Load Balancing in Decoupled Look-ahead: A Do-It-Yourself (DIY) Approach

Raj Parihar
University of Rochester, Imagination Tech
parihar@ece.rochester.edu

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

Abstract
Despite the proliferation of multi-core and multi-threaded architectures, exploiting implicit parallelism for a single semantic thread is still a crucial component in achieving high performance. Look-ahead is a “tried-and-true” strategy in uncovering implicit parallelism. However, a conventional, monolithic out-of-order core quickly becomes resource inefficient when looking beyond a small distance. One general approach to mitigate the impact of branch mispredictions and cache misses is to enable deep look-ahead. A particular approach that is both flexible and effective is to use an independent, decoupled look-ahead thread on a separate thread context guided by a program slice known as a skeleton. While capable of generating significant performance gains, the look-ahead agent often becomes the new speed limit. We propose to accelerate the look-ahead thread by skipping branch based, side-effect free code modules that do not contribute to the effectiveness of look-ahead. We call them Do-It-Yourself or DIY branches for which the main thread does not get any help from the look-ahead thread, instead relies on its own branch predictor and prefetcher. By skipping DIY branches, look-ahead thread propels ahead and provides performance-critical assistance down the stream to improve the performance of decoupled look-ahead system by up to 15%.

Keywords Implicit parallelism, Single-thread performance, Decoupled look-ahead, Do-It-Yourself branches

1. Introduction and Motivation
A decoupled look-ahead approach that employs an independent thread on a separate thread context to target branch mispredictions and cache misses is both flexible and effective [1]. Intuitively, there is a fundamental trade-off between the speed and helpfulness of the look-ahead thread. In a two-core look-ahead system, the look-ahead thread often becomes the new speed limit [2-3]. To better understand the speed limits of a decoupled look-ahead system, we studied four configurations to tease apart various performance bottlenecks. The result of this study essentially separates applications into two groups. In the first group, (about half of applications) the existing methodology of decoupled look-ahead is already effective. The applications sustained high IPC (around 3 or more in a 4-wide machine) and the performance difference from an ideal (perfect branches and caches) system is very small—less than 5%. For the remaining applications, the potential (performance gap between decoupled look-ahead and ideal system) is still large. For the majority of these applications, the decoupled look-ahead architecture is entirely limited by the look-ahead thread. This suggests that the look-ahead thread, while doing a good job eliminating performance hurdles for the main thread, is not running ahead enough. Speeding it up will further speed up the whole system.

2. Do-It-Yourself (DIY) Branches
First, we observed that certain branch based modules do not contribute to the effectiveness of look-ahead. These modules are devoid of hard-to-predict data-dependent branches and cache misses. Also they are quite frequent and mainly consist of a bunch of ALU operations. skipping DIY branches, look-ahead thread propels ahead and provides performance-critical assistance down the stream to improve the performance of decoupled look-ahead system by up to 15%.

Second, all iterations of a loop may not be required in the skeleton for prefetching, if the accesses from consecutive iterations happen to the same cache line. We also observed that a wide variety of library calls including printf, _OtsMove, _OtsFill and reduction operations are often useless in the look-ahead thread. Combined these together constitute a significant fraction of skeleton that can be avoided without diluting the quality of the look-ahead. In subsequent sections, we describe our methodology to systematically exploit the DIY branches in the context of look-ahead thread.

3. Methodology
While it may appear hard to identify the potential DIY candidates using static analysis upfront, we rely on an empirical method to identify and exploit the DIY branches. We start by removing a few dynamic instances/iterations of a branch based code module (e.g., iteration of a loop, dynamic instance of an if-then-else block and subroutine calls) from the skeleton and measure the performance impact. If removal of an instance improved the overall performance, we call the static module a DIY seed module. We also associate a duty cycle number with every DIY seed module which represents the fraction of dynamic instances/iterations skipped to achieve the best individual performance. We experimented with a few different duty cycles (in the range of 5–100%) for each seed to determine best dithering. We considered a number of code modules to be potential DIY, but for brevity discuss a few that we felt to be the easier to support and reason about.

• DIY calls: If a dynamic instance of a call is DIY then look-ahead thread will treat the call instruction as a NOP and proceed sequentially. DIY call, through branch queue (BOQ), will indicate to the main thread that no branches from this point onward are predicted by look-ahead thread until the matching return. Note that DIY property is associated with the call site as opposed to function itself. Every branch in the function body of a DIY call also becomes a DIY branch as shown in Figure–1(A).

1 Language support routine for character operations.
• **DIY loops**: A loop iteration is a DIY if it accesses the same cache line as previous iteration. Our skeleton is represented by a bit vector which masks off instructions not included in the skeleton. To support DIY loops, we have another set of masks that skips all but those instructions in the loop-carried dependence chain e.g. loop index computing instructions. We also include the exit branches and returns in the secondary mask – as shown in Figure–1(B). For DIY loop iterations, we simply switch to secondary mask in the look-ahead thread.

• **DIY branches**: If a forward branch (e.g. if-then-else) is DIY then the look-ahead thread will treat it as a NOP and all the enclosed branches as DIY – as shown in the Figure–1(C). Similar to DIY loops, DIY branches also have a secondary mask which only keeps exit branches, returns and jumps that may take the control outside the branch scope. Function calls inside DIY branches or loops should be treated as DIY as well.

**Final Skeleton**: After testing individual DIY seed modules for the best duty cycle we sort them based on their performance. Then we pick the top n (n=25 in this study) DIY seed modules and combine them in incremental manner. At this point, we could have used sophisticated metaheuristic i.e. genetic algorithm or simulated annealing – as proposed by [3]. Instead, we rely on one-pass incremental combining to create \( \sum n \) different configurations starting from each DIY seed and go all the way to the end of the list adding one seed at a time. Finally, we pick the configuration with the best performance. Because of interaction between individual DIY modules, this is only a sub-optimal combining and we plan to experiment with better heuristics in our future explorations.

4. Experimental Analysis

We conduct our experiments on a 4-wide, detailed out-of-order core that has support for look-ahead thread and detailed memory hierarchy. We measure performance over a window of 200 million instructions after fast-forwarding and warming the caches over the initial region of about 2 billion instructions. Other microarchitectural features are similar to the baseline design in [3].

**Overall Performance Impact**: Our proposed technique boosts the baseline decoupled look-ahead to achieve 1.43x speedup (dark bars in Fig–2) over single-thread baseline. This is a significant improvement of 11% from the baseline look-ahead speedup of 1.32x (lighter bars) over single-thread. Best speedup comes for 176.gcc where decoupled look-ahead performance increases by 15%. Overall, we improve the performance of baseline look-ahead by 8% (geomean) for the look-ahead thread bound applications.

**Duty Cycle Sensitivity**: Loops tend to have better performance for low duty cycles (5–20%), while functions that do minor adjustments are better off when skipped completely. For brevity, we present one case of DIY call from 179.art that has varying performance for different duty cycles (Fig–3). In this case, executing only a few instances (1 out of 10) is sufficient for the look-ahead and results into best performance. This shows that duty cycle for a given module is a critical parameter and must be chosen diligently.

5. Conclusions and Future Work

While it is hard to reason about what code should not be executed in the look-ahead thread, it is straightforward to verify the impact experimentally. In this paper, we have presented a case of DIY branches and a methodology to identify and eliminate them from the skeleton without diluting the look-ahead quality. Given only two cores, performance improvement of 1.43x is a very compelling speedup for hard-to-parallelize SPEC benchmarks and a strong option for turbo boosting at relatively lower cost.

Unlike previous approaches, we do not rely on expensive, time-consuming metaheuristics to reach to a final solution, instead use simple heuristics to combine individual DIY seed modules. We also discuss the properties of a code module that help us in determining the usefulness for the look-ahead. As a future work, we plan to classify and identify DIY modules at source code level so that a helper thread compiler framework (e.g. Intel’s icc compiler) can exploit these insights to generate better code for helper threads.

**References**

