Iridescent: A Framework Enabling Online System Implementation Specialization

Vaastav Anand
Deepak Garg
Antoine Kaufmann
Max Planck Institute for Software Systems
Saarbrücken, Germany

Abstract

Specializing systems to specifics of the workload they serve and platform they are running on often significantly improves performance. However, specializing systems is difficult in practice because of compounding challenges: i) complexity for the developers to determine and implement optimal specialization; ii) inherent loss of generality of the resulting implementation, and iii) difficulty in identifying and implementing a single optimal specialized configuration for the messy reality of modern systems.

To address this, we introduce Iridescent, a framework for automated online system specialization guided by observed overall system performance. Iridescent lets developers specify a space of possible specialization choices, and then at runtime generates and runs different specialization choices through JIT compilation as the system runs. By using overall system performance metrics to guide this search, developers can use Iridescent to find optimal system specializations for the hardware and workload conditions at a given time. We demonstrate feasibility, effectivity, and ease of use.

1 Introduction

Specializing system implementations to workload characteristics and hardware can significantly improve performance and efficiency [3, 6, 15, 17, 19, 22, 25, 26, 33, 34, 43, 44, 46, 51, 53, 56, 57, 60, 64]. Achieving these benefits requires manual modification the system implementations and recompilation. Part of the performance benefit arises from cascading compiler optimizations, e.g. by removing a feature enabling the compiler to eliminate dead code, in turn enabling further optimizations [22, 47, 61]. Today, system specialization for performance is manual, developer-driven, and iterative.

Building optimal systems in practice is a challenge because of three compounding factors: First, specialization is difficult for developers to implement, as it requires trial-and-error and maintaining multiple versions of key code paths. Next, specialization fundamentally comes at the cost of generality — a specialized system either performs poorly outside its regime, or completely fails. Finally, predicting performance implications of specialization choices for different workloads and hardware is almost impossible. To make matters worse, workload and platform conditions are dynamic and optimal specialization choices depend on them.

We propose a fundamentally different approach: *automated online system specialization guided by observed overall system performance*. Our goal is to specialize performance critical system systems at runtime to the exact momentary workload and hardware conditions, *without the human developer on the critical path*. To enable this, we re-distribute tasks between ahead of time development and runtime operation. Ahead of deployment, rather than choosing a concrete set of specializations, *developers specify the space of possible specializations*, along with a search strategy. After deployment, our runtime iteratively *generates different specialized code* versions, choosing the current optimum based on observed overall system performance. This turns compile-time specialization into runtime parameter tuning, suitable for search-based auto-tuning [57, 60].

To enable this, we introduce Iridescent, a framework for practical development of fast and efficient systems with online specialization. Iridescent enables both the incremental specialization of existing code-bases with minimal modifications, and in-depth design for specialization for maximizing performance. Iridescent builds on LLVM to target systems written in typical (non-managed) languages such as C(++) or Rust. Concretely, Iridescent supports the developer in separating the code-base into a specialized, performance critical core, and the remaining generic code. Iridescent then provides the developer with a specialization API to annotate possible specializations in the code thereby specifying the space of possible specializations. Additionally, Iridescent provides hooks to optionally customize JIT code generation in depth. Finally, Iridescent provides the runtime API to control the exploration of different specialized implementations as the system runs.

We integrate Iridescent with multiple systems and show that with Iridescent enabled specializations, systems can gain a performance boost of upto 30×. Moreover, with Iridescent, developers can easily configure the system to automatically explore the specialization space to adaptively find the best performing specialization for different workloads at runtime.

2 Motivation & Challenges

2.1 Illustrative Example: Matrix Multiply Server

We illustrate the benefits and challenges of specialization with an example system: a server executing square matrix

Machine / Workload	N=1024	N=256	N=64
IceLake	32	32	32
IvyBridge	16	16	4
CoffeeLake	32	4	4
AlderLake-p	32	4	2
AlderLake-e	64	4	4

Table 1. Optimal configurations for our block matrix multiply, across 5 hardware platforms and 3 workloads.

multiplications for clients. To optimize cache locality we use a blocked matrix multiplication [9, 62, 63]. We identify one key workload parameter, N the matrix size, and a key configuration parameter, B the block size.

Code specialization improves performance. Both parameters offer a substantial opportunity for optimization through specialization. Unsurprisingly, we find fixing B as a compile-time constant instead of leaving it a variable improves throughput by up to 6.5×, by enabling the compiler to unroll and vectorize the inner loops. Note that B is an internal parameter that may affect performance, but any valid choice results in correct behavior with every workload. Similarly, assumptions about the workload, N, can also simplify the algorithm, e.g. by assuming that N is a multiple of B we avoid the need for copying and 0-padding partial blocks.

Optimal configuration depends on workload and HW. We now compare different block sizes for different workloads (matrix sizes) on 5 different processors. As Table 1 shows, different block sizes yield optimal performance for different workload and processor combinations. Picking a fixed B ahead of a practical deployment in a dynamic environment will not yield optimal performance. Moreover, even with single single-size workloads it is difficult to predict what block size will be ideal for a concrete processor.

2.2 Code Specialization is Effective but Complex

There is a long line of prior work that has established the performance and efficiency benefits of specializing code, and systems code more specifically. For example, interpreted languages may generate specialized instances of functions with constant parameters and optimize them accordingly [16, 40]. This is a generic optimization and typically done automatically and transparently by the runtime. Other optimizations are specific to individual systems and concrete concerns, such as inlining small table lookups in software network functions as if statements [43]. As most compiler optimizations, these approaches are typically guided by simple heuristics around local metrics (e.g. frequency of the same parameter value, or table size) based on developer intuition.

Specialization effects often cascade. A key aspect of specialization techniques is that the effects combine. For example, PacketMill [22] first de-virtualizes function pointers [36] for the Click modular router [37], and then based on this further eliminate dead code and data structure fields. The analysis for the latter optimization is impossible before devirtualization as the virtual calls prevent the compiler from analyzing the complete packet handling code.

2.3 Specialization is Challenging in Practice

(C1) Developer Complexity. A key barrier to specialization is the increased complexity for developers. Developers now need to reason about what assumptions will definitely hold and can help specialize the system for better performance. Next the developer needs to implement the required specialization and evaluate it. In practice, this frequently results in having to maintain multiple code versions. Unsurprisingly, is complex but also laborious and frequently also frustrating, since it is often an iterative hit-and-miss process.

(C2) Loss of Generality. This is further complicated since specializations may hurt performance or break the system when the underlying assumptions are violated. For example, a network function may be processing 99.999% of packets with simple processing and data structures that perform vastly better with specialization. However, specializing the system to this class of packets, and thereby simplifying code and data structures breaks handling of the rare 0.001% case. The need to gracefully handle rare or unexpected cases forces developers to specialize conservatively.

(C3) Optimal Specialization Choices. Finally, as we have seen above, determining the right choices to actually yield optimal overall system performance is a challenge. Modern system performance is a complex product of the emergent behaviors across all system components, it is also not feasible for developers to distill this into simple local cost models. Optimally specializing systems statically is fundamentally impossible since concrete workload and hardware parameters affect these choices, and both are dynamic in practice.

3 Approach

We aim to enable practical specialization of systems for improved performance and efficiency. Our key idea to enable this given the challenges above, is to *automatically explore a space of possible specializations provided by the developer at runtime based on observed overall system performance.*

Simple specialization with simple annotations. We first observe, that while developers struggle with determining optimal specializations and implementing a system based on them, it is much easier for developers to suggest *possible specialization assumptions.* We thus enable developers to annotate system code with simple annotations to this end. Figure 2a shows an example for the matrix multiplication server above. Additionally, developers can also provide LLVM transformations to perform intrusive app-specific specializations, and thereby generate further candidate configurations in the overall specialization space. Crucially, we do not rely on the developer to filter or rank specializations.

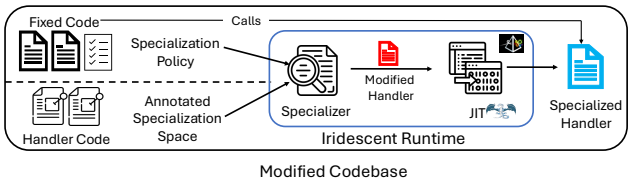

Figure 1. Architecture of a Iridescent-specialized system.

Guarded specialization with a fall-back. Later, when automatically instantiating specialized system versions through JIT compilation, we inject guard conditions into the code and fall back to the unspecialized version of the system code for that invocation. While triggering this guard incurs overhead, it does ensure correctness. And if the guard triggers sufficiently rarely, in combination with the benefit for the common case, the specialization may still be a net win.

Online exploration guided by developer policy. Finally, we completely forgo cost models and heuristics for predicting which choice in the specialization space is optimal, and instead explore different points online as the system is running. Our guards ensure that the system always behaves correctly in this process. We rely on the developer to provide a policy for guiding this exploration. The developers control when to explore which points, manually, through existing autotuning solutions or by leveraging simple search strategies from our library. A key task of this policy is comparing the (application specific) overall system performance metrics, e.g. request throughput, when running different specialization points. Overall system performance metrics also implicitly factor in any overheads, e.g. triggering specialization guards for some calls. This ensures that the system converges to the optimal specialization point for the concrete combination of dynamic system, workload, and hardware conditions.

4 Iridescent Design

We now present the design of our approach in Iridescent.

4.1 Anatomy of a Iridescent System

A key component of the Iridescent mechanism is to modify code of the running system. Since Iridescent targets performance critical systems not implemented in managed languages, this is non-trivial in general. We observe though that typical performance critical systems code are almost exclusively architected with an outer loop handling a sequence of events, such as processing requests or data elements. We leverage this structure for pragmatic specialization, by requiring developers to separate the system code into two parts (Figure 1): (i) the *fixed code*, including initialization and outer loop; (ii) the performance-critical event *handler code*. The developer compiles the fixed code as before, but integrates calls to the Iridescent API (Table 2). For the handler code, the developer adds lightweight Iridescent specialization API calls and compiles to intermediate representation, LLVM IR specifically. In the fixed code, the developer obtains function pointers for the handlers through the Iridescent policy API,

```
void matmul(u64 *L, u64 *R, u64 *O, int N, int B) {
  B = spec_enum("B",B,2,4,8,16,32,64);
  N = spec_general("N",N);
  for (int kk = 0; kk < N; kk += B)
    for (int jj = 0; jj < N; jj += B)
      for (int i = 0; i < N; i++)
        for (int jjj = jj; jjj < jj + B; jjj++)
          for (int kkk = kk; kkk < kk + B; kkk++)
            M3[i*N+jjj] += L[i*N+kkk] * R[kkk*N+jjj];
}
```

(a) Handler Code

```
Iridescentrt("handler_code.ll");
void spec_policy() {
  Conf N_cs[2] ={{}, {{"N", 256}}};
  do { for (c : cartesian(rt->spec_space(), N_cs)) {
         rt.specialize(c); tput_counter = 0;
         sleep(...); // Sleep for some time to monitor
         if (tput_counter > best) {
           best = tput_counter; best_c = c; } }
    rt.specialize(best_c);
  } while(every 1min);
}
int main() {
  auto *matmul = rt.handler("matmul");
  rt.reg_opt_pipeline(opt_passes);
  launch_thread(spec_policy);
  while(true) { // Main processing loop
    Req *req = get_req(); Resp *res = new Resp();
    matmul(req.A, req.B, res.C, 2, req.N);
    tput_counter++;
  }
}
```

(b) Fixed Code

Figure 2. MMulBlockBench microbenchmark

Iridescent Specialization API	
`spec_enum(lbl, x, ...)`	Enumeration spec. point (x∈...)
`spec_range(lbl, x, l, h)`	Range spec. point (l≤x≤h)
`spec_generic(lbl, x)`	Policy-controlled spec. point
`spec_assume(lbl, cond)`	Specialization assumption
`spec_custom_*(lbl, ...)`	Custom spec. point
Iridescent Policy API	
`Iridescent(handler_ir)`	Initialize runtime
`.handler(h)`	Get specialized handler
`.spec_space()`	Obtain code spec. space
`.specialize(c)`	Choose specialization; `c` maps spec. point labels ↦ values
`.add_custom_spec(n, gen)`	Add app specialization
`.customize_opts(passes)`	Modify codegen optimizations

Table 2. Iridescent API overview.

and calls these as before. During dynamic code generation, Iridescent will link handlers against symbols in the fixed code, to enable calls and accesses to global state. Since Iridescent will repeately rebuild handlers, *Iridescent requires that the handler code does not include definitions of global variables that must be preserved.* The developer has to move all global state to the fixed code, and reference from the handlers. Figure 7 illustrates this with our matrix multiplication server running example.

4.2 Defining Specialization Space

The developer defines the specialization space through *specialization point* annotations in the handler code. At runtime, Iridescent replaces these annotations with specialized code, instrumentation, or disables them, guided by the policy. Iridescent offers three types of specialization points:

Value Specialization Point. A value specialization point signals to Iridescent that the handler code could be specialized to a constant for the wrapped expression. `spec_enum` indicates the value could be one of the specified values; `spec_range` instead specifies a range; while `spec_generic` leaves it to the policy to determine possible values.

Assumption Specialization Point. This point provides a *possible specialization assumption* to Iridescent. The condition is a boolean predicate that Iridescent can provide to compiler optimizations to further optimize, through LLVM's builtin `llvm.assume`. Crucially, the assumption need not always hold, unlike for the LLVM intrinsic where incorrect behavior results, Iridescent catches this with its specialization guards.

Custom User-Defined Specialization Point. These specialization points, annotated through `spec_custom_*`, invoke developer-defined specializations registered in the fixed code through `add_custom_spec`. An example is generating an if-else chain as a special-case fast-path for an expensive data structure lookup or computation (§5.1).

4.3 Defining Specialization Policy

The developer implements the specialization policy in the system fixed code using the Iridescent policy API. Through the policy, the fixed code decides what to specialize and when. Iridescent provides a simple periodic exhaustive search strategy for simple cases as a library routine, with a simple callback for providing the strategy with the system performance metric. But we also found in our evaluation there are typically system-specific insights to more intelligently decide when to trigger exploration, and how to prioritize different configurations. As a result, we expect that most systems will implement custom strategies maximizing overall system performance, using the Iridescent building blocks.

Figure 2b shows how through a simple specialization policy, developers can control the exploration and selection of specializations. In line 4, the `spec_space` API returns the specialization space generated by the annotations in the handler code. Here, we then combine this space, with other specializations through a cartesian product to obtain a complete set of specializations. In lines 4-8, the fixed code automatically tries out the different combinations in the set of specializations and chooses the best performing combination. To choose the best, the specialization policy uses the target end-to-end performance metric, in this case the throughput. Here, we then trigger re-exploration at a fixed-time interval to adapt to workload changes.

The specialization policy provides developers fine-grained control for specialization. This also includes enabling instrumentation for specialization points to dynamically identify opportunities for specialization. For example, the policy for the matrix multiplication server may specify that the matrix size, N, should only be specialized if $\geq$70% of the workload has the same value. The developer can also optionally configure custom optimization passes through the `customize_opts` API to further modify code generation.

4.4 Specialization Runtime

The specialization runtime has two key components: (i) the specializer generating the specialized code by applying selected specializations; (ii) the JIT compiling and optimizing the modified specialized code and making it available to the fixed code.

4.4.1 Specializer. The specializer generates the specialized intermediate representations for the handler code. It operates at function granularity, and generates versions of functions with specialization points replaced depending on the chosen specialization configuration.

For each specialization point, Iridescent performs the following actions, depending on the configuration the policy supplied for this point. If the policy marks a point as disabled, the Iridescent specializer simply removes the annotation and replaces it by the original value, or skips over assumption points. To specialize an enabled value specialization point, Iridescent replaces the specialization point annotation in the handler code with the constant value supplied by the policy. By default, the specializer will also insert a specialization guard, which the developers may explicitly disable. To specialize a custom specialization point, the specializer replaces the specialization point with the custom source code provided by the previously registered handler for that custom specialization point type. The recompilation process then enables typical compiler optimization passes to take effect that now operate with the new constants.

Instrumentation. Some specialization policies benefit from collecting runtime data to generate possible specialization configurations. To support such data collection, the specializer can optionally enable instrumentation for each specialization point. For a specialization point with instrumentation enabled, Iridescent additionally generates code for collecting

and storing the observed actual values. The policy retrieves this information included in the result of the spec_space call.

4.4.2 JIT. Compiling the specialized code.

The specialization runtime adds the specialized code generated by the specializer to the JIT. The JIT compiles the code and runs the developer-specified pipeline of optimization passes, including any custom developer-provided optimization passes, to generate an optimized version of the specialized code. By default, the JIT applies the default O3 optimization pipeline if the developer has not provided a specific pipeline. In addition, the JIT also keeps a copy of the original, non-specialized version of the specialized function. This generic function acts as the fallback option for situations where the specialized function is not applicable.

Using the specialized code. To use the specialized handlers, the fixed code obtains function pointers from the specialization runtime. The fixed code then uses to invoke the specialized handlers. If no specialization has been enabled the handlers default to the non-customized generic version of the functions. For simplicity, the fixed code only needs to do this once at the very start of execution. The JIT creates a trampoline function which calls the most recent specialized version of the function. The trampoline function is stored at a fixed address and does not change across runtime updates.

4.4.3 Correctness with Specialization Guards.

Iridescent by default inserts a runtime check called a specialization guard for an enabled specialization point. The guard can be disabled by the policy. The specialization check ensures that the condition for using the specialized version of the code holds during execution. For this, specializer inserts code to perform an early exit from the specialized version of the code on check failure by throwing an exception. The JIT's trampoline function catches the thrown exceptions and re-routes the control flow to the original non-specialized version of the function transparently without exposing the exception to the fixed code.

Restoring state and side effects. To ensure correct restoration of the state, the specializer will call a user-defined cleanup function for reversing any side-effects before throwing the exception. It is critical to note here that not all side-effects are reversible (e.g. sending a packet to a neighbor), so Iridescent performs a best-effort clean-up.

4.5 Prototype Implementation

We have implemented Iridescent in $5K$ lines of C++. Our implementation uses the LLVM IR and JIT [38] to generate the specialized code at runtime. The specializer is implemented as a set of LLVM transformation passes that operate on LLVM IR of the instrumented handler code.

5 Case Studies

For our evaluation we use four open-source systems and libraries as our evaluation target and the MMulBlockBench microbenchmark. We explain each system along with the changes we made for integrating Iridescent below.

In addition to the benchmarks, we also implemented a generic version of the hot key specialization of Morpheus [43] called the fast-path specialization. The fast-path specialization works in two phases: (i) the instrumentation phase: Iridescent inserts instrumentation code that samples (sampling rate selected by the user) invocations of the target function to find the most popular inputs to the function along with their calculated outputs; (ii) the specialization phase: Iridescent updates the target function to have a specialized if-else chain where the top-N inputs along with their outputs are converted into a series of if-else checks to avoid paying the cost of re-computation for the heavy-hitting inputs. The value of N maybe user provided or it could also be configured as a runtime constant specialization point. If the input does not match any of the branches in the if-else chain, then the computation simply defaults to the generic version.

5.1 LibLPM

LibLPM [49] is an open-source Longest Prefix Match (LPM) library written in C with built-in support for IPv4 and IPv6 addresses. LPM is an important operation that is used in packet routing for finding the best and most accurate matches in routing tables for incoming packets.

To integrate LibLPM with Iridescent, we create two different specializations - (i) LibLPM-FP, and (ii) LibLPM-NI.

LibLPM-FP specializes the lookup function to add a fast-path specialization point. The target is the lookup function where the input-output pairs of the lookup function executions are captured.

LibLPM-NI creates a code-generation specialization point in the LPM library. The code-generation specialization point generates a new version of the lookup function which generates a nested-if-else tree of checks consisting of prefix match checks for the incoming address for each lpm entry. The nested-if-else tree starts at the least specific matching rule (based on prefix length) and progressively checks for the most specific matching rule until it can't find one anymore. By performing this code generation specialization, the lookup function can embed the prefix rules directly into the codebase allowing for more optimized checks.

We also create an additional specialization, LibLPM-NI-FP, which combines the previous two specializations by adding a fast-path specialization point in the generated nested-if specialized lookup function.

5.2 TAS

TAS [35], TCP acceleration as a service, is a lightweight software TCP network fast-path that is optimized for common-case client-server RPCs. TAS executes common-case TCP operations in an isolated fast path that uses DPDK [20], while handling corner cases in a slow path.

To integrate TAS with Iridescent, we convert the `BATCH_SIZE` as a runtime constant specialization point. In TAS, the `BATCH_SIZE` variable is used in three different scenarios: (i) to determine the number of packets to be read from the NIC, (ii) to determine the number of packets read from the application queues, and (iii) to determine the number of packets read from the queue manager. For more fine-grained control, we convert each of these usage instances of the `BATCH_SIZE` as three separate runtime constant specialization points - (i) `rx_batch`, (ii) `queues_batch`, and (iii) `qman_batch`.

5.3 FastClick

FastClick [5] is an extended version of the Click Modular Router [37] with improved Netmap support and DPDK support for running the modular router in userspace. A FastClick (or Click) router is assembled from individual packet processing modules called elements. Each element implements simple router functions such as packet classification, scheduling, routing, and interacting with network devices. Users can configure the router with different element pipelines to achieve different router behaviors.

To integrate FastClick with Iridescent, we modify the LinearIPLookup element of the FastClick to create a Iridescent-enabled LinearIPLookup element. The original element uses a linear search algorithm to find the best matching route for an incoming packet based on the routine table. We modify the packet-processing function of the LinearIPLookup element to add a fast-path specialization point. The target of this specialization point is the internal lookup function of the LinearIPLookup element.

5.4 Network Functions

Network Functions is a suite of the network functions that include DPDK and ebpf implementations of common network functions such as a NAT, Router, policer, among others. These functions have been developed and used for evaluation by various research projects such as Pix [30] and Vigor [65]. We extract the implementations from the Pix artifact [29].

To integrate the Network Functions with Iridescent, we convert the `BATCH_SIZE` used by the DPDK version of these functions into a runtime constant specialization point. In the network functions, the BATCH_SIZE controls the number of packets that are read from a network port at a given time.

6 Evaluation

In this section, we evaluate how well Iridescent improves performance of systems by enabling online specializations. We showcase how we can use online specialization with Iridescent for three use-cases: (i) enabling compile-time optimizations at runtime; (ii) enabling incremental specializations at runtime; (iii) design exploration at runtime. We answer the following questions:

- Do Iridescent-enabled compile-time optimizations at runtime improve performance? (§6.2)
- Can Iridescent find an optimal design through exploration at runtime under dynamic conditions? (§6.3)
- What is the cost of using Iridescent? (§6.4)

Machine (Processor)	**Constant** (c) (cycles/op)	**Variable** (v) (cycles/op)	**Benefit** (v/c)
IceLake	175297	284944	61%
IvyBridge	250434	661295	246%
CoffeeLake	168817	581130	348%
AlderLake-p	173350	583724	336%
AlderLake-e	206924	557572	269%

Table 3. Impact of turning s as compile-time constant at runtime for N=64 for different hardwares (Table 1)

6.1 Experimental Setup

Testbed. For our experiments, unless otherwise stated, we configure two identical machines as client and server. They are directly connected with a pair of 100 Gbps Mellanox ConnectX-5 Ethernet adapters. Both machines have two Intel Xeon Gold 6152 processors at 2.1 GHz, each with 22 cores for a total of 44 cores and 187 GB of RAM per machine. We run Linux kernel 5.15 with Debian 11.

6.2 Compiler Optimizations at Runtime

Compile-time optimizations often produce more performant code. These compile time optimizations include constant propagation, loop unrolling, dead-code elimination, and use of vectorization instructions. With Iridescent, we can enable compile time optimizations at runtime.

Iridescent enables Cascading Compiler Optimizations. First, we show the potential benefits of enabling compile-time optimizations at runtime for the MMulBlockBench microbenchmark shown in Table 1. Specifically, for the workload $N = 64$, we compare the performance of keeping the optimal block size,B, as a runtime parameter to that of converting it into a constant at runtime using Iridescent. For both configurations, we execute the function for a fixed number of times and measure the amount of cycles spent for executing 1 execution of the function. Table 3 shows the benefit for converting B into a compile-time constant for different hardwares. For each hardware, we get at least 50% reduction in consumed cycles, and greater than 240% reduction in consumed cycles for four of the five operating conditions. This improvement is a direct impact of Iridescent's specialization allowing different compiler optimizations to cascade and

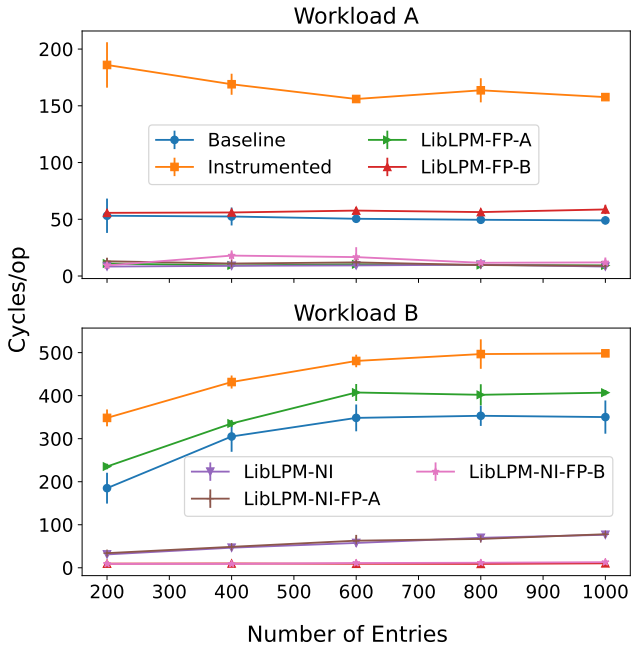

Figure 3. Average number of cycles required to perform one lookup for different number of entries in the lookup table for two different workloads.

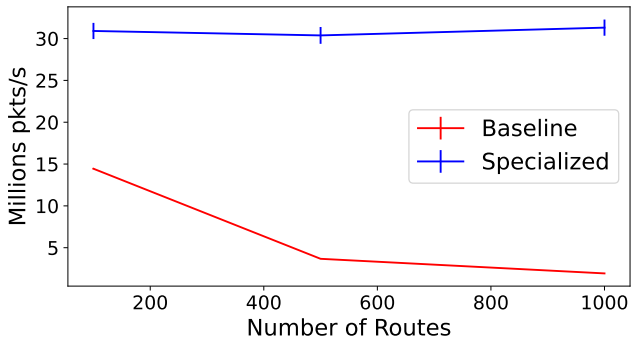

Figure 4. Change in overall throughput of the FastPath router with Iridescent-specialized LinearIPLookup element for different sizes of the overall throughput.

combine together to produce a more efficient version of the code. In this specific case, the specialization of *B* allows the JIT compiler to first easily unroll the loops in the matrix multiplication function and then replace multiple instructions with more optimized vector instructions.

Iridescent incorporates existing specializations. To showcase Iridescent's ability to easily incorporate existing runtime specializations, we implement the map hot-keys specialization of Morpheus [43] as a fast-path specialization point in packet processing function of FastClick's LinearIPLookup element. We configure the fast-path specialization point with

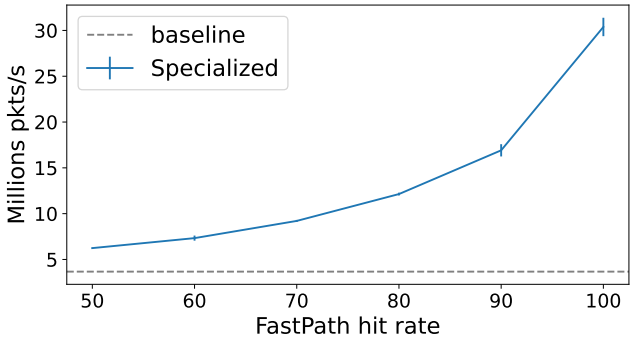

Figure 5. Change in overall throughput of the FastPath router with Iridescent-specialized LinearIPLookup element for different fast-path hit rates.

a 0.01% sampling rate for the instrumentation phase. We execute the FastPath router on 1 machine and treat that machine as the Device Under Test and execute Pktgen to execute an open-loop workload to generate packets with destination ip addresses from a given set of IP addresses. We repeat this experiment for different sizes of the routing table in the LinearIPLookup as well as different sizes of the fast-path to measure the throughput of the router under different conditions. Figure 4 shows that Iridescent achieves upto 15× improvement in the throughput based on the size of the table at 100% fast-path hit rate. The improvement increases with the increase in the table size as for larger tables, more linear scans are required. Figure 5 shows that for a router, with a routing table of size 1000, even with a 50% fast-path hit rate, Iridescent achieves approximately 5× improvement in the throughtput. The improvement increases with the increase in the fast-path hit rate.

Iridescent enables custom specializations. Next, we show the potential benefits of enabling custom compile-time optimizations for LibLPM. For this experiment, we set up eleven different configurations resulting from a combination of five different LPM table sizes and two different workloads - Workload A and Workload B. In Workload A, the incoming IP address is an IP address that matches a very specific prefix entry in the routing table. In Workload B, the incoming IP address is an IP address that only matches the LPM prefix entry of prefix length 0. Thus, Workload A and Workload B cover the best and worst cases respectively for the LPM lookup function. We execute these workloads on seven different specializations - (i) Baseline: no Iridescent-enabled specialization, (ii) Instrumented: Iridescent-enabled specialization which captures the most popular inputs (with a 10% sampling rate); information collected by this configuration is used by Iridescent for the fast path specializations, (iii) LibLPM-FP-A: fast-path specialization of size 1 specialized for Workload A, (iv) LibLPM-FP-B: fast-path specialization of size 1 specialized for Workload B, (v) LibLPM-NI: the

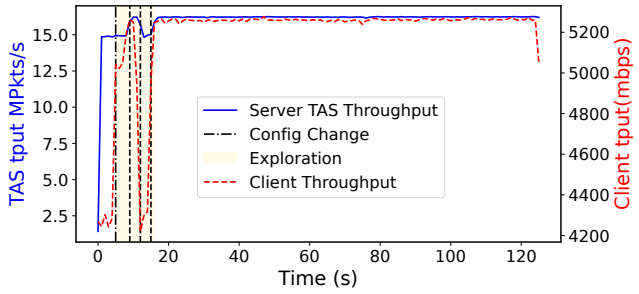

Figure 6. Automatic Exploration and Specialization of different configurations for TAS

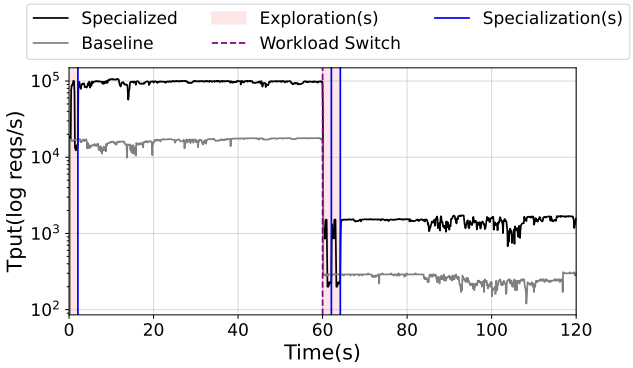

Figure 7. Automatic Exploration and Specialization of MMulBlockBench for two different workloads

nested-if code generation specialization, (vi) LibLPM-NI-FP-A: Combination of LibLPM-NI and LibLPM-FP-A, and (vii) LibLPM-NI-FP-B: Combination of LibLPM-NI and LibLPM-FP-B. Figure 10 shows the cycles required for completing one execution on average of the lpm lookup function for all the different combinations of configurations and specializations. For Workload A, all three of LibLPM-FP-A, LibLPM-NI, LibLPM-NI-FP-A specializations achieve upto 6× reduction in cycles per execution. However, unlike the LibLPM-FP and LibLPM-NI-FP-A specialization, this LibLPM-NI specialization does not require an additional instrumentation phase to sample executions to find the optimal candidates for the fast path. LibLPM-NI improves performance across the board for all workloads by upto 6×. Despite this performance boost, LibLPM-NI is not the best specialization for Workload B. For Workload B, the best optimizations are the *-FP-B specializations as they get upto a 30× performance boost.

6.3 Runtime Design Exploration

Iridescent enables runtime design exploration. We show runtime design exploration for TAS [35]. We setup our experiment with a typical server-client setting, with the server providing an echo service. The client is multi-threaded and executes an open-loop workload of 64 byte packets. Both the server and client are enabled with TAS. In our experiment, we modify the server-side TAS and measure the throughput of the server side TAS as millions of packets processed per second. With Iridescent we explore different values of the `rx_batch` in the TAS source code on the server side. Figure 6 shows how Iridescent can automatically select the best-configuration on the server side by automatically exploring the different values of the batch size for different locations in the source code.

Iridescent can automatically adapt to changing workloads. To show that Iridescent can automatically adapt to workloads, we first use the MMulBlockBench microbenchmark. We execute a Iridescent-enabled version of the function and a non-Iridescent enabled baseline version. In the non-baseline version, the block size s is a runtime parameter. In the Iridescent-enabled version, Iridescent explores different values of s by specializing the value in the code and converting s into a compile-time constant. We execute each version with two different workloads in succession, each lasting one minute. We measure the overall throughput (executions/s) for both configurations. Figure 7 shows how Iridescent can easily explore the different block sizes and find a performant configuration as compared to the baseline. Moreover, Iridescent can automatically detect the workload change based on the drop in throughput and restart the A/B testing process.

Next, we show runtime design exploration and adaptation for two different network functions from vigor [65]. We setup our experiment with a packet generator on one machine and a device under test (DUT) on the other machine. We use two ports for the DUT. The first port acts as the internal facing port where as the second port as the external facing port. The DUT runs executes the specific network function for every incoming packet generated by the packet generator. The network function implementation has different execution pathways for packets depending on whether they arrive on the internal port or the external port. The network function is enabled with the Iridescent specialization policy to trigger an exploration phase to find the best performing `BATCH_SIZE` whenever there is a change in the workload. We execute the experiment in two phases: in the first phase, packets arrive only on the external facing port and in the second phase, packets arrive only on the internal facing port. Figure 8 shows the results for the NAT and the Policer network functions. Iridescent can find the best performing configuration in each phase even if the best configuration is different.

Iridescent transparently enables exploration-based adaptation with custom specializations. We show runtime design exploration for FastClick LinearIPLookup. We execute the FastPath router on 1 machine and treat that machine as the Device Under Test and execute Pktgen to execute an open-loop workload to generate packets with destination ip

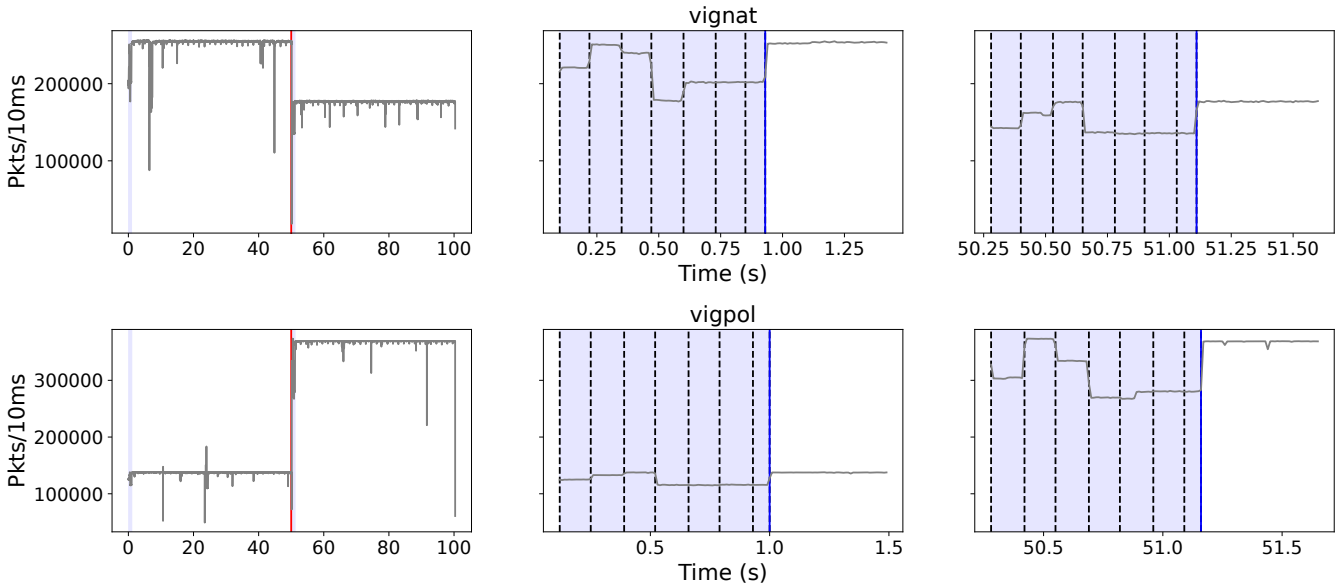

Figure 8. Batch Size exploration with Network Functions

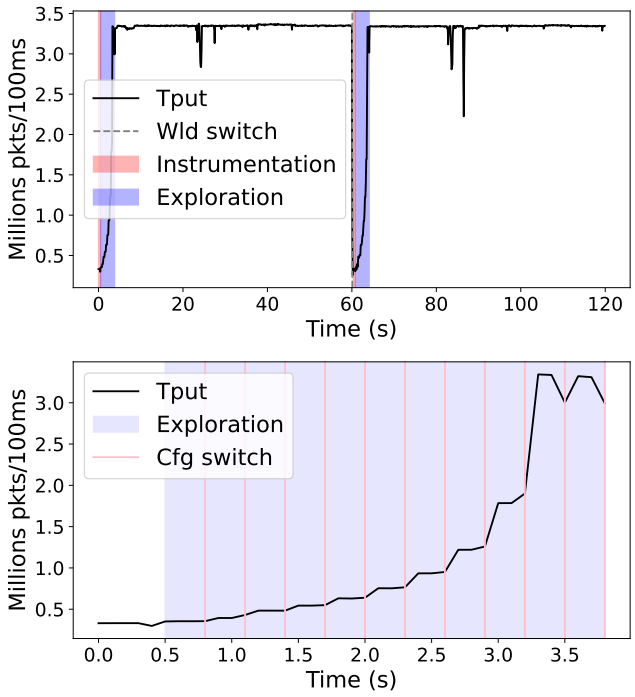

Figure 9. Optimal Fast Path size exploration with Iridescent

addresses from a given set of IP addresses. At the 1 minute-mark, we completely switch the destination IP address set with no overlap with the initial address. The router is configured with a Iridescent specialization policy that triggers an exploration whenever it detects a large change ($\geq$ 25%) in the measured throughput. Figure 9 shows how the throughput of the router changes for the different IP sets. At the start, the specialization policy triggers an exploration as it detects a large in-flux of packets signifying a change from 0 incoming packets. First, Iridescent run a quick instrumentation phase that runs for around 100ms to find the most popular incoming destination IP addresses. After the instrumentation phase, Iridescent switches to an exploration phase to explore the size of the fast-path i.e. how many addresses should be included in the generated if-else source code. Once the exploration finishes, Iridescent switches to the best performing fast-path size and generates the specialized code which persists until the workload switch at the 1-minute mark. Due to the workload change, the specialization policy kicks in automatically, and Iridescent restarts the instrumentation and exploration phase. As shown in Figure 9, Iridescent can quickly adapt to changes in workload to generate optimal specialized code under dynamic conditions.

6.4 Iridescent but at what Cost?

Compilation Cost. Table 4 shows the JIT compilation time for each of the target systems. Note that, this compilation happens off the critical path so its not a performance bottleneck. However, this cost does dictate the propagation delay, i.e., the time taken for the specialized version of the code to be available for the execution.

To analyze how the compilation time changes with the increase in the size of the JIT, we consider the code generated by Iridescent for the LibLPM-NI specialization. Figure 10 shows the change in the compilation time for the LibLPM-NI specialization as a function of the number of the elements in the lpm lookup table. The compilation time increases linearly with increase in the number of lpm entries. This is because

System	JIT Compilation Time (ms)
MMulBlockBench	10 ± 1
LibLPM-FP	72 ± 9
LibLPM-NI	Varies with lpm size
LibLPM-NI-FP	Varies with lpm size
Network Functions	98 ± 5
TAS	340 ± 5
FastClick	11 ± 1

Table 4. JIT compilation time

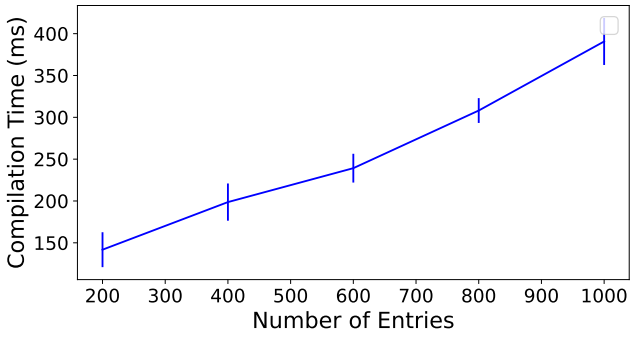

Figure 10. Compilation Time of LibLPM-NI (nested-if specialization) as a function of the number of elements in the LPM lookup table.

System	Handler Annotations (LoC)	Spec Policy (LoC)
MMulBlockBench	2	47
LibLPM	4	9
Network Functions	5	43
TAS	8	38
FastClick	5	88

Table 5. Lines of Code change required to integrate Iridescent

Iridescent generates one basic block in the specialized code for each lpm entry. As a result, the code size generated by Iridescent grows linearly with the total number of entries.

Developer Cost. Table 5 shows the lines of code changes required to integrate Iridescent specializations into the target systems. For all cases, the lines of code required are fewer than 100. Implementing the custom LPM specialization only required 162 lines of code for enabling the custom user-defined LPM specialization.

Instrumentation Overhead. To measure the overhead of adding instrumentation, we devise a microbenchmark called SimpleBench, consisting of two very simple functions, `f` and `g`. `f` computes the square of the input and `g` computes the

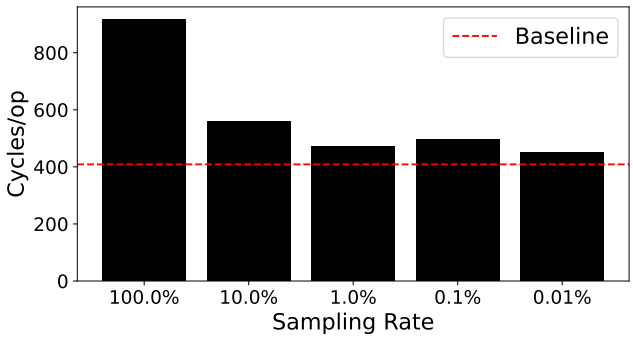

Figure 11. Impact of sampling rate on instrumentation overhead for LibLPM-FP configuration for Workload B

product of its two inputs. For `f`, we add a general specialization point for its input `a`. For `g`, we add a range-based specialization for its second input `b`. For both the specialization points `a` and `b`, we first execute their respective functions, `f` and `g`, in normal mode and then use Iridescent to add instrumentation. We execute each configuration one million times. As a baseline, each function on average takes upto 8 cycles per execution in normal mode. For `a`, which is a general specialization point, adding instrumentation adds an overhead of around 450 to 500 cycles per execution. For `b`, which is a specialization point designed for more efficient instrumentation for specialization points with values in fixed ranges, adding instrumentation adds an overhead of around 1 extra cycle per operation. When possible, Iridescent users should opt to use the more efficient specialization point. However, it is expected that this might not always be possible and in most cases users would be forced to use the general specialization point, Iridescent allows users to specify a sampling rate to avoid incurring the instrumentation cost for low-latency computations. Figure 11 shows how decreasing the sampling rate for the LibLPM-FP configuration when executed with Workload B can decrease the instrumentation overhead.

Specialization Guards and Failures. To measure the cost of specialization guards and failures, we use the SimpleBench microbenchmark. For function `f`, we use Iridescent to specialize the value of `a` and generate two different versions. In the first version, we disable specialization check at the entrypoint of `f`. We then execute this version of the function with the specialized value as input one million times. Each execution of this version takes 7 cycles. In the second version, we enable the specialization check at the entrypoint of `f`. If the specialization check fails, i.e. the input value of `a` does not match the specialized value of `a`, then this version of `f` throws an exception that is caught by Iridescent-inserted trampoline code which subsequently calls the generic version of the function. We first execute this version of the function with inputs such that the specialization guards always pass. This check adds an extra cycle per execution of

the function. Next, we execute this version of the function with inputs such that the specialization guards always fail. This results in exception handling behavior which adds approximately 5000 cycles of overhead per execution. Thus, the raw cost of a specialization check is 1 extra cycle whereas that of a specialization failure is approximately 5000 cycles.

7 Related Work

Runtime Specializations. Runtime Specialization techniques [14, 15, 17, 27] produce more efficient versions of the code by exploiting values and invariants that only exist at runtime. Today, these specializations manifest as optimizations either through binary rewriting [2] or as optimizations in JIT compilers for interpreted languages in the form of Type Specialization [12, 13, 45] for specializing the types of data for dynamically typed languages, or as Value Specialization for interpreted languages [16, 40] where the parameter values of hot functions are converted into constants. These specializations may be optionally applied depending on the computation context [23, 31, 48]. generic optimization and typically done automatically and transparently by the runtime based on internal cost models which may not be appropriate for specific application contexts. Moreover, these specializations are limited in scope and usually do not take the system's end-to-end performance into consideration when applying these specializations.

Transparent Dynamic Optimization (TDO) techniques such as Dynamo [4], The Transmeta Code Morphing Software [18], DynamoRIO [7], optimize code at runtime without requiring any modifications by capturing and optimizing traces (sequences of instructions) that are commonly executed. These optimization techniques are broadly applicable optimization techniques that target a broad spectrum. As a result, do not typically leverage any domain-specific, situation-specific optimizations as these techniques do not have a way of obtaining that knowledge.

Domain-Specific Specializations. In recent times, value specializations have extended beyond interpreted languages for specific use cases such as value specializations for GPU kernels [24], or through increment specializations for packet processing frameworks [22, 43, 50]. Iridescent provides a general framework for configuring, applying, exploring, and selecting such specializations at runtime.

Feedback-Driven Optimizations (FDO). FDO [11, 18, 28, 45, 47, 52, 58], also known as PGO (Profile-Guided Optimizations), are typically multi-run optimizations consisting of multiple executions of the program to be optimized. First the optimization collects relevant metrics from a benchmarking run and then uses these metrics to generate optimized versions of the code. FDO techniques in their current form lack flexibility as they are specialized to work well for only one workload (the workload it was trained on). FDO optimizations typically do not adapt well to change in workload or environment settings.

AutoTuning. AutoTuning allows developers to automatically find the best values of various configuration parameters at runtime without requiring the developers to manually try all possible combinations. Similar to FDO techniques, AutoTuning techniques [57, 60] often perform tuning through trial runs to find optimal values for different parameters. Some AutoTuning techniques such as OPPerTune [55] and SoftSKU [56] apply the AutoTuning paradigm of measurement-driven search at runtime without any trials. However, these techniques are limited to environmental parameters and knobs as they work transparently with respect to the deployed application. Iridescent enables the AutoTuning paradigm of measurement-driven search at runtime to find the best combination of specializations for the system.

8 Discussion

Debugging. Iridescent makes it harder for developers to issues. This is because the generated specialized code is often very different to the code that was originally written by the developer which makes it harder for the developers to fully understand the execution sequences. This is further exacerbated by the fact that different workloads may lead to different specializations making it difficult to reproduce issues seen in production. Combining Iridescent with monitoring and distributed tracing techniques could offer developers a solution to their debugging problems.

Profile-Guided Optimizations at runtime. Iridescent enables developers to utilize traditional PGO techniques at runtime without requiring separate benchmarking runs. For example, prefetch injection optimizations such as APT-GET [32] and RPG2 [66] could be integrated with Iridescent in the future.

Leveraging scale & ML for exploration. Several ML-based techniques exist for tuning configurations [1, 10, 21, 32, 39, 41, 42, 50, 54, 55, 59, 60]. OPPerTune [55] provides access to a plethora of ML techniques toolbox for automatically searching through large configuration spaces. We believe Iridescent can be integrated with OPPerTune to utilize state-of-the-art exploration techniques. Moreover, Iridescent could further incorporate SmartChoices [8] to allow different ML techniques to apply to specific specialization points.

9 Conclusions

In this work, we presented Iridescent, a framework enabling online workload-driven runtime specialization of systems to improve performance. Iridescent provides a toolkit for developers to implement system-specific specializations that utilize runtime data and invariants to generate more performant systems. Iridescent provides the necessary tooling for developers to configure automatic runtime exploration of the

space of potential specializations to find the best performing configuration at each instant. By combining developer insight with automated compiler optimizations, Iridescent enables systems to adapt efficiently to changing workloads and environments with minimal manual effort to improve performance.

Acknowledgments

We thank Yuchen Qian for his contributions to an early prototype Iridescent implementation.

References

[1] Ibrahim Umit Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. 2023. Improving storage systems using machine learning. *ACM Transactions on Storage* 19, 1 (2023), 1–30.

[2] AP Arif Ali and Erven Rohou. 2017. Dynamic function specialization. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 163–170.

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 469–482.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 1–12.

[5] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 5–16.

[6] Sapan Bhatia, Charles Consel, A-F Le Meur, and Calton Pu. 2004. Automatic specialization of protocol stacks in operating system kernels. In *29th Annual IEEE International Conference on Local Computer Networks*. IEEE, 152–159.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 265–275.

[8] Victor Carbune, Thierry Coppey, Alexander Daryin, Thomas Deselaers, Nikhil Sarda, and Jay Yagnik. 2018. SmartChoices: hybridizing programming and machine learning. *arXiv preprint arXiv:1810.00619* (2018).

[9] Steve Carr and Ken Kennedy. 1989. Blocking Linear Algebra Codes for Memory Hierarchies.. In *PPSC*. Citeseer, 400–405.

[10] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. 2021. Cgptuner: a contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1401–1413.

[11] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 12–23.

[12] Maxime Chevalier-Boisvert and Marc Feeley. 2015. Interprocedural type specialization of javascript programs without type analysis. *arXiv preprint arXiv:1511.02956* (2015).

[13] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. 2010. Optimizing MATLAB through just-in-time specialization. In *International Conference on Compiler Construction*. Springer, 46–65.

[14] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. 2005. A uniform approach for compile-time and run-time specialization. In *Partial Evaluation: International Seminar Dagstuhl Castle, Germany, February 12–16, 1996 Selected Papers*. Springer, 54–72.

[15] Charles Consel and François Noël. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 145–156.

[16] Igor Rafael de Assis Costa, Henrique Nazaré Santos, Péricles Rafael Alves, and Fernando Magno Quintao Pereira. 2014. Just-in-time value specialization. *Computer Languages, Systems & Structures* 40, 2 (2014), 37–52.

[17] Jeffrey Dean, Craig Chambers, and David Grove. 1995. Selective specialization for object-oriented languages. *ACM SIGPLAN Notices* 30, 6 (1995), 93–102.

[18] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta Code Morphing/spl trade/Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 15–24.

[19] Bangwen Deng, Wenfei Wu, and Linhai Song. 2020. Redundant logic elimination in network functions. In *Proceedings of the Symposium on SDN Research*. 34–40.

[20] DPDK Project. 2022. Data Plane Development Kit. http://www.dpdk.org/. Retrieved Feb 2, 2022.

[21] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[22] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2021. PacketMill: toward per-Core 100-Gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1–17.

[23] Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Ječmen, Jakob Hain, and Jan Vitek. 2020. Contextual dispatch for function specialization. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–24.

[24] Giorgis Georgakoudis, Konstantinos Parasyris, and David Beckingsale. 2025. Proteus: Portable Runtime Optimization of GPU Kernel Execution with Just-in-Time Compilation. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 507–522.

[25] Hamid Ghasemirahni, Tom Barbette, Georgios P Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q Maguire Jr, and Dejan Kostić. 2022. Packet order matters! improving application performance by deliberately delaying packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 807–827.

[26] Hamid Ghasemirahni, Alireza Farshin, Dejan Kostic, and Marco Chiesa. 2024. Just-in-Time Packet State Prefetching. *arXiv preprint arXiv:2407.04344* (2024).

[27] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J Eggers. 1999. An evaluation of staged run-time optimizations in DyC. *ACM SIGPLAN Notices* 34, 5 (1999), 293–304.

[28] Urs Hölzle and David Ungar. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. 326–336.

[29] Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance Interface eXtractor (PIX) artifact. Accessed November 2024 from https://github.com/dslab-epfl/pix.

[30] Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance interfaces for network functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 567–584.

[31] Era Jain and Subhajit Roy. 2016. Phase directed compiler optimizations. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 270–279.

[32] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 747–764.

[33] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA) *(NSDI)*.

[34] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. 2023. {SelfTune}: Tuning Cluster Managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1097–1114.

[35] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *14th ACM European Conference on Computer Systems* (Dresden, Germany) *(EuroSys)*.

[36] Eddie Kohler, Robert Morris, and Benjie Chen. 2002. Programming language optimizations for modular router configurations. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 251–263. doi:10.1145/605397.605424

[37] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Transactions on Computer Systems* 18, 3 (Aug. 2000), 263–297.

[38] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *9th International Symposium on Code Generation and Optimization* (Palo Alto, CA) *(CGO)*.

[39] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. 2018. Metis: Robustly tuning tail latencies of cloud systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 981–992.

[40] Caio Lima, Junio Cezar, Guilherme Vieira Leobas, Erven Rohou, and Fernando Magno Quintão Pereira. 2020. Guided just-in-time specialization. *Science of Computer Programming* 185 (2020), 102318.

[41] Chen Lin, Junqing Zhuang, Jiadong Feng, Hui Li, Xuanhe Zhou, and Guoliang Li. 2022. Adaptive code learning for spark configuration tuning. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1995–2007.

[42] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2020. {OPTIMUSCLOUD}: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 189–203.

[43] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2022. Domain specific run time optimization for software data planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1148–1164.

[44] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. 2016. Dataplane specialization for high-performance OpenFlow software switching. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 539–552.

[45] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 151–165.

[46] Heng Pan, Peng He, Zhenyu Li, Pan Zhang, Junjie Wan, Yuhao Zhou, XiongChun Duan, Yu Zhang, and Gaogang Xie. 2024. Hoda: a High-performance Open vSwitch Dataplane with Multiple Specialized Data Paths. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 82–98.

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

[48] Gabriel Poesia and Fernando Magno Quintão Pereira. 2020. Dynamic dispatch of context-sensitive optimizations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[49] Mindaugas Rasiuckevicius. 2022. Longest Prefix Match (LPM) library. Accessed June 2025 from https://github.com/rmind/liblpm.

[50] Fabian Ruffy, Zhanghan Wang, Gianni Antichi, Aurojit Panda, and Anirudh Sivaraman. 2024. Incremental Specialization of Network Programs. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*. 264–272.

[51] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[52] Marija Selakovic, Thomas Glaser, and Michael Pradel. 2017. An actionable performance profiler for optimizing the order of evaluations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 170–180.

[53] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 329–339.

[54] Gagan Somashekar, Amoghavarsha Suresh, Saurabh Tyagi, Vikas Dhyani, K Donkada, Anurag Pradhan, and Anshul Gandhi. 2022. Reducing the tail latency of microservices applications via optimal configuration tuning. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 111–120.

[55] Gagan Somashekar, Karan Tandon, Anush Kini, Chieh-Chun Chang, Petr Husak, Ranjita Bhagwan, Mayukh Das, Anshul Gandhi, and Nagarajan Natarajan. 2024. {OPPerTune}:{Post-Deployment} Configuration Tuning of Services Made Easy. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1101–1120.

[56] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. 513–526.

[57] Akshitha Sriraman and Thomas F Wenisch. 2018. {μTune}:{Auto-Tuned} Threading for {OLDI} Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 177–194.

[58] Mark Stephenson and Ram Rangan. 2021. PGZ: automatic zero-value code specialization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 36–46.

[59] Adith Swaminathan, Akshay Krishnamurthy, Alekh Agarwal, Miro Dudik, John Langford, Damien Jose, and Imed Zitouni. 2017. Off-policy evaluation for slate recommendation. *Advances in Neural Information Processing Systems* 30 (2017).

[60] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[61] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. 2020. P2GO: P4 profile-guided optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 146–152.

[62] Michael Wolfe. 1987. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. 357–361.

[63] Michael Wolfe. 1989. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. 655–664.

[64] Shaofeng Wu, Qiang Su, Zhixiong Niu, and Hong Xu. 2024. Tomur: Traffic-Aware Performance Prediction of On-NIC Network Functions with Multi-Resource Contention. *arXiv preprint arXiv:2405.05529* (2024).

[65] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 275–290.

[66] Yuxuan Zhang, Nathan Sobotka, Soyoon Park, Saba Jamilan, Tanvir Ahmed Khan, Baris Kasikci, Gilles A Pokam, Heiner Litz, and Joseph Devietti. 2024. Rpg2: Robust profile-guided runtime prefetch generation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 999–1013.