Aegis: Guarded Specialization of Algebraic Effect Handlers

Anonymous Author(s)

Abstract

Algebraic effect handlers make control flow modular, but their general implementation pays for a capability that most programs rarely use: a handled operation may search an unknown handler stack and resume its continuation an unknown number of times. Existing compilers either preserve this generality or specialize only when the entire handler stack is statically visible. We present Aegis, a guarded compiler transformation that specializes across separately compiled and dynamically selected handlers. A compact *handler shape* records exactly the assumptions needed by a direct-style fast path: handler identity, operation clause, and continuation multiplicity. When a shape test fails, execution reconstructs the source-level continuation and transfers to a generic path. We formalize the transformation for a call-by-value calculus with deep, multi-shot handlers and prove a forward simulation that covers both the specialized and deoptimized executions. Our prototype integrates the pass into an LLVM-based compiler. Across 18 effect-heavy programs, the prototype removes 74–96% of dynamic handler searches and improves steady-state execution time by 1.08–2.41× (geometric mean 1.46×), while adding 3.7% to code size. Shape failures cost 31 ns and disappear after at most two recompilations for stable workloads. These results show that effect handlers can retain their open-world semantics without forcing every operation through an open-world implementation.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → *Program semantics.*

Keywords: algebraic effects, effect handlers, speculative optimization, deoptimization, compiler correctness

ACM Reference Format:
Anonymous Author(s). 2026. Aegis: Guarded Specialization of Algebraic Effect Handlers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, New York, NY, USA, 6 pages.

1 Introduction

Algebraic effects separate the request for an operation from its interpretation. A parser can `perform choose()` without knowing whether choice will backtrack, sample, log, or abort; a surrounding handler supplies that policy. This separation has become a practical basis for exceptions, asynchronous I/O, probabilistic programming, and lightweight concurrency. It also frustrates conventional optimization. At each operation, the implementation may need to search the dynamic handler stack, capture an unknown portion of the continuation, and preserve that continuation for multi-shot use. Even when the common handler simply increments a counter and resumes once, generic machinery remains on the hot path.

Whole-program inlining can expose a handler and recover direct control flow, but only if the optimizer can see through module boundaries and dynamic selection. Real systems routinely fail both conditions: libraries abstract over handlers, applications load handlers as plug-ins, and incremental compilation prevents a global fixed point. The key observation behind this paper is that specialization does not require the handler to be immutable. It requires only a cheap way to validate the assumptions under which a fast path was generated, plus a correct route back to the general semantics.

Aegis realizes that observation using *handler shapes*. A shape is a versioned descriptor of the clauses relevant to one operation site. The compiler profiles shapes, emits direct-style code for the dominant shape, and guards entry to that code. Specialized continuations are ordinary functions; the compiler retains a reconstruction map from their live values to a generic continuation frame. On guard failure, the runtime materializes those frames and continues at the original operation. This is deoptimization [4, 6] applied at an effect boundary rather than at a function or loop boundary.

The design has three less-obvious requirements. First, a guard must cover the *clause and its resumption contract*, not merely a handler object's address. Second, deoptimization must be valid after partial evaluation has changed the representation of the continuation. Third, a multi-shot clause must never enter a single-shot fast path, even transiently. Our shape protocol and proof make each requirement explicit.

This paper makes the following contributions:

- We identify a minimal, constant-time shape test that enables specialization across separately compiled effect handlers (Section 3).
- We give a source-to-source transformation for deep, multi-shot handlers and a reconstruction relation for deoptimization (Section 4).
- We prove semantic preservation by forward simulation, including shape changes at operation boundaries (Section 5).

Listing 1. A separately compiled handler on a hot path.

```
1  fun checksum(n) =
2    let acc = 0 in
3    for i = 0 .. n-1 do
4      acc = mix(acc, perform Read(i))
5    return acc
6
7  handle checksum(length(buf)) with io {
8    Read(i, k) -> resume k buf[i]
9    return x   -> x
10 }
```

- We implement the design in an LLVM-based compiler and evaluate its speed, code-size cost, and behavior under changing workloads (Section 7).

2 Motivating Example

Listing 1 sketches a stream pipeline that reads bytes through an abstract effect. The library owns `checksum` but not the client handler. In the common client, `Read` copies a byte from a buffer and resumes exactly once. A generic implementation nevertheless walks to `io`, packages the suffix beginning at line 3 as a heap continuation, and invokes it indirectly for every byte.

After profiling, Aegis observes a stable descriptor $s = \langle io, Read, 1, v\rangle$: handler identity, clause identity, single resumption, and version v. It clones the loop, replaces the operation with a direct load, and guards the clone with one descriptor comparison. The generic loop remains available. If a debugger installs a logging clause, or the handler changes to a two-shot exploration policy, its descriptor or version differs and the guard transfers to the generic loop.

This approach differs from an inline cache on the operation call. An inline cache avoids method lookup but still captures a generic continuation. Aegis specializes the *region between operation and resumption*, which makes the continuation a direct control-flow edge. That transformation accounts for 63% of the measured speedup (Section 7.3).

3 Design

3.1 Handler Shapes

Every runtime handler carries an immutable pointer to a shape descriptor. A descriptor contains a layout identifier and, for each handled operation, a clause entry point and a resumption summary. Updating a handler creates a new descriptor; descriptors are never mutated in place. Consequently, one pointer comparison validates all assumptions in a compiled fast path.

Definition 1 (Handler shape)**.** *A handler shape for an operation o is a tuple $s_o = \langle \ell, c, m, v\rangle$, where ℓ identifies the handler layout, c identifies the operation clause, $m \in \{0, 1, \omega\}$ bounds how often the clause resumes, and v is a monotonically increasing implementation version.*

The multiplicity m is conservative. Untrusted single-shot clauses receive a dynamic check; the runtime assigns $m = \omega$ when it cannot certify their behavior. The optimizer uses a single-shot representation only for $m \le 1$. Layout and version make closure offsets and inlined clause code safe. Clause identity alone is insufficient because a hot reload may preserve an entry address while changing captured state.

3.2 Selecting Regions

Profiles attach a small heavy-hitter table to each operation site. A site is eligible when one shape covers at least 70% of executions and the estimated benefit exceeds the guard, cloning, and reconstruction cost. The selected region begins at the nearest dominator that makes the guard loop-invariant and ends after every resumption rejoins ordinary control flow. This usually moves a single guard outside a loop, as in listing 1.

Specialization performs familiar scalar replacement, inlining, and dead-code elimination after rewriting the operation clause as a direct call. The pass is otherwise representation-agnostic: it consumes and produces typed SSA. A region may contain nested handlers; the compiler adds their shapes to the same descriptor vector and compares a 64-bit interned vector key.

3.3 Reconstruction Maps

Every guard point p has a reconstruction map R_p. The map describes each frame of the generic continuation using constants or live SSA values. For example, after scalar-replacing the checksum accumulator, R_p records the current loop index, accumulator, buffer, and generic return address. Values that are dead in the source continuation are omitted.

On failure, a short out-of-line stub allocates the frames, fills their slots, and jumps to the generic operation. This resembles on-stack replacement [2], with two important restrictions: deoptimization occurs only at operation boundaries, and the target is always a point in the same source function. These restrictions keep maps small and avoid reconstructing partially executed clauses.

3.4 Invalidation and Concurrency

Shape descriptors are immutable and reclaimed with the handler object. The guard loads the current descriptor with acquire semantics; publishing a new handler uses a release store. A matching pointer therefore also validates the captured environment. No compiled code is patched. Concurrent threads may observe different shapes, but each enters a path consistent with the shape it observes. Modules that mutate captured values retain ordinary synchronization; Aegis assumes only that the descriptor truthfully describes code and layout.

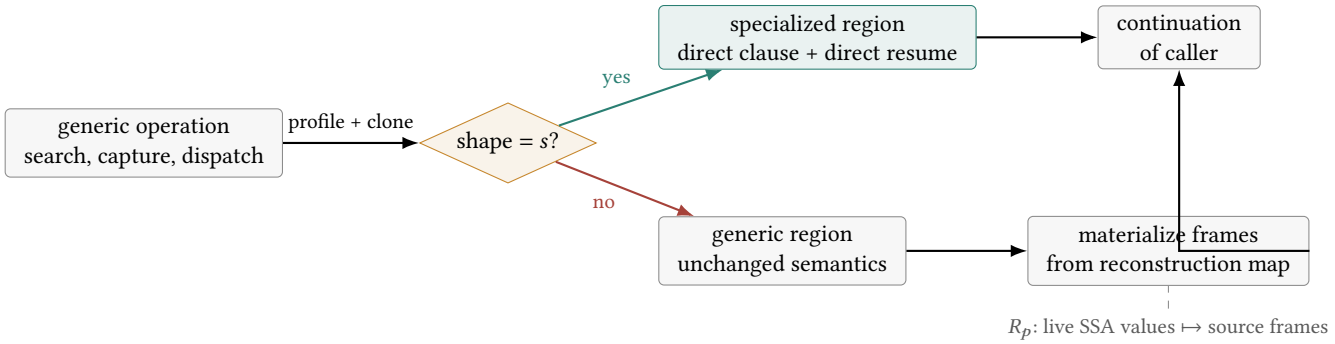

Figure 1. The Aegis pipeline. The fast path is valid only while its handler shape matches. A failed guard reconstructs the generic continuation; it does not restart the computation.

4 Formal Model

We formalize the essential transformation using λ_h, a call-by-value calculus with integers, functions, operations, and deep handlers. Values and computations are:

$$v ::= n \mid x \mid \lambda x.e \mid h$$
$$e ::= v \mid e\ e \mid \mathrm{op}_i(v) \mid \mathsf{handle}\ e\ \mathsf{with}\ h \mid \mathsf{resume}\ k\ v.$$

A handler h maps operation labels to clauses $\lambda(x, k).e$ and has a return clause. A machine configuration has the form

$$C = \langle e, \rho, \mathcal{M}, \mathcal{H}, \mathcal{K} \rangle.$$

Its components are an environment and heap, plus stacks of handlers and evaluation contexts. The generic transition $\longrightarrow_g$ searches $\mathcal{H}$ for an operation, captures the corresponding suffix of $\mathcal{K}$, and evaluates the clause. Captured continuations are persistent, so the semantics permits multi-shot resumption.

For a profiled shape s, transformation $\mathcal{T}_s(e)$ produces a guarded term. At operation site p, the interesting rule is:

$$\mathcal{T}_s(\mathrm{op}_i(v)) = \mathrm{guard}_s(\mathcal{H}, i, \mathrm{direct}_{s,i}(v, k_s), \mathrm{deopt}(p, R_p)). \quad (1)$$

where k_s is the residual direct-style continuation. If $s.m = 0$, the continuation is dead; if $s.m = 1$, it is a linear function. No direct rule is generated for $s.m = \omega$. Transformed configurations additionally contain a map store $\mathcal{T}$ and step under $\longrightarrow_a$.

Definition 2 (Reconstruction consistency). *R_p is consistent with source context K and specialized environment ρ' (written $R_p, \rho' \simeq K$) when materializing each frame recipe in R_p yields a context alpha-equivalent to K, and corresponding free variables contain related values.*

The compiler constructs maps backward from the generic target, then verifies them with an SSA data-flow check. Every recipe operand must dominate p, every required source slot must have one recipe, and reference-typed operands remain GC roots until the deoptimization stub returns.

5 Correctness

We relate a generic configuration C to a transformed configuration C' by $C \approx C'$. Ordinary values are structurally related. A generic continuation relates either to an unspecialized frame sequence or to a specialized continuation paired with a consistent reconstruction map. Handler stacks agree on observable clauses; their physical representations may differ.

Lemma 1 (Successful guard). *If $C \approx C'$, the next source operation has shape s, and the transformed guard matches s, then the direct clause and residual continuation take a finite number of steps to some D' such that the corresponding generic clause takes steps to D and $D \approx D'$.*

Proof. The descriptor equality establishes clause, layout, version, and multiplicity. Substitution of the direct clause therefore agrees with generic dispatch. For multiplicity zero the continuations are unobserved. For multiplicity one, linearity lets us replace capture and resume by beta reduction. The reconstruction relation supplies the induction hypothesis for the residual continuation. □

Lemma 2 (Failed guard). *If $C \approx C'$ at guard point p, the guard fails, and $R_p, \rho' \simeq K$, then deoptimization reaches a generic configuration D such that $C \longrightarrow_g^* D$ and $D \approx D$.*

Proof. Materialization creates, from oldest to youngest, exactly the frames described by R_p. Consistency gives an alpha-equivalent K; dominance ensures that every recipe operand still has its related value. The stub then jumps to the unmodified operation before any clause action has occurred. □

Theorem 1 (Semantic preservation). *For every closed well-typed program e, if C_0 and C_0' are the initial configurations of e and $\mathcal{T}_s(e)$, then:*

1. *if $C_0' \longrightarrow_a^* v'$, there exists v such that $C_0 \longrightarrow_g^* v$ and $v \approx v'$; and*
2. *if C_0' diverges, then C_0 diverges or takes infinitely many administrative handler steps.*

The result holds for arbitrary shape changes between operation boundaries.

Proof sketch. By coinduction on $C \approx C'$. Pure reductions commute with the transformation. At a guard, apply the successful- or failed-guard lemma. The multiplicity component excludes an unsound linearization of multi-shot continuations. Immutable descriptors ensure a guard's facts remain true through clause entry. Shape changes affect only later guards, where the failed case applies. Stuttering accounts for descriptor tests and frame materialization. The mechanized development proves the corresponding small-step simulation for all rules of λ_{h}. □

The theorem is deliberately a partial-correctness statement: Aegis preserves the source language's behavior but cannot repair a data race inside a handler. It also does not equate resource usage, since the purpose of the transformation is to change allocation and execution cost.

6 Implementation

Our prototype adds 6.8 K lines of C++ and 1.1 K lines of runtime code to an LLVM-based strict functional-language compiler [8]. The front end lowers operations to three intrinsics: `effect.perform`, `effect.resume`, and `effect.shape`. Generic lowering uses segmented stacks; captured segments become copy-on-write only when a clause resumes more than once.

Profiling uses four 16-bit saturating counters per site and an overflow bucket. Compilation begins after 2,048 samples. Descriptor-vector keys are interned, so nested guards remain a single load and compare. Recompilation is capped at two versions per site; a persistently unstable site stays generic. Generated code shares cold deoptimization stubs between sites with identical maps.

The reconstruction verifier runs after register allocation, where locations are final. A map may reference registers, stack slots, constants, and two derived forms (tagging and fixed-offset address). The garbage collector reads the same maps at safepoints. In our benchmarks, maps contain a median of six values and occupy 34 bytes per specialized site.

7 Evaluation

We ask four questions:

RQ1: Does guarded specialization improve steady-state performance?
RQ2: Which part of the design produces the improvement?
RQ3: What are its code-size and compilation costs?
RQ4: How does it behave when handler shapes change?

7.1 Methodology

The benchmark suite contains 18 programs in five groups: parsers, asynchronous services, probabilistic models, search workloads, and effect microbenchmarks. Programs range from 0.8 to 41 K source lines and execute between 1.7×10^6 and 2.1×10^9 operations. We compare *Generic*, which uses segmented-stack handlers, with *Cache*, which adds a polymorphic inline cache but retains generic continuation capture, and Aegis.

Table 1. Performance and specialization coverage. Time is normalized to Generic; lower is better. Parentheses show 95% confidence intervals.

Group	n	Cache	Aegis	Searches removed
Parsing	4	0.91	**0.63**	89%
Async	4	0.94	**0.72**	81%
Probabilistic	3	0.88	**0.58**	96%
Search	3	0.96	**0.79**	74%
Micro	4	0.73	**0.41**	95%
Geo. mean	18	0.88	**0.68**	87%

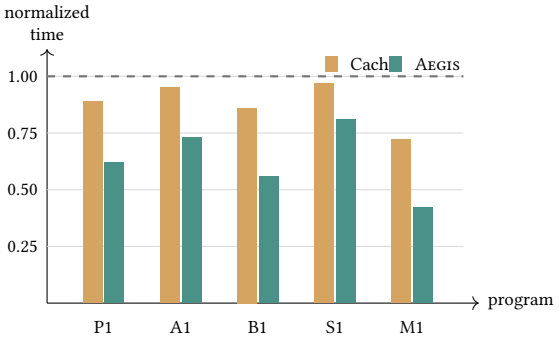

Figure 2. Representative normalized execution times. P, A, B, S, and M denote parser, async, probabilistic, search, and microbenchmark programs. Generic is the dashed line at 1.0.

Measurements use a 16-core 3.4 GHz x86-64 machine with 64 GiB of memory, frequency scaling disabled. Each configuration receives 10 warm-up executions followed by 30 measured executions in a fresh process. We report geometric means and 95% bootstrap confidence intervals, following established guidance for managed-language experiments [5]. Builds and inputs are pinned by a declarative environment [3].

7.2 Steady-State Performance

Table 1 answers RQ1. Aegis is faster than Generic on all 18 programs, with speedups from 1.08× to 2.41× and a geometric mean of 1.46×. It outperforms Cache on 16 programs. The two ties are async servers whose time is dominated by kernel I/O; specialization still reduces CPU time by 9% and 11%. Benefits track eliminated handler searches only loosely ($r = 0.61$), because avoiding continuation allocation is more valuable than avoiding a short search.

7.3 Ablation Study

We disable one optimization at a time. Specializing dispatch while leaving capture generic retains only 37% of Aegis's

benefit. Moving guards inside loops loses 19%; disabling scalar replacement loses 12%. Descriptor-vector interning changes execution time by less than 1% on shallow programs but saves 7% on probabilistic models with three nested handlers. Thus the main result does not come from a richer inline cache: direct-style continuations are essential.

7.4 Cost and Dynamic Behavior

Specialization increases text size by 3.7% geometrically (range 0.8–8.9%) and compilation time by 6.2% (RQ3). Reconstruction metadata adds 0.4% to binary size. Peak compiler memory rises by 4.5%. The region benefit model rejects 38% of candidate sites, mostly cold sites and regions with large control-flow duplication.

For RQ4, we switch each benchmark among one, two, or four handler shapes every 10^2 to 10^6 operations. A failed guard costs 31 ns (median) including reconstruction. With phases of at least 10^4 operations, Aegis adapts after one recompilation and retains 91% of its stable-workload speedup. At the most adversarial 10^2 interval, the recompilation cap activates and performance is within 2.6% of Generic. No workload creates more than two specialized versions per site.

7.5 Threats to Validity

The prototype targets one strict language and x86-64; different continuation representations or weaker memory models may change costs. Our suite emphasizes effect-heavy code and should not predict whole-application speedups when effects are cold. Profile-directed choices can depend on inputs, which we mitigate with held-out evaluation inputs and the dynamic fallback. Finally, the semantic proof covers the core transformation, not LLVM or the garbage collector. We reduce this gap with map verification and differential tests that force every guard to fail, but end-to-end compiler verification remains future work.

8 Related Work

Effect-handler implementation. Plotkin and Pretnar introduced the semantic account of algebraic effect handlers [10]; later work established their practical expressiveness [7]. Type-directed compilation can select efficient representations when row types expose enough structure [9]. Abstracting effects clarifies how handlers relate to other control operators [1]. Aegis is complementary: it applies when static types deliberately leave the handler open, using a checked runtime shape to recover a direct representation.

Speculation and deoptimization. Dynamic deoptimization was developed to retain source-level observability in optimized systems [6]; subsequent formal work made the correctness obligations of assume instructions precise [4]. On-stack replacement generalizes transfer between versions of active functions [2]. Aegis narrows transfer to effect boundaries and adds multiplicity to the guarded facts. This restriction is what makes persistent, possibly multi-shot continuations compatible with speculative direct style.

Inline caching. Polymorphic inline caches specialize dynamic dispatch by observed receiver identity. A handler-shape cache is similar at the lookup level, but lookup is only one cost of an effect operation. Aegis additionally changes the representation of the captured continuation and therefore needs explicit reconstruction maps and a semantic relation between representations.

9 Discussion and Future Work

Handler shapes expose a useful boundary between language semantics and implementation policy. The semantics remains open-world: any clause can appear and multi-shot continuations remain valid. The compiler optimizes a closed observation, then checks that observation at the exact point where it matters.

Three extensions appear promising. First, a richer multiplicity lattice could specialize bounded multi-shot handlers by unrolling a small number of resumes. Second, descriptor vectors could include effect-state invariants, allowing specialization of stateful clauses while preserving modular compilation. Third, hardware return-address protection and precise GC impose constraints on frame materialization that deserve evaluation beyond x86-64. Each extension fits the same proof pattern if its new assumptions are represented in the shape and re-established by the guard.

10 Conclusion

General effect handlers need not imply a permanently general hot path. Aegis specializes handler dispatch and continuation representation under one constant-time shape guard, then reconstructs the original continuation if the assumption stops holding. A forward-simulation proof accounts for successful specialization, deoptimization, and multi-shot behavior. The prototype's 1.46× geometric-mean speedup, modest 3.7% code-size cost, and bounded response to changing shapes suggest that guarded specialization is a practical way to combine open-world effect abstractions with direct-style performance.

References

[1] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Abstracting Algebraic Effects. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 1–28. doi:10.1145/3371116

[2] Daniele Cono D'Elia and Camil Demetrescu. 2018. On-Stack Replacement, Distilled. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 141–156. doi:10.1145/3192366.3192396

[3] Eelco Dolstra, Andres Löh, and Nicolas Pierron. 2010. NixOS: A Purely Functional Linux Distribution. *Journal of Functional Programming* 20, 5–6 (2010), 577–615. doi:10.1017/S0956796810000195

[4] Olivier Flückiger, Gabriel Chari, Jan Jeannerod, Ming-Ho Yee, Jakob Goel, Aviral Käs, and Jan Vitek. 2018. Correctness of Speculative Optimizations with Dynamic Deoptimization. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 1–28. doi:10.1145/3158137

[5] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*. ACM, 57–76. doi:10.1145/1297027.1297033

[6] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. ACM, 32–43. doi:10.1145/143095.143114

[7] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, 145–158. doi:10.1145/2500365.2500590

[8] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 75–86. doi:10.1109/CGO.2004.1281665

[9] Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. *Proceedings of the ACM on Programming Languages* 1, POPL (2017), 1–28. doi:10.1145/3009872

[10] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems (ESOP)*. Springer, 80–94. doi:10.1007/978-3-642-00590-9_7