Bootstrapping: Using SMT Hardware to Improve Single-Thread Performance

Sushant Kondguli
University of Rochester
Rochester, NY
sushant.kondguli@rochester.edu

Michael Huang
University of Rochester
Rochester, NY
michael.huang@rochetser.edu

Abstract

Single-thread performance improvement remains a central design goal for general purpose processors. Microarchitectural designs for the core have reached a plateau over the past years. However, we are still far from exhausting the implicit parallelism available in today's programs. One approach is to use a separate thread context to improve data and instruction supply to the main pipeline. Such decoupled look-ahead (DLA) architectures have been shown to be an effective way to improve single-thread performance. However, a default implementation requires an additional core. While an SMTfl avor is possible, a naive implementation is inefficient and thus slow. In this paper, we propose an optimized implementation called Bootstrapping that makes DLA just as effective on a single (SMT) core as using two cores. While fusing two cores can improve single-thread performance by 1.22x, Bootstrapping provides a speedup of 1.48 over a broad range of benchmark suites, making it a compelling microarchitectural feature for general-purpose microarchitectures.

CCS Concepts • **Computer systems organization** → **Architectures**; *Pipeline computing*; *Multicore architectures.*

Keywords Decoupled Look-Ahead (DLA) architectures, Simultaneous Multi-Threading (SMT), single thread performance.

ACM Reference Format:
Sushant Kondguli and Michael Huang. 2019. Bootstrapping: Using SMT Hardware to Improve Single-Thread Performance. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304052

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304052

1 Introduction

Single-thread performance improvement remains a central design goal for general-purpose processors. Over the years, microarchitectural designs have reached a plateau for the core: while there are more cores, features, and bigger structures today [1], the basic out-of-order design is no different from more than 20 years ago [2]. Correspondingly, performance measured by IPC has also stagnated.

Yet, we are far from exhausting available implicit parallelism: a simple modeling of some idealized systems shows at least a 5x performance potential remaining [3]. While it is unclear how much of this potential is *easily* extractable, we believe there are still plenty low-hanging fruits. In particular, the data and instruction supply chain remains a significant bottleneck. Addressing this bottleneck will not only lead to immediate performance gain but also amplify or enable other mechanisms that exploit implicit parallelism.

While more powerful branch predictors [4] and prefetchers [5, 6] are reasonable options moving forward, we focus on decoupled look-ahead (DLA) architectures for two interrelated reasons. First, DLA is more general-purpose as it does not depend on a particular access pattern to issue prefetches. Second, since the major resource needed is another thread context, there is thefl exibility to allocate the resource either to improve a single thread or to execute a different thread.

DLA does carry an overhead both in terms of hardware used (two cores) and the energy cost, but note two points: first, optimizations are being made and will continue to be in the future. Second, assuming energy doubles is a gross exaggeration, as we will discuss in more detail later. Indeed, DLA is already competitive against some current practices that boost single-thread performance [7]. In particular, using a wide-issue superscalar is one such practice. With the concern of energy efficiency, such wide cores are often treated as multiple narrower cores/clusters/slices by default, and only amalgamated to function as a single wide core when the situation permits. In this paper, we explain that this clustered SMT architecture offers a natural platform for a DLA system. We make the following contributions:

- Earlier designs such as Slipstream explored this space [8]. It is also easy to implement a more recent DLA model [9] onto an SMT substrate. However, we show that both

cases leave significant performance potentials unexploited (Section 4).
- We propose Bootstrapping which takes a few simple tweaks of a DLA design, but significantly improves resource utilization. One such tweak on the cache control allows both contexts to be efficiently contained by the cache. Another tweak dynamicallyfi nds a resource allocation that maximizes the performance of the thread pair. The design makes it all but unnecessary to use an additional core for look-ahead. This design acts as a high-performance foundation for continued innovation in the area of DLA.
- Over a wide-range of workloads, Bootstrapping achieves significant performance benefits, with a speedup of 1.21 (geometric mean). This is on top of the single-thread speedup of 1.22 by fusing two narrower cores into a single, wider core. The result is an average 1.48 combined speedup over a single (narrower) core.

The rest of the paper is organized as follows: wefi rst discuss the variants of DLA design and other related concepts like helper threads in Section 2. Then we provide an overview of the DLA design in Section 3.1 and discuss the Bootstrapping design in Section 3.2. Finally, experimental analysis is performed in Section 4 and ourfi nding are summarized in Section 5.

2 Background and Related Works

On-demand caching system alone provides a limited degree of anticipatory data fetching. A number of techniques complement such a basic system by following expected access patterns in a hard-wiredfi nite state machine, launching micro helper threads, or executing a single, more self-sustained look-ahead thread for targeted prefetches as well as branch prediction. We call the last type a decoupled look-ahead (DLA) approach.

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 [10–37]. Although these micro helper threads are an immensely useful concept, marshaling a very large number of micro threads can bring practical issues [38].

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, 7–9, 39–47]. 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 [48]. 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 [45] 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 [39]. Garg and Huang experimented with a more purpose-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 promising 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.

Other than explicitly launching a helper thread, many proposals have dealt with reducing the chance a conventional microarchitecture is blocked [27, 49–60]. Many designs share a theme of checkpointing important state, clean up some structures to allow further (speculative) execution. Sometimes the sole purpose of the execution is warm-up [58]. In this latter case, the design is more closely related to helper threading. Finally, there are recent incarnations of the basic concept of DAE to separate the computation part of the program from memory accesses [61, 62].

3 Bootstrapping Architecture

Wefi rst provide a brief discussion of the DLA execution model and a basic implementation (Section 3.1); then discuss new support to allow efficient DLA execution on an SMT (Section 3.2).

3.1 Overview of DLA Baseline

Note that numerous DLA designs have been proposed in the past (*e.g.*, [8, 9, 39, 42]). Most share the general working principle, which by itself does not guarantee large performance gains. The key challenge in DLA design is to keep the look-ahead thread fast and yet accurate in order to achieve *deep, sustained* look-ahead [63].

Our baseline DLA architecture is largely based on [9]. Specifically, a *skeleton* of the original program binary is generated offline through an automated binary analysis. The skeleton includes all the control instructions and their backward dependence chain. A subset of memory instructions are also included in the skeleton as prefetch payloads along with their backward dependence chain. Figure 1 shows a pseudo code that generates a basic skeleton used by DLA. Note that the process is almost identical to the one used in [9], which includes more detailed discussions about the choice of design parameters and additional optimizations.

Basic Skeleton Mask Generation Algorithm

```
1:  profiled_instructions = profile ( execute ( application (training_input) ) )
2:  seeds = { }
3:  for each instruction in profiled_instructions:
4:      if instruction.l1_misses > instruction.l1_accesses/100   :
5:          seeds.insert(instruction)
6:      if instruction.l2_misses > instruction.l2_accesses/1000 :
7:          seeds.insert(instruction)
8:      if instruction.is_branch :
9:          seeds.insert(instruction)
10: mask[ length (profiled_instructions) ] ← { 0 }
11: for seed in each seeds:
12:         mask[ seed ] ← 1
13:         mask ←   mask | backward_dependents(seed)
```

Figure 1. Pseudo code for the skeleton generation process used by DLA. Note that the runtime profile uses training inputs to identify memory instructions that are likely to experience cache misses. The binary is parsed to identify control instructions. The skeleton includes these memory and control instructions along with their backward dependencies.

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). It may appear wasteful to execute the same code twice, however, many actions are not repeated at runtime *e.g.*, off-chip accesses are only time shifted and not repeated, wrong path instructions are limited to LT and LT only executes a subset of the original program.

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

1. Containment of speculation: LT cannot be allowed to update architectural state since its execution often involves speculative optimizations. For the most part, LT's state is naturally confined to its thread context. The only additional support needed is about the dirty lines in the private caches. In the look ahead mode, dirty lines are not used to supply coherence requests from other cores and simply discarded upon eviction. In other words, we only need to make the private cache state of LT invisible to the rest of the system.
2. Communication of look-ahead results: LT already warms up the caches it shares with MT without any additional support. However, a mechanism to send over additional results from LT to MT is valuable. We include two FIFO queues (Fig. 2) Branch Outcome Queue (BOQ) and Footnote Queue (FQ). The BOQ serves a multitude of purposes.
   - First, it passes LT's branch outcomes as predictions to MT. This ensures that in the steady state, the majority of the branch mispredictions are experienced only in LT.
   - Second, it is a simple and effective mechanism to detect incorrect look-ahead controlfl ow. When a branch prediction fed by LT turns out wrong (which is relatively rare at about 0.06 per kilo instructions), MT can *reboot* it.
   - Third, 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).
   - Fourth, 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. FQ is used for other less frequent but wider data like prefetch addresses and indirect branch targets.
3. Support for instruction masking: In our design, the skeleton executed by LT only includes a subset of instructions from the original program binary. This allows us to use only the original program binary along with a set of bits to mask off instructions not included in the skeleton. These bits are immediately deleted upon fetch. While the mask bits can be generated offline or online through dependence analysis of the program binary, we model a system where mask bits are generated offline and stored in the program binary. During runtime, LT will separately fetch the mask bits along with the instructions in the I-cache.[1]

Runtime Operations: With this architectural support, we now briefly describe the overall operation of the baseline system in DLA mode (Fig. 2). In this setup, the program binary is profiled offline to generate the skeleton, which is expressed by mask bits. The two threads (LT and MT) run

[1]Instructions and the corresponding mask bits arrive asynchronously in the I-cache. In the case the instructions arrive before the mask bits from the L2 cache, we simply set all bits to 1, effectively assuming that the all instructions in that cache line are present in the skeleton. When the mask bits arrive, the updated version will be used thereafter.

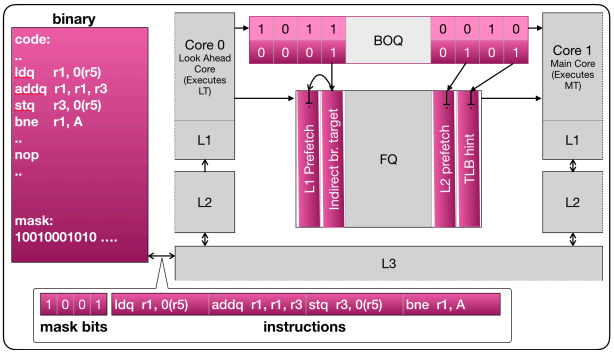

Figure 2. Architectural support for baseline DLA. The mask bits are stored with program binary and fetched separately by LT. Branch outcome queue (BOQ) helps LT communicate the controlfl ow data to MT. Footnote queue (FQ) helps pass on additional runtime information from LT to MT.

on two separate cores which are connected by the queues discussed above and the system always runs in DLA mode. As we will see later, these are not intrinsic requirements to implement DLA but help describe its most basic incarnation. When a regular thread (MT) 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 their architectural and microarchitectural states, with the following exceptions:

1. Fetch: The fetch unit of the core executing the actual program thread (MT) bypasses its branch predictor and draws its branch predictions from BOQ. The fetch stage is stalled when BOQ is empty. If the footnote bit of the BOQ entry is set, one or more entries are dequeued from FQ and actions are performed according to the content type. For example, a branch target entry prompts MT to use the corresponding entry for branch target prediction and a prefetch entry prompts MT to launch prefetch for the corresponding address. Finally, the highly accurate branch predictions from BOQ makes decoupled fetch more practical and useful. MT[2] thus employs a fetch queue and stalled decode stage does not stall the fetch stage from populating the fetch queue.
   The fetch unit of the core executing LT issues I-cache requests for skeleton mask bits along with instructions. In case of an I-cache miss, the cache controller issues two requests to the L2 cache: to the instructions at the address A_i and their masks stored in address $A_m = f(A_i)$. The fetch unit will skip the instruction if its mask bit is "off".[3]
2. Commit: LT writes hints for MT into the queues (BOQ and FQ) during its commit stage. Specifically, outcomes of the conditional branches ("taken" or "not taken") are stored in the BOQ in FIFO order. In addition to the continuous branch direction hints, occasionally LT has other hints. Whenever it encounters a branch target miss or a data cache miss, it will pass the relevant address through the FQ and set the footnote bit in the most recent BOQ entry.
   At commit, if MT realizes that the branch prediction it received via BOQ was incorrect, it triggers a *reboot*. A reboot signal is sent to LT asking it tofl ush its pipeline and stop using FQ to send data. MT then sequentially copies its architectural registers to FQ.
3. Finally, LT may occasionally receive a reboot signal from MT. It means that LT has veered off the control flow of MT and needs course correction. In such cases, LT squashes all instructions from its pipeline and re-initializes its architectural state by copying the values it receives from MT.[4]
   Note that it is possible that a fault occurs in LT execution. In our experiments, wefi nd this to be exceedingly rare (with the median and maximum observed rate of 0.4 and 1.2 per million instructions respectively). Most of these are page faults. In these cases, we simply halt the LT and triggers a reboot when MT catches up.

3.2 Bootstrapping in SMT

3.2.1 Overview of Design Updates

Fundamentally, LT does not require an extra core. A thread context in an SMT core works too in principle. In practice, however, when sharing the same execution hardware, LT competes with MT for resources and can thus reduce or even negate the look-ahead benefits. But as we will see later with quantitative analyses, judicious use of the shared resource can allow a far more efficient mode of look-ahead. As a result, having a dedicated thread for look-ahead becomes an effective "bootstrap" to pull MT ahead, hence the name.

To support DLA on an SMT core, wefi rst need to ensure correctness. In a dual-core DLA, speculative data by LT are naturally contained by its private cache. In an SMT core, this is only slightly less straightforward as simple modifications can achieve isolation of LT's speculative data.

Most of Bootstrap's architectural support is about resource utilization. First, with two threads, there is an increase in

[2]A decoupled fetch unit can in principle improve performance for the conventional microarchitecture. Its performance benefits depend on the exact microarchitectural detail. In our setup, a decoupled fetch unit has negligible impact on either LT or the underlying microarchitecture.

[3]The use of masks bits slightly reduces the effective storage space of the unified caches. We model this in our simulations but note that its effect on performance is negligible.

[4]We model a 64-cycle delay for the copying, but the overall cycle impact of handling a reboot is around 200 cycles. Adding another 200 cycles to the delay would only slow the system down by about 2%.

data footprint. We discuss our cache control approach in Sec. 3.2.2 that tries to minimize the negative impact.

Second, the execution engine needs to support the throughput of both threads. In a DLA system, the speed is determined by the slower of the thread pair. Careless execution resource allocation can exacerbate the problem and make the bottleneck thread even more of a bottleneck. We found that a simple adaptive resource allocation approach works quite well. We discuss this in Sec. 3.2.3.

Finally, the SMT hardware provides opportunities to further optimize the DLA execution. We briefly discuss some examples in Sec. 3.2.4.

3.2.2 Cache Control

SMT already builds in support to isolate threads and naturally supports the two-threaded execution in DLA. The only remaining issue is that of speculative, LT-written data in the cache. Just like in the baseline DLA, when LT writes to the cache, the data is fundamentally speculative and can not be used by the rest of the system. One way to support this is to have a speculative bit to lines written by LT and differentiate them from other lines. Again like in the baseline DLA, when a speculative line is evicted from L1, it is discarded; and upon a reboot, all speculative lines are gang-invalidated. While this support is enough for correctness guarantee, there are optimization opportunities.

Thefi rst opportunity is to realize that LT can use MT's data for the most part. So both threads can access the same set of data. The only exception is dirty, speculative data from LT, which can not be accessed by MT. So when LT writes to a cache line for thefi rst time, we make a copy of it and set the speculative bit. Subsequent reads and writes from LT will only use the speculative copy. On the other hand, only a non-speculative copy can be used to service an MT request. Again, the speculative line will be discarded upon eviction. When managed thus, the only increase in cache footprint is due to the lines LT writes to. For a very rough estimation: Due to load and store instruction ratio in typical programs, on average about 22% of data in cache are dirty; Since the skeleton is only a portion of the original code and brings about 66% of stores, LT will bring in about 14% extra footprint. In other words, increasing the cache size by about 1/7 would compensate for this footprint expansion. But even this resource pressure can be reduced, making cache expansion unnecessary.

Consider a store instruction s that is on the skeleton (Fig. 3-a). When s is committed by LT at time t_1, a new, speculative version of cache line x is created. At time t_2, when the same instruction is committed by MT, the speculative version is no longer necessary and should be evicted to save space. Indeed, the speculative data is possibly incorrect and thus ought to be evicted as soon as possible.

Intuitively, a simple versioning system can provide a good hint on the appropriate time to evict a speculative data. Every

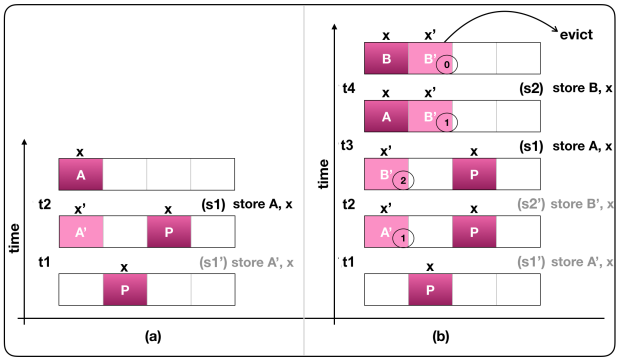

Figure 3. Example of a desired cache version control. x and x' indicates some address and its LT-produced speculative version, while P, A, and B indicate values. (a) When the value of x goes from the previous value P to A after time T2, x' is no longer useful and we want to keep only the new x. (b) A slightly more complex cases in which the two versions are only supposed to merge back to one after time T4.

time LT writes to a cache line, the speculative data becomes a version newer.[5] Similarly, when the same store instruction writes in MT, the non-speculative version got newer and when the two versions match (*e.g.*, at time t_4 in Fig. 3-b), the speculative version becomes unnecessary.

In practice there is a complicating factor to maintaining such version tracking: The addresses used in the two threads are not guaranteed to be the same. In the example, if s_1 and s_1' update different cache lines ($x \neq x'$), then we should not simply allow s_1 and s_1' to update the version of their corresponding cache line (x and x'). This is not a mere theoretical possibility: about 12% of the stores in LT mismatch the address of their counterpart in MT. If we ignore the case, the mismatched lines will linger in the cache longer than necessary and result in a small but tangible performance penalty of about 2%.

There are two different ways of addressing the issue. The first is more direct and conventional: make sure version adjustment is always about a pair of store instructions (s_1 and s_1'), not their respective cache lines. We let s undo the version increment due to s'. When a version drops to 0, it becomes unnecessary and will be evicted. This can be implemented, for example, by using a FIFO queue to communicate the cache line identity from LT to the trailing MT. Clearly, the extra hardware requirement is undesirable.

The second way is to use a best-effort, approximate approach discussed below which is also used in our evaluations. The idea is to place a speculative line in a position in the LRU stack that would likely result in its eviction at the right time: to coincide with the commit of the non-speculative version

[5] In our experiments, the largest version difference observed is 6, suggesting a 3- or 4-bit counter – together with some overflow prevention mechanism – might be sufficient if indeed a version-based solution is pursued.

of the store in MT. Keep in mind: there is no correctness concerns and this gives usfl exibility of choice in a wider design space.[6]

This position can be estimated as follows. At the commit time of store s' in LT, we know how far behind the non-speculative version of the same store (s) in MT is: the number of entries in the BOQ (N_{BOQ}) indicates how many basic blocks MT is trailing behind. If we further measure the average number of misses per basic block (MPBB), we can know roughly how many misses (and thus replacements) we expect to occur to each set when MT catches up to store s: $n = \lfloor \frac{N_{BOQ} \times MPBB}{N_{set}} \rfloor$. We move the line to the n^{th} least recently used position.

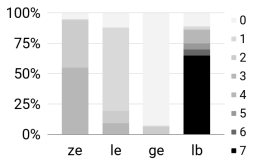

Figure 4. Distribution of insertion positions for speculative cachelines. 0 (white) and 7 indicate LRU and MRU positions respectively.

Again, the approach does not aim to eliminate any line the moment it becomes unwanted but tries to reduce its unnecessary time in the cache in a very simple way. Wefind this approach to provide a comparable performance to one that employs exact tracking. As it turns out, for the vast majority of applications, the decision is almost always to insert into the LRU position. Fig. 4 shows the distribution of the inserted position for the few outlier applications. Always inserting into the LRU position is thus another approximate solution, though at a cost of about 1% on average and up to 6% performance degradation, making it hard to justify in our opinion.

3.2.3 Adaptive Resource Allocation

Executing two (almost) identical threads in an SMT processor is a familiar concept, such as for redundancy (*e.g.*, [40]). More relevantly, an SMT-based Slipstream design has been evaluated [8]. Similarly, our baseline DLA can be easily implemented on an SMT substrate. Indeed, if we already have the ability to fuse two cores into a wider one that supports SMT [1], we can build a straightforward DLA-on-SMT system, which simply fetches instructions from both threads in a round-robin fashion and gives no priority to either thread. This system is in general faster than running DLA on two cores, largely because sharing resources improves utilization. However, such a naive implementation is suboptimal to a significant extent, as we will show in Section 4. This is due to the opportunity costs: resources could be utilized much more effectively by favoring the bottleneck thread. The question becomes that of the mechanism and policy of resource allocation. In both cases, wefi nd that simple approaches work reasonably well.

Mechanism: We allocate execution resource by controlling the number of re-order buffer (ROB) entries occupied by each thread. We do so at increments of 1/16 of the capacity. Thus MT and LT can divide up ROB at a ratio of 1:15, 2:14, and so on. The fetch stage will fetch instructions from both threads in a round robin fashion, skipping the thread that has reached its designated capacity. Finally, it is possible to give LT no resources at all. In that case DLA degenerates into conventional execution.

Policy: Since the best allocation depends on the program behavior, we divide the execution into recognizable code segments (*e.g.*, a loop) and adapt for each segment independently. The policy is based on the heuristic of repeating instances of the same code segment are similar in behavior. Thus, we can adopt a trial-and-error approach: testing out each different configuration, and picking whichever seems to work the best for all future instances. The implementation of the policy includes architectural support for marking the code-segment instance boundaries and an algorithm governing the search through the configuration space.

Implementation: In our experience, a simple and effective way to divide code is to identify certain backward branches that we call *eigen-branches*. Take a simple loop for example. The typicalfi nal backward branch marks the boundary between repeating iterations of the loop body, which often show repeating behavior. When we broaden the definition of branch to include function calls and unconditional jumps, any repetition of the same static code – including recursion – will necessarily involve backward branches. They therefore serve as a convenient way of demarcating the repeating instances.

Not all backward branches necessitate repetition. We only focus on those that do repeat, which often manifest as back-to-back ones, *i.e.*, those without other backward branches in between two consecutive instances. These are easy to detect: a *loop-branch register* (LBR) keeping the PC of a detected backward branch is all we need. But some genuine loop branches do not occur back-to-back, for example, when there is another backward branch within the loop body. Tofilter these unwanted backward branches out, we augment the LBR to also keep track of the target of the backward branch. If we encounter a new backward branch, then there are a few different possibilities: ① the new branch is the "real" loop branch and the one in the LBR is part of the body, ② the other way around, ③ both are loop branches with one of them being the outer branch, and so on. While the detailed decision logic is shown in Fig. 5, the general heuristic is that

[6] This is yet another small benefit we gained in the DLA design style. While the upfront cost of some redundancy is obvious, these (sometimes small) benefits are just as real.

a backward branch is considered to be nested within another one if its address falls between the PC and the target of the other branch. This heuristic does not cover all possibilities but works well for normal code.

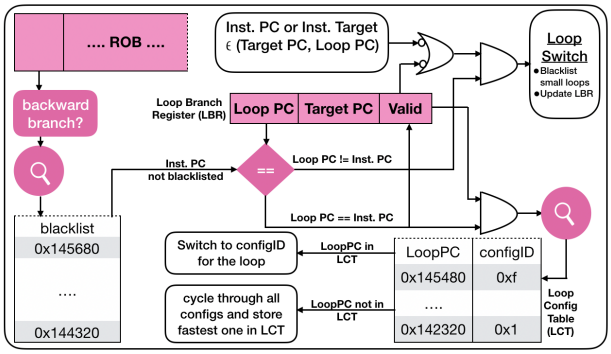

Figure 5. Loop identifier decision logic. As a backward branch retires, the loop branch register is updated to mark the beginning of a new loop or new iteration of the same loop provided the retiring branch is not already blacklisted or nested within the branch currently occupying the loop branch register.

After blacklisting some inner branches (treating them as if they are not backward branches), all other back-to-back backward branches are identified as eigen-branches. Thus, the entire execution is marked by a sequence of eigen-branches. The segment between two neighboring instances of the same eigen-branch is, for our purposes, an iteration.[7] Depending on the eigen-branch, a single iteration may be too short to accurately measure its behavior. We thus use a period of an integer number of iterations longer than a certain length, say, 10,000 dynamic instructions. For notational convenience, we call such a period a *shift*.

To decide on the best allocation, we simply go through trial-and-error testing all configurations: when wefi rst encounter an eigen-branch, we test the IPC of thefi rst configuration on thefi rst shift, the second configuration on the next shift, and so on. After 16 shifts, we would know what appears to be the fastest configuration. From then on, when we encounter that eigen-branch, we just select this fastest configuration. This algorithm can be easily implemented in afi nite-state machine with insignificant storage need. In our empirical observations, eigen-branches are limited in numbers (1-42 in 1B instruction windows) and thus do not require significant storage for state. Even in the occasional displacement of a state for an eigen-branch, a median loop lasts over 5 million instructions, 30 times longer than the time needed for testing.

[7]Segments between two neighboring but different eigen-branches are ignored because they are statistically insignificant. (less than 0.001%)

Alternatives: Note that there are other proposals to track program phase behavior based onfi xed lengths [64]. In our experience, our approach provides noticeably superior results. Specifically, we have experimented with using a design based on [64]. Such a design generally leads to performance loss regardless of the granularity used (including the default case of 10 million instructions). Indeed, the best we can achieve after tuning the phase granularity is a mixed bag with a small (1%) average performance gain. In contrast, our design improves performance in all cases. A detailed analysis of the reason is beyond the scope of this paper but has been dealt with to some extent elsewhere [65].

3.2.4 Other Optimizations

There are a number of other possibilities to optimize DLA execution on an SMT substrate. We briefly discuss some.

- **Fast reboot:** When LT veers off the right path, we need to reboot it. On an SMT substrate, initializing the register state can be as quick as afl ash copy of the register alias table (RAT), greatly expediting the reboot. The only change to the renaming logic after the copying is to ensure correct recycling. In the original design, a physical register (say, P_{102}) is guaranteed to be mapped to one logical register (say, R_{12}) when it is allocated and will be recycled when the next instruction writing to R_{12} retires. With RAT copying, P_{102} will appear in both LT and MT's RAT, and LT needs to prevent releasing these initialized physical registers. This can be achieved by having a single do-not-release bit set for each entry of RAT upon copying. These bits will be reset when a new version allocated in LT updates their corresponding entries.
- **Prioritization:** We can assign a higher priority to the thread allocated more resources. However, we found the performance benefit to be small($< 1\%$).
- **Register utilization:** LT does not have correctness constraints and thus can recycle registers early for better utilization. However, at our current configurations, the potential is insufficient ($< 1\%$) to justify any support.

4 Experimental Analysis

In this section, we perform the experimental analyses of the proposed design. The simulation setup is detailed in Sec. 4.1. We show the bottom-line results of a complete system in Sec. 4.2 and provide more detailed analyses of the design in Sec. 4.3.

4.1 Simulation Setup

The main question we want to answer is whether Bootstrapping can be competitive with some of the practices of enhancing single-thread performance. Our main reference point will be the dynamically-formed-wide-core approach. Take a

leading example of IBM POWER9: two cores (a.k.a. superslices) can form a wide (12-decode) SMT core [1]. We model a wide-issue SMT system loosely after POWER9's SMT-8 core configuration and refer to this as a "full" core (FC) fused from two "half" cores (HC).[8]

	Half Core (HC)	Full Core (FC)
Cores + L1	8/6/8/8 (f/d/i/c), 256 ROB, 192 LSQ, 128-int/128-fp PRF, 8-int, 4-mem, 8-fp FUs, 32kB/4way dcache	16/12/16/16 (f/d/i/c), 512 ROB, 384 LSQ, 256-int/256-fb PRF, 16-int, 8-mem, 16-fp FUs, 64kB/8way dcache
	12-16 stages, out-of-order, 3 GHz branch predictor: SC-L-TAGE [67], 4k-entry BTB, 32-entry RAS dcache: 64B blocks, 3 ports, 1ns, 32 MSHRs, LRU, stride prefetcher	
L2	256KB, 8-way, 64B blocks, 2 ports, 3ns, 32 MSHRs, LRU, BOP [68]	
L3	2MB, 16-way, 64B blocks, 12ns, LRU	
RAM	DDR3-1600MHz, 4GB, 2 channels, 2 ranks/ch, 8 banks/rank, t_{RCD}=13.75ns, t_{RAS}=35ns, t_{FAW}=30ns, t_{WTR}=7.5ns, t_{RP}=13.75ns	

Table 1. System configuration.

We use Gem5 [69] for simulation purposes. We modify Gem5 to model DLA and Bootstrapping. Our baseline FC is an aggressive out-of-order pipeline with state-of-the art prefetchers and branch predictors. We model a 256kBits SC-L-Tage branch predictor as described in [67] to predict branches. A stride prefetcher that can identify 32 different strides and prefetches them with a degree of 4 is modeled to prefetch data into L1. Additionally, BOP [68] is modeled to prefetch data into L2. This prefetcher configuration was chosen because it was found to provide the best average baseline performance out of various other state-of-the-art prefetcher designs we evaluated [38]. The DRAM is modeled using DRAMCtrl embedded in Gem5. More details about the parameters we used to configure the system are shown in Table 1.

For comparison, we have also modeled a few similar approaches: Slipstream [8], B-Fetch [59] and CRE [60]. Slipstream and CRE can be easily ported on to an SMT substrate. For Slipstream the on-chip resources are equally split between the A-stream and R-stream. Similarly, for CRE the resource are equally split between the runahead engine and main core. This prompts CRE to prefetch data in L1, which, in our evaluations, provides higher performance gains than just prefetching data into LLC as suggested in [60]. B-Fetch requires a special purpose pipeline. For fairness, we model the main core to use only half of the on-chip resources on the SMT substrate and model the B-Fetch pipeline as presented in [59] to speculate the controlfl ow and launch prefetches.

[8]Note that in the original plan of the FC, it can devote all resources to just a single thread. But the plan was not carried out in the implementation. [66].

We use McPAT [70] to model CPU's energy consumption and assume a 22nm technology node. We modified McPAT to more accurately model the baseline along with DLA and Bootstrapping. DRAMPower [71] is used to compute the energy expenditure of main memory.

We evaluate our proposal over a broad set of benchmark suites. The SPEC2006 [72] benchmarks are evaluated using reference inputs. A graph application benchmark suite, CRONO [73], is evaluated using graph input data structures from google, amazon, twitter, mathoverflow and california road-networks. This suite consists of the following applications: ap (all pairs shortest path), bc (betweeness centralities), bf (breadthfi rst search), co (community detection), cc (connected components), df (depthfi rst search), pr (pagerank) and tr (triangle counting). We also use embedded applications from STARBENCH [74] and use large inputs provided by the benchmark suite to evaluate them. The applications belonging to STARBENCH are km (kmeans), md (md5), rg (rgbyuv), st (streamcluster) and ti (tinyjpeg). Finally, we also pick applications from NPB (NAS Parallel Benchmarks) to represent scientific applications and evaluate them using C class of workloads provided by the suite.

All of the benchmarks are compiled using gcc with -O3 flag. To reduce simulation time we use SimPoint Tool [75] to generatefi ve simpoints per benchmark with 10 million instruction intervals. We warm up the caches for 100 million instructions before beginning the simpoint interval. All the simulation results we report are obtained from these simpoints.

4.2 Overall Benefits

Performance: Wefi rst compare single-thread performance of the four main ways of using a FC plus a reference point of using two FCs: ① FC: using the FC monolithically; ② DLA (2xHC and 2xFC): using the two HCs (and for reference two FCs) to run baseline DLA; ③ Bootstrapping : using the Bootstrapping design on the FC; and ④ Throughput (SMT): using the SMT of FC to run two copies of the same application for better throughput. In Fig. 6 we plot their throughput normalized to a half core (HC).

Thefi gure shows performance of each of the individual applications along with an overall geometric mean which is included in the bottom half of thefi gure and indicated as "GM". Thefi gure contains a lot of information that can be summarized as follows:

1. A wider core (FC) and baseline DLA (2xHC) both provide high performance compared to an aggressive underlying microarchitecture sporting state-of-the-art branch predictor and prefetchers. Neither FC nor DLA ever slows down the system in our experiments and can achieve speedups as high as 1.74x and 2.08x respectively. Overall, FC achieves an overall geometric mean speedup of 1.22x and DLA (2xHC) achieves an overall

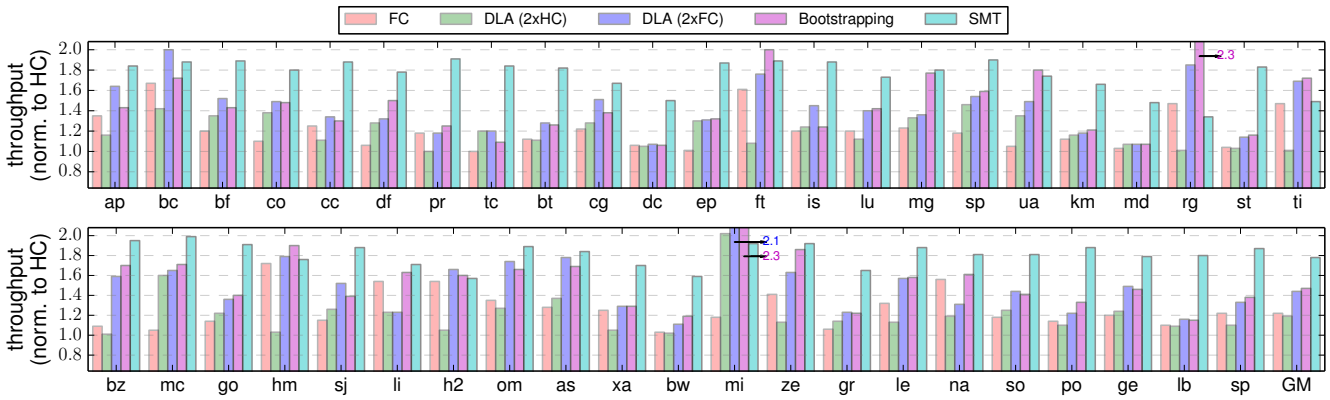

Figure 6. Comparison of throughput (normalized to HC) obtained by FC, by DLA using 2 half cores (DLA_2xHC) or 2 full cores (DLA_2xFC), by Bootstrapping, and by using SMT to run two copies of the same application. The top half of thefi gure contains applications from crono, starbench, and npb and the bottom half contains applications from spec2k6. Geometric mean across all applications is also included in the bottom half and is indicated by "GM".

geometric mean speedup of 1.20x. However, their impact on performance can vary significantly depending on the application. For instance, choosing the right optimization (DLA or FC) depending on the application raises the overall speedup over HC to 1.31x.
Note that this comparison is somewhat unfair as making a monolithic FC is perhaps far more challenging beyond a certain scale than linking together two largely independent HCs. However, making such a wider core to enhance single-thread performance has been one (if not the) chosen route for industry [66]. A simple DLA system is not yet a clear winner in performance. We need many optimizations. Fortunately, there seem to be plenty of conceivable optimizations, and they are often easier to implement thanks to the lack of absolute correctness constraints on LT.

2. Bootstrapping significantly outperforms both FC and DLA (2xHC). On overage Bootstrapping is 21% faster than FC and 24% faster than DLA (2xHC). Effectively, it is able to combine the benefits of a wider core and DLA. Overall, Bootstrapping achieves a geometric mean speedup of 1.48x over HC. Later (in Sec. 4.3.2) we will see that Bootstrapping also outperforms a number of other related approaches and the performance is rather insensitive to configuration details.
3. With Bootstrapping, it is no longer necessary to employ two cores for DLA. Using DLA on two full cores (2xFC) provides a performance benefit of 1.18x over FC compared to 1.21x obtained from Bootstrapping. This is because with a reasonably wide pipeline, the benefit obtained from efficiently utilizing and sharing on-chip resources is more pronounced than having more resources that multiple cores offer. We will show in Sec. 4.3.2 that Bootstrapping is comparable with dual-core DLA with much narrower pipelines.
4. Finally, we see that if absolute throughput is the goal, running multiple independent threads is still likely a better choice than Bootstrapping with an average of 21% performance advantage. However, note that running an additional thread brings in extra pressure for shared resources and sometime this can make the throughput less than using Bootstrapping. In 6 out of 44 applications Bootstrapping is already the better choice. We believe this gap will continue to shrink as the design of DLA is further optimized and new techniques are being invented.

Efficiency: One commonly expressed concern about DLA architectures is the energy cost. While it may be tempting to assume that executing the same program twice will double the energy cost in DLA, it would be a gross exaggeration even for baseline DLA. Table 2 shows the normalized instruction throughput in different stages and average normalized energy from four different configurations: HC, FC, DLA (2xHC), and Bootstrapping (BTSP). All throughputs are normalized to the commit stage of HC.

		D	X	C	Energy	Speedup
HC		1.25	1.16	1.00	1.00	1.00
BTSP (1xHC)	LT	0.32	0.31	0.27	1.12	1.19
	MT	1.03	1.02	1.00		
FC		1.32	1.20	1.00	1.52	1.22
DLA (2xHC)	LT	0.54	0.52	0.48	0.52	1.2
	MT	1.03	1.03	1.00	0.82	
BTSP (1xFC)	LT	0.36	0.35	0.30	1.62	1.48
	MT	1.02	1.01	1.00		

Table 2. Average activities in Decode, eXecution, and Commit stages, and energy for both threads in DLA and Bootstrapping. All activities are normalized to commit stage activity of HC. All energies and speedups are normalized to HC. ST indicates conventional single-threaded execution.

As we can see, LT executes only a subset of the instructions committed by MT. It also significantly reduces wrong-path instructions experienced by MT in all pipeline stages. The relatively low activity overhead coupled with the reduction in execution time and corresponding energies results in only 7% energy overhead over FC for a significantly higher speedup.

4.3 Detailed Analyses

We now perform a number of experiments to help understand the effects and utility of each element (Sec. 4.3.1); how well the design performs across different configurations of the underlying microarchitecture and how well it compares with other designs (Sec. 4.3.2); and show underlying impacts of the design choices (Sec. 4.3.3).

4.3.1 Performance Impact of Optimizations

Wefi rst compare performance of different design points in Fig. 7, incrementally adding elements discussed in Sec. 3. For brevity, we only show geometric mean speedup (over FC) with the range of the suite as an I-bar (same for some subsequentfi gures). For comparison, the dashed horizontal line shows the average speedup of a basic DLA architecture using two FCs.

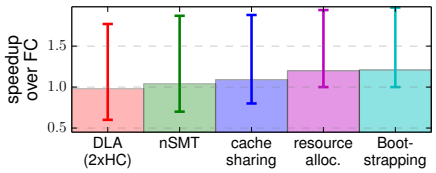

Figure 7. Speedup over FC. From left to right, the configurations correspond to basic DLA, nSMT, adding cache control, then adding adaptive resource allocation, andfinally Bootstrapping. The box and the bar show, respectively, the geometric mean and the range.

We start with the basic DLA (2xHC) configuration. We already discussed earlier that this basic design is slightly (2% on average) slower than FC. Next, for the sake of understanding, we look at a configuration that we call naive SMT (nSMT), which adds basic DLA support to the SMT substrate on FC and treats the threads as independent. Here LT and MT each gets half of the ROB. Compared to DLA (2xHC), the only benefit of nSMT is better use of shared execution resources of two HCs fused together. As can be expected, nSMT is marginally more effective and has about the same performance as FC. We see that even though SMT is a natural platform to implement DLA, a naive implementation is far from optimal.

Next, we progressively add the optimizations discussed earlier: cache control (Sec. 3.2.2); resource allocation optimization (Sec. 3.2.3); and fast reboot (Sec. 3.2.4). We see that performance incrementally improve to 1.21x over FC. Note that the exercise is to show that each bit of optimization helps a little in improving speed. The apparent amount each contributes depends on the order of adding these optimizations. Specifically, adding cache sharingfi rst produces a 5% performance boost while adding it last produces a gain of 11%. For resource allocation, these numbers are 6% and 14% respectively.

4.3.2 Comparison and Sensitivity Analysis

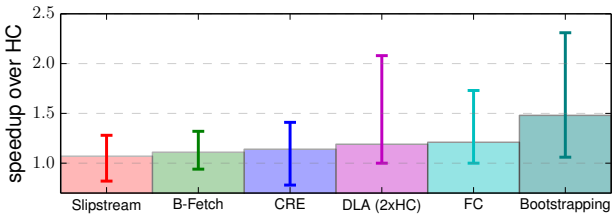

Figure 8. Comparison of speedup (normalized to HC) obtained by B-Fetch, Slipstream, CRE, DLA (2xHC), FC, and Bootstrapping.

While DLA can be a very effective paradigm for exploiting implicit parallelism, Bootstrap's performance is the result of many techniques either built into the baseline DLA or the ones discussed in this paper. Running two threads is not a new concept nor is it the reason for performance gain. Indeed it is mostly a compromise given the predominance of monolithic microarchitecture substrates. What really determines the performance is how effectively we can decouple the look-ahead activities. We briefly compare the overall performance among a set of related approaches (B-Fetch [59], Slipstream [8] and CRE [60]) in Fig. 8. We can see that the advantage of Bootstrap is clear and non-trivial.

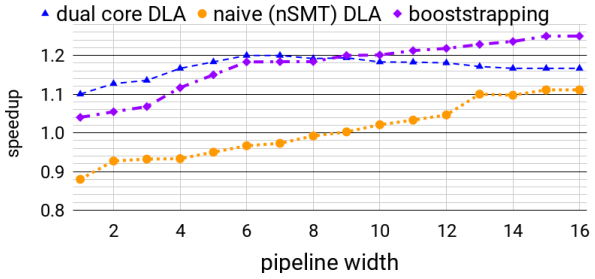

Figure 9. The speedups as a function of pipeline decode width (other resources scaled proportionally).

The utility of Bootstrapping can be viewed in a different way: DLA reduces pipeline bubbles at a price of extra execution bandwidth. In nSMT (a naive configuration which only adds basic DLA support to the SMT baseline), this price directly cuts into the benefit. Fig. 9 compares the speedup of DLA (on two cores), nSMT, and Bootstrapping as pipeline

(decode) width changes over a baseline configuration of corresponding width. We see that below 8-wide, nSMT does not provide a net benefit. In other words, in these configurations, execution resources are better devoted to the semantic thread than to any (inefficient) look-ahead thread. Bootstrapping makes more efficient use of the available resources and is consistently and significantly faster than nSMT. With the improvement, on average, it always pays to do decoupled look-ahead. At about 8-wide, there are enough resources that a single-core Bootstrapping is just as effective as a dual-core DLA. In fact, beyond that point, the advantages of Bootstrapping, like cache sharing and fast reboot overpower the advantages of increasing on-chip resources. Hence, beyond 8-wide, Bootstrapping, which uses a single core, is even more effective than DLA on two cores.

Finally, we note that the performance impact of Bootstrapping relatively to the underlying microarchitecture (FC) is rather insensitive to the configuration details. For instance, if we change the number of ROB entries in FC from 512 to 256 and 128, the speedup (over FC) changes from 1.21 to 1.19 and 1.16, respectively. If we reduce the L1 associativity from 8 to 4, the speedup changes from 1.21 to 1.20. If we reduce the L1 size of FC from 64KB to 32KB, the speedup remains unchanged.

4.3.3 Underlying Impacts of Design Choices

Cache control: In Bootstrapping, the cache is used in a different way than in the dual-core version of DLA. This impacts the cache miss rates for both LT and MT. Fig. 10 summarizes this information.

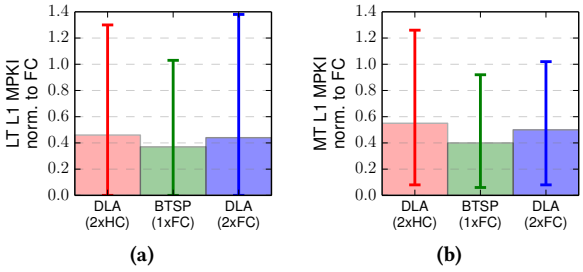

Figure 10. L1 cache's misses per kilo instructions (MPKI) experienced by LT and MT in three different configurations: DLA versions with 2 HC and 2 FC respectively and Bootstrapping. The box and the bar show the overall average and the range.

As we can see, Bootstrapping is noticeably better than DLA including the two-FC version which has twice the L1 capacity. For LT, the benefit is straightforward as it is expected to source data produced from MT. In the dual-core DLA, this can cause LT an L1 miss serviced (ultimately) by the MT. In Bootstrapping, some of these cases become a simple cache hit. For MT, the benefit comes from timing of LT prefetch. In dual-core DLA, the cache line address from an L1 miss in LT will be entered into the footnote queue and dequeued by MT later for prefetching. When LT is not sufficiently ahead of MT, the prefetch may not arrive in time. In Bootstrapping, LT directly fetches into the shared L1 and is thus generally faster. In dual-core DLA (2xFC), on average, MT enjoys 106 L1 hits (per kilo instructions) thank to LT's prefetching hints. In Bootstrapping, this number improves to 112.

	split	shared	smartShared
mean	50%	18%	12%
median	50%	12%	7%

Table 3. Percentage of L1 cache space occupied by LT specific data.

The downside of sharing a cache, of course, is the reduction in effective capacity. However, with the cache control discussed in Sec. 3.2.2, the capacity impact is small. Table 3 shows the percentage of capacity occupied by LT-specific data with and without the cache control. We see that with our cache control, LT's capacity impact is about 16% on average (broadly in line with thefi rst-order approximation in Sec. 3.2.2). Keep in mind that even this small portion tends to occupy the least recently used ways so their impact on hit rate due to capacity is even less. In our observations, in DLA (2xFC), MT sees 122.4 hits per kilo instructions on data brought in by MT. With Bootstrapping, this reduces only slightly to 122.2 hits.

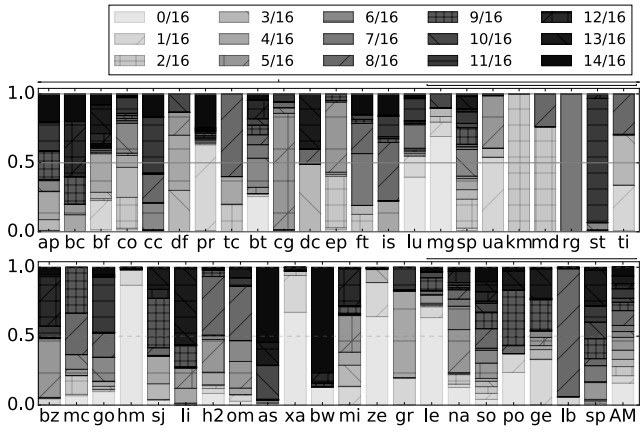

Figure 11. Runtime distribution of the configuration made by the resource allocation algorithm. Y-axis represents the portion of time a configuration is used. The legend shows all chosen choices. 1/16 mean that 1/16th of the ROB entries were allocated to LT. The rest were allocated to MT.

Adaptive resource allocation: In this paper, we adopt a trial-and-error approach for allocating resources to LT and MT. Fig. 11 shows the result of this control in percentage of

time each configuration is adopted. In thefi gure, the fraction indicates that of the ROB entries allocated to LT. Thus 0/16 refers to the configuration where LT is disabled. A few applications (e.g., `hm` and `lb`) have only a small number of configurations used. The remaining applications all used a large number of options based on the behavior of the code.

To see whether this adaptation is really necessary, we compare to two other levels of adaptivity. Thefi rst one selects a constant allocation for the system. It turns out the optimum configuration is 10/16. In the second level, we allow the allocation for each application to differ, but keeping it constant throughout the application. We pick the result of the best-performing configuration for each application. The result is shown in Fig. 12.

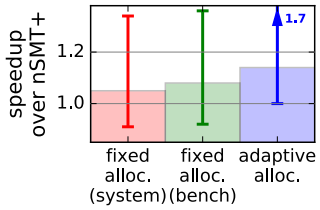

Figure 12. Performance benefit of adaptive resource allocation scheme compared againstfi xed resource allocation schemes that use one optimum configuration for the entire system and one one optimum configuration per benchmark. The speedup is normalized to nSMT+, a relatively naive configuration which adds basic DLA support to the SMT baseline and enables support for fast reboot and cache control.

As we increase the level of adaptivity, the performance gain goes from 4% forfi xed system-wide configuration, to 7% forfi xed application-specific configuration, to 14% in our design. This suggests that there is still significant intra-application behavior variation that warrants a morefine-grained adaptation. Of course, our design is just one possible approach to exploiting this variation.

Impact on reboots: In DLA, reboots are a cause of poor look-ahead. When LT is on the wrong path, its execution is less useful and potentially pollutive. On the other hand, allowing optimizations that is conventionally unsafe is a key advantage of DLA. The frequency of reboot is thus an important diagnostic metric in analyzing design tradeoffs. In our setup, depending on the application, the frequency of reboots ranges widely from 0.04 to 975 per million instructions, with a mean of 61.7 and median of 4.8. In general, Bootstrapping helps reducing the frequency of reboot (Table 4).

There are two contributing factors to this reduction. First, we use an adaptive mechanism to allocate resources between the two threads. The side effect of this adaptation is that at certain program points, not engaging LT turns out to be the best choice. As a result, no reboots will occur during these periods. Second, LT executes using a skeleton and thus can not produce complete memory states on its own. When residing on different cores, memory updates from MT do not propagate to LT via coherence actions. When they share the same cache, these updates naturally occur and help reduce the chance LT uses a stale value. The reduction in wrong-path execution for LT helps reduce the activity and energy overhead as shown earlier in Table 2.

	min	max	median	mean
DLA (2xHC)	0.04	974.6	4.8	61.7
DLA (2xFC)	0.06	943.9	4.2	62.5
BTSP (1xFC)	0.04	472.2	3.6	48.7

Table 4. The number of reboots per million instructions in different configurations.

5 Conclusions

In this paper we propose Bootstrapping, an optimized DLA architecture on an SMT substrate. It achieves the benefit of DLA without using an additional core as before. While this is conceptually straightforward, a study of design details reveals a few lessons:

- Naive porting of the DLA model on to an SMT platform is insufficient;
- State of the art in DLA already exposes quite significant parallelism that careful resource allocation is a necessity;
- Only a few tweaks of the basic DLA design are needed to make SMT a good platform for continued innovation of the DLA paradigm.

Overall, Bootstrapping can be made into an on-demand feature just like dynamically forming wider cores being attempted in commercial products. It is significantly more effective than the latter, achieving an average of 1.21 speedup over the underlying FC and 1.48 over HC. Bootstrapping also makes it more practical to support DLA on narrower SMT microarchitectures. Overall, it is a compelling architectural feature for general-purpose microprocessors.

Acknowledgments

The authors would like to thank the reviewers for their valuable feedback. This work is supported in part by NSF under grants 1514433 and 1533842.

References

[1] S. Sadasivam, B. Thompto, R. Kalla, and W. Starke. IBM POWER9 Processor Architecture. *IEEE Micro*, 37(2):40–51, 2017.

[2] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[3] S. Kondguli and M. Huang. R3-DLA (Reduce, Reuse, Recycle): A More Efficient Approach to Decoupled Look-Ahead Architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2019.

[4] A. Seznec. A 256 kbits l-tage branch predictor. In *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, volume 9, pages 1–6, 2007.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[11] 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.

[12] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, pages 115–126, October 1998.

[13] A. Roth, A. Moshovos, and G. Sohi. Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation. In *Proceedings of the International Conference on Supercomputing*, pages 356–364, June 1999.

[14] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the International Symposium on Computer Architecture*, pages 186–195, May 1999.

[15] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 37–48, January 2001.

[16] C. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–13, June 2001.

[17] M. Annavaram, J. Patel, and E. Davidson. Data Prefetching by Dependence Graph Precomputation. In *Proceedings of the International Symposium on Computer Architecture*, pages 52–61, June 2001.

[18] C. Luk. Tolerating Memory Latency Through Software-Controlled Pre-execution in Simultaneous Multithreading Processors. In *Proceedings of the International Symposium on Computer Architecture*, pages 40–51, June 2001.

[19] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proceedings of the International Symposium on Computer Architecture*, pages 14–25, June 2001.

[20] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice-processors: an Implementation of Operation-Based Prediction. In *Proceedings of the International Conference on Supercomputing*, pages 321–334, June 2001.

[21] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *Proceedings of the International Symposium on Microarchitecture*, pages 306–317, December 2001.

[22] P. Wang, H. Wang, J. Collins, E. Grochowski, R. Kling, and J. Shen. Memory Latency-Tolerance Approaches for Itanium Processors: Out-of-Order Execution vs. Speculative Precomputation. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 167–176, February 2002.

[23] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, October 2002.

[24] R. Chappell, F. Tseng, A. Yoaz, and Y. Patt. Difficult-Path Branch Prediction Using Subordinate Microthreads. In *Proceedings of the International Symposium on Computer Architecture*, pages 307–317, May 2002.

[25] S. Liao, P. Wang, H. Wang, G. Hoflehnerg, D. Laveryg, and J. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–128, June 2002.

[26] R. Chappell, F. Tseng, A. Yoaz, and Y. Patt. Microarchitectural Support for Precomputation Microthreads. In *Proceedings of the International Symposium on Microarchitecture*, pages 74–84, November 2002.

[27] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-Performance Throughput Computing. *IEEE Micro*, 25(3):32–45, May/June 2005.

[28] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's Rock Processor. In *Proceedings of the International Symposium on Computer Architecture*, pages 484–295, June 2009.

[29] J. Collins, S. Sair, B. Calder, and D. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the International Symposium on Microarchitecture*, pages 62–73, November 2002.

[30] I. Atta, X. Tong, V. Srinivasan, I. Baldini, and A. Moshovos. Self-contained, accurate precomputation prefetching. In *Proceedings of the International Symposium on Microarchitecture*, pages 153–165, 2015.

[31] 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.

[32] D. Kim and D. Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Transactions on Computer Systems (TOCS)*, 22(3):326–379, 2004.

[33] P. Wang, J. Collins, H. Wang, D. Kim, B. Greene, K. Chan, A. Yunus, T. Sych, S. Moore, and J. Shen. Helper Threads via Virtual Multithreading. *IEEE Micro*, 24(6):74–82, November 2004.

[34] Y. Song, S. Kalogeropulos, and P. Tirumalai. Design and Implementation of a Compiler Framework for Helper Threading on Multi-core Processors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 99–109, September 2005.

[35] 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.

[36] W. Zhang, D. Tullsen, and B. Calder. Accelerating and Adapting Precomputation Threads for Efficient Prefetching. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2007.

[37] C. Madriles, P. López, J. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez. Boosting Single-thread Performance in Multi-core Systems Through Fine-grain Multi-threading. In *International Symposium on Computer Architecture*, pages 474–483, 2009.

[38] 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.

[39] 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.

[40] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 84–91, June 1999.

[41] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W. Hwu. Beating In-Order Stalls with "Flea-Flicker" Two-Pass Pipelining. In *Proceedings of the International Symposium on Microarchitecture*, pages 387–399, December 2003.

[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.

[43] B. Greskamp and J. Torrellas. Paceline: Improving Single-Thread Performance in Nanoscale CMPs through Core Overclocking. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 213–224, September 2007.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

[49] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the International Symposium on Microarchitecture*, pages 423–434, December 2003.

[50] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Hiding L2 Misses with Checkpoint-Assisted Value Prediction. *IEEE TCCA Computer Architecture Letters*, 3, December 2004.

[51] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proceedings of the International Conference on Supercomputing*, pages 68–75, July 1997.

[52] A. Hilton, N. Eswaran, and A. Roth. CPROB: Checkpoint processing with opportunistic minimal recovery. In *18th International Conference on Parallel Architectures and Compilation Techniques, 2009*, pages 159–168, 2009.

[53] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Checkpointed Early Load Retirement. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 16–27, February 2005.

[54] T. E. Carlson, W. Heirman, O Allam, S. Kaxiras, and L. Eeckhout. The Load Slice Core Microarchitecture. In *International Symposium on Computer Architecture*, pages 272–284, 2015.

[55] 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.

[56] 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.

[57] M. Hashemi and Y. Patt. Filtered runahead execution with a runahead buffer. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 358–369, 2015.

[58] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 129–140, February 2003.

[59] 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.

[60] 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.

[61] T. Ham, J. Aragón, and M. Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 191–203, 2015.

[62] C. Ho, S. Kim, and K. Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 118–130, 2015.

[63] R. Parihar and M. Huang. Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2014.

[64] T. Sherwoood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proceedings of the International Symposium on Computer Architecture*, pages 336–347, June 2003.

[65] W. Liu and M. Huang. EXPERT: Expedited Simulation Exploiting Program Behavior Repetition. In *Proceedings of the International Conference on Supercomputing*, pages 126–135, June–July 2004.

[66] H. Le, J. Van Norstrand, B. Thompto, J. Moreira, D. Nguyen, D. Hrusecky, M. Genden, and M. Kroener. IBM POWER9 processor core. *IBM Journal of Research and Development*, 62(4), 2018.

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

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

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

[70] 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.

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.