RPG^2: Robust Profile-Guided Runtime Prefetch Generation

Yuxuan Zhang
zyuxuan@seas.upenn.edu
University of Pennsylvania

Nathan Sobotka
nsobotka@seas.upenn.edu
University of Pennsylvania

Soyoon Park
soyoon@seas.upenn.edu
University of Pennsylvania

Saba Jamilan
sjamilan@ucsc.edu
University of California, Santa Cruz

Tanvir Ahmed Khan
tk3070@columbia.edu
Columbia University

Baris Kasikci
baris@cs.washington.edu
University of Washington & Google

Gilles A Pokam
gilles.a.pokam@intel.com
Intel

Heiner Litz
hlitz@ucsc.edu
University of California, Santa Cruz

Joseph Devietti
devietti@cis.upenn.edu
University of Pennsylvania

Abstract

Data cache prefetching is a well-established optimization to overcome the limits of the cache hierarchy and keep the processor pipeline fed with data. In principle, accurate, well-timed prefetches can sidestep the majority of cache misses and dramatically improve performance. In practice, however, it is challenging to identify which data to prefetch and when to do so. In particular, data can be easily requested too early, causing eviction of useful data from the cache, or requested too late, failing to avoid cache misses. Competition for limited off-chip memory bandwidth must also be balanced between prefetches and a program's regular "demand" accesses. Due to these challenges, prefetching can both help and hurt performance, and the outcome can depend on program structure, decisions about what to prefetch and when to do it, and, as we demonstrate in a series of experiments, program input, processor microarchitecture, and their interaction as well.

To try to meet these challenges, we have designed the RPG^2 system for online prefetch injection and tuning. RPG^2 is a pure-software system that operates on running C/C++ programs, profiling them, injecting prefetch instructions, and then tuning those prefetches to maximize performance. Across dozens of inputs, we find that RPG^2 can provide speedups of up to 2.15×, comparable to the best profile-guided prefetching compilers, but can also respond when prefetching ends up being harmful and roll back to the original code – something that static compilers cannot. RPG^2 improves prefetching robustness by preserving its performance benefits, while avoiding slowdowns.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04…$15.00
https://doi.org/10.1145/3620665.3640396

ACM Reference Format:
Yuxuan Zhang, Nathan Sobotka, Soyoon Park, Saba Jamilan, Tanvir Ahmed Khan, Baris Kasikci, Gilles A Pokam, Heiner Litz, and Joseph Devietti. 2024. RPG^2: Robust Profile-Guided Runtime Prefetch Generation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620665.3640396

1 Introduction

As modern applications scale to ever-larger datasets, the pressure on the memory hierarchy to provide data to the processor pipeline efficiently continues to grow. In particular, data center applications can spend most of their cycles on cache misses, waiting for data to be fetched from main memory [30]. Modern processor techniques such as out-of-order scheduling can frequently hide the latency of L1 misses but struggle with misses in deeper levels of the cache hierarchy.

Data prefetching is a popular strategy to improve the performance of the memory system by proactively bringing unrequested lines into the cache in anticipation (and hopefully avoidance) of future misses. The ability to predict and efficiently prefetch data depends on the workload's memory access patterns [6], which often include stride accesses (`a[i]`), indirect memory accesses (`a[b[i]]`), and random accesses (pointer-chasing). While numerous academic proposals exist [10, 13, 18–20, 22, 23, 25, 26, 29, 33, 37, 39, 41, 46, 47, 49, 53, 56, 58, 59, 65, 68] to prefetch these diverse access patterns, few have been implemented in real hardware. In particular, we have found that modern processors such as Intel Cascade Lake can prefetch stride accesses well but still struggle to prefetch indirect memory accesses efficiently.

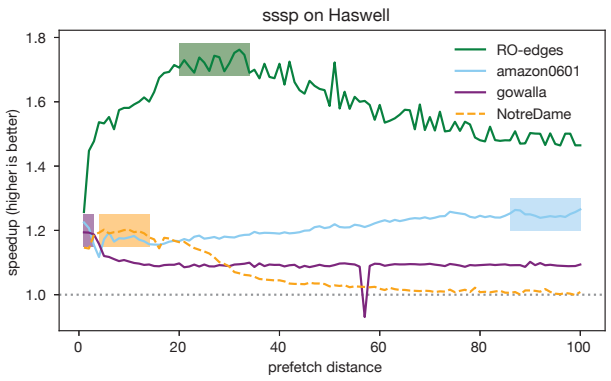

Figure 1. The sssp benchmark from CRONO [2] has very different optimal prefetch distances (shaded regions) with different inputs.

To address the challenge of prefetching complex access patterns, CPU vendors have introduced software prefetch instructions that can be utilized via compiler intrinsics such as __builtin_prefetch() in gcc and clang. As software developers are generally aware of the memory access patterns exhibited by their code, these powerful instructions can theoretically prefetch any pattern.

Unfortunately, these software prefetch instructions are also challenging to use efficiently. First, the developer needs to extract the *prefetch kernel* for computing the address that should be prefetched. Second, the prefetch instruction (and its kernel) must be inserted into the correct code location to enable *timely* prefetches. In particular, the time between the prefetch and the usage of a data item must match the time needed to load data from the main memory which represents an almost impossible-to-resolve task for developers due to out-of-order execution and the complex memory hierarchies employed by modern CPUs. Finally, the timeliness of a prefetch depends not only on the source code of an application but also on its inputs. For instance, the average vertex degree of a graph may affect the time between (indirect) memory accesses. While many automatic compiler passes exist to insert prefetching as well [3, 4, 12, 15, 21, 28, 34], they struggle with this same set of challenges.

In this work, we shed light on the scope of the data prefetching challenge via a large-scale study of program behavior across dozens of inputs, much more than have been considered in previous work. This study reveals that prefetching can be very sensitive to both microarchitecture *and* program input, making it very challenging to perform effective software prefetching via static compiler instrumentation.

For example, consider Figure 1, which shows the speedup obtainable from prefetching in the sssp benchmark from the CRONO suite [2], running on a 16-core Haswell machine. The x-axis shows the *prefetch distance*: essentially, how many loop iterations ahead we prefetch for. Each line illustrates, for a different input from the Stanford Network Analysis Platform (SNAP) [38], the speedup over a no-prefetch baseline when prefetching for 1 to 100 iterations ahead. The best performing range of prefetch distances is shaded, which reveals that all inputs show substantially different behaviors: RO-edges (the top line) performs best with a distance of 20-34, while gowalla is best with 1-2 (which would fare poorly on RO-edges). The prefetch distance, however, must be baked into the instructions that perform prefetching (typically as a displacement in x86 addressing), and any single choice for these sssp inputs may work well in some cases but can leave a lot of performance on the table in others.

In response to this finding, we propose the pure-software RPG^2 system for dynamic prefetch insertion and tuning. RPG^2 profiles a running C/C++ program, adds prefetch instructions as indicated by the profile, and tunes prefetch distance online using the program's real-time performance to guide adjustments. RPG^2 is, to our knowledge, the first system to enable *dynamic* software prefetching. RPG^2 can adapt prefetching to program inputs while the program runs, avoiding the pitfalls of ahead-of-time profiling and static compiler optimization. RPG^2 can also disable prefetching if it causes slowdown, as we see in many cases, and restore baseline performance. Like other compiler-based prefetching optimizations [3, 28, 34], RPG^2 targets programs with 1-2 small hot loops, each containing a small number of load instructions that are potentially prefetchable. Unlike prior work, RPG^2 builds on the BOLT [50, 51] binary optimization tool and thus does not need access to source code: RPG^2 operates on a program binary and the process launched from it. Prefetching is a well-known "double-edged sword" that can as easily lift as lower performance. RPG^2 provides a safer framework for data cache prefetching that makes that sword much easier to wield.

2 Background

In this section, we describe the main elements of effective memory prefetching, some of the performance challenges that prefetching can present, and current compiler techniques for automatic prefetching.

2.1 Prefetching Basics

A data prefetch is used to proactively request data that are used by a subsequent *demand* memory request; we use the "demand" qualifier to distinguish requests that are part of the semantics of the program from prefetch requests that are logically NOPs. We typically care only about demand loads, as the latency of store misses can be hidden in most cases via out-of-order execution and a hardware store buffer.

The success of a prefetch is determined by three qualities: its accuracy, coverage, and timeliness. **Accuracy** means that there exists a future demand load to the same address as the prefetch; prefetches (since they are NOPs) may also be

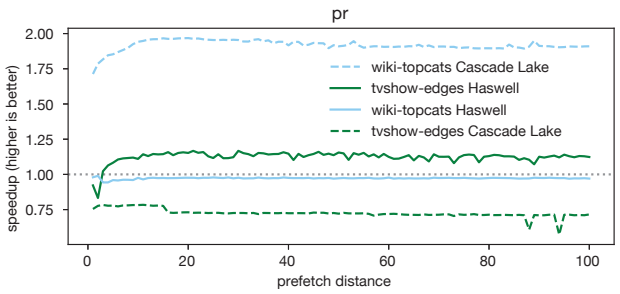

Figure 2. The pr benchmark from CRONO [2] sees a speedup or a slowdown with prefetching, depending on the microarchitecture.

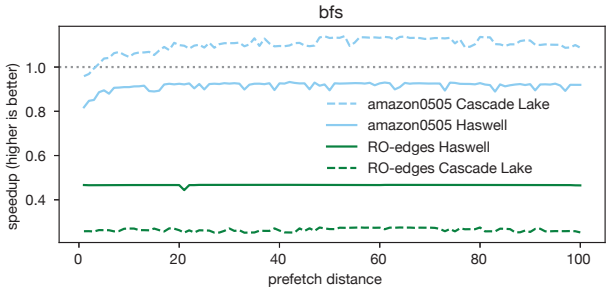

Figure 3. The bfs benchmark from CRONO [2] often (but not always) suffers significant performance slowdown with prefetching.

issued speculatively for addresses that may or may not match a future demand load. Any prefetch for an address that is never used later wastes memory bandwidth and pollutes the on-chip caches. **Coverage** refers to the fraction of cache misses that are prefetched – higher coverage leads to more misses being transformed into cache hits. Finally, **timeliness** refers to the requirement that a prefetch occur far enough in advance of the demand load to bring the load's data into the L1 cache from wherever it currently resides (which may be DRAM, hundreds of cycles away). Furthermore, it also must not occur too far in advance of the demand load, as in that case, it may be evicted from the cache before the demand load occurs. As a result, prefetches are most effective within a certain window.

Figure 1 shows these timeliness windows visually in terms of the **prefetch distance**, a measure of how far ahead in the execution we are prefetching. The typical unit of time for prefetch distance is a loop iteration. In Figure 1, the timeliness windows are the width of each shaded region, which show the best-performing ranges of prefetch distances. Sometimes these windows are quite narrow, as for the gowalla input, where prefetching too far ahead quickly triggers interference. Timeliness can be tuned by varying the number of instructions between the execution of the prefetch and demand load. However, as modern CPUs support variable clock frequencies, execute at different instructions per cycle (IPC), and deploy out-of-order execution, no analytical approach to determine the optimal prefetch injection site or prefetch distance exists.

Furthermore, the three prefetch properties are hard to improve simultaneously: higher coverage often leads to lower accuracy, and improving timeliness may require sacrificing coverage or accuracy. This interplay makes it challenging to diagnose performance problems with prefetching. Modern hardware performance monitoring mechanisms offer few insights, making it hard to know whether prefetching is always bad for a given program/input, or merely misconfigured.

2.2 Performance Pitfalls

In addition to the challenges of input-dependent prefetch distances, which we showed in Figure 1, our study with SNAP [38] inputs has also revealed that prefetching's performance can be microarchitecture-dependent. Figure 2 shows results for two different inputs with the pr benchmark from CRONO [2], where on an Intel Cascade Lake machine (dashed lines), the wiki-topcats input (light blue lines) experiences a nearly 2x speedup with prefetching, across a wide range of prefetch distances (x-axis). The tvshow-edges input (dark green lines), however, experiences a mild slowdown with prefetching. The story is reversed on an Intel Haswell machine (solid lines) where tvshow-edges sees a moderate speedup with prefetching and wiki-topcats sees a moderate slowdown with prefetching. Unless the program is re-profiled and recompiled for each machine, some performance will be left on the table.

We also found a significant number of situations where prefetching hurts performance. Figure 3 shows for the bfs benchmark from CRONO not only the machine-dependent behavior of prefetching (where the amazon0505 input in light blue benefits from prefetching on Cascade Lake but not on Haswell) but also significant machine-dependent slowdowns of 50-70% for the RO-edges input (dark green lines).

2.3 Current Approaches to Software Prefetching

There exists a large body of work on software prefetching, including automatic compiler approaches that analyze only static code [4, 12, 15, 21] or additionally leverage profiling information [3, 28, 34] to insert prefetch instructions without programmer intervention. All of these schemes need to balance accuracy, coverage, and timeliness by adjusting the placement of prefetch instructions and the chosen prefetch distance. Ultimately, these schemes produce a single binary that contains one fixed set of prefetch instructions. While it is possible to re-profile and produce additional binaries tuned for different inputs or microarchitectures, there is a significant operational burden for doing so. Profiling can be very time-consuming: the recent *APT-GET* [28] system

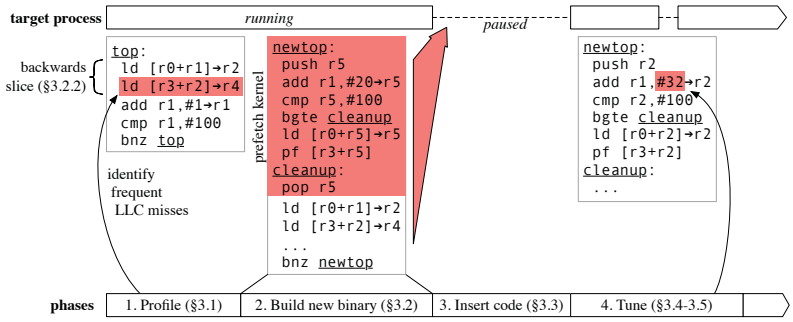

Figure 4. RPG^2 proceeds through four phases when optimizing a target process.

for profile-guided static recompilation-based prefetch insertion can spend several minutes profiling and compiling a benchmark, though admittedly no attempt has been made to optimize this offline processing. Even once a set of binaries has been produced, they must be stored and organized for future retrieval, and something must decide which binary to use at program launch time. Given the variability we have seen across inputs and machines, it seems very challenging to predict the gain or loss from prefetching *a priori*.

Ultimately, prefetching is an unreliable optimization with benefits that are benchmark-, input- and microarchitecture-dependent. Its fragility makes developers rightfully wary, as prefetching can cause significant harm by evicting useful data from the cache and stealing memory bandwidth from demand loads. While additional research into better static prefetching schemes can surely help, in this paper, we instead embrace prefetching's mercurial nature and build the RPG^2 runtime framework for adding, tuning, and removing prefetch instructions while a program is running. We describe how RPG^2 works next.

3 Design of RPG2

We first give an overview of RPG^2 by describing at a high level how it optimizes the example code in Figure 4 by inserting prefetch instructions. Subsequent sections then examine in detail each of RPG^2's main phases.

RPG^2's first phase starts by profiling a running process to discover where last-level cache (LLC) misses are occurring and also to establish the baseline IPC for the running application. RPG^2 uses Intel's Precise Event-Based Sampling (PEBS) hardware performance monitoring feature for this task, as it can track the instruction PCs that trigger LLC misses. In Figure 4, an example program in pseudo-assembly has a load into `r4` that is identified as a frequent source of LLC misses.

RPG^2's second phase builds on Meta's Binary Optimization and Layout Tool (BOLT) [50], which lifts an executable into a low-level IR format, performs optimizations, and then produces a new BOLTed executable. BOLT ships with a variety of code layout and peephole optimizations. Thanks to BOLT, RPG^2 does not need access to the application source code and only requires the program binary that launched the target process we wish to optimize. Our new BOLT pass investigates the example code and sees a loop it can optimize, with `r1` as the loop induction variable. The LLC-miss-causing load is an indirect load, whose data address depends on the value returned in `r2` by the previous load – indirect loads are RPG^2's main optimization target, though it can target direct loads as well as we discuss in § 3.2. As a result, RPG^2's inserted prefetching code, which we call a *prefetch kernel* and is shaded in Figure 4, must replicate this indirect structure. Moreover, this prefetch kernel must also ultimately act as a `NOP`, leaving the semantics of the original code unchanged.

Figure 4 shows the resulting prefetch kernel for our example program. While we discuss this kernel in much more detail in § 3.2.3, we provide a brief overview here: first, we compute the address we want to access, which is 20 iterations ahead. Because this may be beyond the bounds of the original loop, we must insert a bounds check to guard the demand load that we insert – otherwise, a program crash could arise from accessing unmapped virtual memory. Finally, we add the prefetch instruction for the indirect load and run the original loop body.

Having produced a new binary with prefetch code added, RPG^2 now enters its third phase: injecting the prefetch code into the running process and transferring control to it. RPG^2 adds code at function granularity, writing a new version of the optimized function into the address space. This requires a short pause of the target process to add the optimized function's code and to patch any references to the old function that exist in program counters, call sites, or stack return addresses to refer to the new function instead.

The target process now resumes, and RPG^2 enters its fourth phase: tuning the prefetch distance. RPG^2 performs a bounded search, from a randomized starting point, over a set of possible distances and examines how IPC responds. Each adjustment of the prefetch distance requires a brief pause of the target process to edit the few bytes representing the prefetch distance as an offset in the machine code. For example, in Figure 4, RPG^2 changes the initial prefetch distance of 20 to 32 by adjusting the immediate value of the

add instruction. After the search completes, RPG^2 re-enables the best-performing prefetch distance. If the search does not discover any prefetch distance that outperforms the original IPC, RPG^2 steers execution back to the original code instead. As Figure 4 shows, most of RPG^2's work happens concurrently with the execution of the target process to minimize performance interference.

In the remainder of this section, we describe in detail each major phase of RPG^2's operation, starting with profiling.

3.1 Phase 1: Profiling

RPG^2 employs Linux's perf utility to profile the execution of a running process. To identify candidate loads for prefetching, RPG^2 uses perf's Processor Event-Based Sampling (PEBS) event MEM_LOAD_RETIRED.L3_MISS/ppp to identify LLC misses. PEBS samples LLC misses at a specified rate and stores a PEBS record in memory for each sampled miss. This PEBS record contains information about the miss including its data address and PC. RPG^2 filters PEBS records by only considering as prefetch candidates the instructions that cause at least 10% of the misses within their respective function. For an online scheme like RPG^2, there exists a trade-off between collecting additional profiling data to make better optimization decisions and implementing optimizations quickly to accelerate as much of the program execution as possible. By default, RPG^2 samples 2 seconds of execution; we examine the impact of this sampling period in § 4.

3.2 Phase 2: Code Analysis & Generation

RPG^2 adds a new InjectPrefetchPass pass to BOLT to add prefetch kernels to a given binary. The binary code analysis and transformation in InjectPrefetchPass begins by identifying different code patterns that are amenable to prefetching.

3.2.1 Prefetch Categories. RPG^2 can prefetch both indirect memory access and direct memory access. The prefetchable memory accesses are grouped into three categories shown in Table 1 where `a[]` and `b[]` are two arrays, `i` is the induction variable for the outer loop, `j` the induction variable for the inner loop, `d` the prefetch distance, and $f()$ represents the data dependency chain from the load of `b[]` to the demand load that needs to be prefetched.

demand access	prefetch	description
`a[j]`	`a[j+d]`	direct access using inner loop induction var
`a[`f`(b[j])]`	`a[`f`(b[j+d])]`	indirect access using inner loop induction var
`a[`f`(b[i])+j]`	`a[`f`(b[i+d])+j]`	indirect access using inner and outer loop induction vars

Table 1. Memory access categories that RPG^2 supports

RPG^2 currently only supports loads that fall into these three categories, but we have found these patterns general enough to match many code patterns, like dense arrays and stencils (category 1) and sparse arrays (categories 2 and 3).

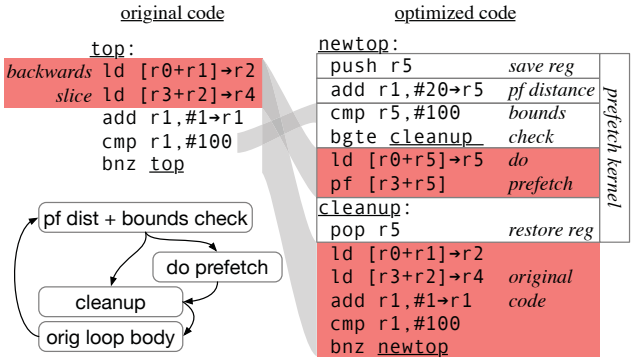

Figure 5. Annotated example of RPG^2 code transformations.

Indirect loads, represent a frequent source of LLC misses that cannot be covered efficiently by hardware prefetchers. We leave prefetching of additional memory access patterns (*e.g.*, pure pointer-based data structures without arrays) for future work.

RPG^2's prefetch strategy for the first two load types in Table 1 is straightforward. At every iteration `i`, RPG^2 prefetches the data to be used for iteration `i+d` where `d` is the prefetch distance (measured in loop iterations). For `a[j]` (direct) accesses, line-level spatial locality and hardware prefetchers typically work well, though if RPG^2 sees significant LLC misses (likely due to a sub-optimal choice of the prefetch distance by the hardware prefetcher) then it can improve performance further even for this simple access pattern.

For indirect memory accesses in the `a[`f`(b[i])+j]` category, RPG^2's strategy is to prefetch the data feeding a future iteration's indirect load `a[`f`(b[i+d])]` instead of prefetching a future iteration of the inner loop with `a[`f`(b[i])+j+d]`. While we experimented with both, we found that the former approach performs better since it attacks a more difficult access pattern.

In order to identify which prefetch category a load falls into, RPG^2 starts by analyzing the two innermost loops within each loop nest and identifying their loop induction variables. RPG^2 places its prefetch kernel in the loop header, which runs at the beginning of each loop iteration. A key decision is whether to place the kernel in the header of the inner loop or (if there is one) the outer. The decision depends on what data are used by the prefetch operation, which we discuss next.

3.2.2 Backwards Slicing. To prefetch for indirect memory accesses, RPG^2 must check whether the memory address is prefetchable and, if so, compute the address for prefetching. Our detection algorithm computes the backward slice [62] starting at the demand load that causes the most LLC misses in the function. The slice extends until it reaches an instruction whose source registers are either loop invariant or loop induction variables. Figure 5 zooms in on the example code

from Figure 4: `ld [r3+r2]→r4` is the demand load where we begin computing a backward slice, which encompasses only `ld [r0+r1]→r2` since `r0` and `r3` are loop-invariant and `r1` is the loop induction variable.

RPG^2 analyzes the backwards slice to see which category (Table 1) it matches, using the presence of indirect loads and loop induction variables as key indicators. If a slice does not match one of the supported categories, RPG^2 cannot currently optimize it. For loads like `a[`f`(b[i])+j]`, the prefetch kernel will be inserted in the outer loop since, as was mentioned in § 3.2.1, this performs better. Otherwise, the prefetch kernel is added to the inner loop.

RPG^2's slice computation can traverse dependencies via stack memory at fixed offsets from the stack pointer where there is no intervening stack pointer manipulation, however, dependencies via non-stack memory or that involve conditional control flow (such as multiple reaching definitions) are currently unsupported. These cases did not arise in our evaluation benchmarks. Some of these cases (such as conditional dependencies) could be supported with additional engineering effort; BOLT already provides dominator and reaching definition analyses as a starting point.

3.2.3 Code Generation. Once we have generated the backward slice, we can create the code for prefetching, referred to as a *prefetch kernel*. Our correctness criterion is that the prefetch kernel behaves like a `NOP`. Prefetch instructions themselves are natural `NOP`s, however, the kernel also contains supporting code needed to enable the prefetch, *e.g.*, instructions in the backward slice must be run as real instructions to compute the correct prefetch address.

As our example code in Figure 5 falls into the `a[`f`(b[j])]` category, our prefetch kernel requires a scratch register to assist with the address computation, which we obtain by spilling a register (here r5) to the stack. The kernel then uses the prefetch distance to compute the element `b[j]`, which is being prefetched, and additionally needs to perform a bounds check to ensure that `b[j]` refers to a mapped, accessible address. We copy the bounds check condition from the original loop latch, inverting the condition (from bnz to bgte) since we are checking whether to *skip* the prefetch and adjust for the prefetch distance. The demand load of `a[]` that we wish to prefetch for is converted to a prefetch instruction. If the bounds check succeeds, the backward slice (which includes the prefetch) is executed. Afterwards, we run cleanup code to restore the scratch register we commandeered and then run the original loop body.

These code transformations are implemented in the InjectPrefetchPass BOLT pass, which produces a new BOLTed binary containing the new code with the prefetch kernel.

3.3 Phase 3: Runtime Code Insertion

With our optimized code in hand, we must insert this new code into the target process's address space. This requires overcoming several challenges. Consider that the unoptimized original loop resides within a function f_0. If we overwrite f_0 with the optimized function directly in memory, we must consider that PCs are shifted due to the insertion of the prefetch kernel, and hence branch targets must be adjusted. Furthermore, if the optimized function f_1, containing the prefetch kernel, is larger, it may no longer fit within the space allocated to f_0. To sidestep these issues, RPG^2 instead places f_1 at a new location in the address space, leaving f_0 intact in its original spot. This adds the benefit of automatically preserving all code pointers to f_0, no matter how exotic they are, including function pointers, setjmp buffers, etc. Also, in the event that prefetching causes a performance regression, which sometimes happens, we can steer the execution back to f_0 to restore the original performance.

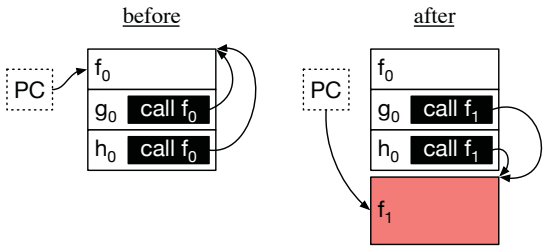

Figure 6. The PC register and code memory of the target process before and after, RPG^2's code replacement.

Inspired by Google's XRay [9] function call tracing system, we also initially considered ahead-of-time compiler support to add regions of NOPs to code where prefetches were likely to prove useful, allowing these NOPs to be quickly overwritten with a prefetch kernel at runtime. However, these NOPs inside hot loops had a noticeable runtime cost of up to 5% in some cases, causing us to prefer our current pay-as-you-go approach, which has zero ongoing runtime costs when prefetching is disabled.

RPG^2 leverages Linux's ptrace API to perform the actual code insertion. ptrace allows RPG^2 to pause or resume the target process and update its register and memory contents. While ptrace is powerful, it is also somewhat slow, so to accelerate code insertion, we have also developed a library libpg2 that is loaded into the address space of the target process when it launches via Linux's LD_PRELOAD mechanism. Because libpg2 runs inside the address space of the target process, it can edit target process memory directly with low overhead; in contrast, ptrace would require a series of system calls to accomplish the same. Nevertheless, ptrace is still required for some operations, such as pausing and resuming process execution and changing register values.

libpg2 does nothing until code injection is triggered via ptrace. To begin code injection, RPG^2 uses ptrace to pause the target process, moves the PC to the relevant function within libpg2, bumps the stack pointer to avoid clobbering

the red zone of the target process, and then resumes the process to let libpg2 run. mmap is then used to allocate new memory and copy the f_1 code into it.

3.3.1 On-Stack Replacement. Our task now becomes to redirect execution to f_1. Figure 6 illustrates how this works. Direct calls to f_0 from other functions are patched to refer to f_1 instead. Much more challenging is translating thread PCs which refer to f_0: these are cases where execution is in the middle of an invocation of f_0. While it is tempting to simply wait until a function call boundary – optimizing the *next* invocation of f_0 instead of trying to optimize the currently-running one – we have found that such waiting is a non-starter for our workloads since they often spend most of their execution within a single function invocation which contains a hot loop that must be optimized. Waiting would thus be a huge missed optimization opportunity.

The problem of moving between different versions of a function at runtime is known in the managed languages literature as *on-stack replacement* [24] (OSR), and many sophisticated language VMs use it to switch to a more optimized version of a running function, or to deoptimize a function when the memory layout of a class changes. While OSR is commonplace in managed languages, RPG^2 is the only system we are aware of to support it for unmanaged languages like C/C++, as mapping both code and data across function versions is complex, and there is essentially no existing compiler support. Even the recent Ocolos [69] system, which provides online code layout optimization for unmanaged languages, does not support OSR.

RPG^2 is able to support OSR for two reasons. First, RPG^2 prefetch kernels are designed with OSR in mind. As they have a logical NOP structure, it is relatively easy to slot them into an existing function without changing semantics. In initial versions of RPG^2 we tried to leverage prior prefetching compilers like *APT-GET* [28] that operate at the compiler IR level. However, we discovered that adding a prefetch kernel at the IR level triggers myriad small changes in the resulting machine code. For example, a program variable v would be allocated to different registers in f_0 versus f_1 (or stack-allocated in one and register-allocated in the other); moving execution to f_1 then requires understanding how variables are mapped to data, which is not transparent in existing unmanaged compilers. Inspired by these challenges, RPG^2's code transformations instead incur zero data layout changes.

The second reason RPG^2 can support OSR is that, even though RPG^2 necessarily makes code layout changes, BOLT provides a handy BOLT Address Translation Table (BATT) that maps PCs between an original function and the version modified by BOLT passes. The BATT is embedded in an ELF section in binaries produced by BOLT. BOLT uses the BATT to support re-optimizing a binary that it has optimized before; RPG^2 uses the BATT to map PCs from f_0 to f_1 during OSR. Once code insertion completes, libpg2 raises a SIGSTOP signal to send a notification via ptrace. Upon receiving the SIGSTOP, RPG^2 moves the stack pointer back to the original value and updates thread PCs to point to their corresponding instructions in the f_1 code. Then RPG^2 resumes the target process, which from now on executes f_1 instead of f_0 code.

3.4 Phase 4: Monitoring And Tuning

After the prefetch kernel has been inserted, RPG^2 determines the optimal prefetch distance via binary search. In particular, RPG^2 changes the program code to implement a new prefetch distance and then monitors the resulting performance impact. Prefetch distance adjustments require rewriting just a few bytes of program code (and accompanying system calls to enable code edits and disable them again afterward) and are performed via libpg2. RPG^2 monitors performance by measuring IPC via perf stat.

The prefetch distance search algorithm has three stages. In the first stage, RPG^2 decides the direction for searching. Starting from a random number r drawn from the interval $[1, 100]$ as the initial prefetch distance, RPG^2 takes three measurements of the IPC of $r-5$, r, and $r+5$ to determine a gradient that identifies the direction towards higher IPC. We empirically determined that most optimal prefetch distances are smaller than 100, so we chose it as an upper bound for the initial prefetch distance. In the second stage, RPG^2 samples coarsely in the identified direction to find a region of promising prefetch distances.

In stage 2, RPG^2 keeps on doubling the jump size to compute the new prefetch distance in the chosen direction, so long as RPG^2 sees increasing IPC. Prefetch distances are capped to be within [1,200]; if the algorithm attempts to step outside this range, the search terminates, and the best prefetch distance is chosen from among the measurements taken so far. Otherwise, once RPG^2 finds an IPC decrease, it sets the n^{th} prefetch distance and the $n-1^{\text{th}}$ prefetch distance as the upper and lower bounds of the interval for the third stage, which is binary search within this interval to identify a local optimum.

If a program has multiple prefetch locations, the prefetch distance can, in principle, be tuned separately for each location. As we show in Figure 13, there is sometimes a benefit to such asymmetric prefetch distances, though our search algorithm scales exponentially in the number of prefetch locations. For efficiency, RPG^2 currently restricts all prefetch locations to have the same distance.

After RPG^2 determines the best prefetch distance d, it pauses the target process one final time to install d and then detaches from the process to run without ongoing overheads except for those of the prefetch kernel itself.

3.4.1 Rolling Back Prefetches. In some cases, inserting prefetches can harm program performance, irrespective of the prefetch distance. In such cases, after the prefetch distance search completes, RPG^2 rolls back to the original f_0

code, which remains in the address space (Figure 6). When RPG^2 decides to roll back, it pauses the target process to undo its previous changes: reverting changes to direct call sites and program counters. The BATT is crucial for translating f_1 locations into their corresponding f_0 locations. However, there is one additional corner case to consider. If a thread is currently inside the prefetch kernel, there will be no BATT entry because there is no corresponding f_0 location. So, RPG^2 instead single-steps the target process via ptrace until the PC hits an address that is stored in the BATT, which can then be translated into an f_0 location.

4 Evaluation

Our evaluation has five main parts. After explaining our experimental setup, we show RPG^2's performance compared to a range of baselines. Next, we examine how accurate RPG^2's prefetch distance search is, how much profiling data it needs to work well, and how long key RPG^2 operations take. Then, we measure RPG^2's behavior at the microarchitectural level by measuring the impact on LLC MPKI (misses per kilo-instruction) and instruction count. Finally, we expand on the data in Figures 1-3 to demonstrate additional facets to the prefetching challenge.

4.1 Experimental Setup

We run our experiments on an Intel Xeon Gold 6230R Cascade Lake and an Intel Xeon(R) CPU E5-2618L v3 Haswell server. The Cascade Lake server has two sockets with 26 cores and 52 threads per socket, all running at 2.1GHz. Each core has a 32 KiB L1i, a 32 KiB L1d, a 1 MiB L2, and access to a shared 36 MiB L3 and 384GiB of RAM. The Haswell server has 2 sockets with 8 cores and 16 threads per socket, all running at 2.3GHz. Each core has a 32 KiB L1i, a 32 KiB L1d, a 256 KiB L2, and access to a shared 20 MiB L3 cache and 128GiB of RAM. All available hardware prefetchers are enabled on both machines. Both machines run Linux version 5.40. Our BOLT is built based on commit `56ff67ccd907` from BOLT's GitHub repository [1]. We use Clang version 10.0 to compile all workloads. We use BAT-dump from LLVM's GitHub repository on commit `2b88298c2ab2`.

We use the CRONO [2] benchmark suite's BFS, PR, BC, and SSSP benchmarks. For inputs, we use both real-world graph data sets from the Stanford Network Analysis Platform (SNAP) [38] and synthetic inputs from *APT-GET* [28]. We run the IS, CG, and randAccess benchmarks and inputs from Ainsworth and Jones [3], and call these the "AJ" benchmarks. In this work, we focus on graph workloads as they frequently exhibit indirect memory access patterns that RPG^2 optimizes.

All benchmarks are compiled with their default optimization level -O3 and with the linker flag -Wl,–emit-relocs, which enables BOLT's function relocation. We extend the runtime of the measured workloads by adding iterations so that they last at least 1 minute. To avoid profiling the initialization phase of a workload, which can be very different from the main application phase, we modify each benchmark to signal the end of its initialization phase. Future work could leverage program phase detection techniques [16, 57] to do this automatically. We run each benchmark+input combination until we find 5 successful results. If none of the first 5 runs can activate RPG^2, we just record the execution reported by the original binary. RPG^2 collects LLC misses via PEBS for 2 seconds at the maximum supported sampling frequency of 25,750 samples/sec on Haswell and 12,500 on Cascade Lake. During the tuning phase, RPG^2 measures IPC for 0.3 seconds.

4.1.1 Baselines.

We accurately gauge RPG^2's performance, we compare it to the following alternative schemes.

- ***APT-GET***, the latest profile-guided static compiler for automatic prefetch injection [28]. *APT-GET* profiles a randomly chosen input and the resulting binary is run on the remaining inputs. This configuration captures a real-world use case where it is not feasible to build a new binary for each individual input. *APT-GET* data is missing for sssp, bfs, and randacc as *APT-GET* would not reliably generate prefetch instructions for these benchmarks after we extended their running time, and our consultation with the authors was not able to resolve these issues.
- **manual** prefetching by the benchmark developers. This configuration is only available for the AJ benchmarks.
- the **active-only** subset of RPG^2 runs when it gathers enough profiling data to enable optimization (which does not always occur, as we explore later in § 4.3). The main RPG^2 bars, in contrast, also include runs where RPG^2 starts but does not receive enough profiling information to justify enabling prefetching.
- an **offline** version of RPG^2 where a binary is produced for each input using the manually chosen best prefetch distance. This scheme represents in some ways an upper bound on RPG^2's performance, as the best prefetch distance is always chosen, sidestepping any shortcomings in the prefetch distance search algorithm, and the optimized code is running the entire time, avoiding the delay of RPG^2's online profiling, code generation, code injection, and prefetch tuning. However, prefetching is always enabled in this configuration, so for benchmarks where prefetching is harmful, this offline scheme may experience a net slowdown.

4.2 Performance

Figure 7 shows how RPG^2 performs on our Cascade Lake and Haswell machines. All results are normalized to the original, non-prefetch code. For the CRONO benchmarks (left graphs), pr, bfs and sssp's results are averaged across 71 inputs from SNAP [38] on Cascade Lake and 67 inputs

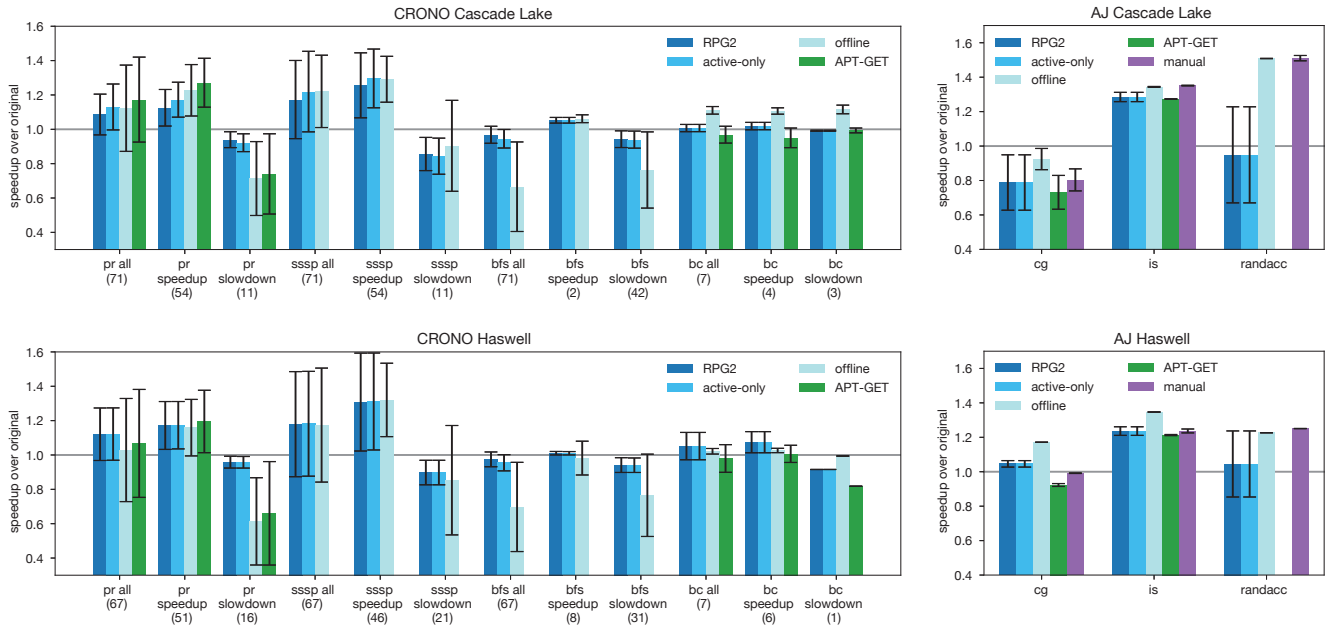

Figure 7. RPG^2 and baseline performance on Cascade Lake (top) and Haswell (bottom).

on Haswell, as some of the largest inputs exceeded our 10-minute timeout on the older machine. The bc benchmark does not support the graph formats used in SNAP and only runs on a smaller number of synthetic graphs drawn from the *APT-GET* [28] evaluation. The graphs on the right side show the AJ benchmarks, where we use the single inputs used in [3]. Error bars show the standard deviation, which is often large for the CRONO benchmarks where we *aggregate* data from many distinct inputs due to space reasons. The runtime variance on each particular input, however, is low.

For the CRONO benchmarks, given the large number of inputs, we show three groups of bars. The first group (*all*) shows speedup averaged across all inputs. The second *speedup* group shows results averaged across the subset of inputs where RPG^2 outperforms the original code. The final *slowdown* group includes only inputs where RPG^2 detects a performance regression and rolls back to the original code. In some cases RPG^2 neither improves over the original performance nor rolls back, so the all group includes some inputs that are in neither the speedup group nor the slowdown group. The number of inputs in each group is shown in parentheses below each group name on the x-axis.

We break out the speedup and slowdown groups to highlight the gap between cases where prefetching helps and hurts: this discrepancy is easily lost when averaging performance but can be important in settings where predictable tail latency is required. Considering the speedup group first, RPG^2's performance gains are on par with what is achievable via a state-of-the-art static approach like *APT-GET* or offline. A greater separation appears in the slowdown groups, where RPG^2 does a better job of preserving the original's performance than *APT-GET* or offline. This is especially noticeable in the pr slowdown group as well as with bfs, where all except from a few inputs suffer a slowdown from prefetching. RPG^2 offers higher average performance and much lower standard deviation in these cases.

The gap between RPG^2 and the active-only bars arises because of the noise intrinsic to online profiling, where sometimes there are an insufficient number of sampled LLC misses to activate RPG^2's optimization phases. The gap between active-only and offline shows the price of RPG^2's online phases, which incur a delay before the optimized code can begin to execute. With longer-running programs, these costs can be more effectively amortized.

The results also show some immediate opportunities for improvement. For example, we noticed that sssp running the as20000102 input on Cascade Lake suffers a significant slowdown with prefetching, but RPG^2 fails to roll back to the original code. This is due to insufficient profiling: a low-IPC program phase during the profiling period, followed by a higher-IPC phase, causes RPG^2 to incorrectly attribute the IPC improvement to prefetching instead of the phase transition. Without prefetching, IPC would be higher still. This case indicates that IPC alone is not always a good performance indicator. There is also a sizeable gap between RPG^2 and optimal and manual for the randacc benchmark. We found this benchmark, which randomly jumps around an array with indirect accesses, exhibits a very peculiar behavior

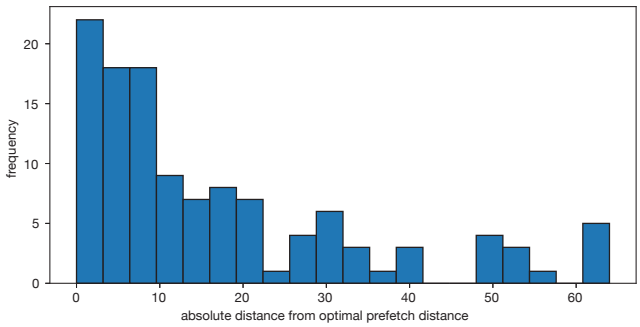

Figure 8. How close RPG^2 gets to the optimal prefetch distance, for inputs with a single optimal distance.

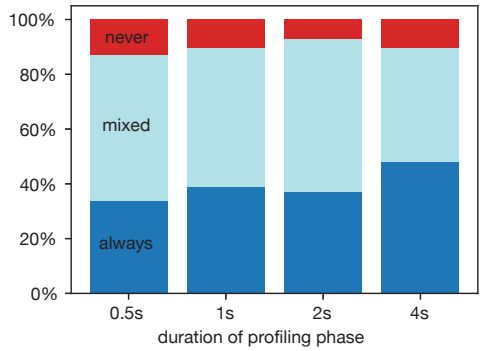

Figure 9. The impact of profiling phase duration on RPG^2's optimization activation.

where prefetch distances that are multiples of 8 perform very well, and all other distances perform much worse. RPG^2's prefetch search assumes the search space is relatively smooth and, therefore, sometimes misses these special distances.

4.3 RPG2 Characterization

In this section, we explore different aspects of RPG^2's operation, starting with its ability to find the optimal prefetch distance for a given input. Figure 8 shows, for just the 120 inputs across all our benchmarks that exhibit a clear single, optimal prefetch distance d, how far away RPG^2's search result was from d in absolute terms. The single optimal distance makes it easy to measure how well RPG^2 is doing. The results are summarized as a histogram, so the leftmost bar shows that, for 22 inputs, RPG^2 was within 3 of the correct distance and within 10 for just over half of the inputs. Being closer to the optimal distance is generally better, but the performance gain from additional proximity is not always very high. Overall, Figure 8 shows that RPG^2's prefetch distance search (§ 3.4) does well most of the time. The biggest impediment to improving the search results is noisy IPC measurements, which sometimes give a misleading view of the search space, causing the search to terminate prematurely.

In Figure 9, we evaluate RPG^2's sensitivity to the duration of its initial profiling phase (§ 3.1), for the pr benchmark

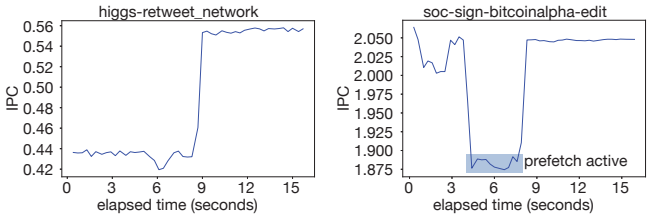

Figure 10. RPG^2's impact on IPC over time.

on Cascade Lake. While RPG^2 profiles for 2 seconds by default, we also measured the effect of profiling for shorter and longer periods. Each bar shows, across all runs of each pr input, how often RPG^2's optimization phases (code generation, injection and tuning) were always, sometimes ("mixed") or never activated. As the profiling phase increases, RPG^2 optimizations are activated more often, though the influence is mild. Longer profiling also diminishes, per Amdahl's Law, the time that optimizations can accelerate execution. We have found two seconds to be a reasonable trade-off in practice, but longer-running benchmarks could amortize longer profiling for further improved performance.

4.3.1 RPG2 Latencies. The left side of Figure 10 shows a "live" view of RPG^2 in action, with IPC measured every 300 milliseconds for an execution of pr with the higgs-retweet-network input on our Haswell machine. The process has an initial IPC of about 0.44, which dips slightly around the 6-second mark as RPG^2 enters code injection and tuning. After tuning, RPG^2 settles on 62 as the prefetch distance which boosts IPC by over 25%.

The right graph in Figure 10 shows a similar view for the soc-sign-bitcoinalpha-edit input with pr on Cascade Lake, where RPG^2 discovers that prefetching harms performance instead. Prefetching remains active for a few seconds (the shaded region) while the prefetch distance search attempts to find a beneficial distance, before rolling back at around the 8-second mark, restoring the original code's performance.

Table 2 examines in more detail the latency of key steps within RPG^2's execution. Each cell shows an average across all inputs for the given benchmark. The first row shows RPG^2's overall execution time, from when profiling begins until RPG^2 detaches having completed prefetch distance tuning. For most of this time, the target process can continue running in parallel. The next row shows that BOLT takes about 30 milliseconds to generate a binary containing the prefetch kernel (§ 3.2.3), which also occurs in the background.

The last three rows show the latency of RPG^2's stop-the-world operations where the target process must be paused: initial code insertion (§ 3.3) takes 3-4ms, a single prefetch distance edit takes 1.1-1.4ms and 11-12 distances are explored during RPG^2's prefetch distance search. These low latencies contribute to RPG^2's mild impact on performance during code insertion and tuning, as reflected in Figure 10.

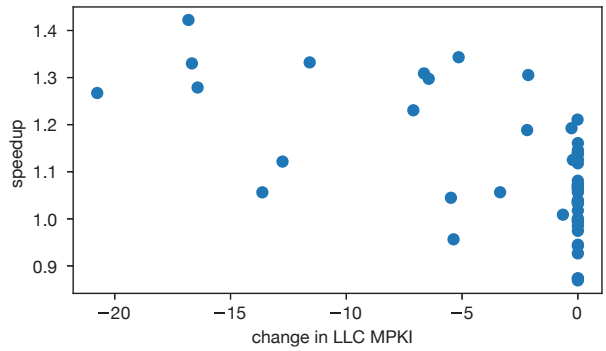

Figure 11. The relationship between speedup and LLC MPKI for pr on Cascade Lake.

benchmark	pr	sssp	bfs	bc	is	randacc	cg
RPG^2 exec (s)	7.6	7.9	7.8	8.5	8.6	7.9	8.8
BOLT (ms)	30.3	28.4	28.6	29.1	27.2	32.0	26.4
code insert (ms)	3.9	3.3	3.1	3.4	3.3	3.0	2.9
1x pd edit (ms)	1.2	1.1	1.4	1.4	1.2	1.1	1.1
# pd edits	11.1	11.3	11.5	12.7	11.4	8.8	12.0

Table 2. Average latency of RPG^2 operations.

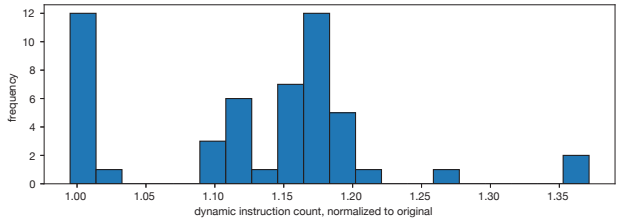

Figure 12. RPG^2's impact on dynamic instruction count for pr on Cascade Lake.

4.4 Performance Counter Validation

To validate that RPG^2's speedup comes from reducing cache misses via prefetching, we measured LLC MPKI via perf stat for all of pr's inputs on Cascade Lake and show the results in Figure 11. Speedup is shown on the y-axis. While RPG^2 reduces LLC MPKI, the relationship between the amount of LLC MPKI reduction and speedup is not especially strong. We initially experimented with using LLC MPKI, instead of IPC, as our performance metric during the tuning phase but were unable to find good prefetch distances, as different distances had little impact on MPKI despite a large impact on performance. We believe that a reduction in misses elsewhere in the hierarchy, as well as changes in DRAM bandwidth consumption, can explain the rest of the change in speedup.

We also measure RPG^2's impact on the dynamic instruction count for the extra instructions needed to run the prefetch kernel. We measured dynamic instruction count, normalized to the original no-prefetch execution, for all inputs of pr on Cascade Lake. Depending on the size of a program's prefetchable data structures (which is input-dependent) and the chosen prefetch distance, varying number of prefetches will be in-bounds and thus executed, affecting the dynamic instruction count. Figure 12 is a histogram showing how frequently we observed a particular increase in instruction count. Overall, half of the inputs see an increase beneath 15%, and the worst-case was a 37% increase. RPG^2's speedup results already include (and overcome) the overhead of these extra instructions.

	Cascade Lake				Haswell			
type	pr	sssp	bfs	bc	pr	sssp	bfs	bc
single optimal	26	25	1	2	18	23	0	1
range optimal	2	7	1	3	7	9	0	0
asymptotic	15	13	2	2	18	14	1	6
both bad	11	10	52	0	11	10	52	0
Haswell bad	-	-	-	-	1	1	5	0
Cascade bad	2	2	5	0	-	-	-	-
noisy	4	3	1	0	7	2	4	1
other	2	2	0	2	0	3	0	1

Table 3. The influence of prefetch distance on performance.

4.5 Prefetch Distance Sensitivity

To better quantify the challenge of doing prefetching well, we used the offline configuration to measure the performance of all prefetch distances in the range [1,100]. While a subset of these results were presented earlier in Figures 1-3, we give a more comprehensive view in Table 3. For each (benchmark,input) pair, we manually examined its prefetch distance versus runtime data and classified it into one of eight types: *single optimal* where there is a clear single prefetch distance that performs best, *range optimal* where a bounded range of distances all perform equivalently, *asymptotic* where performance saturates as distance increases (*e.g.*, all of the curves in Figure 2), *both bad* where prefetching hurts performance on both machines, *Haswell bad* and *Cascade bad* where prefetching is harmful on one machine but beneficial on the other, *noisy* where the behavior is too erratic to be cleanly classified, and *other* for all remaining cases. Table 3 quantifies the challenge of identifying good prefetch distances – generally, fewer than half of inputs exhibit the asymptotic shape that makes it especially easy to find a good prefetch distance. For single and range optimal cases, some kind of prefetch distance search is necessary. The behavior of inputs is also not especially stable across machines, *e.g.*, 27 pr inputs are single optimal on Cascade Lake, but only 17 are on Haswell. This input- and microarchitecture-variability makes it hard to prefetch well in the absence of dynamic feedback.

While most of our benchmarks present only a single demand load that can be accelerated via prefetching, sssp has

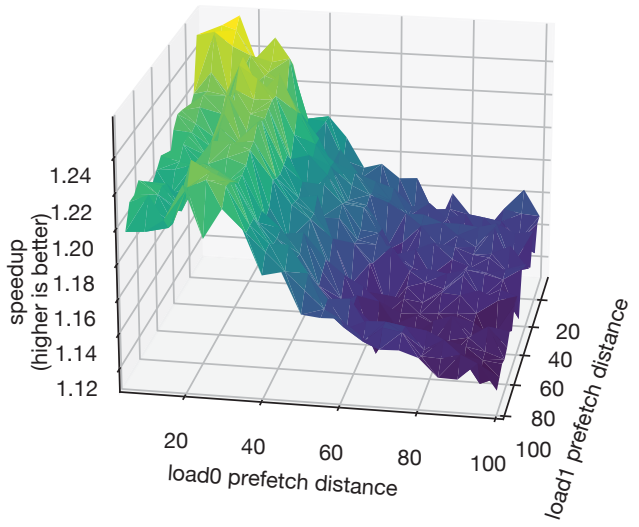

Figure 13. The relationship between multiple loads' prefetch distances and performance for sssp running the p2p-Gnutella05 input on Cascade Lake.

two such loads. The prefetch distance for each prefetch can thus be set differently. We found a variety of behaviors for sssp across different inputs. Sometimes using a "symmetric" configuration with the same distance for both prefetches (which is RPG^2's behavior) was optimal; sometimes symmetric performed the worst, but all asymmetric configurations performed equally well. Figure 13 shows another interesting asymmetric case where performance depends largely on load0's distance (x-axis), though getting load1's distance right is worth a few additional percentage points of speedup as well. Efficiently searching the space of prefetch distances with multiple loads may present an interesting direction for future work.

5 Related Work

As the gap between processor and memory speeds is the underlying cause of many performance problems in today's computer systems, researchers have proposed a myriad of hardware, compiler, and operating systems techniques to improve data locality. We discuss the most closely related work in software prefetching, hardware prefetching, and in dynamic program optimization.

Software prefetching mechanisms. The most related work to RPG^2 are compiler-based approaches to software prefetch injection [4, 12, 15, 17, 21, 32, 40, 43–45, 54, 55, 60, 63, 64, 67]. These systems analyze source code to identify patterns that are amenable to prefetching and can automatically add software prefetch instructions accordingly, and can suggest new source code patterns that RPG^2 can support (§ 3.2). Some of these schemes [3, 28, 34] can leverage profiling information to make more informed decisions about what and how to prefetch. However, once a binary is produced, the prefetch location and distance within it are fixed, which prevents adapting to runtime conditions as RPG^2 can.

Hardware prefetching mechanisms. Hardware prefetching techniques include a wide-range of prefetchers [18–20, 22, 25, 26, 29, 37, 41, 46, 47, 49, 53, 56, 58, 59, 68] that can capture a variety of patterns, including indirect prefetchers [61, 65] and criticality-aware prefetchers [39, 52]. Modern processors adopt a combination of these hardware prefetchers [5, 27, 31, 36, 48], while also relying on software prefetching techniques to cover other complex memory access patterns [3, 28, 34]. RPG^2 is complementary to existing hardware techniques, and all hardware prefetchers were enabled for our experiments.

Programmable hardware prefetching mechanisms. Others have proposed hybrid hardware-software approaches to prefetching [8, 24, 35, 61, 66], where the hardware prefetcher can be programmed by software. If such prefetchers one day appear in commercial processors, they would be an ideal complement to RPG^2 that could provide low-overhead ways to implement prefetch kernels, avoiding some of the software costs that RPG^2 incurs to support on-stack replacement.

Dynamic software optimizations. While we are not aware of any other dynamic prefetch injection systems for unmanaged languages, there are related systems for dynamic code optimization. OCOLOS [69] performs code layout optimizations at runtime for unmanaged code and also uses BOLT [50], though it does not support on-stack replacement. HP's Dynamo system [7] performs optimizations on unmanaged code at runtime, though it does not insert data prefetches. DynamoRIO [11] and Intel's Pin [42] are dynamic binary instrumentation platforms that can be used to implement optimizations though that is not their primary focus. LiteInst [14] is another instrumentation platform focused on low-overhead trampolines to instrumentation code, though with a higher runtime cost (and lower fixed cost) than our current approach. It would be interesting to consider using LiteInst to quickly insert prefetches and judge their efficacy, only resorting to BOLT once we are confident that prefetching is beneficial.

6 Conclusion

In this paper, we have described the RPG^2 system to add, monitor and adjust prefetching online while a program is running. We showed that prefetching can be highly sensitive to program input and microarchitecture and that RPG^2 can adapt the prefetching configuration to the current environment. RPG^2 is especially effective at adding guardrails around the performance cliffs that prefetching can expose by automatically restoring the speed of the original no-prefetching baseline within a few seconds if we happen to fall from one of these cliffs.

Acknowledgments

We thank the anonymous reviewers for their insightful suggestions and feedback. We thank Pranoti Dhamal for help on earlier versions of this work. This work was supported by generous gifts from Intel Labs, the Intel TSA project, Google, NSF/Intel joint grant #2011168, NSF #1942754, NSF #2346057, NSF #1841545, and the PRISM Research Center, a JUMP Center cosponsored by SRC and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] BOLT: Binary Optimization and Layout Tool, 2019. https://github.com/facebookarchive/BOLT.

[2] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*, pages 44–55, 2015.

[3] Sam Ainsworth and Timothy M. Jones. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 305–317, 2017.

[4] Hassan Al-Sukhni, Ian Bratt, and Daniel A. Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, page 91, USA, 2003. IEEE Computer Society.

[5] Erika S Alcorta, Mahesh Madhav, Scott Tetrick, Neeraja J Yadwadkar, and Andreas Gerstlauer. Lightweight ml-based runtime prefetcher selection on many-core platforms. *arXiv preprint arXiv:2307.08635*, 2023.

[6] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 513–526, 2020.

[7] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 1–12, New York, NY, USA, 2000. Association for Computing Machinery.

[8] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. Analysis and optimization of the memory hierarchy for graph processing workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–386, 2019.

[9] Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. XRay: A Function Call Tracing System, 2016. https://research.google/pubs/pub45287/.

[10] Peter Braun and Heiner Litz. Understanding memory access patterns for prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, 2019.

[11] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, page 265–275, USA, 2003. IEEE Computer Society.

[12] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, page 40–52, New York, NY, USA, 1991. Association for Computing Machinery.

[13] Chandranil Chakraborttii and Heiner Litz. Learning i/o access patterns to improve prefetching in ssds. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 427–443. Springer, 2020.

[14] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 320–332, New York, NY, USA, 2017. Association for Computing Machinery.

[15] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wen-mei W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, MICRO 24, page 69–73, New York, NY, USA, 1991. Association for Computing Machinery.

[16] Meng-Chieh Chiu, Benjamin Marlin, and Eliot Moss. Real-time program-specific phase change detection for java programs. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer cache assisted prefetching. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pages 62–73, 2002.

[18] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 14–25, 2001.

[19] Fredrik Dahlgren and Per Stenstrom. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture*, pages 68–77, 1995.

[20] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, page 68–75, New York, NY, USA, 1997. Association for Computing Machinery.

[21] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, page 354–368, New York, NY, USA, 1990. Association for Computing Machinery.

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

[23] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In *International Conference on Machine Learning*, pages 1919–1928. PMLR, 2018.

[24] Urs Hölzle and David Ungar. A third-generation self implementation: Reconciling responsiveness with performance. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Language, and Applications*, OOPSLA '94, page 229–243, New York, NY, USA, 1994. Association for Computing Machinery.

[25] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 397–408, 2006.

[26] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 499–500, New York, NY, USA, 2009. Association for Computing Machinery.

[27] Majid Jalili and Mattan Erez. Managing prefetchers with deep reinforcement learning. *IEEE Computer Architecture Letters*, 21(2):105–108, 2022.

[28] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 747–764, New York, NY, USA, 2022. Association for Computing Machinery.

[29] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, page 252–263, New York, NY, USA, 1997. Association for Computing Machinery.

[30] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 158–169, 2015.

[31] Hui Kang and Jennifer L. Wong. To Hardware Prefetch or Not to Prefetch? A Virtualized Environment Study and Core Binding Approach. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 357–368, New York, NY, USA, 2013. Association for Computing Machinery.

[32] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. A case for resource efficient prefetching in multicores. In *2014 43rd International Conference on Parallel Processing*, pages 101–110, 2014.

[33] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-guided btb prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 816–829, 2021.

[34] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 163–181. USENIX Association, July 2021.

[35] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–159. IEEE, 2020.

[36] Sushant Kondguli and Michael Huang. Division of labor: A more effective approach to prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 83–95. IEEE, 2018.

[37] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin. Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(9):1309–1324, 2009.

[38] Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection., 2014. https://snap.stanford.edu/data/.

[39] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. Crisp: critical slice prefetching. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 300–313, 2022.

[40] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 180–190, 2003.

[41] Jiwei Lu, A. Das, Wei-Chung Hsu, Khoa Nguyen, and S.G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc/spl reg/ cmp processor. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 12 pp.–104, 2005.

[42] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery.

[43] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, page 222–233, New York, NY, USA, 1996. Association for Computing Machinery.

[44] Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P. Geoffrey Lowney, and Robert Cohn. Profile-guided post-link stride prefetching. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, page 167–178, New York, NY, USA, 2002. Association for Computing Machinery.

[45] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, page 62–73, New York, NY, USA, 1992. Association for Computing Machinery.

[46] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Techniques for efficient processing in runahead execution engines. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 370–381, 2005.

[47] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 129–140, 2003.

[48] Carlos Navarro, Josué Feliu, Salvador Petit, Maria E Gomez, and Julio Sahuquillo. Bandwidth-aware dynamic prefetch configuration for ibm power8. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1970–1982, 2020.

[49] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 96–96, 2004.

[50] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: A practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2019.

[51] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. Lightning bolt: Powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 119–130, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Biswabandan Panda. Clip: Load criticality based data prefetching for bandwidth-constrained many-core systems. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*, 2023.

[53] Tanausu Ramirez, Alex Pajuelo, Oliverio J. Santana, and Mateo Valero. Runahead threads to improve smt performance. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 149–158, 2008.

[54] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, page 115–126, New York, NY, USA, 1998. Association for Computing Machinery.

[55] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, page 111–121, USA, 1999. IEEE Computer Society.

[56] Suleyman Sair, Timothy Sherwood, and Brad Calder. A decoupled predictor-directed stream prefetching architecture. *IEEE Transactions on Computers*, 52(03):260–276, mar 2003.

[57] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 45–57, New York, NY, USA, 2002. Association for Computing Machinery.

[58] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[59] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 171–182, 2002.

[60] Seung Woo Son, Mahmut Kandemir, Mustafa Karakoy, and Dhruva Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, page 209–218, New York, NY, USA, 2009. Association for Computing Machinery.

[61] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, et al. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 654–667. IEEE, 2021.

[62] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[63] Michael Joseph Wolfe, Carter Shanklin, and Leda Ortega. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[64] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 210–221, New York, NY, USA, 2002. Association for Computing Machinery.

[65] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. Imp: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 178–190, 2015.

[66] Chao Zhang, Yuan Zeng, John Shalf, and Xiaochen Guo. Rnr: A software-assisted record-and-replay hardware prefetcher. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 609–621. IEEE, 2020.

[67] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12 pp.–64, 2006.

[68] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. Accelerating and adapting precomputation threads for effcient prefetching. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 85–95, 2007.

[69] Yuxuan Zhang, Tanvir Ahmed Khan, Gilles Pokam, Baris Kasikci, Heiner Litz, and Joseph Devietti. Ocolos: Online code layout optimizations. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 530–545, 2022.