various properties of the applications' address stream [14], [18], [21], [28], [29], [36].

Ultimately, not all accesses can be described by simple address patterns. Obtaining addresses through partial execution of the program represents a broad class of prefetching approaches. On one extreme of the design spectrum, many short threads are launched as helpers to precompute information for data or instruction supply [7], [8], [11], [17], [25], [26], [37]. Although these micro helper threads are an immensely useful concept, marshalling a very large number of micro threads can bring practical issues [22].

On the other extreme of the spectrum, an idle core in a multicore system is used to execute a different copy of the original program on a separate thread context [3], [10], [19], [20], [27], [38], [42]. This copy is often a reduced version of the program (which we referred to as the skeleton) so that it can run faster to look ahead. This style of design can be traced back to the Decoupled Access/Execute architecture [35]. Unlike in DAE, however, the leading thread in this group of designs does not affect the architectural state and only performs look-ahead functionality. We therefore refer to these designs as *Decoupled Look-Ahead* (DLA) architectures.

DLA designs sidestep some of the practical problems facing micro helper threads. But the key challenge becomes how to create a look-ahead thread that is sufficiently autonomous and yet fast enough to permit deep look-ahead. Various ways are devised to improve the look-ahead thread's speed in order to stay ahead of the main program thread. For instance, Slipstream [38] removes predicted dead instructions and biased branches; Dual-core execution skips memory access instructions that miss in the L2 cache [42]; Tandem uses architectural pruning to make the hardware faster [27]. Garg and Huang experimented with a more purposeful-built look-ahead thread using a stripped-down version of the original program [9].

In this past work, only a small number of ideas are discussed at a time. By themselves these ideas have a limited benefit – no different from ideas for conventional microarchitectures. The limited benefit coupled with the perceived disadvantage of doubling the resources needed can hardly make DLA appear as a promissing solution that we believe it is. Keep in mind, the extra thread context is an infrastructure whose cost is amortized over future ideas. As we will show in this paper, there are many conceivable optimizations that can lower the overhead even more while improving performance.

III. Optimizations of DLA Architecture

In this section, we first discuss a basic platform of DLA (Sec. III-A), then discuss four optimizations in detail: reducing look-ahead workload with prefetch offloading (Sec. III-C); reusing value (Sec. III-D1) and control flow information (Sec. III-D2); and recycling the skeleton (Sec. III-E);

A. Baseline DLA

Our baseline DLA architecture is based on the one proposed in [9]. Specifically, a *skeleton* of the original program binary is generated which includes all the control instructions and their backward dependence chain. A subset of memory instructions is also included in the skeleton as prefetch payloads along with their backward dependence chain. During execution, this skeleton forms the static code of the *look-ahead thread* **(LT)** and runs on a different core. It passes relevant information (*e.g.*, branch outcomes) which speeds up the execution of the *main thread* **(MT)**. At first glance, it may seem wasteful to execute the same program twice. But in reality, LT only executes a small portion of the code (about a third in our design). Even when it does execute an instruction, the actions involved may not be redundant. For instance, wrong path instructions are mostly limited to the LT; off-chip accesses are only time-shifted, not repeated; We will present quantitative description on this point later in Sec. IV and only note here that the energy overhead is less than 25%.

Such an architecture requires the following support on top of a generic multi-core architecture, ordered from least to most special-purpose:

i) Containment of speculation: LT usually involves speculative optimizations and thus cannot be allowed to update the architectural state. The support is simple as most of the state is already naturally confined to the thread context. The only additional support needed is about dirty lines in the private caches (in our study, the L1 data cache and the L2 cache). In the look-ahead mode, they are not used to supply coherence requests from other cores and are not written back upon eviction, but simply discarded. In other words, when a core executes in look-ahead mode, its private caches only obtain data uni-directionally from the rest system and never supplies data. It only supplies hints as follows.

ii) Communication of look-ahead results: In a multicore architecture with shared lower level caches, LT can already warm up the shared caches without any additional support. However, a mechanism to explicitly pass on hints from LT is valuable. First, we can send over the branch outcomes via the *Branch Outcome Queue* **(BOQ)**. The BOQ also acts as a natural mechanism to detect when LT veers off the correct control flow; and to keep it from running too far ahead. In the case an incorrect branch outcome is detected, a *reboot* is triggered by MT which re-initializes LT. In addition to the BOQ, we can also send other information such as branch target addresses and prefetch addresses. We use a *Footnote Queue* **(FQ)** for such less frequent but wider data. On average, one footnote is generated every 30 instructions. FQ is also used during reboot to copy the architectural registers from MT to LT.

iii) Support for instruction masking: Finally, we find it convenient to have the code of LT being a subset of MT, thus allowing us to use the same program binary and a set of bits to mask off instructions not on the skeleton. These unwanted instructions are deleted immediately upon fetch in LT. These mask bits can be generated either offline or online through dependence analysis of the program binary. In this paper, we model a system where these bits are generated offline and stored inside the program binary. At runtime, the I-cache will combine the separately fetched mask bits and instructions.

Summary of operations: To put these elements together, we now describe the overall operation of the system (Figure 2). We assume the program binary is analyzed and augmented with mask bits offline, the system always runs in DLA mode, and that the two threads (the real program thread and its look-ahead instance) run on individual cores connected by the various queues discussed above. Note that these are not intrinsic requirements to implement DLA. They describe the most basic incarnation.

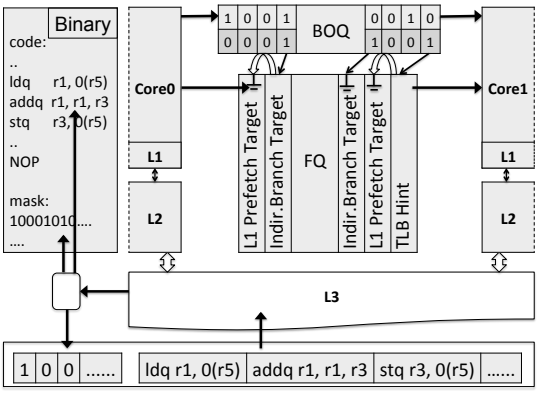


Figure 2. Architectural support for baseline DLA.

When a semantic thread is launched or context-switched in, its architectural state is also used to initialize LT. Both threads proceed to execute the code largely conventionally: fetching, dispatching, executing, and committing instructions according to the content of its architectural and microarchitectural states. They differ from conventional cores in the following way.

For the core executing the actual program thread (MT), its fetch unit draws branch direction predictions from the BOQ instead of its branch predictor. If the queue is empty, we stall the fetch. The dequeued entry of BOQ may have a footnote bit set. In that case, the control logic will dequeue one or more entries from FQ and act according to the content type. For example, if the entry is a branch target hint, then the content from the entry (rather than from the core's own BTB) will be used for target prediction.

For the core executing in look-ahead mode, there are three main differences. First, upon an instruction fetch, the logic will delete unwanted instructions. Beyond the I-cache, however, the masks are assumed to be stored in a different location in the program binary for backward compatibility. The masks are thus stored separately from the instructions in the lower level caches. During an I-cache *miss*, the controller will issue two read requests to the L2 cache for the instructions (at the address A_i) and their masks stored in address $A_m = f(A_i)$.

Second, LT will write hints into the queues for MT. Specifically, at commit time, the outcome of a conditional branch ("taken" or "not taken") will be stored in FIFO order in the BOQ. The BOQ serves a multitude of purposes. ① It passes a branch outcome as a prediction to MT. This ensures that in the steady state, the majority of branch mispredictions are experienced only in LT. ② It is a simple and effective mechanism to detect incorrect look-ahead control flow. When a branch prediction fed by LT turns out wrong, which is relatively rare (0.06 per kilo instructions), it means that LT is executing down the wrong path. We will reboot LT from the current state of MT. ③ We can easily know and control the depth of look-ahead: the number of unread entries in the BOQ equals the number of dynamic basic blocks LT is ahead of MT. To prevent run-away prefetching, we only need to limit the size of the BOQ (512 entries in this paper). ④ It is a convenient way to allow delayed (just-in-time) prefetching. When a prefetch hint is generated, it can be associated with a branch entry and released only upon the dequeuing of that BOQ entry.

Finally, in addition to the continuous branch direction hints, occasionally LT has other hints. Whenever it encounters a miss in TLB, L1 data cache, or BTB, it will pass the relevant address through FQ and set the footnote bit in the most recent BOQ entry.

B. R3-DLA: Overview

Before we discuss the proposed optimizations, it may be helpful to put these ideas into the context of the vision for DLA. As we have seen, there is significant implicit parallelism in normal programs. It is not yet clear to us what the best approach to exploiting this parallelism is. Our current DLA design follows a path that first targets instruction and data supply issue, basically because we have to start *somewhere*. The baseline DLA design is also just a starting point, intuitively with many low-hanging optimization opportunities.

Indeed, as it turns out, there are a number of simple things to do to either make the LT more efficient and/or get more utility from it. In the first category, we can offload certain type of look-ahead code to a finite state machine (Sec. III-C), making the LT smaller and thus faster. In the second category, we find that LT can provide more than just branch prediction and addresses for prefetching. Both the intermediate values and control flow information can be reused, for example for value prediction (Sec. III-D). Finally, the heuristic-driven skeleton in the baseline DLA is clearly not optimal and can be adjusted online to other predefined versions which, depending on the specific situation, can make LT more efficient and/or more effective (Sec. III-E).

C. Reduce: offloading strided prefetch

The first optimization speeds up LT by reducing its workload. The intuition is simple. LT serves as a software-guided prefetch engine which is far more flexible and precise than hard-wired, finite-state machine (FSM) driven prefetchers. The cost, however, is that we need to execute a sequence of instructions to compute the address for prefetching. For simple, strided accesses, such a software-driven approach is an overkill. Instead, we build a hardware FSM (which we call T1) to offload this type of prefetch.

Note that there is an important difference in the design goal between T1 and a traditional stride prefetcher. The latter needs to extract the stride in the presence of unrelated addresses and in the absence of any certainty that there is a strided stream to begin with. Moreover, to improve coverage, practical prefetcher designs target variations of strided accesses. All these are non-trivial challenges and often involves memorizing and cross-comparing a non-trivial number of addresses. T1, on the other hand, merely carries out the mundane task of address calculation and issuing prefetches. In other words, compared to a traditional stride-detecting prefetcher, T1 is only a dumb FSM carrying out simple orders. Additionally, T1 only targets one common situation and does not try to be general-purpose.

1) Overview of operation: The common situation targeted by T1 is a loop with one or more memory access instructions whose address gets incremented by a (run-time) constant every iteration. In such a case, all we need for effective prefetching is the stride (δ), the prefetch distance (n), and the identity of the strided access instructions. When we encounter a strided instruction, we can take its address (A) and simply prefetch $A+n\delta$. Given the identity of the strided instructions, we can easily derive the stride and prefetch distance: The former is simply the difference between the addresses of two consecutive instances of the same static instruction; The latter is the average access latency divided by the time interval between two consecutive instances. Instead of including these necessary instructions in the skeleton to generate proper prefetches, we mark them in MT and let T1 handle the prefetches. LT is thus smaller and faster than otherwise.

2) Instruction marking: To summarize the discussion above, all the T1 hardware needs is the identities of the loop branch and the strided access instructions. These instructions are marked with another bit (the S bit). The S bits are generated the same time the skeleton masks are generated, however they are a marker on the binary for MT[1]. They are fetched from the program binary by MT in the same manner as the skeleton's mask bits. Note that the T1 hardware located in MT's core will use information from MT to produce corresponding prefetches for the instructions marked with the S bit. Hence, the skeleton generation process will completely ignore these instructions and their backward dependencies while generating the skeleton for LT.

state	loop PC	inst. PC	eff. addr.	stride	cur. time	pref. distance

Figure 3. T1 prefetch register fields.

3) Operations of FSM: With marking of S bits done, the run-time operation is to calculate two parameters: stride and prefetching distance. To accomplish these tasks, we use a small prefetch table. An entry of this table is shown in Figure 3. All entries begin in an invalid state. Upon execution of a strided instruction, an entry is allocated in the prefetch table. T1 starts issuing prefetches (with a fixed prefetch degree) as soon as the first instance of a stride is calculated. Each entry in the table gradually moves from an invalid state through transient states to the steady state. Transient states help guard T1 against identifying incorrect strides resulting from out-of-order execution. They also aid in calculating appropriate prefetching distance using the information on iteration time and average memory access latency. When prefetching distance is calculated, T1 launches multiple prefetches to "catch up" with the prefetching distance. T1 then transitions into a steady state in which it launches one prefetch every iteration. All entries in the table are cleared when a loop terminates.

[1]So, the skeleton now includes two bits per instruction: a mask for LT and a mark for T1

D. Reuse of Value and Control Information

The software-controlled nature of the baseline DLA makes it a very flexible branch predictor and prefetcher. The cost for such flexibility is the extra execution. As we will see later, even with offloading discussed above, LT still executes about half of the dynamic instructions of the program. The next two optimizations try to increase the benefit of this work already done. In particular, since a significant portion of the values have already been computed, we seek to reuse them in the form of value prediction. Also, the content of BOQ is highly accurate future control flow information and can help improve instruction fetch for the trailing MT.

1) Value Reuse: A variety of techniques have been proposed to predict values [30], [39]–[41]. Most of these techniques rely on the history of the values produced to predict future values. Unlike branch outcomes, a typical value usually has non-trivial entropy and thus defy easy predictions. However, in our system, many instructions have already been executed in LT. Empirical observations show that over 98% of them have the same result as their counterpart in MT and thus lend themselves to reuse.

The basic support is similar to any value predictor: ① the predicted value will be used to allow dependent instructions to execute early; ② the instruction producing the value will check the outcome with the prediction and, upon disagreement, trigger a replay. In our system, instead of coming from a value prediction table, the predictions are read in FIFO order fed by LT. In our design, we extend the footnote queue for this purpose: every instruction that we decide to apply value reuse will allocate an FQ entry containing the value and an offset indicating distance from the preceding branch.

Again, unlike traditional value predictions, we have an abundance of highly accurate results. Thus the key design issue for our value reuse is to minimize the costs, which includes communication from LT to MT and the performance loss due to incorrect values. Our approach is to limit value only to "slow" instructions with a high confidence of successful reuse. After some experiments, it quickly became obvious that many different heuristics can achieve the goal. We describe one runtime version below.

At the beginning of a new loop (Sec. III-E2), MT spends a few iterations (8 in our experiments) identifying these slow instructions, defined as having dispatch-to-execute latency of at least 20 cycles. Their PCs will be recorded in a bloom filter (let us call that *Slow Instruction Filter* or **SIF**). LT checks this table at commit stage and if the instruction is there, allocates a *value reuse* entry in FQ. The SIF is cleared upon entering a new loop.

Our confidence mechanism is simplistic: when a value prediction turns out incorrect, the entry of that static instruction is deleted from the SIF and LT will no longer provide a prediction for that instruction. However, we observe that this is infrequent (less than twice per million instructions).

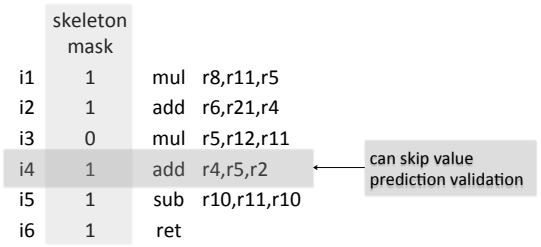


Figure 4. Example of skipping value prediction validation.

There is one small optimization to this basic support. In some cases, we do not need to validate all predicted values. We can skip those ALU instructions that only depend on other instructions that have produced a predicted value. Figure 4 shows an example. i_4 sources from i_1 and i_2, both of which produce a value prediction. When we see this case, we can directly use i_4's value prediction as the outcome and there is no need to execute $i4$ in MT. This is because in our system, we have no speculative optimization that can corrupt functional units in LT. So, if both values are correct, then i_4's result is correct (barring hardware reliability issues). If either value is incorrect, eventually, there will be a value misprediction recovery upstream. i_5, on the other hand, cannot skip validation as it depends on a value that is not predicted. This optimization will save about 11% of validations.

We implement this optimization with a simple scoreboarding logic at the decode stage in the MT core. When an ALU instruction i produces a value prediction, we mark its destination register as validated. Other instructions (e.g., loads, or instructions not producing a value prediction) will clear the marking for its destination register. If an instruction has a value prediction and its source registers are all marked validated, it will not be executed for validation.

Finally, once the value prediction framework exists, we can add some critical-path instructions back to the skeleton. Clearly the trade-off is faster execution of MT at the expense of slowdown of LT. In general, whether adding an instruction to the skeleton speeds up the whole system or not depends on the balance of the duo. It opens up a general optimization problem of choosing the right skeleton that maximizes system performance. In this paper, we only follow a simple heuristic to find candidates: they have a long dispatch-to-execute latency (more than 20 cycles on average) and have more than one dependent instruction. The skeleton construction algorithm will include the necessary backward dependence chain.

2) Control flow information reuse: Instruction fetch can sometimes be a source of pipeline bubbles. In DLA, the presence of future control flow information allows us to ameliorate this problem to some extent for MT. For instance, a trace cache [32] can increase the number of fetched instructions per cache access. Having a highly accurate

stream of branch predictions from the BOQ is a significant advantage to leverage when using a trace cache. However, trace cache is an expensive form of instruction caching. In this paper, we opt for a simpler approach that is in fact more effective in our setup.

The basic idea is to reduce idling for the instruction fetch unit by allowing it to continue even if the decode stalls. In other words, we want to decouple the fetch unit from the rest of the pipeline – or in the case where this has already been done to the baseline architecture, increasing the degree of decoupling with a bigger buffer. The key point to emphasize is that the BOQ offers a much higher degree of branch prediction accuracy. Without this accuracy, fetching too much down the predicted control flow is unlikely to pay off, and indeed can even backfire and slow the whole processor down. In fact, in a conventional architecture, sometimes a more constrained fetch unit is beneficial as it slows down the pollution created by the relatively common wrong-path instructions. In other words, the benefit of having a fetch buffer is clear for DLA, but not necessarily so for a conventional architecture. We will show this in the experimental analysis later in Sec. IV-C.

E. Re-cycling the Skeleton

In DLA, the skeleton is constructed using simple heuristics. Given this basic skeleton, if we add one more memory instruction (and its backward dependence chain), LT is likely to run a bit slower but potentially helping MT avoid more misses. Depending on which thread tends to be the bottleneck, this small change may increase or decrease system performance. We can see that there is a vast number of possible variations and the basic skeleton is unlikely to be optimal. The question is, therefore, are there significantly better options than our default? If so, how can we systematically and efficiently arrive at such options?

These are all questions beyond the scope of this paper. Nevertheless, we do know that simple tunings can effectively improve the performance. The general approach is to create a few versions of skeleton and cycle through them to find out the best empirically.[2]

1) Versions of skeleton: The most basic version of the skeleton includes all branches and their backward dependence chains and is produced with a binary parser. From this starting point, we may add or subtract instructions using a few broad heuristics coupled with static-time profiling. In our experiments we collect these statistics by executing the programs with training inputs and use them to build skeletons that are used during the actual run. We experimented with five options:

- L2 prefetch targets: Instructions that account for significant portions of L2 misses can be added to the skeleton;
- L1 prefetch targets: Instructions that account for significant portions of L1 misses can be added to the skeleton;
- Value reuse targets: Instructions that have a long dispatch to execute latency can be added to the skeleton;
- T1 targets: Memory instructions that are handled by T1 are by default removed from the skeleton. However, they may be added back (as they might warm up cache for LT).
- Biased branches: Conditional branches with a bias over a threshold can be converted to unconditional branches in the skeleton.

These independent options naturally lead to many different combinations. Our empirical observation shows that a very small number of combinations need to be searched to obtain noticeable benefits. We evaluate a design that cycles through six versions of skeleton empirically observed to be most often helpful. Changes to the number of options, the number of versions, or the thresholds used in identifying target instructions will likely affect the exact outcome. The key point to note is that this is not an effort to find the optimal points in the design space, but simply an attempt to pick a few different points so that we can avoid poor design points due to simplistic heuristics. Also note that the skeleton generation process (Fig. 5) is an offline, automated process just as in the baseline, except it produces more than one skeleton.

2) Controller: With a number of skeleton options, the goal of the controller is to find the skeleton version that maximizes the benefit. To do this, we divide the execution into repeating code units, or loosely speaking, loops. For each loop, we cycle through different versions to find out the best.

To identify the current loop, we capture the backward "loop branch" (Figure 6). Two consecutive instances of the loop branch without an interleaving instance of another loop branch marks the two ends of an iteration. Note that units need to be of sufficiently coarse granularity, otherwise we can neither accurately measure execution statistics nor profitably adjust the system configuration. So, a unit of execution is one or more iterations lasting, say, at least 10,000 instructions.

Note that recursive functions can represent a significant portion of the execution time without having a detectable loop. To deal with these cases, we treat certain function call instructions as if they are loop branches. In such a case, an "iteration" may not have the same meaning as we are used to. But it is still a valid strategy to observe the behavior of a unit of multiple iterations to predict that of a future unit.

During execution, the controller will run each loop for enough iterations under a particular skeleton. This will allow an accurate measurement of the speed (instructions committed per unit time) of that loop under that skeleton. After cycling through all skeletons, the controller selects the skeleton showing the highest speed and use it for the

[2]This process may repeat a few times to average out noise. Admittedly, the analogy with recycling in the normal sense is tenuous.

steady state. The identity of the loop (PC of the loop branch) and the corresponding best configuration is then stored in a Loop-Config Table (LCT as shown in Figure 6). If a loop PC is found in the LCT, the controller selects the corresponding configuration.

Note that all the steps involved in re-cycling skeleton can be done either on-line as the application runs, or off-line using training runs. For the simple recycling discussed in this paper, we believe the offline approach is more advisable as we need no architectural support (other than performance counters). However, an online recycling support (like the one we discussed) may be a better alternative in a more dynamic environment. We will compare the effect later in Sec. IV-C.

F. Recap

To sum up, a basic DLA design uses one core to execute a look-ahead thread and passes information through two queues (BOQ and FQ) to help accelerate MT. On top of this basic design, we propose to add a number of supporting elements (Figure 7) to accelerate either LT or MT:

- **T1:** A prefetching FSM to offload prefetching of loop-based strided accesses;
- **Value reuse:** logic to pass register values from LT through FQ and used as predictions in the front-end of MT;
- **Fetch buffer:** using an extended buffer to fetch instructions down the path predicted in the BOQ;
- **Re-cycle controller** (hardware support optional): cycles through a number of skeleton mask bits to pick the best performing configuration.

With these elements, the R3-DLA becomes significantly more effective as we will show next. More importantly, these optimizations are merely examples of what can be done to make the DLA models more effective. We believe that there are plenty of opportunities in DLA to keep on extracting more implicit parallelism.

IV. Experimental Analysis

In this section, we perform experimental analyses of the proposed design. After detailing the simulation setup (Sec. IV-A) we first show bottom-line results of a complete system (Sec. IV-B) and then provide more detailed analyses to gain insight of the individual design decisions (Sec. IV-C).

A. Simulation Setup

For simulation purposes, we use Gem5 [4] simulator to model our proposed architecture. Our baseline is an aggressive out-of-order pipeline with a Best Offset [28] prefetcher (BOP) at L2. This prefetcher is selected because in our experiments it provided the best average performance gain among a group of 7 state-of-the-art prefetchersover all application suites experimented [22]. The prefetcher is configured with 256 RR table entries and 52 offsets as described in [28]. Additional technical details about the baseline are provided in Table I. Unless otherwise mentioned, we use this baseline configuration in all of our experiments. For DLA reboots, we add a 64 cycle delay to account for copying of the architectural registers from the trailing thread to the look-ahead thread. Note that since the chances of reboots are rare in DLA (0.6 on average in a 10k instruction window), their impact on performance is minimal *e.g.*, increasing the reboot cost to 200 cycles will degrade the overall performance of R3DLA by less than 2%.

For comparison, we have also modeled a number of similar approaches [11], [16], [31] ranging from earlier

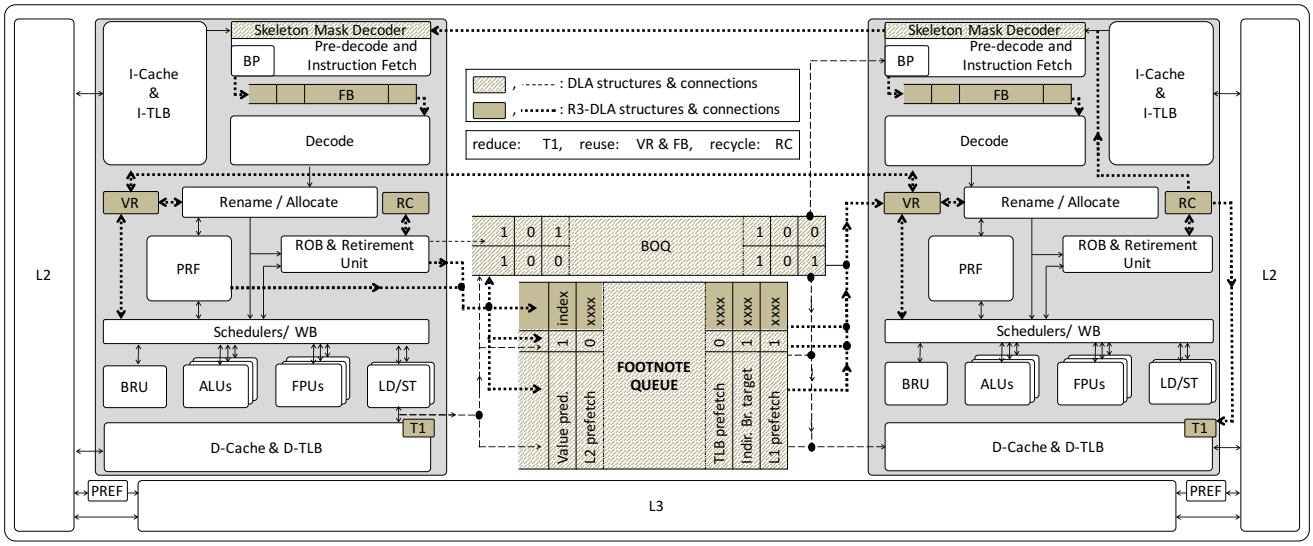


Figure 7. Architectural support for R3-DLA. The gray colored rectangle on the left represents the lead core and the one on the right represents the main core. Structures included by DLA are patterned and the connections are indicated with the dashed lines. Structures included by R3-DLA are shaded and the connections are indicated with the dotted lines.

```
Skeleton Generation Algorithm
1:  Function Profile (condition)
2:      s ← clear()
3:      for each instruction in program
4:          if instruction & condition then  s ← insert(instruction)
5:      return s
6:  Function GenerateSkeleton (targets)
7:      mask[ length (program) ] ← { 0 }
8:      for each instruction in targets
9:          mask[ instruction ] ← 1
10:         mask ←   mask | backward_dependents(instruction)
11:     return mask
12: // profile collection
13: vector[0] ← Profile(probability(L1 miss) > 0.01)
14: vector[1] ← Profile(probability(L2 miss) > 0.001)
15: vector[2] ← Profile(prefetch_accuracy (T1) > 0.95)
16: vector[3] ← Profile(average(dispatch_to_execute_delay) > 20 cycles)
17: vector[4] ← Profile(branch_bias > 0.99)
18: vector[5] ← Profile(branches)
19: // skeleton generation
20: base_targets ← vector[5] – vector[4]
21: versions  ← [00000, 01000, 11101, 11000, 01010, 11010]
22: skeletons[ length(versions) ] [ length (program) ] ← { 0 }
23: for i in range ( length (versions) )
24:     targets ← base_targets
25:     for j in range (5)
26:         targets ← targets +  ( versions[ i ][ j ] x vector[ j ] )
27:     skeletons[ i ] ← GenerateSkeleton (targets)
```

Figure 5. A pseudo code outlining the steps involved in generating different skeletons used by the recycle optimization. A few seed vectors are constructed using profiling information obtained from the program binary and training runs. A skeleton is generated from a seed vector by including backward dependencies of each seed present in it. Multiple combinations of these seed vectors can therefore produce multiple skeletons. The recycling optimization we used in our evaluations uses six of these skeletons (Line 21 in the figure).

design of SlipStream to the recently proposed state-of-the-art runahead execution scheme called Continuous Runahead Engine (CRE). Under our baseline configuration (Table I), CRE outperforms other designs. It generates its helper threads at runtime and executes them on a custom processor located at the memory controller. We modified CRE's design to prefetch data into L1 which on average provides higher overall performance than just prefetching into LLC. Note that since we do not ignore any of the applications in our evaluations and since our baseline configuration uses 3 levels of cache hierarchy with BOP as a L2 prefetcher, our performance numbers for CRE appear different than the ones reported in [11]. For similar configuration and applications, the average performance gain of CRE on our platform is within 5% of the ones reported in [11]. In the case of [9], the factors like memory model, improved prefetcher/branch predictor and an overall aggressive baseline all contribute to the variation in the reported performance benefits.

For CPU's energy consumption modeling, we use McPAT [24] and assume a 22nm technology node. We modified McPAT to correctly model our proposed architecture and additional hardware structures shown in Figure 7. To compute main memory energy, we use DRAMPower [5].

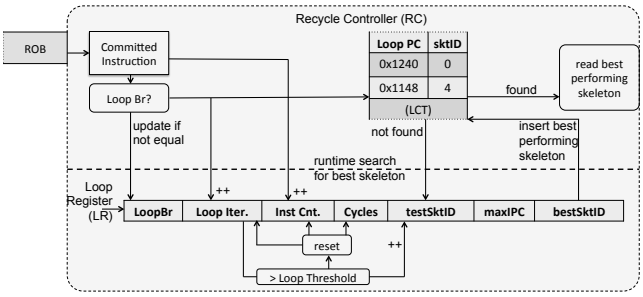


Figure 6. Skeleton recycling flow chart. As a loop branch retires, the Loop Config Table (LCT) is queried for the skeleton that is optimum for the current loop. If none of the entries in LCT match the loop branch, different fields in the Loop Register (LR) are used to cycles through each of the available skeletons for a few iterations of the loop and identify the optimum skeleton for the loop. The LCT is updated when an optimum skeleton is found and that skeleton is used by lead thread until a new loop branch retires from ROB.

Processing Node	20-stage pipeline, out-of-order, 4-wide, 192 ROB, 96 LSQ, 128INT/128FP PRF, 4INT/ 2MEM/ 4FP FUs, Tage SC-L Predictor (configured as the 256kBits predictor described in [33]), 4K Entry BTB, 32-entry RAS
Operating Points	0.8V, 3.0GHz
L1 Caches	32KB I-cache and 32KB D-cache, 4-way, 64B blocks, 3 ports, 1ns, 32 MSHRs, LRU
L2 Cache	256KB, 8-way, 64B blocks, 2 ports, 3ns, 32 MSHRs, LRU, BOP [28]
L3 Cache	2MB, 16-way, 64B blocks, 12ns, LRU
Main Memory	4GB, DDR3 1600MHz, 1.5V, 2 channels, 2 ranks/channel, 8 banks/rank, t_{RCD}=13.75ns, t_{RAS}=35ns, t_{FAW}=30ns, t_{WTR}=7.5ns, t_{RP}=13.75ns
DLA Support	
BOQ	512 entries (512x2 bits = 128B)
FQ	128 entries (128x64 bits = 1KB)
R3-DLA Support	
T1	16 prefetching entries (512B)
FB	32 instructions (256B)
VPT	32 Entries (32x64 bits = 256B) (used by VR)
LCT	16 Entries (136B) (used by RC)

Table I. System configuration.

We evaluate our proposal on a broad set of benchmark suites. In addition to the SPEC2006 [12] benchmark suite, we use CRONO [1] (a graph application suite), STARBENCH [2] (embedded applications), and scientific workloads from NAS Parallel Benchmarks (NPB). For SPEC2006, we use reference inputs. For STARBENCH we use large inputs. NPB is simulated with C class of workloads. For CRONO we use graph input data structures from google, amazon, twitter, mathoverflow and california road-networks. All benchmarks are compiled using gcc with -O3 option. To reduce simulation time, we use SimPoint sampling methodology. To accurately capture all phases of the application we use the SimPoint Tool [34] to generate five simpoints per benchmark with 10 million instruction intervals. We warm up the caches for 100 million instructions before beginning each of the simpoint intervals. All the simulation results are obtained from these simpoints.

B. Overall Benefits

It is worth noting that in this paper, we assume DLA is only used as a turbo-boosting technique – when there is an idle core/thread. We assume otherwise exploiting explicit parallelism yields better results.[3]

1) Performance: We first measure the overall performance of R3-DLA and compare it to that of the underlying microarchitecture and that enhanced by our baseline DLA. We show these three configurations both without a hardware prefetcher (left group of 3 bars In Figure 8-a) and with BOP prefetcher (right group of 3 bars). All performance results are normalized to the microarchitecture with BOP, which represents the best we can do today without using DLA techniques.

For clarity, we summarize the results of an entire benchmark suite into a single bar showing the geometric mean of the whole suite and an I-beam showing the range of values. This figure contains a lot of information that can be organized into a number of observations:

i) R3-DLA provides high performance compared to the underlying microarchitecture with an advanced prefetcher. The speedup ranges from 1.06x to 2.24x with a geometric mean of 1.4x. While the average performance gain is significant, there is also a wide range of result. DLA is most likely to be selectively applied when the benefit is large. For instance, for the top half of the applications, the geometric mean speedup would be 1.51x.

ii) R3-DLA is also significantly faster than more basic DLA designs. On average, R3-DLA outperforms the baseline DLA by about 1.25x. This shows that the proposed optimizations are effective. Figure 8-b briefly compares the overall performance among a set of related approaches: B-fetch [16], SlipStream [31] and CRE [11].

iii) DLA is a fully-flexible prefetcher and thus has overlapping targets with a standalone prefetcher such as BOP. When used with R3-DLA, BOP can still help as it frees up DLA's attention to better handle the remaining targets, making the system a bit more efficient. However, the "collaboration" between the two mechanisms is unplanned for and the benefit is somewhat limited: while BOP can improve the baseline architecture by 1.27x, its effect on an R3-DLA system is only 1.13x. We conjecture that a more conscious collaboration between a standalone prefetcher and DLA will be more effective.

2) Efficiency: One common concern of DLA architectures is the energy cost. While it is tempting to assume the energy cost (or at least power) doubles in DLA due to executing the program twice, it would be a significant overestimation even for the baseline DLA design, not to mention R3-DLA, which further lowers the overhead.

First and foremost, LT is a much lighter thread, with an average length of only 36% that of MT. Second, not all LT activities are overheads. Some are time-shifted activities (*e.g.*, most memory accesses). Others help MT avoid almost all wrong-path instructions. Finally, faster execution lowers fixed energy costs. Note that LT-to-MT communication is insignificant (averaging 2.2 bits per instruction) and is faithfully modeled. To see this in a bit more detail, in Table II, we show the amount of activities, the resulting dynamic and static power in both LT and MT, all normalized to the baseline microarchitecture. We see that LT expends much less dynamic energy or power than baseline. Also, despite running much faster than baseline, MT's power is comparable to the latter since it significantly reduces waste.

		D	X	C	Dyn. Energy	Dyn. Power	Static Power	Power
DLA	LT	49%	48%	48%	48%	54%	94%	71%
	MT	77%	86%	100%	88%	96%	99%	97%
R3-DLA	LT	35%	29%	29%	30%	42%	93%	64%
	MT	77%	82%	100%	80%	110%	95%	103%

Table II. Average of activities (in Decode, eXecution, and Commit stages), energy, and power for both threads in DLA and R3-DLA all normalized to baseline. Note that for every instruction committed in the baseline processor, 1.16 are executed and 1.25 decoded.

Combining these factors together, our energy estimates (Figure 9) suggest that the average normalized energy for R3-DLA is 1.11x for the processor and 0.9x for memory (all geometric means). There is significant variation among individual benchmarks (with arithmetic mean 1.19x and 0.92 respectively.) In terms of energy delay product, DLA is 6% worse than baseline while R3-DLA is 19% better on average.

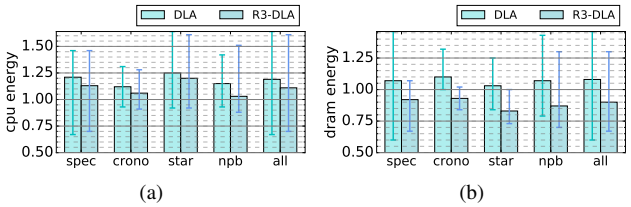
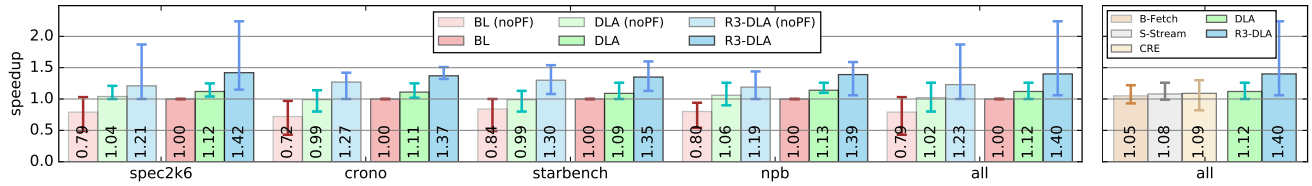


Figure 9. Comparison of energy normalized to baseline spent in (a) cpu (b) dram by DLA and R3-DLA.

3) Application in SMT cores: Finally, with increased efficiency and sophistication, R3-DLA opens up the DLA architecture to more usage scenarios *e.g.*, in SMT cores. Since SMT presents new opportunities for optimizing DLA architectures, a more complete treatment of the subject is left in [23].

[3]Note that this is not always true anymore. And as more ideas are developed, we may reach a point where the decision whether to use resource for one type of parallelism or the other becomes a non-trivial one.



(a) (b)

Figure 8. Performance gain over an aggressive baseline with BOP at L2. NoPF shows the normalized performance of a baseline configuration with no prefetcher.

C. Detailed Analysis

We now look at the contribution of each individual element in the design and some aspects of their interaction.

1) Offloading strided prefetch: With offloading stride prefetching, we move the comparatively simpler task of prefetching for certain strided accesses to a hardware FSM. This alone reduces the skeleton size (from 66% to 45% on average, more on that later), allowing LT to run faster, and in turn making it more likely to succeed in other prefetches. To understand the effect, we compare three different options of prefetching these strided accesses: a modified stride prefetcher [15],the baseline DLA, and R3-DLA. We show the mean and median L1 MPKI (misses per kilo instructions) for strided and the remaining accesses in Table III.

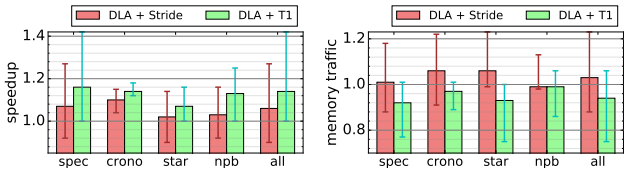
	strided				others			
config.	BL	BL + stride	DLA	DLA + T1	BL	BL + stride	DLA	DLA + T1
mean	12.4	8.4	5.9	2.1	7.4	6.9	6.1	4.8
median	10.0	4.8	4.0	1.1	3.9	3.5	2.8	3.2

Table III. L1 MPKI divided between strided accesses and non-strided accesses, corresponding to four different configurations. For brevity, only means and median are shown.

We can imagine that T1 is far from perfect, requiring the loop to start a few iterations before catching up. This is why there is still a non-trivial 2.1 MPKI remaining for strided accesses. But in comparison, both baseline DLA and a hardware prefetcher are worse with 5.9 and 8.4 MPKI remaining respectively. Additionally, the offloading improved DLA's ability to target non-strided misses, reducing it from 6.1 to 4.8 MPKI on average. The medians show a similar trend.

Figure 10 evaluates both performance and memory traffic metrics among the various choices. For brevity, we only show the aggregate result as the suite-wide average (represented by the bar) and range among individual applications (represented by the I-beams).

First, we see that offloading works very well with DLA across all four suites, achieving a geometric mean speedup of 1.14x over all benchmarks. Second, this offloading arrangement is noticeably more effective as well as more efficient than simply adding a hardware stride prefetcher. In terms of performance, offloading never slows down the system in



(a) (b)

Figure 10. Comparison of (a) speedup and (b) normalized memory traffic of two different configurations: DLA with a stride prefetcher and DLA with offloading (DLA+T1).

any benchmarks and has a high mean speedup (compared to 1.06x for adding a stride prefetcher). This is because the T1 hardware does a much more limited and easier job than a conventional stride prefetcher [6]. This can be seen by the memory traffic result shown in Figure 10-b: the total memory traffic is lower with adding T1 than with adding a stride prefetcher. Some of the extra prefetches from the stride prefetcher are useless and create pollution, which contributes to the lower performance.

2) Reusing control flow information: Using a fetch buffer to decouple fetch stage and decode stage is not a new idea. The key point here is that DLA makes it far more effective due to the much higher branch prediction accuracy. Figure 11-a shows the performance gains obtained by adding a fetch buffer over a baseline system and a DLA system.

We see from the figure that the impact of a fetch buffer can be negative in the baseline system. In fact, for NPB suite, the overall effect is negative. When averaging over all applications, the benefit is relatively small (4% improvement). In contrast, when driven by the highly-accurate prediction sequence from BOQ, the fetch buffer almost never hurts and the benefit can be as high as 1.28x. Overall, the speedup due to this addition is 1.08x.

3) Re-cycling skeleton: Re-cycling does not add new direct mechanisms for better look-ahead. It merely searches the configuration space for a better solution. Figure 12 shows the distribution of the skeleton being chosen in re-cycling of skeleton. Each color shows a different configuration being chosen. Although some simulation windows have a single choice for a major portion of the window, all of them have chosen a number of different solutions for different loops. This suggests that using simplistic heuristics to design

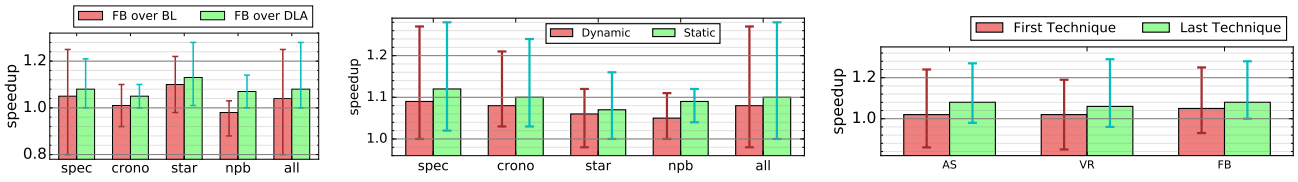


Figure 11. (a) Comparison of performance gains obtained by a Fetch Buffer over BL system and over DLA system (b) The speedup differences between dynamic and static tuning (c) Speedup when an optimization is applied first or after other optimizations.

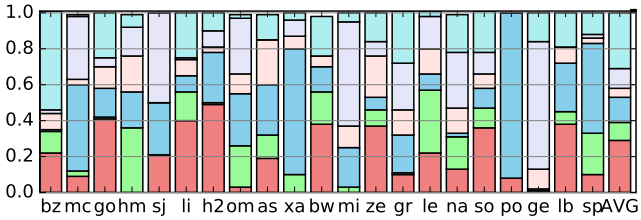


Figure 12. The distribution of skeleton versions chosen during online tuning.

the default skeleton is unlikely to pick an optimal design for a particular situation. Some dynamic tuning is perhaps necessary.

Our experiments show that re-cycling skeleton improves performance by about 1.08 on average and up to 1.27x as shown in Figure 11-b. The figure also compares the difference between dynamic on-line and static off-line (using training inputs) tuning. We see that static tuning consistently shows better result. This is partly due to the fact that in dynamic tuning, more time is spent trying out suboptimal configurations. We note that in both approaches, the tuning is done in a very crude way and in a coarse-grain manner. This observation suggests that a more methodical, fine-grain tuning may be able to further improve the performance of a DLA system.

4) Synergy of individual optimizations: While some optimizations proposed improve the speed of the look-ahead thread, others extract more benefit from the look-ahead thread to improve the main thread. There is an additional synergy when all these techniques are combined: in a DLA system, the overall speed in a given phase is limited by the slower of the two threads. So, if a technique speeds up only one thread, the system performance will increase but only to the point where the other thread becomes the new bottleneck. The rest of the benefit will only manifest when something is done to improve the other thread. So, when multiple techniques are applied, their combined benefit will usually be higher than implied by the benefit of each individual technique measured in isolation. We show this visually in Figure 11-c.

In this experiment, we take the baseline DLA platform and measure the performance impact of applying only one of the three techniques. We compare that to applying the same technique last, that is, when the platform already incorporated other techniques. We see that in all three cases, if we measure the technique's benefit when it is applied as the first step, then none looks especially promising averaging about 2-5% gain. However, if we measure the difference it makes as the last technique to be applied, the same technique now appears to have a noticeably higher 6-8% benefit. Thus, as we add more optimizations to the design, more performance benefit may be unlocked.

V. Conclusions

Today's general-purpose applications continue to have significant levels of implicit parallelism. However, data and instruction supply subsystem presents significant barriers to exploiting this parallelism in a conventional microarchitecture. Decoupled look-ahead systems are a potential solution. In this paper, we have explored a number of optimizations to such an architecture. They include ① reducing the look-ahead thread workload by offloading simple prefetch pattern to a finite state machine; ② reusing available values and control flow information to improve execution and instruction supply to the main thread; and ③ fine-tuning by cycling through a number of pre-made skeletons. Each of these techniques makes a seemingly limited contribution when applied in isolation. But combined together, they improve the performance of a basic DLA system by 1.25x and achieves a speedup over a conventional architecture with a state-of-the-art prefetcher by 1.4x on average. This performance advantage differs from application to application and can be as high as 2.24x, suggesting that if used selectively and judiciously, an optimized R3-DLA system is already a high-performance solution of exploiting the available implicit parallelism. Furthermore, analyses of the system suggest the potential for further improvements.

References

[1] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *IEEE International Symposium on Workload Characterization*, pages 44–55, 2015.

[2] M. Andersch, B. Juurlink, and C. Chi. A benchmark suite for evaluating parallel programming models. In *Proceedings of Workshop on Parallel Systems and Algorithms (PARS)*, volume 28, pages 1–6, 2013.

[3] A. Ansari, S. Feng, S. Gupta, J. Torrellas, and S. Mahlke. Illusionist: Transforming lightweight cores into aggressive cores on demand. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2013.

[4] N. Binkert et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[5] K. Chandrasekar, C. Weis, Y. Li, B. Akesson, N. Wehn, and K. Goossens. DRAMPower: Open-source DRAM power and energy estimation tool, 2012. http://www.drampower.info.

[6] T. Chen and J. Baer. Effective hardware-based data prefetching for high-performance processors. In *IEEE Transactions on Computers*, volume 44, pages 609–623, 1995.

[7] M. Dubois and Y. Song. Assisted execution. Technical Report, Department of Electrical Engineering, University of Southern California, 1998.

[8] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proceedings of the International Symposium on Microarchitecture*, pages 59–68, November–December 1998.

[9] A. Garg and M. Huang. A Performance-Correctness Explicitly Decoupled Architecture. In *Proceedings of the International Symposium on Microarchitecture*, pages 306–317, November 2008.

[10] A. Garg, R. Parihar, and M. Huang. Speculative Parallelization in Decoupled Look-ahead. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 412–422, October 2011.

[11] M. Hashemi, O. Mutlu, and Y. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[12] J. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.

[13] W. Hwu. Top Five Reasons Why Sequential Programming Models Could the Best to Program CMPs. Keynote Speech, International Symposium on Microarchitecture, December 2006.

[14] Y. Ishii, M. Inaba, and K. Hiraki. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism*, 2011.

[15] B. Janssens J. Fu, J. Patel. Stride directed prefetching in scalar processors. In *Proceedings of the International Symposium on Microarchitecture*, December 1992.

[16] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez. B-Fetch: Branch prediction directed prefetching for Chip-Multiprocessors. In *Proceedings of the International Symposium on Microarchitecture*, December 2014.

[17] D. Kim, S. Liao, P. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, M. Gikar, J. Shen, and D. Yeung. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 27–38, March 2004.

[18] S. Kondguli and M. Huang. T2: A Highly Accurate and Energy Efficient Stride Prefetcher. In *Proceedings of the International Conference on Computer Design*, November 2017.

[19] S. Kondguli and M. Huang. A Case for a More Effective, Power-Efficient Turbo Boosting. *ACM Transactions on Architecture and Code Optimization*, 15(1):5–22, March 2018.

[20] S. Kondguli and M. Huang. Bootstrapping: Using SMT Hardware to Improve Single-Thread Performance. *IEEE TCCA Computer Architecture Letters*, 17(2):205–208, July 2018.

[21] S. Kondguli and M. Huang. Division of Labor: A More Effective Approach to Prefetching. In *Proceedings of the International Symposium on Computer Architecture*, June 2018.

[22] S. Kondguli and M. Huang. "R3-DLA (Reduce, Reuse, Recycle): A More Efficient Approach to Decoupled Look-Ahead Architectures". *arXiv preprint arXiv:1812.04514*, December 2018.

[23] S. Kondguli and M. Huang. Bootstrapping: Using SMT Hardware to Improve Single-Thread Performance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 2019.

[24] 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 *Proceedings of the International Symposium on Microarchitecture*, December 2009.

[25] J. Lu, A. Das, W. Hsu, K. Nguyen, and S. Abraham. Dynamic Helper Threaded Prefetching on the Sun UltraSPARC CMP Processor. In *Proceedings of the International Symposium on Microarchitecture*, pages 93–104, December 2005.

[26] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *Proceedings of the International Symposium on Microarchitecture*, pages 3–14, November 2002.

[27] F. Mesa-Martinez and J. Renau. Effective Optimistic-Checker Tandem Core Design Through Architectural Pruning. In *Proceedings of the International Symposium on Microarchitecture*, pages 236–248, December 2007.

[28] P. Michaud. Best-Offset Hardware Prefetching. In *International Symposium on High Performance Computer Architecture*, pages 469–480, 2016.

[29] K. Nesbit, A. Dhodapkar, and J. Smith. Ac/dc: An adaptive data cache prefetcher. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, September 2004.

[30] A. Perais, F. Endo, and A. Seznec. Register sharing for equality prediction. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[31] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A Study of Slipstream Processors. In *Proceedings of the International Symposium on Microarchitecture*, pages 269–280, December 2000.

[32] E. Rotenberg, J. Smith, and S. Bennett. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the International Symposium on Microarchitecture*, pages 24–34, December 1996.

[33] A. Seznec. Tage-scl branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, pages 45–57, October 2002.

[35] J. Smith. Decoupled Access/Execute Computer Architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.

[36] S. Somogyi, T. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-Temporal Memory Streaming. In *Proceedings of the International Symposium on Computer Architecture*, June 2009.

[37] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, , and M. Upton. Continual Flow Pipelines. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, October 2004.

[38] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, pages 257–268, November 2000.

[39] R. Thomas and M. Franklin. Using dataflow based context for accurate value prediction. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 107–117, 2001.

[40] D. Tullsen and J. Seng. Storageless Value Prediction Using Prior Register Values. In *Proceedings of the International Symposium on Computer Architecture*, pages 270–279, May 1999.

[41] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 281–290, 1997.

[42] H. Zhou. Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 231–242, September 2005.