Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach*

Raj Parihar and Michael C. Huang
Dept. of Electrical & Computer Engineering
University of Rochester, Rochester, NY 14627, USA
Email: {parihar@ece., 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, but a conventional, monolithic out-of-order core quickly becomes resource-inefficient when looking beyond a small distance. A more decoupled approach with an independent, dedicated look-ahead thread on a separate thread context can be a more flexible and effective implementation, especially in a multi-core environment. While capable of generating significant performance gains, the look-ahead agent often becomes the new speed limit. Fortunately, the look-ahead thread has no hard correctness constraints and presents new opportunities for optimizations.

One such opportunity is to exploit "weak" dependences. Intuitively, not all dependences are equal. Some links in a dependence chain are weak enough that removing them in the look-ahead thread does not materially affect the quality of look-ahead but improves the speed. While there are some common patterns of weak dependences, they can not be generalized as heuristics in generating better code for the look-ahead thread. A primary reason is that removing a false weak dependence can be exceedingly costly. Nevertheless, a trial-and-error approach can reliably identify opportunities for improving the look-ahead thread and quantify the benefits. A framework based on genetic algorithm can help search for the right set of changes to the look-ahead thread. In the set of applications where the speed of look-ahead has become the new limit, this method is found to improve the overall system performance by up to 1.48x with a geometric mean of 1.14x over the baseline decoupled look-ahead system, while reducing energy consumption by 11%.

1 Introduction

Despite the ubiquity of multi-core and multi-threaded architectures, high single thread performance is still an important processor design goal. It not only provides *automatic* performance benefits and thereby improving productivity, but also is important for the overall performance even for explicitly parallel programs in speeding up serial sections and bottleneck threads.

*This work is supported by NSF CAREER award CCF-0747324 and also in part by NSFC under the grant 61028004.

Unfortunately, the two main drivers for single-thread performance – faster clocks and advancements in microarchitecture – have all but stopped in recent years. The future for single-thread performance might appear bleak. That appearance can be misleading. There is no evidence of a fundamental lack of parallelism in sequential codes. In fact, limit studies (similar to [37]) we have conducted confirm that modern codes are just like older codes and have significant potentials in implicit parallelism: As shown in Figure 1, even the integer codes have a geometric mean parallelism of about 20 instructions per cycle even if we only look ahead a few hundred instructions. The real question is whether we can realize the potential without undue costs.

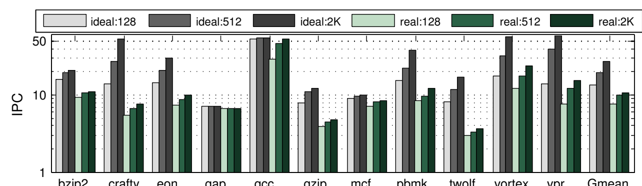

Figure 1. Implicit parallelism of integer applications. The three bars on the left indicate the amount of available parallelism (instructions per cycle) in the application when inspected with a moving window of 128, 512, or 2048 instructions. The three bars on the right repeat the same experiments only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

Conventional out-of-order microarchitecture tightly integrates look-ahead with the rest of the processing, which makes looking too far ahead too expensive to be practical [29]. One way to improve look-ahead effectiveness is to decouple the inspection activities for look-ahead purposes from the normal processing. In particular, given the proliferation of multi-core architectures, it is conceivable in a "turbo-boost" mode to launch a look-ahead thread on a different core and pass on relevant information to the original program thread [16, 30, 36, 38]. Given that the look-ahead thread is only intended to assist, it can execute a "skeleton" version (rather than a complete copy) of the program, which allows significant acceleration [16]. Even so, the look-ahead thread often becomes the new bottleneck [17] and its acceleration will directly translate to system performance gains.

A unique opportunity to speed up the look-ahead thread is that it is not bound by unyielding correctness constraints as in a regular thread. In particular, we hypothesize that

among the apparent dependences in a thread, there are *plenty* of weak links. Cutting them from the look-ahead thread allows us to speed it up without much loss of effectiveness. However, it is not a simple task to identify weak links as the effect of removing a weak link often depends on the dynamic context and whether other links are removed, much like in a Jenga game. In our exploration, we found that simple heuristics based on the static code are unlikely to be an acceptable solution as they lead to non-negligible false positives with considerable performance costs – again akin to taking out the wrong block in the Jenga game. Verifying the suspected weakness by actual measurements is probably an indispensable component of any heuristics. Given such a framework of trial-and-error, we argue that human analysis to find heuristics in code patterns becomes unnecessary. Instead, we can apply a metaheuristic approach and let the system perform the search. We will discuss our proposed design in Section 4 and present experimental analysis in Sections 5. But first, we will discuss background related work and motivation for our approach in more detail in Sections 2 and 3, respectively.

2 Background and Related Work

Architecture and system designs have long incorporated look-ahead techniques. Upon encountering a long-latency instruction, out-of-order microarchitecture does not stall the entire pipeline. The front-end continues to look into the future instruction stream to find tasks to accomplish. Software- or hardware-based prefetching achieves similar goals of looking beyond currently executed instructions. These traditional designs are either limited in look-ahead distance (out-of-order pipeline) or flexibility (prefetch based on certain types of access patterns), which led to newer forms of look-ahead.

In one group of ideas, the distance of look-ahead is extended by unblocking the various buffers. This can be done by checkpointing state [27] or moving certain entries to less powerful secondary buffers [9, 10, 13].

A significant body of work explored the concept of (micro) helper threads (or subordinate threads), where small threads are launched ahead of the targeted problematic instructions in order to alleviate the problem [2, 7, 11, 14, 23, 25, 32, 40, 41]. Compared to the aforementioned systems, they are flexible in target benefits (mispredictions or cache misses even with irregular patterns) and in the look-ahead distance. These helper threads can use regular thread contexts, but are perhaps more suited to thread contexts designed specifically for them [7].

A different flavor of helper threading uses a single, (near) complete copy of the program on a separate thread context to look ahead [5, 19, 24, 30, 31, 38]. We call this type decoupled look-ahead. The main attraction of this approach is that it sidesteps some of the subtle implementation issues of the micro thread approach. For instance, micro helper threads are often hand crafted and carefully inserted at the right locations, perhaps after lengthy manual tuning. A fully automated mechanism to make these decisions may have difficulty achieving similar effectiveness as the hand-crafted versions reported in the literature. Micro helper threads are numerous. Without substantial hardware support, spawning these threads, passing necessary initial values, and receiving results from them can create significant overheads for the main thread, which offset their benefits. In contrast, a decoupled look-ahead thread is simple to generate and relies on less extra hardware support or micromanagement from the main-thread.

Although it seems wasteful to run the program twice, in reality, the overhead is far smaller [16]. First, the look-ahead thread can be a stripped-down version of the main thread, reducing redundancy. Second, successful look-ahead prevents the main thread from wasting resource on wrong-path instructions or idling. Besides, decoupled look-ahead is intended only as an on-demand mechanism when single-thread performance is important and even then only to those applications benefiting sufficiently. An optimal look-ahead thread provides large performance benefits at small energy costs and this paper is but one more step in this direction. As the general approach matures, decoupled look-ahead should represent a powerful turbo-boost mechanism.

Finally, a metaheuristic strategy such as genetic algorithm we used is a common approach in searching an otherwise intractably large design space. It has been used to good effects in other system optimization problems [1, 3, 26].

3 Motivation

Look-ahead can be used to extract all types of information to help exploit implicit parallelism. Using it to mitigate cache misses and branch misprediction is a familiar approach and is the focus of our exploration for a number of reasons: (a) misses and mispredicts introduce large bubbles that significantly "dilute" available parallelism (see Figure 1); (b) these bubbles are particularly difficult for conventional out-of-order cores to tolerate effectively; and (c) targeting them in a decoupled system is relatively easy to accomplish. In the following, we discuss the rationale of focusing on accelerating the look-ahead thread (Sec. 3.1), the motivation to exploit weak dependences (Sec. 3.2), and the reason to use a metaheuristic approach (Sec. 3.3).

3.1 Speed Limit of Look-ahead

Intuitively, there is a fundamental tradeoff between the speed and helpfulness of the look-ahead thread. With simple transformations, a look-ahead thread can be constructed from the original program thread and successfully reduce misprediction and cache misses for the main thread. But the look-ahead thread often becomes the new speed limit. As an example, let us look at our baseline decoupled look-ahead system illustrated in Figure 2.

In this system, we use a parser to analyze the program binary and create a skeleton version as the look-ahead thread [16]. At runtime, this look-ahead thread runs on a different core and pipes outcomes of committed conditional branch instructions through a FIFO to the trailing

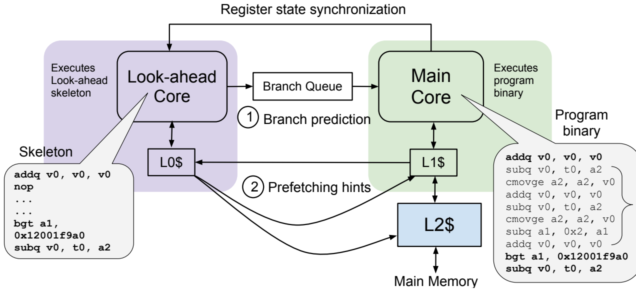

Figure 2. A generic decoupled look-ahead architecture.

main thread, which uses the outcomes as branch predictions. The look-ahead thread also naturally prefetches data into the shared L2 cache.

To understand the performance impact of this decoupled look-ahead system, in Figure 3, we show the speed of benchmarks in four configurations (the details of the experimental setup are shown in Sec. 5.1). The first configuration is the baseline where the benchmarks run on a single core. The second configuration is the decoupled look-ahead system. One performance upperbound of this second configuration is the speed of an ideal system without any mispredictions or cache misses, which is shown as connected dots above the bars. The applications are sorted from left to right with increasing gap from this upperbound.

As we can see from the figure, for 9 benchmarks (to the left of the dashed line), the potential is small (less than 5%), showing that the decoupled look-ahead architecture has successfully achieved the goal. We can also see that, after using decoupled look-ahead, these benchmarks are reaching a high sustained IPC (around 3 or more). Further increasing the speed probably requires addressing the throughput bottleneck of the pipeline.

For the remaining benchmarks, the potential can be quite large. To understand the reason why the decoupled look-ahead system falls short of the potential, we show the result of a fourth configuration, where we measure the (approximate) speed of the look-ahead thread on its own. Because of the approximation involved, this upperbound is not exact [29]. We indicate this inexactness with shaded sidebands around the simulated results.

The figure clearly suggests that for the vast majority of the remaining cases, the look-ahead thread is the new bottleneck. Indeed, among the 16 benchmarks, the decoupled look-ahead architecture can run 14 of them (with the exception of *apsi* and *bzip2*) almost exactly at the speed of running look-ahead thread alone. For the rest of the paper, we will focus on these 14 (problematic) applications.

3.2 Weak Dependences

To further speed up the look-ahead thread, a number of approaches can be taken. Speculations commonly used in helper threads [8, 39, 40] are an example. And in theory, deep transformations beyond mechanistic slicing can be performed to generate an optimal skeleton code solely for the purpose of look-ahead. In this paper, we search for solutions that do not require rewriting the skeleton code, but simply skip those instructions that contribute least to its look-ahead purposes. Intuitively, not all computations are equal. For instance, some merely add small adjustments to addresses which are inconsequential if all we wanted to do is to prefetch the right cache line. Put in a different way, for our purpose of look-ahead, the final address value *weakly* depends on such small adjustments. Dependence strength is not a consideration in a traditional analysis, which we used to generate the look-ahead thread.

Weak dependence is not a mere possibility. We have observed many examples in actual code. A simple litmus test can be used to identify a weak dependence in isolation: if we remove the instruction(s) from the look-ahead thread and the overall system runs faster, that instruction (chain) can be considered to be weakly depended upon (or a weak instruction). In Figure 4, we list a number of easy-to-explain examples from two applications (*vpr* and *mcf*).

- *Mostly silent loads (and stores)*: These instructions often load or store the same values to their target register or memory location. In example ①, the value loaded from memory location happens to match the content of the register most of the time. One particular root cause of

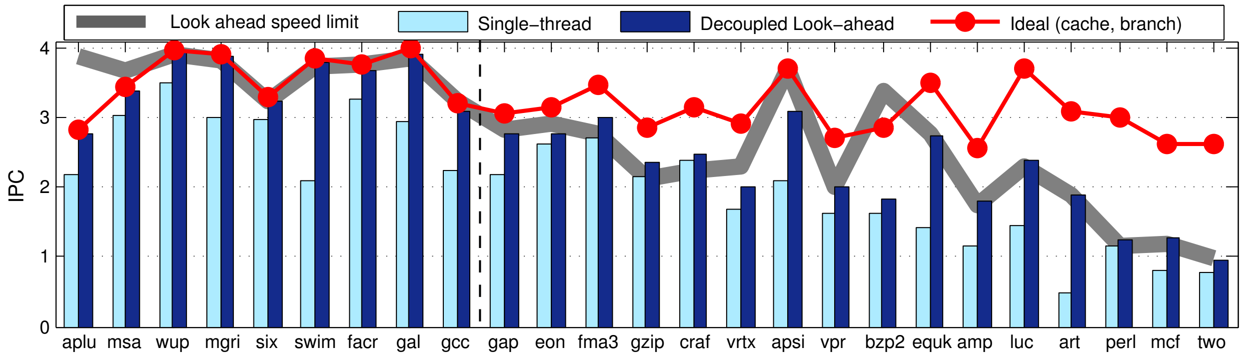

Figure 3. Performance comparison of 4 configurations. Shown in the bars are baseline single core (left) and a decoupled look-ahead system (right). Two upper-bounds are shown: the performance of a single core with idealized branch predictions and perfect cache accesses (curve with circles), and the the approximate speed limit of the look-ahead thread (gray wide curve indicating approximation). The applications are sorted with increasing performance gap between the decoupled look-ahead system and the prediction- and accesses-idealized single-core system.

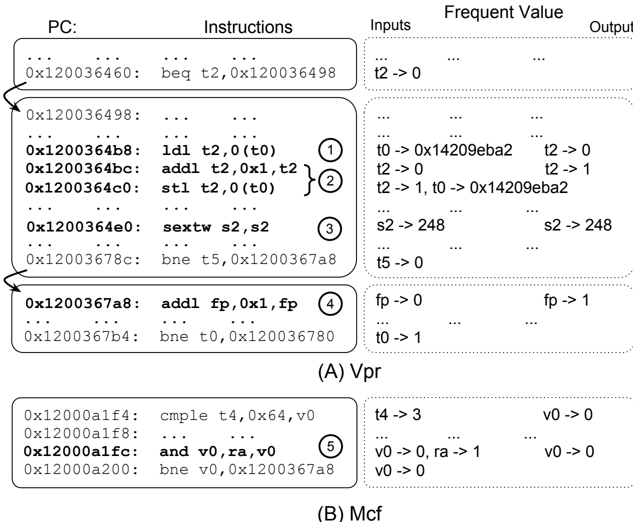

Figure 4. Example of weak dependences in the original program binary. In these examples from applications *vpr* and *mcf*, each box represents a basic block. For clarity, only a subset of instructions are shown. Frequent values of inputs and output registers – captured using a profiler – are shown on the right hand side. Shown in bold are instructions that are *weak* and can be safely removed without impacting the quality of look-ahead.

such behavior is unnecessary register spilling that led to silent loading back later.

- *Inconsequential adjustments*: In both ② and ④, the value was incremented by 1. In these two particular cases, those updates are inconsequential and removing them ended up being helpful.
- *Dynamic NOP:* Similar to the silent loads/stores, an arithmetic and logical instruction may end up not changing the result most of the time. Particular examples are sign extension as in ③ and logical AND operation in ⑤: When the value is predominantly non-negative, sign extension has no effect. Similarly, when one input is zero the other input does not matter.

Not only do weak dependences exist, they also have non-trivial impact on system performance. In fact, in one particular application (*art*), we found that removing just a single instruction from the look-ahead thread resulted in a 9.4% performance improvement of the entire system! Clearly, systematically identifying and exploiting weak dependences is useful.

3.3 Challenges of Predicting Weak Dependences

While weak dependences can be experimentally identified and we can assign reasons to explain them *after the fact* (as in the examples above), predicting them ahead of time is an entirely different matter. After comprehensive testing of individual instructions to round out all single weak instructions, we attempted to identify their uniqueness based on the static instruction. Unfortunately, they consist of all types of instructions and do not appear to be special at all (see Sec. 5.3.4). Ultimately, it is the specific computation and values involved that make their computation less consequential than others.

Admittedly, some instructions are more likely than others to be weak, but even in those cases, a single false positive can negate all gains from correct predictions. Case in point: instruction `zapnot`[1] is a common occurrence in experimentally identified weak instructions. In *gap*, a whopping 83.3% of all `zapnot` (static) instructions are weak in isolation. Removing all of them allows the program to speed up by 3.4%. Unfortunately, if we falsely identify even one `zapnot` from the remaining cases as weak, we can lose up to 5.9% performance, more than wiping out all the gain from the right selections. If we have more than one false positives, the performance loss can go up to 13%.

Indeed, the very concept of weakness is context-dependent. Fundamentally, we are performing a tradeoff: removing some instructions can shorten the dependence chain for the look-ahead thread. But the resulting approximation leads to other costs. For instance, an incorrectly prefetched cache line can hurt performance through pollution. In particular, when the approximation eventually causes the look-ahead thread to deviate from the original program's control flow, we need to "reboot" (or re-initialize) it. This is a costly operation. It is all but impossible to analytically attribute the true cost to the exact root cause.

Finally, even if it is possible to identify weak dependences in isolation, the effect of removing multiple instructions is certainly complex and non-linear. To illustrate this effect, we experimentally identify all weak instructions (in isolation) in application *perlbmk* and sort them by the number of cycles saved due to their removal. In Figure 5, we show the cumulative effect of removing these 300+ instructions one by one. Note that all 300 instructions are by themselves weak, but once we pile on about 50 of them, the overall impact becomes negative. It goes from bad to worse, slowing down the program by almost 40% at some point. To continue the Jenga analogy, the tower clearly collapsed when we removed a collection of individually safe-to-remove blocks.

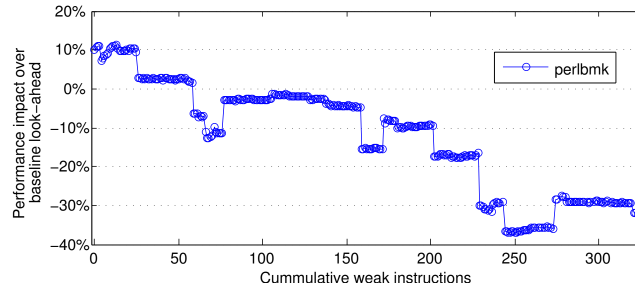

Figure 5. Performance impact of cumulatively removing instructions identified as weak in isolation.

Given the ad hoc nature of dependence strength, the non-linearity of performance effects, and the interdependence of individual chains, using heuristics and code analysis to predict weakness is at best a non-robust mechanism – in our

[1] It sets selected bytes of the source register to zero and copies the result into the destination register.

opinion. However, regardless of the root cause of weakness, the effect of removing instructions is experimentally measurable. Moreover, it appears to be reasonably stable across time and different inputs. Therefore, we can easily envision a self-tuning system that identifies weak instructions through trial and error and figures out – via metaheuristics – the right combination of these instructions to remove in order to maximize performance gain.

Genetic algorithm is a good metaheuristic method to use in such a self-tuning system. Developed in the 1970s to solve optimization problems, genetic algorithm mimics natural evolution to adapt solutions slowly but steadily towards more optimal versions [20, 21]. While genetic algorithms can be a slow mechanism to derive solutions, they have shown great potentials in finding intelligent solutions of conventional optimizations and problems involving tricky trade-offs [34]. Finally, genetic algorithm is a natural choice for our system: We use a bit mask in our design to indicate which instructions in the binary are part of the skeleton to be executed in the look-ahead thread. The way the bits govern the behavior of the skeleton is analogous to genes governing an organism, making genetic algorithm a natural choice.

4 Genetic Algorithm Based Framework

4.1 Basic Design

Our goal is to find a skeleton that maximizes performance. In our current design, the skeleton is a masked version of the program binary: some instructions are dynamically suppressed at fetch time. The task of the genetic algorithm is to find the best mask. The initial candidate solutions can be generated from any heuristics. Indeed, they can be completely random. From pairs of existing solutions, we use crossover and mutation to create pairs of children solutions as part of the next generation. To guide the evolution towards better solutions, we need to assign better (faster) solutions with a higher fitness score. Evolution can be stopped after a certain number of generations or upon seeing diminishing return.

In a most straightforward case of fitness testing, given a mask to be tested, we run the application with the corresponding skeleton as the look-ahead thread, measure the new execution time, and use the number of cycles saved (compared to the reference run using the default skeleton and the same training input) as the fitness score. Fitness tests can also be performed online while the application is running (Sec. 4.3). As often is the case with genetic algorithm-based problems, fitness tests are time-consuming. But with proper optimizations, most applications could be meaningfully tuned on-line in the first minutes of their cumulative execution, making the strategy practical.

In the following, we discuss the genetic algorithm framework (Sec. 4.2), implementation issues regarding fitness tests (Sec. 4.3), and the sampling framework that accelerates the fitness tests (Sec. 4.4).

4.2 Framework of Evolution

Genes: In our framework, a gene is the fundamental unit of modifications to the skeleton. For the most part, such a modification is masking off one (static) instruction. A "good" gene is one that improves the system's overall performance, *i.e.*, one that removes a weak dependence. Since our baseline system depends on the fact that both threads execute all branches, in this paper, we will not attempt to remove branches. Therefore, for a skeleton with n static non-branch instructions, we will have n possible genes. Out of these n genes, there is a much smaller subset of good genes that speed up the execution. To expedite the convergence to good solutions, we only use those to form chromosomes.

It is entirely possible that removing two (or more) instructions together can improve performance when removing any individual instruction alone slows down the program. The reason is that a weak dependence can be a chain of instructions, of which, any individual instruction will not appear to be weak. In general, if we remove an instruction I from the code, we should remove all instructions on I's *exclusive* backward dependence chain, *i.e.*, all instructions that are only helping instruction I and nothing else. Unfortunately, identifying the entire exclusive backward dependence chain is not easy as we need either complex hardware support or slow software analysis. Note that letting genetic algorithm stumble upon these chains is theoretically possible, but exceedingly inefficient. For simplicity, we choose to use an approximate approach and form multi-instruction genes as possible weak dependence chains to be removed. To limit the number of genes for testing, we only form genes from consecutive instructions and up to five in one gene.

Prescreening and fitness tests: To measure the impact of a gene, we activate it (masking off the corresponding instructions), measure the speed, and compare that to the original system without the modification. Again, in the simpler example of off-line profiling, the fitness score of the gene is the number of cycles saved. Note that the fitness score is only used in a probabilistic way later on. Measurement precision requirement is not high. In particular, in the first step of operation, we will filter out the bad genes from all possible genes. We do not need exact fitness score of a bad gene. In many cases, it is very clear early on that a gene is a bad one (e.g., causing too many look-ahead thread reboot). We can thus terminate the measurement early.

After prescreening single-instruction genes, we screen multi-instruction genes. A multi-instruction gene is only considered positive if its performance benefit is higher than the sum of all constituent instructions' individual benefit.

Initial chromosome pool: Given N positive genes in an application, the chromosome is represented as an N-bit vector. In our experiments, N ranges between 30 and 300. To "jump start" the evolutionary process, in addition to N single-gene chromosomes (Figure 6-a), we seed the initial pool of chromosomes with some heuristically-derived solutions. This is known as hybridization.

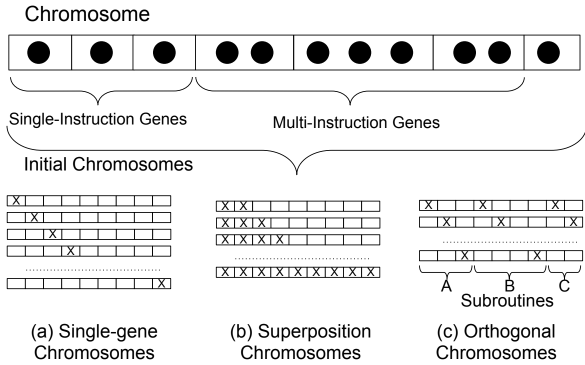

Figure 6. Look-ahead chromosome representation. Genes are basic blocks of chromosomes and can be either single-instruction or multi-instruction. Marking of a gene (shown as X) knocks out the associated instructions from the chromosome (look-ahead binary).

The first heuristic is simply to turn on many genes and create a "superposition" of genes. Not surprisingly, turning on *all* genes is almost never a good design, as too many approximations weaken the function of the look-ahead thread beyond repair. To create partial superpositions, we sort all N genes based on their fitness score and create $N-1$ different chromosomes starting from one that contains the two top-ranking genes, and going down the list, adding one more gene at a time to the previous chromosome (Figure 6-b).

Lastly, we use the heuristic that modifications to different subroutines are likely to be more orthogonal to each other than are those in the same subroutine. We thus select one gene from each subroutine to form a chromosome. Specifically, we sort the genes from each subroutine separately based on their fitness score. We pick the top-ranking gene from each subroutine to form the first chromosome; the second-ranking genes for the second chromosome, and so on. If all the genes from a subroutine have been exhausted, that subroutine will not contribute to later chromosomes. Finally when all but one subroutine have exhausted their genes, the process stops (Figure 6-c).

Population size and parent selection: For simplicity of implementation, we use a fixed population size. Based on several factors such as the number of positive genes, the cost of fitness tests, and convenience of experiments, we chose 96 as the population size. Of all the initial chromosomes formed, only 96 unique members will be selected as generation-1 population. To select these members, we use an approach that can be likened to repeatedly spinning a roulette wheel with a large number of slots to select one winner at a time. Each member occupies a number of slots proportional to its fitness.

Given one generation of population, we go through the parent selection process in order to reproduce. This is again done with the roulette wheel approach. We go through iterations, each time selecting two parents to do crossover and mutation to generate two offsprings.

Crossover and adaptive mutation: Crossover is the process of multiple (two in our case) parents swapping parts to form the same number of offsprings. *Uniform crossover* and *multi-point* crossover are attractive alternatives to one-point crossover as they tend to produce very different solutions from one generation to another [18]. We experimented with single-point, multi-point, uniform mask based, fusion operator, and xor based crossover. We chose fusion operator based on limited experiments. Fusion operator is an enhancement over uniform crossover where each bit of the vector is randomly decided to follow one of the parents based on a probability proportional to that parent's fitness score.

Crossovers allow chromosomes to exchange their genes but does not create diversity outside gene patterns present in the current generation. Mutation is thus added, which randomly flips individual genes based on certain probability. This probability increases in each generation to lower the chance that the algorithm gets stuck at local optima [4].

Survival and uniqueness test: It is possible to go through iterations of reproduction and produce more offsprings than parents. In that case, after performing fitness tests on all offsprings, a mechanism similar to the roulette wheel can be used to determine which ones survive into the next generation to keep the population size fixed. In our case, to minimize expensive fitness tests, we do not generate superfluous offsprings. We perform tests to maintain unique offsprings and all of them survive into the next generation.

We also use *elitism*, which is a type of mechanisms to shield the best solution(s) from destructive evolution. We use one flavor of elitism which remembers the currently known best chromosome. This chromosome may not be present in the current population.

Putting it together: The complete process flow is shown in Figure 7 and can be divided into three parts: baseline look-ahead thread construction, initial chromosome generation, and evolution. We start with program binary and build baseline unoptimized look-ahead binary using conventional approach [16]. We then select positive single- and multi-instruction genes, *i.e.*, weak dependences in isolation. We form some initial chromosomes from these genes based on some heuristics. Finally, we start the genetic algorithm to gradually fine-tune the solutions.

4.3 Online and Offline Fitness Tests

The tuning performed by the genetic algorithm can be done either offline or online. In the offline case, the algorithm will measure the execution time of the same application (window) many times, each time with a different skeleton.We found that a few generations of evolution is sufficient to significantly improve the performance. The control program running the genetic algorithm only needs to manipulate bit masks for skeletons to be tested, launch the program, read performance counters, and perform simple calculations and book-keepings. It thus incurs negligible overheads. The bulk of the tuning delay comes from the actual profiling runs. But these runs not be extensive and there is significant parallelism in administering the fitness tests. Overall, it is feasible to have a fully automated self-tuning process complete in min-

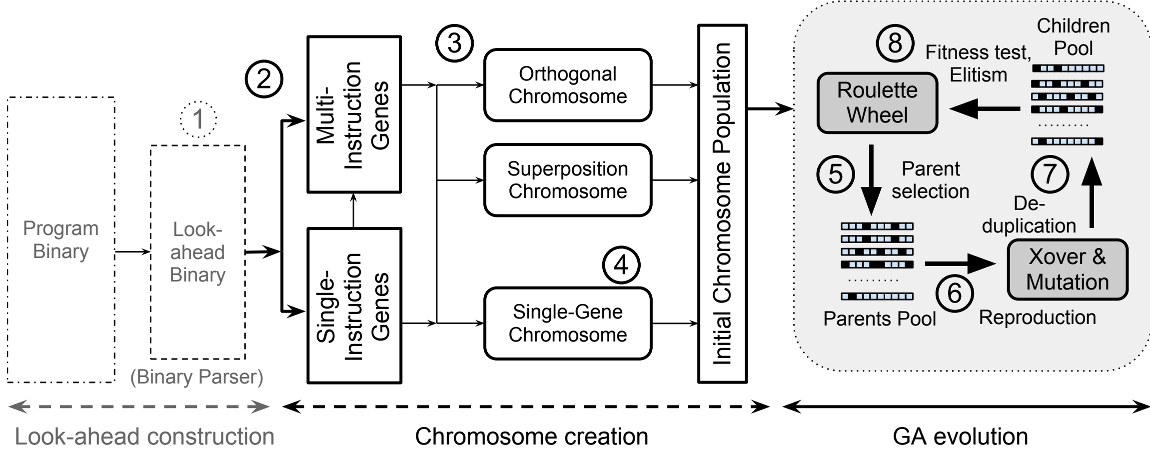

Figure 7. Genetic algorithm based framework to refine look-ahead binary by removing weak dependences.

utes. Such offline-tuning can be done at software release or install time. The cost of tuning is easily amortized over the life cycle of the particular release.

In addition to offline tuning, we can also perform the evolution online as the program runs. For online evolution, the system is similar to diagnosis embedded in software that we see today. Instead of sending certain statistics to the software vendor to improve the next release (or to mine the data for commercial purposes), we keep the statistics as the application's metadata to improve its speed. The metadata will be kept on persistent storage so that the evolution process can span over multiple runs, if needed.

During the initial stage of evolution, the program will likely run slower than without the self-tuning. This is mostly because we will inevitably try bad solutions. But this is only a short-term cost that will be more than compensated for in the later stage of evolution (Sec. 5). Another source of slowdown is the intermittent execution of the control algorithm, which reads counters, calculate probabilities, and so on. However, this overhead is largely theoretical as the magnitude is minuscule. We have measured our implementation of the control program and found that it executes about 16 million instructions for the entire evolution process that can last hundreds of billions of instructions. Overall, the overhead for online self-tuning is negligible. The real question is how fast we can reach an optimal stage. And there are mechanisms to accelerate that as we discuss below.

4.4 Sampling-Based Fitness Tests

To screen for potential genes, we will test static instructions on the look-ahead thread, which can be more than a thousand in some applications. After screening for potential genes we need to perform N_{IC}+m$\times N_P$ fitness tests, where N_{IC} is the number of initial chromosomes, m is the number of generations of evolution, and N_P is the size of population (96 in our case). All together, hundreds of fitness tests are needed in our applications. If we use naive (offline) profiling, hundreds of runs of an application are needed to reach a solution (though using smaller inputs). Significant reduction in test time can be achieved using two simplifications: sampling and multi-gene tests.

Sampling: The code module (loop or subroutine) that contains the gene are invoked countless times during one run of the application. These instances are rather self-similar [22]. Taking enough sample instances will provide a reasonably accurate estimate of the impact of the modification, especially because the sign of performance impact is more important than the exact magnitude – the magnitudes are only probabilistic heuristics. And indeed, as we will show later, sampling errors do not affect the quality of the solution.

Module-based sampling is especially useful for the online version. Unlike offline profiling where we can control what to run when performing fitness tests, in an online system we do not have the choice. Since we need to measure execution speeds with and without a modification at two different time points, we would need the system to behave the same way at these two points except for the modification. This way, the difference in measurement can be attributed to that modification alone, not random variation. If we choose the two time points to be two instances of the same code module (a form of stratified sampling), the random variation will be far lower than if we choose two time windows [22]. Of course, the impact will still be calculated from the measurement of many samples. Once we know the performance impact of a particular chromosome on every module, the program-wide impact is estimated as the weighted-average of the per-module impacts, the weight being that of the execution time for each module.

Multi-gene fitness tests: When we make a modification to some part of the code, intuitively its performance impact is localized to a certain extent. If we assume that the impact is limited to the code module that contains the modification (or gene), then we can perform tests on different genes that appear in different code modules simultaneously. All we need to do is to measure performance (say IPC) of the instances of code modules and attribute the change in performance to the tested gene within the corresponding code module.

When we use both simplifications, we will be able to perform a fitness test not for every profiling run of an applica-

tion, but for every batch of instances of a code module.

5 Experimental Analysis

Using our experimental setup (Sec. 5.1), we first show the ultimate performance benefit of weak dependence removal in Sec. 5.2. We then offer some in-depth discussion in Sec. 5.3 and compare to other look-ahead designs in Sec. 5.4.

5.1 Experimental Setup

We perform our experiments using a cycle-level, execution-driven in-house simulator.[2] We faithfully model support for a decoupled look-ahead system, including when the lead thread diverge from the actual program's control flow. The simulator also faithfully model a number of details in advanced designs such as load-hit speculation (and scheduling replay), load-store replays, keeping a store miss in the SQ while retiring it from ROB [12]. Our baseline core is a generic out-of-order microarchitecture with parameters loosely modeled after POWER5 [35]. An advanced hardware global stream prefetcher based on [15, 28] is also implemented [29]. The configuration parameters are shown in Table 1.

Baseline core	
Fetch/Decode/Issue/Commit	8 / 4 / 6 / 6
ROB	128
Functional units	INT 2+1 mul +1 div, FP 2+1 mul +1 div
Fetch Q/ Issue Q / Reg. (int,fp)	(32, 32) / (32, 32) / (80, 80)
LSQ(LQ,SQ)	64 (32,32) 2 search ports
Branch predictor	Gshare – 8K entries, 13 bit history
Br. mispred. penalty	at least 7 cycles
L1 data cache (private)	32KB, 4-way, 64B line, 2 cycles, 2 ports
L1 inst cache (private)	64KB, 2-way, 128B, 2 cycles
L2 cache (shared)	1MB, 8-way, 128B, 15 cycles
Memory access latency	200 cycles
Look-ahead core:	Baseline core with only LQ, no SQ L0 cache: 32KB, 4-way, 64B line, 2 cycles Round trip latency to L1: 6 cycles
Communication:	Branch Output Queue: 512 entries Reg copy latency (recovery): 64 cycles

Table 1. Microarchitectural configurations.

Applications and inputs: We use applications from SPEC CPU2000 benchmark suite compiled with optimization flag -O3 for Alpha using a cross-compiler built on gcc-4.2.1.[3] We use the *train* input for profiling, and run the applications for 500 million instructions, which generally cover the exercised code region in later non-profiling runs. After the offline evolution, the modified look-ahead thread will be used for performance benefit analysis. We use *ref* input and simulate 100 million instructions after skipping over the initialization portion as indicated in [33].

5.2 Overall Performance Results

Recall that out of all the benchmarks, 14 showed a bottleneck in the speed of the look-ahead thread. For these applications, we compare two systems: baseline decoupled look-ahead system and one that adds a step of offline self-tuning (with a different training input, of course) to remove weak dependences. Their speedups over a single-threaded execution are shown in Figure 8.

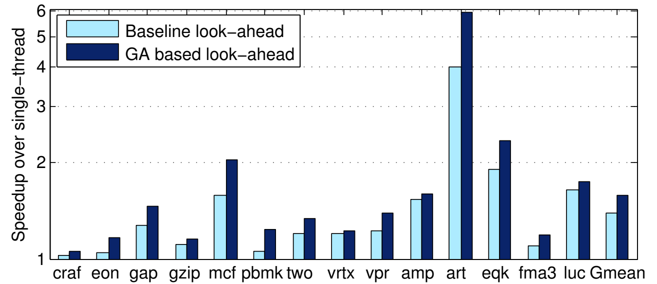

Figure 8. Speedup of baseline look-ahead and genetic algorithm-based self-tuned look-ahead over single-core baseline architecture. The vertical axis is log scale.

Offline self-tuning provided a speedup from 1.02x to 1.48x with a geometric mean of 1.14x (or a harmonic mean of 1.13x). Note that the performance advantage is obtained by executing fewer instructions in the look-ahead thread – even after accounting for additional reboots which have been faithfully modeled. Thus this performance gains comes with energy savings. Our simulations show that total energy is reduced by about 11% from the baseline decoupled look-ahead system. This is mostly due to reduced execution time and less activity (the look-ahead thread retires 9.9% fewer instructions now).

Compared to the single-threaded conventional execution, decoupled look-ahead achieves a speedup of 1.39x. With our proposal's 1.14x (multiplicative) benefit, the total speedup improves to 1.58x. Such a speedup is obtained using a fully automated system without any programmer effort, little additional hardware support, and over all applications not already saturating the pipeline. It is becoming competitive with the speedup achieved using explicit parallelization and certainly more significant than what can be achieved via moderate frequency increases.

Overall, these results show that weak dependence removal is a technique with relatively significant payoff. As another contributing piece, this technique further improves the energy efficiency of decoupled look-ahead and makes it a compelling performance boosting mechanism to add to a general-purpose multi-core architecture.

5.3 Diagnostic Analysis

As a concept, it is almost trivial that we can exploit weakness of dependence in a correctness-insensitive environment such as look-ahead. What is challenging is quantifying it precisely so that we can maximize the gain. The following analysis attempts to shed light on various aspects of the operation: how well it achieves the tradeoff between speed and accuracy of look-ahead (Sec. 5.3.1), how long it takes to find good solutions (Sec. 5.3.2), and the robustness of the solutions and the entire system (Sec. 5.3.3). Finally, we will discuss the strengths and weakness of using a metaheuristic approach relative to relying on conventional analysis and

[2] Instruction interpretation and linux system call emulation are partially borrowed from [6].

[3] Technical issues in cross-compilation have prevented us from obtaining complete results from the SPEC CPU2006 suites at the time of writing of this paper. For the partial results obtained, however, there is no material difference from the experiments presented here [29].

	crafty	eon	gap	gzip	mcf	perlbmk	twolf	vortex	vpr	ammp	art	equake	fma3d	lucas	Average
Baseline look-ahead skeleton (%dyn)	87.14	72.29	76.22	64.72	59.95	78.98	81.05	58.10	67.06	66.60	54.33	32.86	79.50	32.21	65.07
GA tuned look-ahead skeleton (%dyn)	82.66	65.83	67.79	57.35	52.61	70.22	78.25	56.31	59.01	63.22	41.11	30.06	73.50	28.53	59.03
Total program instructions (static)	57568	79730	74650	23205	18286	120529	53936	95121	42089	43154	25588	25639	249464	103235	72299
Instructions in 100m window (static)	12543	6562	4130	1424	381	9692	2456	12230	1061	958	582	1041	3098	319	4034
Individual weak instructions (static)	172	57	211	207	117	417	110	398	261	223	173	628	104	55	224
Instructions removed using GA (static)	51	15	37	56	20	30	24	37	33	24	36	40	35	12	32

Table 2. Dynamic size of baseline and genetic algorithm based self-tuned look-ahead thread relative to the original program binary and static instruction count of the original program and those removed by the framework (last row).

heuristics (Sec. 5.3.4).

5.3.1 Tradeoff between speed and accuracy

In Table 2, we show the detailed instruction counts for the look-ahead thread. For these applications, the baseline look-ahead thread is about 65% that of the main thread. After weak dependence removal, the look-ahead thread is about 90% of its original size. This relatively significant dynamic size reduction is a result of removing between 12 and 56 static instructions from the code.

The removal of these instructions will make the look-ahead thread less accurate as a predictor for future behavior of the main thread. It causes the look-ahead thread to veer off the correct program control flow more often than before and increases the number of reboot events. Table 3 summarizes the number of reboots in both the baseline decoupled look-ahead system and our new proposal. For brevity, we only show the range and average of the rates. We separate the integer and floating-point results since they are rather different. On an average, 3-4 extra reboots are encountered for every 100,000 instructions. This is still an acceptably low rate.

	INT (9 apps)			FP (5 apps)		
	Max	Avg.	Min	Max	Avg.	Min
Baseline look-ahead	79.8	**26.8**	0.5	31.7	**6.7**	0.0
Self-tuned look-ahead	112.0	**31.4**	3.9	43.5	**10.0**	0.0

Table 3. Reboot rate for baseline and the proposed self-tuned decoupled look-ahead systems. The rates are measured by the number of reboots per 100,000 committed instructions in the main thread.

Another sign of reduced accuracy of the look-ahead thread is the change in the coverage of L2 misses averted. In the baseline decoupled look-ahead, 90% of L2 misses are averted in the main thread. The rate actually improved slightly to 90.4%. This suggests that whatever imprecision in address calculation in the new look-ahead thread is more than offset by the speed increase. As a result, the L2 prefetch capability is maintained even under a more challenging environment.

5.3.2 Speed of search

Genetic algorithm convergence: To see how quickly the genetic algorithm can find a good solution, we track the chromosomes generated through seven generations. Figure 9 shows this change. To show results from different applications together, we normalize the performance gain. The solutions are represented as cycles saved and normalized to that of the best solution for that application. A few applications achieved very small performance gain (<5%), making this normalization potentially misleading. They are therefore excluded from this figure.

From the figure, we can see that the convergence is fairly

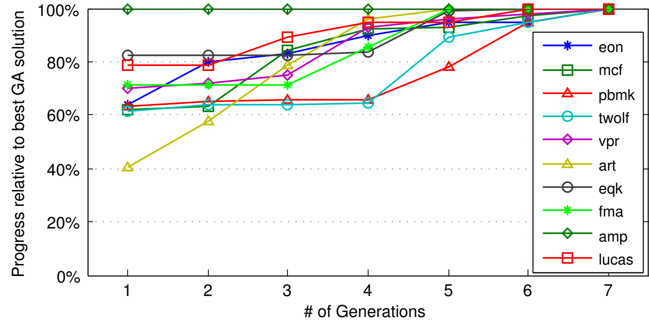

Figure 9. Normalized gains achieved through generations of evolution.

quick: after 2 generations, all applications achieved more than half of the benefit; after 5 generations, at least 90% of benefits are achieved. Nevertheless, we can not ascertain whether these optima are local. We have observed 12 generations and so far have seen no application breaking away from the best result all found within 7 generations.

Profile time: The amount of profiling time it would take in a real system for the complete genetic evolution process can be broken down into the control software overhead and the time it takes to executed certain amount of code in order to measure execution speed. Recall that the control software manipulates chromosomes through crossover and mutations, and performs bookkeeping on the fitness scores. Depending on the number of genes, the software executes between 4.8 million (*lucas*) and 43.2 million (*perlbmk*) instructions with an average of 16.8 million instructions. This overhead is on the orders of millisecond for the entire evolution stage and is clearly negligible.

For offline profiling (with sampling and multi-gene tests), the complete prescreening (for an average of 862 genes) and genetic evolution (7 generations with population size 96 per generation) process involves profiling runs between 16.9 billion (*eon*) and 61.5 billion (*mcf*) instructions with an average of 32.1 billion instructions. On the target machine, this translates to roughly 2 to 20 seconds.

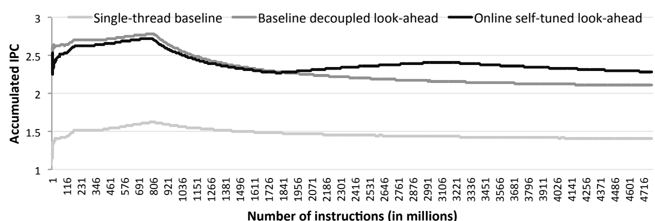

Figure 10. Comparison of single-thread, baseline look-ahead and self-tuned look-ahead using online evolution.

For an online self-tuning system, the overhead mainly comes from testing bad skeleton configurations that actually slow down the system. However, bad configurations are quickly discarded and therefore do no have a lasting impact. Figure 10 shows this effect visually with the cumulative speed (measured by IPC) of *equake* running in a single-threaded version, in the baseline decoupled look-ahead system, and in an online self-tuning system. In the very early stage for *equake*, the online version is noticeably slower than the baseline decoupled look-ahead system, but the gap narrowed afterward and within 1.8 billion instructions the online version broke even and maintained the lead thereafter. By the end of 4.6 billion instructions, its overall cumulative speed was already 10% faster than the baseline decoupled look-ahead system. Our online evolution experiment is performed with brute-force simulation and is thus excruciatingly slow. Additional examples and analysis will be included in the technical report [29].

5.3.3 Robustness

L2 cache sensitivity: Given that L2 misses are significantly reduced, it is natural to assume that the system's overall effect is highly sensitive to the L2 cache size. We carried out a sensitivity study where the genetic evolution was performed on our original platform (with 1MB L2 cache) but the testing of the final product was carried out with 2MB and 4MB L2 caches. The results are shown in the Figure 11. For brevity, we only show the geometric mean.

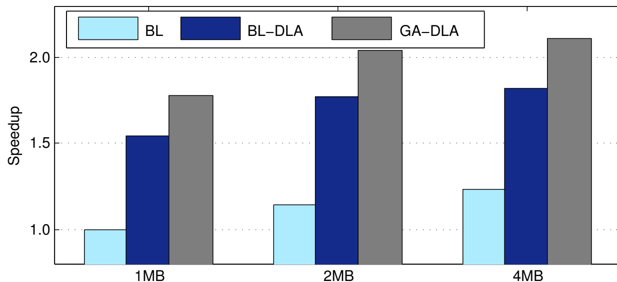

Figure 11. Speedup of various configurations: baseline (BL) baseline decoupled look-ahead (BL-DLA), and genetic algorithm tuned decoupled look-ahead (GA-DLA) for various L2 cache sizes. All results are geometric means of speedup relative to single-threaded execution on a 1MB L2 system.

From the figure, we can see that having larger L2 caches simply raises the bar on all configurations. In fact, the speedup of our proposal remains surprisingly stable: it is 1.139x, 1.133x, and 1.131x for 1, 2, and 4MB configurations respectively. In other words, the difference is less than 1% if the L2 size is quadrupled. This experiment also clearly shows that solutions evolved from a similar but not identical system are still useful. That observation suggests opportunities to further amortize tuning costs given support at the ecosystem level.

Simplified fitness test: We use two simplifications to accelerate fitness tests: sampling and multi-gene tests. In theory, these simplifications can introduce errors and potentially mislead the genetic algorithm into suboptimal solutions. To quantify the impact of these fitness test simplifications, we contrast two systems that only differ in their fitness tests: one uses complete profiling runs of 500 million instructions apiece (and costing us a tremendous amount of simulation cycles) the other with sampling based measurement and multi-gene testings discussed in Sec. 4.4. Specifically, we sample module instances at a rate of 1 per 30. For each application, we take the evolved result from each generation; measure its performance gain (cycles saved); and normalize it to that of the final (generation 7) result found in the full profiling. In Figure 12, we show a representative application.

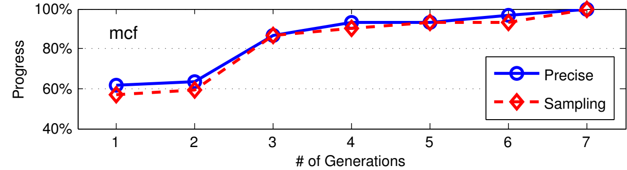

Figure 12. Comparison of generational solutions using full-run fitness tests (precise) and sampling based fitness test (sampling) for one representative application.

The result clearly shows that even though the evolution may go through different paths due to different fitness scores and thus assigned probabilities, the two evolutionary paths are essentially lock-stepped in its overall progress, suggesting that accelerated fitness tests make virtually no difference. The final best solutions are also almost identical in terms of their gene composition. Additionally, we found that the average of the population's fitness also rises in similar ways in both runs. This result also shows that even though randomization is used in the evolution process, with a sufficient population size, the outcome is not chaotic.

5.3.4 Strengths and weaknesses of metaheuristics

An alternative to using metaheuristics is a compiler analysis-based approach. Based on our experience, there are significant challenge in such an approach for our task. Although we have summarized some common patterns of weak instructions, they are hard to generalize. Predicting weakness based on the instruction itself is extremely challenging if we want to avoid false positives, which are very costly. In fact, if we represent all weak instructions as a distribution based on static code attributes (opcode and number of operands), the vector shows a correlation coefficient of 0.958 with that representing all non-weak instructions. To us, this clearly suggests that weakness is not determined by the static instruction. Moreover, weakness is not additive. Removing multiple known weak instructions can slow down the thread as has already been shown in Figure 5. Indeed, hundreds of individually weak instructions are found in almost every program. But the best solution from the evolution process removes only 12 to 56 instructions. With these instructions removed, we go back and retest the hundreds of weak instructions and have found very few to be still weak. This suggests that even if conventional analysis is used to identify potential candidates, trial-and-error is still indispensable.

Like a typical metaheuristic approach, our framework does

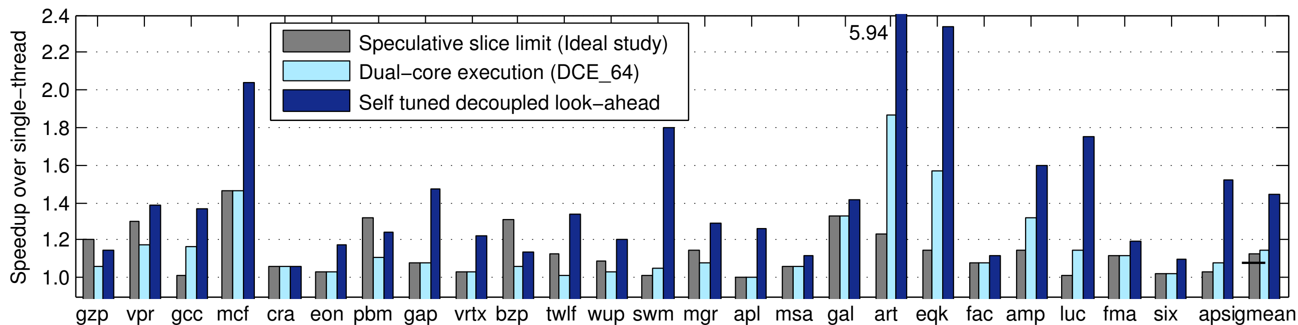

Figure 13. Comparison of state-of-art decoupled look-ahead system with previous proposals. The latency to copy register file is set to 64 cycles for DCE. Speedups are normalized to a 4-wide, out-of-order single-thread baseline with a state-of-art L2 stream prefetcher.

not care about the *meaning* of the adjustments such as removing an instruction. As such, simplistic conclusions (e.g., if A is not needed nor is its exclusive backward slice) can only be stumbled upon by chance. To see how much conventional analysis can contribute to the metaheuristic search, we perform a slicing pass on top of the evolution-derived new skeleton. This pass removes some additional instructions and results in an average of 1% performance gain. For this specific target problem, it appears that just relying on metaheuristics is enough.

5.4 Comparison with Previous Proposals

To better understand the effectiveness of our proposal, we compare it with two existing approaches, one representing a micro helper thread approach [39, 40], the other a decoupled look-ahead approach [38]. In the former case, we follow [40] and only model the upperbound of the resulting system by idealizing the top 10 problematic instructions. This is the effect when the helper threads are always on-time, accurate, and overhead-free. We show the speedup of these configurations as well as our system in Figure 13. Since a more realistic implementation of microhelper thread will only achieve part of the potential (57% according to [40]), we use a horizontal line on the bar representing the average of micro helper thread approach to provide a visual reference.

Compared to the micro helper thread approach, our design is not limited by a small number of targets. In many applications, especially newer codes, “problematic” instructions are numerous and widely spread out. Compared to dual-core execution (or other similar approaches [5,19,30]), our approach is much more effective in allowing the look-ahead thread to run faster and stay ahead of the main thread to be useful.

Recap: To summarize, the current decoupled look-ahead system achieves significant performance compared to previous proposals across diverse set of applications that range from control-intensive integer codes to large loop-based floating point applications.

6 Conclusions

Single-thread performance remains a key processor design goal despite the slowdown of traditional drivers. Look-ahead strategy, though not a silver bullet, still has significant potential. In particular, a flavor of decoupled look-ahead architecture has shown significant benefits but is often limited by the speed of the look-ahead thread. Our study has shown that there are plenty of instructions that contribute marginally to the “precision” of the outcome and thus can be eliminated in the look-ahead thread to speed it up without a significant impact on the look-ahead effectiveness.

In this paper, we have presented a metaheuristic based approach that incrementally removes weak dependences using genetic algorithm. The algorithm provides a straightforward framework to empirically explore the space of interacting opportunities. The result of this search is encouraging. Dozens of weak instructions are removed from thousands of static instructions, resulting in speedup as much as 1.48x and a geometric mean of 1.14x, while reducing the energy consumption of the system. Our study also shows that the mechanism is rather robust in that (a) the performance boost is virtually the same across different L2 sizes; and (b) the system is rather tolerant to measurement noise due to sampling and approximations in fitness tests.

With the help of the evolution framework, the speedup of a decoupled look-ahead system increases from 1.39x to 1.58x (geometric means) using two cores compared to the baseline single-threaded execution. We estimate the time for evolution to be on the orders of seconds to minutes, and during the process the extra control overhead is negligible. In our opinion, this technique further strengthens decoupled look-ahead as at least a compelling mechanism, at least for turbo boosting.

References

[1] D. Abts, N. Jerger, J. Kim, D. Gibson, and M. Lipassti. Achieving predictable performance through better memory controller placement in many-core CMPs. In *Proceedings of the International Symposium on Computer Architecture*, pages 451–461, June 2009.

[2] 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.

[3] J. Ansel, M. Pacula, Y. Wong, C. Chan, M. Olszewski, U. O’Reilly, and S. Amarasinghe. Siblingrivalry: Online Autotuning Through Local Competitions. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages

91–100, October 2012.

[4] Seung-Hee Bae and Byung-Ro Moon. *Mutation Rates in the Context of Hybrid Genetic Algorithms*. Springer Berlin Heidelberg, 2004.

[5] 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.

[6] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] Compaq Computer Corporation. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, September 2000.

[13] 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.

[14] 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.

[15] I. Ganusov and M. Burtscher. On the Importance of Optimizing the Configuration of Stream Prefetchers. In *Proceedings of the 2005 Workshop on Memory System Performance*, pages 54–61, June 2005.

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

[17] 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.

[18] S. Gilbert. Uniform Crossover in Genetic Algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 2–9, 1989.

[19] 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.

[20] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[21] J. Holland. Genetic algorithms. *Scientific American*, pages 114–116, July 1992.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] J. Mukundan and J. Martinez. MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 25–29, February 2012.

[27] 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.

[28] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–33, April 1994.

[29] R. Parihar and M. Huang. Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach. Technical report, Electrical & Computer Engineering Department, University of Rochester, February 2014.

[30] 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.

[31] 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.

[32] 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.

[33] S. Sair and M. Charney. Memory Behavior of the SPEC2000 Benchmark Suite. Technical report, IBM T. J. Watson Research Center, October 2000.

[34] D. Shasha. Future of computing: inspiration from nature. *ACM Crossroads*, 18(3):38–39, spring 2012.

[35] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, September 2005.

[36] 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.

[37] D. Wall. Limits of Instructions-Level Parallelism. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, pages 176–188, April 1991.

[38] 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.

[39] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the International Symposium on Computer Architecture*, pages 172–181, June 2000.

[40] 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.

[41] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the International Symposium on Microarchitecture*, pages 85–96, November 2002.