RPG$^2$: Robust Profile-Guided Runtime Prefetch Generation

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

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

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

Tanvir Ahmed Khan  
tk3070@columbia.edu  
Columbia University

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

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

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

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

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. Competing 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 2.15x, 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.

ACM Reference Format:

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. 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 RPG² system for dynamic prefetch insertion and tuning. RPG² 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. RPG² is, to our knowledge, the first system to enable dynamic software prefetching. RPG² can adapt prefetching to program inputs while the program runs, avoiding the pitfalls of ahead-of-time profiling and static compiler optimization. RPG² 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], RPG² targets programs with 1-2 small hot loops, each containing a small number of load instructions that are potentially prefetchable. Unlike prior work, RPG² builds on the BOLT [50, 51] binary optimization tool and thus does not need access to source code: RPG² 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. RPG² 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 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 load demand 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 input (light green lines) experiences 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 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 RPG2 runtime framework for adding, tuning, and removing prefetch instructions while a program is running. We describe how RPG2 works next.

3 Design of RPG2

We first give an overview of RPG2 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 RPG2’s main phases.

RPG2’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. RPG2 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.

RPG2’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, RPG2 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 RPG2’s main optimization target, though it can target direct loads as well as we discuss in §3.2. As a result, RPG2’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, RPG2 now enters its third phase: injecting the prefetch code into the running process and transferring control to it. RPG2 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 RPG2 enters its fourth phase: tuning the prefetch distance. RPG2 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, RPG2 changes the initial prefetch distance of 20 to 32 by adjusting the immediate value of the add instruction. After the search completes, RPG2 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.

<table>
<thead>
<tr>
<th>demand access</th>
<th>prefetch</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a[j])</td>
<td>(a[j+d])</td>
<td>direct access using inner loop induction var</td>
</tr>
<tr>
<td>(a[f(b[i]+d)])</td>
<td>(a[f(b[i]+j)])</td>
<td>indirect access using inner loop induction var</td>
</tr>
<tr>
<td>(a[f(b[i]+d)+j])</td>
<td>(a[f(b[i]+d)+j])</td>
<td>indirect access using inner and outer loop induction vars</td>
</tr>
</tbody>
</table>

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 
够 to match many code patterns, like dense arrays and 
stencils (category 1) and sparse arrays (categories 2 and 3).
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: \(ld [r3+r2] \rightarrow r4\) is the demand load where 
we begin computing a backward slice, which encompasses

![Figure 5. Annotated example of RPG\(^2\) code transformations.](image-url)
only 1d \([r0+r1]\rightarrow r2\) since \(r0\) and \(r3\) are loop-invariant and \(r1\) is the loop induction variable.

RPG2 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, RPG2 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.

RPG2’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 NOPs, 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 \(bge\)) 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, RPG2 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.

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.

RPG2 leverages Linux’s ptrace API to perform the actual code insertion. ptrace allows RPG2 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, RPG2 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 $f_i$ code into it.

3.3.1 On-Stack Replacement. Our task now becomes to
redirect execution to $f_i$. Figure 6 illustrates how this works.
Direct calls to $f_0$ from other functions are patched to refer
to $f_i$ 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 $f_i$ 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 lit-
erature as on-stack replacement [24] (OSR), and many sophis-
ticated 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 is the only sys-
tem 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 com-
piler support. Even the recent Ocolos [69] system, which
provides online code layout optimization for unmanaged
languages, does not support OSR.

RPG is able to support OSR for two reasons. First, RPG 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 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 $a$ would
be allocated to different registers in $f_0$ versus $f_i$ (or stack-
allocated in one and register-allocated in the other); moving
execution to $f_i$ then requires understanding how variables
are mapped to data, which is not transparent in existing
unmanaged compilers. Inspired by these challenges, RPG’s
code transformations instead incur zero data layout changes.

The second reason RPG can support OSR is that, even
though RPG 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 has optimized
before; RPG uses the BATT to map PCs from $f_0$ to $f_i$ during
OSR. Once code insertion completes, libpg2 raises a SIGSTOP
signal to send a notification via ptrace. Upon receiving the
SIGSTOP, RPG moves the stack pointer back to the original
value and updates thread PCs to point to their corresponding
instructions in the $f_i$ code. Then RPG resumes the target
process, which from now on executes $f_i$ instead of $f_0$ code.

3.4 Phase 4: Monitoring And Tuning

After the prefetch kernel has been inserted, RPG determines
the optimal prefetch distance via binary search. In particular,
RPG 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 monitors performance by
measuring IPC via perf stat.

The prefetch distance search algorithm has three stages.
In the first stage, RPG decides the direction for searching.
Starting from a random number $r$ drawn from the interval
$[1, 100]$ as the initial prefetch distance, RPG 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 samples
cosinely in the identified direction to find a region of
promising prefetch distances.

In stage 2, RPG keeps on doubling the jump size to com-
pute the new prefetch distance in the chosen direction, so
long as RPG 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 finds an IPC decrease, it
sets the $n_{th}$ prefetch distance and the $n + 1_{th}$ 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 loca-
tion. 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 currently restricts all prefetch
locations to have the same distance.

After RPG 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 dis-
tance search completes, RPG rolls back to the original $f_0$
code, which remains in the address space (Figure 6). When RPG² 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 $f_0$ locations into their corresponding $f_6$ 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, RPG² 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 RPG²’s performance compared to a range of baselines. Next, we examine how accurate RPG²’s prefetch distance search is, how much profiling data it needs to work well, and how long key RPG² operations take. Then, we measure RPG²’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 RPG² 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 RPG², we just record the execution reported by the original binary. RPG² 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, RPG² measures IPC for 0.3 seconds.

4.1.1 Baselines. We accurately gauge RPG²’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 RPG² runs when it gathers enough profiling data to enable optimization (which does not always occur, as we explore later in § 4.3). The main RPG² bars, in contrast, also include runs where RPG² starts but does not receive enough profiling information to justify enabling prefetching.

• an offline version of RPG² 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 RPG²’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 RPG²’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 RPG² 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
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 RPG2 outperforms the original code. The final slowdown group includes only inputs where RPG2 detects a performance regression and rolls back to the original code. In some cases RPG2 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, RPG2’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 RPG2 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. RPG2 offers higher average performance and much lower standard deviation in these cases.

The gap between RPG2 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 RPG2’s optimization phases. The gap between active-only and offline shows the price of RPG2’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 RPG2 fails to roll back to the original code. This is due to insufficient profiling: a low-IPC phase during the profiling period, followed by a higher-IPC phase, causes RPG2 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 RPG2 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.
where prefetch distances that are multiples of 8 perform very well, and all other distances perform much worse. RPG\textsuperscript{2}’s prefetch search assumes the search space is relatively smooth and, therefore, sometimes misses these special distances.

### 4.3 RPG\textsuperscript{2} Characterization

In this section, we explore different aspects of RPG\textsuperscript{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\textsuperscript{2}’s search result was from \(d\) in absolute terms. The single optimal distance makes it easy to measure how well RPG\textsuperscript{2} is doing. The results are summarized as a histogram, so the leftmost bar shows that, for 22 inputs, RPG\textsuperscript{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\textsuperscript{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\textsuperscript{2}’s sensitivity to the duration of its initial profiling phase (§ 3.1), for the pr benchmark on Cascade Lake. While RPG\textsuperscript{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\textsuperscript{2}’s optimization phases (code generation, injection and tuning) were always, sometimes ("mixed") or never activated. As the profiling phase increases, RPG\textsuperscript{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 RPG\textsuperscript{2} Latencies

The left side of Figure 10 shows a "live" view of RPG\textsuperscript{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\textsuperscript{2} enters code injection and tuning. After tuning, RPG\textsuperscript{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\textsuperscript{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\textsuperscript{2}’s execution. Each cell shows an average across all inputs for the given benchmark. The first row shows RPG\textsuperscript{2}’s overall execution time, from when profiling begins until RPG\textsuperscript{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\textsuperscript{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\textsuperscript{2}’s prefetch distance search. These low latencies contribute to RPG\textsuperscript{2}’s mild impact on performance during code insertion and tuning, as reflected in Figure 10.
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.

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.

While most of our benchmarks present only a single demand load that can be accelerated via prefetching, sssp has
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 RPG2’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 RPG2 is 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 RPG2 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 RPG2 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]. RPG2 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 RPG2 that could provide low-overhead ways to implement prefetch kernels, avoiding some of the software costs that RPG2 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. Litelnst [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 Litelnst 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 RPG2 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 RPG2 can adapt the prefetching configuration to the current environment. RPG2 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


