From Theory to Practice: How Partial Evaluation Found Its Niche

Xavier Adettu[a,1], Nayan Banik[a,2], Tian Zhao[a,3,*]

[a] *Department of Computer Science, University of Wisconsin–Milwaukee, Milwaukee, WI, USA*

Abstract

Partial evaluation (PE) is a classical program optimization technique. Its underlying principle is embodied in ubiquitous compiler transformations, but PE as a general-purpose methodology has not achieved broad adoption. This survey argues that PE succeeds not as an unconstrained, general-purpose optimization, but in domains that provide explicit structure for specialization. We identify four recurring structural patterns underlying successful applications: explicit staging, effect discipline, suitable intermediate representations, and bounded specialization policies. Using this perspective, we survey practical PE across multiple domains and outline emerging opportunities in machine learning compilation. We conclude with open research challenges for future PE systems.

Keywords: PE, Program specialization, Staging, DSL compilation, JIT specialization

1. Introduction

Partial evaluation (PE) is one of the foundational ideas in program specialization. By specializing a program with respect to *static* information (inputs known before execution) while leaving *dynamic* information (inputs known only at execution time) unresolved, PE produces a *residual program* – a specialized version of the original that eliminates interpretation, dispatch, and unnecessary abstraction overhead associated with the static inputs. Since the pioneering work of Ershov [35, 36] and Futamura [38] and the subsequent development of PE theory, specialization has influenced compiler construction, staged programming, domain-specific languages (DSLs), virtual machines (VMs), symbolic execution [50], and hardware generation. The central promise of PE is compelling: rather than optimizing individual program fragments, a sufficiently general specializer could automatically derive efficient implementations from generic ones.

Despite this elegant theoretical foundation and decades of research, PE as a self-contained, general-purpose specialization methodology has not achieved the mainstream adoption envisioned in the early literature. This is not because its underlying principle is rare: restricted, often unnamed forms of specialization – constant folding, constant propagation, tracing JIT compilation, and basic-block versioning among them – are pervasive in modern compilers and language runtimes. What has proven difficult is PE in its most general form: a single specializer that can be pointed at an arbitrary program and arbitrary static information and reliably produce a predictable, efficient residual program. Unlike many compiler optimizations that operate locally, this general form of specialization depends on a collection of global decisions, including the choice of static and dynamic information, unfolding depth, generalization strategy, and specialization boundaries [47, 58]. These decisions are highly dependent on program structure, workload characteristics, and engineering constraints. Modern software systems further complicate specialization through mutable state, dynamic language features, foreign-function interfaces, concurrency, and other effects that limit the scope of transformations that can be performed safely and predictably.

At the same time, PE has proved successful in a number of domains, including staged programming, DSLs, VMs with guarded specialization, compiler toolchains, and optimizations. These successes sug-

*Corresponding author

Email addresses: xadettu@uwm.edu (Xavier Adettu), nbanik@uwm.edu (Nayan Banik), tzhao@uwm.edu (Tian Zhao)

[1] ORCID: 0009-0006-5986-7380

[2] ORCID: 0000-0003-4563-3314

[3] ORCID: 0000-0001-6456-9763

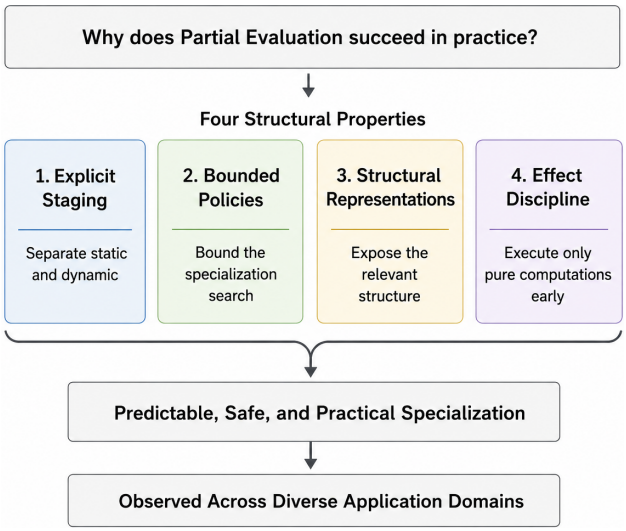


Figure 1: Conceptual overview of this survey. Successful applications of partial evaluation consistently exhibit four recurring structural properties, derived from the qualitative coding procedure described in Section 6.1. These properties provide the organizing framework for the remainder of the survey.

gest that the practical limitations of PE do not arise from specialization itself, but from the conditions under which specialization is applied.

This observation motivates the central question of this survey:

What distinguishes successful applications of partial evaluation from those in which it fails to become a practical optimization technique?

Our thesis is that successful applications consistently provide additional structure that enables specialization to be applied predictably and safely. Across diverse application domains, four recurring structural properties appear repeatedly (Figure 1): (1) explicit staging boundaries separating static and dynamic information; (2) constrained specialization policies that bound the search space; (3) intermediate representations that expose the program structure needed for specialization; and (4) disciplined treatment of computational effects that identifies which computations may be safely executed during specialization and which must remain residual. Rather than viewing these as isolated engineering techniques, we argue that they constitute a common systems-level explanation for why PE succeeds in practice across seemingly diverse application domains.

1.1. Related Work

The classical literature on PE is anchored by several influential surveys and monographs. The tutorial by Consel and Danvy [27] introduced many of the central concepts of program specialization and summarized the state of the field during its rapid development. Jones's survey [46] and the monograph by Jones, Gomard, and Sestoft [47] remain the standard references on the theory, algorithms, implementation techniques, and applications of PE. Among general-purpose partial evaluators, Bondorf's self-applicable Similix system for Scheme [16] is one of the most successful, demonstrating automatic compiler generation from interpreters via self-application. An alternative to building a general-purpose partial evaluator is to hand-write a specializer, or even a compiler generator, for a specific program or language; such hand-written approaches have also proven successful in the classical PE literature and remain a practical alternative when a general-purpose specializer is unavailable or insufficiently predictable.

Subsequent survey-oriented publications broadened this foundation. The Dagstuhl seminar volume *Partial Evaluation* [33] reviewed emerging research directions, while the DIKU summer school volume *Partial Evaluation: Practice and Theory* [41] collected tutorials covering offline and online PE, logic-program specialization, supercompilation, and type-directed specialization. Later tutorials, including Cook and Lammel's introduction to online PE [29], focus on specific technical developments rather than providing a broad synthesis of the field.

This survey differs from previous work in both its perspective and its organizing principle. Existing surveys primarily describe the algorithms, theoretical foundations, and historical evolution of PE. In contrast, we examine PE as a systems methodology: rather than treating specialization as a self-contained algorithm, we treat its success as depending on properties of the surrounding compilation environment – specifically, the four structural properties introduced in Section 1 and the operational success criteria defined in Section 4.4. Rather than organizing the literature chronologically or algorithmically, we synthesize successful applications according to the structural conditions that enable specialization in practice. This perspective allows us to explain why PE became highly successful in certain domains despite never becoming a universally applicable optimization technique, and to con-

nect classical PE with modern compiler-intensive application areas.

1.2. Scope and Contributions

This survey focuses on PE as a practical specialization methodology rather than solely as a program-transformation technique. Our emphasis is on applications in which specialization is embedded within larger compilation, analysis, or artifact-generation pipelines, allowing the surrounding system to constrain and validate specialization decisions.

The principal contributions of this survey are as follows.

- We synthesize the literature to explain why PE has not achieved the broad, self-contained adoption envisioned in the early literature, even as PE-derived techniques such as constant propagation and tracing JIT compilation have become ubiquitous in modern compilers and runtimes.

- We propose a systems-oriented taxonomy that organizes successful applications into four application niches – Multi-Stage Programming (MSP), Domain-Specific Languages (DSLs), Virtual Machines (VMs) and Just-In-Time (JIT) compilation, and program optimization – according to the structural mechanisms that enable specialization.

- We identify four recurring structural properties – explicit staging boundaries, bounded specialization policies, structure-exposing representations, and effect discipline – that provide a unifying explanation for successful PE systems. These properties are derived from a coded corpus of 44 systems through an explicit survey methodology (Section 4).

- We analyze two case studies: GENSYM and AnyHLS, to illustrate how these structural properties appear in modern software-analysis and hardware-generation toolchains.

- We discuss how these observations motivate future directions, including bounded agent-guided specialization, and examine deep-learning compilation as an emerging domain alongside modern data-processing systems.

2. Background and Terminology

Before discussing success and failure, we first define the main terms used in the rest of the survey.

2.1. Partial Evaluation Basics

Partial evaluation (PE) specializes a program with respect to information known before execution. We adopt the standard two-input formulation: a program p takes *static* inputs x, which are available during specialization, and *dynamic* inputs y, which are available only when the residual program executes. A partial evaluator spec produces a specialized residual program

$$p_x \triangleq \mathsf{spec}(p, x),$$

such that, for every dynamic input y,

$$\forall y.\ [\![p]\!](x, y) = [\![p_x]\!](y),$$

where $[\![\cdot]\!]$ denotes the semantics of the language [47]. Intuitively, specialization performs those computations that depend only on the static input while preserving the behavior of the original program for all remaining dynamic inputs.

In practice, PE performs semantic simplifications rather than merely local optimizations. During specialization, known values may be propagated, branches eliminated, function calls inlined, and loops partially evaluated when their behavior depends only on static information. Unlike optimizations such as constant folding or dead-code elimination [62], PE exploits semantic knowledge of the program to eliminate entire layers of interpretation, dispatch, and generic abstraction [47, 27].

We use the term *staging* to refer to the explicit separation of a computation into a sequence of phases – most simply, an earlier specialization-time stage and a later execution-time stage – such that all values available in an earlier stage are resolved before the next stage begins. *Staging boundaries* are the points in a program or system where this separation is made explicit, whether through language constructs (Section 5.1), compiler phases, or deployment structure (Section 5). Binding-time analysis (Section 2.3) is one mechanism for inferring a staging boundary automatically; multi-stage programming languages instead make it a first-class part of the program text.

A useful way to view PE is as the specialization of a general mechanism into an artifact tailored to a particular context. If a generic program contains

decisions intended to support many possible execution scenarios, fixing one scenario in advance allows those decisions to be resolved during specialization rather than at run time. Depending on the application, the resulting residual artifact may be an optimized executable, a specialized interpreter, a symbolic analyzer, or a hardware-generation pipeline. Although these artifacts differ substantially, they all arise from the same underlying principle: computations determined by static information are performed once during specialization, leaving only genuinely dynamic computation in the residual program.

The simplicity of this definition hides the main challenge of PE. A practical specializer must determine which information is sufficiently stable to specialize – that is, which values are fixed at specialization time and remain valid for every execution of the residual program, as a deployment configuration does but a per-run user input does not – how aggressively to unfold computation, how to control code growth, and how to avoid unsafe specialization across effectful computations. These engineering decisions largely determine whether PE remains a theoretical transformation or becomes a practical optimization technique. The remainder of this section reviews the classical concepts needed throughout the survey, including offline and online PE, binding-time analysis, effect management, and termination control.

2.2. Offline and Online Partial Evaluation

A fundamental distinction in the PE literature is between *offline* and *online* specialization [47, 27]. Both aim to evaluate computations using information available before execution, but they differ in when specialization decisions are made.

Offline PE. Offline PE separates analysis from transformation. A binding-time analysis (BTA) first classifies program expressions as static or dynamic, either through programmer annotations or automatic inference [47, 27]. The subsequent specialization phase follows these decisions to evaluate static computations and residualize the remaining program. Because specialization policies are determined before transformation begins, offline PE offers predictable specialization behavior and facilitates explicit control over unfolding, code growth, and specialization time [47].

Online PE. Online PE interleaves analysis with specialization [27, 29]. Decisions such as whether to unfold a function call, evaluate a branch, or residualize a computation are made dynamically as additional information becomes available during specialization. This often enables more precise specialization but requires stronger mechanisms for controlling termination and limiting resource consumption [29].

Many practical systems combine elements of both approaches [47]. A conservative offline analysis may establish an initial specialization boundary, while online decisions further refine specialization as additional information becomes available. Throughout this survey, we treat the offline–online distinction primarily as one dimension of specialization design rather than as a strict classification of PE systems.

2.3. Binding-Time Analysis

Binding-time analysis (BTA) determines which parts of a program may be evaluated during specialization and which must be deferred to run time. In classical offline PE, BTA precedes specialization by classifying program constructs – variables, expressions, statements, and, in higher-order settings, function definitions – as *static* or *dynamic*, either through programmer annotations or automatic inference [47, 27]. The resulting binding-time information guides the specializer to evaluate static computations while residualizing computations that depend on dynamic inputs.

The precision of BTA has a direct impact on the effectiveness of specialization. An overly conservative analysis leaves optimization opportunities unexplored, whereas an overly aggressive or incorrect analysis risks unsafe specialization; aggressive unfolding in particular risks excessive code generation or nontermination even when the underlying analysis is correct. Practical systems therefore combine BTA with mechanisms such as user annotations, polyvariance, effect analyses, or explicit staging constructs to balance optimization opportunities against safety and predictability [47, 29]. A specializer is *polyvariant* if it may generate several specialized versions of the same source function or program point, one for each distinct combination of static values it encounters, and *monovariant* if it generates at most one; more variants usually improve the precision of specialization but enlarge the residual program and lengthen specialization [47].

Although BTA originated as a compiler analysis [47], its underlying role is more general: it es-

tablishes the boundary between computations that can be performed ahead of time and those that must remain in the residual program. Throughout this survey, we revisit this boundary in different forms, including explicit staging, domain-specific intermediate representations, runtime specialization, and effect-aware execution.

2.4. Termination and Specialization Control

A PE system is useful only if specialization terminates and produces residual programs whose compilation and execution costs justify the specialization effort. Without appropriate control, unrestricted unfolding may diverge, generate excessive code, or create an impractical number of specialized variants [47].

Classical PE employs several complementary mechanisms to control specialization. Memoization avoids repeatedly specializing equivalent program configurations by reusing previously generated residual code [47]. Generalization merges overly specific specialization states to limit code growth and ensure convergence, trading some specialization precision for predictability [47]. Many systems further employ principled termination criteria such as homeomorphic embedding [58], widening techniques inspired by abstract interpretation [31], or explicit limits on unfolding depth and polyvariance [47].

Across these mechanisms, a common purpose emerges: they bound the specialization search space. Rather than allowing unrestricted symbolic execution, successful PE systems constrain specialization so that optimization remains predictable in both time and space. This observation reappears throughout the survey as one of the recurring characteristics of practical PE systems.

2.5. Futamura Projections

The Futamura projections provide one of the most influential conceptual results in partial evaluation by relating interpretation, compilation, and program generation through specialization [38, 47]. Let int denote an interpreter for a source language implemented in some implementation language. Executing a source program src on input input can be written as

$$\mathsf{int}(\mathsf{src}, \mathsf{input}).$$

The *first Futamura projection* observes that specializing the interpreter with respect to a fixed source program produces an executable equivalent to compiling that program:

$$\mathsf{spec}(\mathsf{int}, \mathsf{src}) = \mathsf{compiled_src}.$$

Here compiled_src is expressed in the same implementation language as int: specialization can remove the interpretive overhead of int but does not change its implementation language, so the residual program is compiled *to* that language rather than to some independently chosen target. Conceptually, compilation can therefore be viewed as partial evaluation of an interpreter.

The *second Futamura projection* specializes the specializer itself with respect to the interpreter:

$$\mathsf{spec}(\mathsf{spec}, \mathsf{int}) = \mathsf{compiler},$$

where $\mathsf{compiler}(\mathsf{src}) = \mathsf{compiled_src}$. Self-application thus turns an interpreter into a compiler without any hand-written compiler. The *third Futamura projection* specializes the specializer with respect to itself:

$$\mathsf{spec}(\mathsf{spec}, \mathsf{spec}) = \mathsf{cogen},$$

where cogen is a *compiler generator*: applied to an interpreter it yields a compiler, $\mathsf{cogen}(\mathsf{int}) = \mathsf{compiler}$. The three projections therefore produce, in turn, a compiled program, a compiler, and a compiler generator, using only a specializer that can be applied to itself (a *self-applicable* specializer) [38, 47]. Bondorf's Similix, a self-applicable partial evaluator for a higher-order subset of Scheme, realizes the third projection: it derives a compiler generator from the specializer by self-application, rather than directly from an interpreter [16].

Self-application is not the only route to a compiler generator. In the *cogen approach*, the compiler generator is written by hand instead of being derived from a specializer. Given a program annotated with binding times (Section 2.3), a hand-written cogen produces a *generating extension*: a program that, when run on the static inputs, directly emits the residual program with no interpretive specializer in the loop; the generating extension of an interpreter is a compiler. Because a hand-written cogen never has to specialize itself, it can be simpler to build than a self-applicable specializer and can avoid its restrictions on the subject language [42, 14, 78]. When later sections refer to a "cogen-based" approach, they mean this construction.

Although all three projections are historically important, this survey primarily adopts the first projection as a systems perspective, since it corresponds directly to the recurring pattern examined throughout this survey: specializing a general interpretive or configurable mechanism into a form dedicated to one deployment target. The second and third projections describe how self-application can automate the construction of compilers and compiler generators from interpreters, but the systems we survey mainly apply an existing specializer once, to a fixed interpreter or engine, rather than deriving compilers or compiler generators through self-application; we therefore do not extend the treatment of self-application further, though it remains an active area in the classical PE literature [47]. Many successful applications of PE can be understood as specializing an interpreter-like engine for a restricted domain, such as a DSL, symbolic analysis framework, query processor, or hardware-generation toolchain. We revisit this perspective throughout the survey when discussing interpreter specialization, embedded compilation pipelines, and domain-specific code generation.

The remainder of this survey examines how modern systems realize this vision under practical engineering constraints. Rather than relying on unrestricted specialization, successful systems expose explicit structure that makes specialization predictable, bounded, and effective.

3. Why General-Purpose PE Remains Uncommon

The promise of partial evaluation is compelling: by exploiting information available before execution, a program can be specialized into a simpler and often substantially faster residual program. Yet despite decades of research and numerous successful systems, PE as a self-contained, general-purpose specialization methodology has not achieved the broad adoption envisioned in the early literature. This disparity between theoretical elegance and practical adoption is one of the central questions motivating this survey.

This claim should be read narrowly. Restricted, often unnamed forms of the same underlying principle – constant folding, constant propagation, loop unrolling, tracing JIT compilation, and basic-block versioning – are widely deployed in production compilers and language runtimes. What has remained comparatively rare is PE in its most general form: a single, explicitly named pass that specializes arbitrary programs with respect to arbitrary static information.

The difficulty is not that PE is ineffective. Rather, profitable specialization in its most general form depends on a collection of global decisions – what information is static, how aggressively to unfold computation, when to generalize, and which computations may safely be executed early. In unrestricted software systems, these decisions are difficult to make automatically and predictably. The following subsections examine the principal factors that have limited the widespread adoption of general-purpose PE.

3.1. Effectful and Dynamic Programs

The first obstacle is determining what computations can be safely performed during specialization. In purely functional settings, specialization can freely evaluate computations whose inputs are known. Mainstream software, however, is dominated by computations that interact with the surrounding execution environment. Input/output operations, mutable state, exceptions, concurrency, and other observable effects constrain when computations may be executed without changing program behavior. Executing such operations during specialization may perform them at the wrong time, duplicate them, or alter their ordering, thereby violating program semantics [47, 61].

Dynamic language features introduce additional uncertainty. Dynamic dispatch, reflection, runtime code generation, and foreign-function interfaces make future program behavior difficult to predict statically because part of the computation lies outside the specializer's model [27]. Consequently, the amount of computation that can be safely classified as static is often substantially smaller than the amount that would be profitable to specialize. Rather than reflecting a weakness of PE itself, these limitations arise from the difficulty of identifying sufficiently stable computations in general-purpose software systems.

3.2. Policy Explosion

Even when specialization is semantically safe, a practical PE system must make numerous interdependent policy decisions [47, 27]. These include determining the static–dynamic boundary, choosing how aggressively to unfold function calls, deciding

how many specialized variants to generate, selecting when to generalize symbolic states, and allocating budgets for specialization time and residual code size. None of these decisions can be made independently: increasing unfolding may require more aggressive generalization, while increasing polyvariance (Section 2.3) often improves specialization quality at the cost of larger residual programs and longer specialization times.

The central difficulty is that no single policy is uniformly optimal. Effective specialization depends on the structure of the program, the expected workload, available computational resources, and interactions with downstream compiler optimizations. Consequently, PE is better viewed as a search over a large policy space than as a single optimization algorithm. This observation helps explain why successful PE systems typically restrict or expose these policy decisions rather than attempting to optimize them universally.

Two observations temper this difficulty in practice. First, the policies that make specialization succeed often depend only on the program being specialized and not on the static data it is specialized to. When specializing an interpreter, for example, the interpreter may have to be adjusted once so that specialization produces good residual code – a *binding-time improvement* in the classical terminology [47] – but once it has been adjusted, specialization typically succeeds for whichever source program is supplied. The tuning effort is paid once per specialized program or engine rather than once per input. Second, and for the same reason, effective specialization usually requires that the code be written with specialization in mind. This is a modest, one-time engineering cost, and it recurs throughout the application niches surveyed in Section 5 (see in particular the summary of Section 5.4).

3.3. Uncertain Cost Models

Determining whether specialization will be profitable is itself a difficult optimization problem, closely related to the classical inlining and cost-model problems in compiler construction [47, 79]. Aggressive specialization may substantially improve execution time, but it can also increase compilation cost, residual code size, instruction-cache pressure, and memory consumption. Conversely, conservative specialization may preserve predictable compilation behavior while yielding little performance improvement.

Unlike many classical compiler optimizations, the profitability of PE depends on complex interactions between specialization policies, program characteristics, and subsequent optimization passes. Reliable cost models that generalize across diverse software systems are therefore difficult to construct. This unpredictability conflicts with the design goals of production compilers, which emphasize bounded compilation time, stable code quality, and consistently positive optimization outcomes across a broad range of workloads [47].

3.4. Mismatch with Compiler Ecosystems

Modern optimizing compilers are built around collections of relatively modular analyses and transformations, including inlining, constant propagation, dead-code elimination, common subexpression elimination, loop optimizations, and profile-guided optimization [62, 79, 12]. These optimizations generally operate within mature intermediate representations, compose well with one another, and offer reasonably predictable compilation costs and performance improvements.

Partial evaluation makes stronger assumptions about the surrounding software ecosystem. To achieve substantial specialization, PE benefits from explicit staging boundaries, constrained specialization policies, well-structured intermediate representations, and disciplined treatment of computational effects. General-purpose software rarely exposes these properties explicitly, making aggressive specialization difficult to apply reliably. By contrast, systems that do provide such structure – including staged programming languages, virtual machines and JIT compilers that specialize under explicit deoptimization guards, domain-specific compilation frameworks, and hardware generation toolchains – create conditions under which PE becomes both predictable and economically attractive.

Taken together, these observations suggest that the historical limitations of PE stem less from the specialization technique itself than from the environments in which it is applied. The following sections examine domains where the surrounding system supplies the additional structure needed to make specialization explicit, bounded, and reliable.

4. Survey Methodology

4.1. Corpus Construction

The literature examined in this survey was assembled in three steps. We began from a seed set

consisting of the standard monographs, tutorials, and earlier surveys of partial evaluation [47, 27, 46, 33, 41]. From this seed set we performed backward snowballing over the works they cite and forward snowballing over later works citing them. We complemented snowballing with keyword searches, such as *partial evaluation*, *program specialization*, *multi-stage programming*, *staging*, *runtime specialization*, *JIT specialization*, over standard bibliographic databases including ACM digital library, IEEE Xplore, DBLP, and Google Scholar. Because the goal of this survey is to explain practical success rather than to enumerate the field exhaustively, we prioritized works describing implemented systems over purely theoretical treatments.

4.2. Inclusion and Exclusion Criteria

A work was included if it satisfies three criteria: (a) partial evaluation or program specialization is a central mechanism of the work rather than an incidental implementation detail; (b) the specialization mechanism is described well enough to answer, from the published description alone, four questions: where the static–dynamic boundary is drawn and by whom (staging), what limits unfolding, specialized variants, or code growth (specialization policy), what program representation the specializer operates on (representation), and how side effects are treated (effects) – a work whose text leaves any of the four unanswered does not satisfy this criterion; and (c) the work reports a quantitative evaluation or a documented deployment. Works failing these criteria were not necessarily discarded: purely theoretical treatments are retained as background material (Sections 2 and 3), and works on emerging platforms whose specialization mechanisms are not yet mature are discussed as future directions (Sections 8, 9, and 9) rather than used as evidence for the structural patterns.

4.3. Corpus Composition.

The resulting bibliography contains 90 works. Of these, 42 satisfy the inclusion criteria and form the surveyed corpus analyzed in Sections 5 and 6; 25 provide background, terminology, and historical context; 19 support the discussion of agentic and emerging directions; and 4 are examined exclusively in the case studies of Section 7. Figure 2 shows the publication-year distribution of all cited works. The distribution reveals a foundational period through the late 1990s anchored by the classical PE literature, sustained systems-oriented activity between 2000 and 2014 as specialization techniques matured within multi-stage programming, DSL frameworks, and virtual machines, and a pronounced recent concentration (2015–2026) reflecting the growth of compiler-intensive domains such as language runtimes, program analysis, and hardware generation. Appendix A lists the 42 works of the surveyed corpus, grouped by niche, with the subsection of Section 5 in which each is discussed. The coded corpus and the script that regenerates Figure 2 directly from the bibliography accompany the paper's sources.

4.4. Criteria for Practical Success

Because terms such as *success* are subjective, we use explicit criteria throughout this survey. We consider an application of PE practically successful if it (a) reports a quantitative improvement on at least one primary metric such as execution time, code size, compilation time, or implementation effort, and (b) shows evidence of sustained use, such as production deployment, adoption by downstream systems, or a maintained implementation. Under this definition, success does not require that PE appear as a named optimization pass in a general-purpose compiler; it requires that specialization deliver measurable benefit within a working system.

4.5. Scope and Limitations

This process has limitations that bound the claims we make. Classification and coding involve judgment, and the published literature overrepresents successful systems, so the corpus cannot establish that any structural property is strictly necessary for practical PE. Accordingly, we present the four patterns of Section 6 as *recurring enabling conditions* observed across this corpus, not as an exhaustive or necessary set; other properties, alone or in combination with those identified here, may support practical specialization in settings not represented in the corpus. The pattern-extraction procedure itself is described in Section 6.1.

5. Application Niches for Practical PE

PE has found practical success in a diverse range of domains, yet not all applications have been equally successful. This section examines the niches where PE has thrived, with the goal of understanding what distinguishes these domains from those

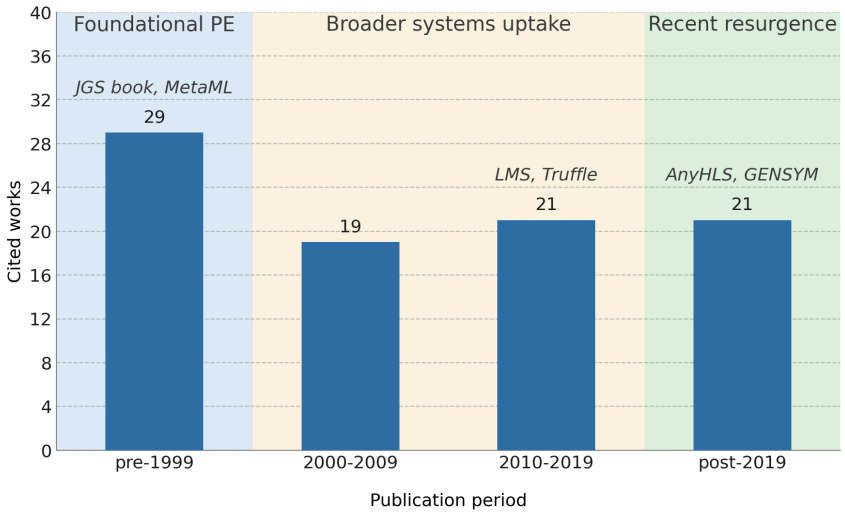


Figure 2: Publication-year distribution of the 90 works cited in this survey, grouped by period (pre-1999, 2000–2009, 2010–2019, post-2019). Shaded bands mark three eras of PE research; annotated labels identify landmark systems discussed in the text.

where PE has struggled to gain adoption. We organize the niches around four applications: multi-stage programming (Section 5.1), domain-specific languages (Section 5.2), virtual machines and JIT compilation (Section 5.3), and program optimization (Section 5.4). Although the surveyed systems were developed independently and pursue different goals, they repeatedly converge on the same underlying principles. Some make staging explicit through language constructs, others constrain specialization through domain restrictions or runtime guards, others expose rich intermediate representations, and still others isolate effectful computation. The implementation mechanisms differ, but the structural patterns remain remarkably consistent. Section 6 synthesizes these patterns; the remainder of this section provides the concrete evidence, examining each niche in turn and identifying the specialization mechanisms and structural conditions that enable practical success.

5.1. Multi-Stage Programming

Multi-stage programming (MSP) provides one of the clearest demonstrations of how explicit structure enables practical partial evaluation. Rather than requiring a specializer to infer which computations may be performed early, MSP languages distinguish code-generation time from execution time through explicit staging constructs. Computations whose inputs are available in an earlier stage are evaluated immediately, while computations that depend on future inputs are represented as residual code to be executed later [74, 73].

This explicit separation greatly simplifies specialization. Classical PE relies on binding-time analysis to approximate the static–dynamic boundary, whereas staged languages make that boundary part of the program itself. Many staged systems further enforce well-formedness through type systems that prevent illegal cross-stage references, ensuring that generated programs are both syntactically and semantically valid. This section surveys key contributions to MSP that have influenced the broader PE landscape, organized by their central technical contributions.

Staging is not limited to dedicated staged languages. C++ template programming, for example, has been used to specialize code at compile time: template instantiation evaluates computations over types and compile-time constants before the program runs, and Veldhuizen showed that this process can be understood as a form of partial evaluation [80]. The staging boundary is the language's own distinction between compile-time template arguments and run-time values, and the specialization space is bounded by the instantiation mechanism, which compilers limit in recursion depth. BuildIt, discussed below, is a more recent, library-based approach to staging in C++.

5.1.1. Foundations: MetaML and Explicit Annotations

The MetaML family of languages, introduced by Taha and Sheard [74], established the core abstractions for MSP. MetaML provides three high-level constructs: *brackets* `<e>` to delay computation, *escapes* `~e` to force immediate evaluation within brackets, and `run` to compile and execute generated code. These constructs allow programmers to break down computations into distinct stages while automatically avoiding accidental variable capture and statically ensuring that generated programs are well-typed [74, 73].

The key insight of MetaML is that staging constructs are not hints but imperatives: they determine precisely when computation occurs. For example, specializing a power function to a specific exponent proceeds by evaluating the recursive definition at compile time, producing residual code that computes the result directly. This gives programmers full control over specialization while maintaining type safety – properties that distinguish MSP from other metaprogramming approaches.

5.1.2. Type-Based Staging: Lightweight Modular Staging and BuildIt

Building on the foundations of MetaML, Lightweight Modular Staging (LMS) [68] brought staging to Scala through a library-based approach. LMS represents staged computations using a type constructor `Rep[T]` that distinguishes compile-time from run-time values. Scala's type inference acts akin to a local binding-time analysis: expressions of type Int execute when the host program runs, while expressions of type `Rep[Int]` construct residual code. This approach enables the implementation of DSLs as libraries of higher-order functions, with PE removing abstraction overhead.

However, LMS requires programmers to write code in a style that explicitly distinguishes staged from unstaged values. For data types unknown to LMS, programmers must implement cogen (code generation) logic themselves, introducing type variables for each desired staging combination. This limitation can be partially alleviated through parametric polymorphism, but at the cost of additional type annotations.

BuildIt [19] takes a complementary approach in C++. Rather than relying on a type constructor, BuildIt uses two new types, `static<T>` and `dyn<T>`, to indicate whether variables should be evaluated in the static or dynamic stage. The framework extracts abstract syntax trees (ASTs) from programs by executing them repeatedly to explore all control-flow paths. This novel technique allows BuildIt to extract conditionals and loops, including side effects on unstaged variables, without requiring compiler changes or specialized syntax. A static-tag memoization strategy bounds the extraction process, ensuring that AST construction remains polynomial in the number of branches.

5.1.3. Monadic Staging: Controlling Code Duplication

A central challenge in staging is avoiding code duplication: without careful control, recursive staging can generate exponentially large residual programs. Swadi et al. [72] address this problem in the context of memoized functions, where direct staging "undoes the effect of memoization" and leads to exponential code growth.

Their solution employs a staged monadic combinator library based on a state-continuation monad. The key insight is that monadic bind operations can name intermediate results using let-statements, preventing duplication when values are used multiple times. The combinators `retN` and `bindN` ensure that generated programs are in A-Normal Form, with all intermediate results explicitly named. This approach generalizes beyond memoized functions: the same combinators have been used to stage hardware circuit generation (FFT) and Gaussian elimination, demonstrating the broader applicability of monadic staging techniques.

Experimental results show that the combinator-based approach produces residual programs whose performance is competitive with handwritten C implementations of dynamic programming algorithms, confirming that monadic staging effectively avoids exponential code growth.

5.1.4. Cross-Stage Persistence and Delimited Control

A recurring theme in MSP research is cross-stage persistence (CSP): the ability to use values from an earlier stage in a later stage. Early staged languages like MetaML restricted CSP to base-type values to maintain the existence of a best staging annotation. More expressive CSP mechanisms require more sophisticated control.

Kameyama, Kiselyov, and Shan [49] explore staging with delimited control operators, showing that

delimited continuations can shift computations between stages. This work demonstrates that control effects can shape how computations move across stages, enabling patterns that are difficult to express with ordinary staging constructs. The implementation in terms of delimited control operators also provides a foundation for efficient code generation in the presence of advanced control flow.

5.1.5. Binding-Time Analysis and Automatic Staging

While manual staging provides precise control, automatic staging through binding-time analysis can reduce programmer effort. Asai [10] and Asai and Kameyama [11] relate the 2-level λ-calculus (used for offline PE) to the staged λ-calculus (used in MetaOCaml) via an intermediate staged 2-level λ-calculus. This relationship enables the automatic conversion of binding-time annotations to staging annotations.

The key insight is that the "best staging annotation" can be defined in terms of binding times: a staging annotation is best if it yields the most evaluated code. For a calculus with CSP restricted to base-type values, Asai and Kameyama prove that the best staging annotation exists and can be derived automatically from binding-time analysis. However, when CSP is allowed for higher-type values, no single best annotation exists in general – different staging choices may be incomparable.

This work provides a formal foundation for introducing binding-time analysis into full-featured staged languages like MetaOCaml, enabling automatic staging while preserving the programmer control that makes MSP practical.

5.1.6. Tag Elimination and Jones-Optimality

A long-standing challenge in typed MSP is achieving Jones-optimality: the property that a partial evaluator removes all interpretive overhead when specializing an interpreter. Taha, Makholm, and Hughes [75] address this through tag elimination, a program transformation that removes unnecessary tagging and untagging operations from generated programs.

In typed staged languages, universal datatypes introduce tagging overhead that cannot be eliminated by the type system. Tag elimination analyzes annotated terms to determine when tags are superfluous, removing them while preserving semantics. The authors show that tag elimination achieves Jones-optimality for a typed self-interpreter, solving a problem that had remained open for over thirteen years. Experimental results demonstrate speedups of 1.9x to 2.6x on sample programs, and the technique has been validated in a Scheme implementation.

5.1.7. Scoped and Typed Staging by Evaluation

Recent work by Allais [7] gives an intrinsically scoped-and-typed presentation of a two-level simply typed calculus with static and dynamic stages. Using normalization by evaluation, the staging function partially evaluates static parts of a term. The model construction yields a staging algorithm that is both sound and complete, with the additional feature that static datatypes need not have counterparts at runtime – static pairs, for example, can be evaluated away entirely.

This work extends to practical applications including digital circuit generation, where the static layer is a higher-order functional language and the dynamic layer corresponds to circuit descriptions. The dependently typed host language (Agda) ensures that generated code is both well-scoped and well-typed, and the staging algorithm guarantees that type information is erased at runtime.

5.1.8. Summary

MSP research has established the core principles of explicit staging: separating code-generation time from execution time through language constructs, maintaining type safety across stages, and providing mechanisms to control code growth. The surveyed work spans foundational languages (MetaML), practical frameworks (LMS, BuildIt), formal analyses (binding-time analysis, tag elimination), and applications (hardware generation, dynamic programming). A recurring theme is the importance of making staging boundaries explicit, whether through language constructs, type systems, or monadic combinators, to ensure predictable and efficient specialization.

5.2. Domain-Specific Languages (DSLs)

DSLs provide one of the most fertile application areas for partial evaluation (PE). DSLs deliberately restrict the semantic space in which programs are expressed, focusing on well-defined classes of computations such as database queries, numerical kernels, image processing pipelines, or hardware descriptions. As a result, many performance-critical

parameters, including data schemas, tensor shapes, scheduling decisions, and target architectures, are known before execution, making them attractive candidates for specialization [44].

This section surveys DSL implementations that leverage PE, ranging from shallow and deep embedding approaches to frameworks that automatically generate compilers from interpreters. A recurring theme is that DSLs naturally exhibit the structural properties (explicit staging boundaries, bounded specialization policies, structure-exposing representations, and effect discipline) that make PE practical.

5.2.1. Shallow Embedding and PE

A shallow embedding represents DSL constructs directly as host-language functions, avoiding the overhead of building an intermediate representation (IR). The challenge is that shallow embeddings cannot easily manipulate the embedded program, limiting optimization opportunities. PE addresses this by specializing the host-language implementation with respect to static DSL parameters.

Leissa et al. [56] present a shallow embedding approach using online PE on a continuation-passing style (CPS) IR called Thorin. The key innovation is a termination policy that makes PE predictable: when encountering a dynamic branch, the evaluator skips to a post-dominator, avoiding induced divergence. This is complemented by a local control-flow analysis that computes post-dominators in higher-order programs, making the approach practical for realistic DSLs.

The evaluator is driven by explicit annotations in the style of Schism filters [26]: annotations attached to a function that decide whether a call to it is unfolded or kept residual and specialized, and which of its arguments it is specialized on. This puts the programmer in total control over which parts of the program should be specialized under which conditions, eliminating the imprecision of static analyses [63] that guide other evaluators. The approach is evaluated on stencil computations and V-cycle multigrid solvers, demonstrating performance competitive with hand-tuned OpenCV implementations and HIPAcc-generated code. Notably, merging residual and restrict components yields speedups of 11–20% on CPU and GPU targets.

AnyDSL [57] extends this work to a complete programming system built around an online partial evaluator for the CPS-based IR Thorin. The system exposes code generation for accelerators (vectorization, parallelization, and GPU mapping) via compiler-known higher-order functions that can be subjected to PE. This design allows generic implementations to be instantiated with target-specific code at compile time, achieving performance within 10% of industry-grade, manually-optimized expert codes in image processing, ray tracing, and genome sequence alignment.

A key insight of AnyDSL is that PE of higher-order programs enables a specific kind of shallow DSL embedding. Following the tagless interpreter approach of Carette et al. [21], the abstract syntax of the embedded program is represented as a tree of closures rather than data. Partially evaluating this closure tree corresponds to the first Futamura projection and leads to the same code a syntax-directed compiler would emit. This enables the implementation of a DSL in a library of higher-order functions, without requiring a dedicated DSL compiler.

5.2.2. Deep Embedding and DSL Compilation

Deep embeddings represent DSL programs as data structures (abstract syntax trees) in the host language, enabling domain-specific optimizations. However, deep embeddings introduce usability hurdles: convoluted interfaces, difficult debugging, incomprehensible type errors, and runtime overhead.

Jovanovic's thesis [48] proposes Yin-Yang, a framework that unites shallow and deep embeddings through an automatic translation from end-user-friendly shallow DSLs to better-performing deep DSLs. The translation uses Scala macros to reliably convert direct (shallow) EDSL programs into corresponding deep EDSL programs. During program development, the user is presented with user-friendly features of a shallow EDSL. Before execution in production, programs are reliably translated into the high-performance deep EDSL with equivalent semantics.

The translation is type-directed: it virtualizes host-language constructs (if statements, loops, function definitions) into method calls, then intrinsifies DSL operations and types from the direct embedding to their deep counterparts. A configurable type translation maps direct types `T` to deeply embedded types `[T]`, with several options including identity translation, generic polymorphic embedding, eager inlining, and custom types.

Yin-Yang also improves error reporting by compiling DSL programs at host-language compile time, enabling domain-specific error reporting before runtime. It reduces runtime overhead

by caching reified DSL programs, avoiding re-reification on every execution. The framework is evaluated on the Slick database query DSL, where building a new frontend required only 120 hours of development and reduced compilation overhead from 380μs to under 1μs per execution.

5.2.3. Tagless Final Interpreters

Carette, Kiselyov, and Shan [21] introduced the "finally tagless" approach to DSL embedding, solving the long-standing problem of tagless type-preserving interpretation in a typed metalanguage without GADTs or dependent types. The key insight is to represent object terms not in an initial algebra (using data constructors) but using the existing coalgebraic structure of the λ-calculus: object programs are encoded as combinator functions that abstract uniformly over the family of ways to interpret them.

The approach defines a type class `Symantics` that gives the syntax of the object language. Instances of the class provide different semantics: an evaluator (`R`), a compiler (`C`), a partial evaluator (`P`), and CPS transformers. The partial evaluator uses a type function `static` that maps source types to target types, simulated through type classes in Haskell and through an additional type argument in OCaml.

This approach achieves several desirable properties: no type tags or pattern matching in the interpreter, static assurance of type preservation, and the ability to process the same term using multiple interpreters. The partial evaluator avoids polymorphic lift by building dynamic terms alongside static terms, and delays binding-time analysis for functions until application. The resulting code is portable across Haskell 98 and ML, demonstrating that advanced type-system features are not necessary for tagless type-preserving interpretation.

Biri [13] extends the tagless final approach with dependent types in Idris. The key contribution is an encoding that addresses the composition of effectful language fragments. Instead of a monolithic interpreter type, Biri uses a context-parameterized interpreter: `Inter context` $\equiv$ (`ctx` : `context`) $\rightarrow$ (`tag` : `Type`) $\rightarrow$ `Type`. Each DSL fragment interacts with the context through type-level interfaces (`ReadContext`, `VarContext`), enabling modular composition of language features.

Quantitative types (0-quantity) guarantee that type information is erased at runtime, ensuring that the solution is truly tagless. The approach preserves staging capabilities: an interpreter can produce `Elab` values (Idris's metaprogramming representation) at compile time, generating optimized code. The paper demonstrates rewriting rules that improve code quality by embedding a read-only interpreter within a more general one, enabling constant computation at compile time.

5.2.4. PE of DSL Interpreters

Brady and Hammond [18] show that PE of DSL interpreters in a dependently typed host language (Idris) yields efficient residual programs whose performance is competitive with Java and C. The key insight is that dependent types allow precise statement of what the type of a residual program should be, removing the need for tagging of intermediate values. I/O and side effects pose no problem because computation and execution are distinguished (IO is treated as an EDSL for describing interaction with the operating system).

The authors identify four rules for maximizing the effect of PE:

1. Index the EDSL representation by its type, to avoid needing to tag the result.
2. The interpreter must only pattern match on the EDSL program to be translated.
3. Auxiliary functions which match on dynamic data must not be passed EDSL code unless it has been interpreted first.
4. Ensure that EDSL program construction, generation and transformation can be evaluated statically.

Experimental results show that specializing the interpreter provides significant speedup and reduction in space usage. For `Expr` programs, speedups are dramatic (e.g., factorial from 538.7s to 3.15s for 2M iterations). For `File` programs, PE improves execution time and memory usage, with the specialized version of a file copy program running faster than its Java equivalent. The paper also demonstrates that the approach can be applied to resource-safe network protocols.

5.2.5. Generating Compilers from Executable Semantics

Melancon, Feeley, and Serrano [60] present an executable semantics for Python operators that enables reuse in optimizing compilers. The key innovation is the `compiler_intrinsics` statement, which extends Python with low-level primitives for describing operations at a lower level. This allows

the definition of *behaviors*: specializations of an operator for a given combination of built-in types.

The semantics is written in a format that highlights type checks, primitive values boxing and unboxing, and function calls – all known sources of overhead. The tool `semPy` is a partial evaluator that generates *behaviors* by removing redundant type checks, boxing and unboxing, and method calls. The partial evaluator uses three transformations: aggressive inlining of method calls, branch resolution based on type information, and removal of redundant boxing and unboxing.

The approach is evaluated in Zipi, an ahead-of-time optimizing Python compiler prototype. Zipi dispatches operations to behaviors at runtime, increasing execution speed. On arithmetic-heavy programs, Zipi rivals PyPy in performance (e.g., Ackermann: 38x faster than CPython, exceeding PyPy). While the speedup is limited to arithmetic-heavy programs, the paper shows that behaviors can be extended to other operations or used alongside other optimization techniques.

5.2.6. PE for DSLs in Haskell

Coutts [32] provides a comprehensive survey of partial evaluation for DSLs in Haskell, identifying the challenges and proposing a cogen-based approach (Section 2.5). The thesis argues that fully-automated partial evaluators suffer from "clever compiler syndrome": programmers must be intimately familiar with the compiler's implementation to reliably take advantage of optimizations. Instead, the author advocates a semi-automated approach with programmer control.

The prototype transforms explicitly binding-time annotated programs into their generating extensions using Template Haskell. The cogen function is split into two phases: conversion to memoizing form and then encoding. The memoizing transformation ensures that recursive functions are specialized without infinite unfolding, by caching specialized versions and reusing them. The encoding phase handles double encodings of code, converting the IR (with static/dynamic annotations) into valid Haskell code.

The thesis identifies several technical challenges for future work: proper binding time checking, user-assisted binding time inference, support for higher-order static values, and partially static data structures. These challenges are addressed in subsequent work by Carette et al. [21] and others.

5.2.7. PE and Reflection for DSALs

Hutchins [45] explores the combination of PE and reflection for building domain-specific aspect languages (DSALs). The key insight is that PE with first-class functions is excellent for writing type-safe, well-integrated code generators for DSLs. Adding reflection allows automatic generation of "boilerplate" code. Combining PE with both reflection and first-class classes may enable full aspect-weaving, though type safety concerns remain.

The paper identifies a crucial limitation: mutable state in the heap makes everything partially static data. The proposed solution is to treat aspects as functions from immutable class sets to new transformed class sets, similar to feature-oriented programming. This approach enables step-wise refinement and allows different parts of a program to use different class extensions.

The DEEP programming language, introduced in the paper, integrates dependent types, singleton types, and PE to deal with partially static data. The type of an expression holds whatever information is known about it at compile time: static data has singleton types, while dynamic data uses dependent types. Classes are prototypes, treated as types at compile time and ordinary objects at runtime, simplifying code generation while maintaining static type safety.

5.2.8. Summary

DSL implementations that leverage PE consistently exhibit the four structural patterns identified in Section 6. *Explicit staging boundaries* appear in shallow embeddings (AnyDSL's annotations [57]), deep embeddings (Yin-Yang's translation between shallow and deep [48]), and tagless final approaches (the `Symantics` interface). *Bounded specialization policies* are evident in AnyDSL's user annotations controlling termination, Yin-Yang's compilation guards and code caches, and the memoising transformation in cogen-based approaches. *Structure-exposing representations* include Thorin's CPS IR, LMS's `Rep[T]` types, tagless final's combinator functions, and the AST representations used in deep embeddings. *Effect discipline* is present in the treatment of I/O in Idris DSLs (IO as an EDSL), in Yin-Yang's effect annotations, and in the pure functional semantics of Haskell-based approaches.

5.3. Virtual Machines and JIT Compilation

VMs and JIT compilation represent a domain where PE has found significant practical success. Unlike classical PE, which performs specialization before execution, JIT systems exploit information collected during program execution, including runtime types, object layouts, branch behavior, and execution frequencies. This additional information enables optimizations that would be difficult or impossible to perform statically, while allowing the system to adapt to changing workloads.

A key engineering innovation in modern VMs is the use of *guarded specialization*. Rather than assuming that specialization decisions will always remain valid, the JIT generates optimized code under explicit runtime assumptions and inserts guards to verify those assumptions during execution. If a guard fails, execution transfers to a more general representation, typically an interpreter, and the system may later generate a different specialized version. This "speculate, guard, and de-optimize" model allows aggressive specialization without sacrificing correctness or robustness [43, 86].

This section surveys VM and JIT systems that leverage PE, ranging from interpreter specialization frameworks to full VM generators. A recurring theme is that successful systems make specialization explicit, bounded, and checkable, whether through annotations, profiling information, or generated compilers.

5.3.1. Truffle/Graal: Practical PE in Production

The Truffle framework [85] represents the most comprehensive practical application of PE to dynamic language VMs. Rather than requiring language-specific JIT compilers, Truffle implements a generic JIT compiler that generates code by partial evaluation of a guest language AST interpreter on the input AST program at runtime. This realizes the first Futamura projection in a production system.

The interpreter is written in a high-level managed language (Java) using the Truffle API, which provides building blocks for self-optimizing AST interpreters. AST nodes start in an uninitialized state and replace themselves with specialized versions geared towards specific input data. For example, an addition node may specialize to integer addition, then generalize to handle floating-point values, and eventually become fully generic. This progressive specialization means the AST encodes the minimal functionality needed for inputs encountered so far.

The connection to PE is made explicit through a set of core primitives that communicate information from the interpreter to the partial evaluator:

1. `@PEFinal`: instructs PE that a field is stable during compilation. PE treats the field like a final field, replacing loads with constants and performing constant folding. The interpreter may write the field, but only after calling `transferToInterpreter`, which transfers execution from optimized code back to the interpreter.
2. `@PEBoundary`: explicitly marks where PE should stop, separating interpreter code from runtime code. Calls to annotated methods remain as runtime calls, preventing code explosion from inlining large or rarely-used code.
3. `Assumption`: supports method-external invalidation of compiled code. When global information changes (e.g., a method is redefined), all compiled code depending on that assumption is invalidated. No runtime checks are emitted; instead, the compiler registers dependencies and the runtime invalidates code when assumptions are violated.
4. `transferToInterpreter`: causes a transfer from optimized machine code back to the interpreter. This intrinsic is intrinisified to a deoptimization node; since control never returns, all dominated code is dead and eliminated during PE.
5. `inInterpreter`: distinguishes interpreted from compiled execution. The intrinsic returns true in the interpreter but is intrinisified to false during PE, allowing profiling code to be excluded from compilation.

These primitives are sufficient to implement high-level VM features such as polymorphic inline caches, typed local variables, and on-stack replacement (OSR) for loops. For example, polymorphic inline caches are implemented by chaining cache entries using `@PEFinal` fields; PE treats the chain as constant, removing dispatch overhead. When the cache exceeds a maximum size, the whole list is replaced with a generic entry behind a `@PEBoundary`, preventing code explosion.

The PE algorithm follows standard online PE: it starts with the first instruction of the interpreter method, with the AST node object as a con-

stant. Virtual method calls are devirtualized, memory loads from `@PEFinal` fields are constant-folded, and dead branches are not parsed. The algorithm stops at `@PEBoundary` methods and indirect calls that cannot be devirtualized.

The Graal compiler then optimizes the resulting IR. Escape analysis is particularly important: the interpreter uses objects pervasively (e.g., to store local variables), and scalar replacement eliminates allocations when objects do not escape. Without escape analysis, performance degrades by an order of magnitude.

Evaluation on JavaScript, Ruby, and R [85] shows that the approach is competitive with best-in-class production VMs. Truffle's JavaScript implementation is within 17% of V8's performance (0.83x speedup), Ruby is 3.8x faster than JRuby, and R is 5x faster than GNU R. While warmup times are longer (30-90 seconds), peak performance is achieved within a minute, sufficient for server-side applications.

5.3.2. CompGen: The Second Futamura Projection in Practice

While Truffle's first Futamura projection (F1) achieves high peak performance, PE itself consumes significant compile time – up to 50% of overall compilation time. Latifi et al. [53, 52] address this with CompGen, an approach loosely based on the second Futamura projection that generates language-specific compilers from interpreter implementations.

CompGen generates source code for specialized versions of the partial evaluator for selected AST interpreter nodes ahead of time. If a compiler was generated for a particular method, PE uses it directly, removing method parsing and instruction dispatching overhead. Otherwise, the algorithm falls back to the first Futamura projection.

The compiler generation algorithm parses the target execute method of an AST node, propagates compile-time constants, schedules the IR, and outputs PE code that carries out simplification logic specific to each visited IR node. The generated code is Java source code (not bytecode), improving debuggability and development experience.

A hybrid mode combines the existing F1-based PE with CompGen for important AST nodes. Based on profiling, the hottest 750 AST interpreter methods (out of 10,000) are selected for compiler generation, yielding 3.1MB of bytecode and achieving 2× average PE time improvements across Octane benchmarks. Generating compilers for all methods would produce 40MB of bytecode, which is impractical, but the hybrid approach provides a practical trade-off between code size and PE speed.

The key insight is that CompGen leverages as much static information as possible at compiler generation time. Values available during generation (e.g., primitive literals, Truffle annotations) are optimized into the generated PE code. Values constant during PE time (e.g., child node arrays) are checked before instantiating IR nodes. Only values unknown until runtime are emitted as IR instructions. This shifts optimization from runtime to compiler generation time.

5.3.3. Deegen: Automatic JIT-Capable VM Generation

Deegen [87] takes VM generation to its logical conclusion: a meta-compiler that automatically generates a two-tier VM execution engine from C++bytecode semantics at build time. In Futamura's taxonomy, Deegen is a second Futamura projector: it generates language-specific compilers from interpreter implementations.

The user describes the execution semantics of each bytecode in C++using Deegen APIs. The description is compiled to LLVM IR using the unmodified Clang compiler. Deegen then performs analysis, transformation, and optimization at the LLVM IR level to generate:

1. An interpreter with continuation-passing, register pinning, bytecode specialization, dynamic quickening, and inline caching.
2. A baseline JIT with polymorphic inline caching, hot-cold code splitting, and OSR-entry.
3. The tier-switching logic connecting them into a self-adaptive system.

The interpreter uses LLVM's GHC calling convention to achieve register pinning and guaranteed tail calls. Six VM states are pinned in callee-saved registers, and the interpreter uses a custom stack to support stackful coroutines. Generic inline caching is implemented through a set of APIs that allow users to express arbitrary IC semantics; Deegen lowers these to optimized implementations customized for each VM tier.

The baseline JIT employs polymorphic IC based on self-modifying code. Through an improved Copy-and-Patch technique, Deegen automatically

generates the JIT compiler from bytecode semantics. The JIT compiles at 19.1 million Lua bytecodes per second, generating machine code at 1.62GiB/s. The generated JIT code includes inline caching, hot-cold code splitting, and jump-to-fallthrough optimizations.

The key technical innovation is the use of LLVM's `CallBr` IR node to model polymorphic IC inside LLVM. The `CallBr` node models an opaque piece of logic that takes in the IC key and branches to one of several LLVM labels representing the IC miss logic or IC hit logic. At the textual assembly level, Deegen identifies the assembly payload, removes it, and extracts the implementations of the main logic and each IC stub logic. This enables LLVM to generate the main logic while Deegen extracts IC stubs for dynamic patching.

Evaluation on LuaJIT Remake (LJR), a standard-compliant Lua 5.1 VM, shows that LJR's interpreter is on average 179% faster than PUC Lua and 31% faster than LuaJIT's interpreter. The baseline JIT has negligible startup delay and is on average 360% faster than PUC Lua, only 33% slower than LuaJIT's optimizing JIT, and faster on 13/44 benchmarks. The semantic descriptions require only 1,500 lines of logical code, compared to 4,500 lines of assembly in LuaJIT and 900 lines in PUC Lua.

5.3.4. Meta-Tracing: PyPy's Tracing JIT Compiler

Bolz et al. [15] pioneered the application of tracing JIT compilers to interpreters in the PyPy project. Instead of tracing the user program directly, PyPy's tracing JIT operates "one level down": it traces the execution of the interpreter, as opposed to the execution of the user program.

This approach has a fundamental problem: tracing one iteration of the interpreter's bytecode dispatch loop records only the execution of one opcode. Since it is unlikely that the same opcode is executed many times in a row, the resulting trace is not useful. The solution is to unroll the bytecode dispatch loop so that the unrolled version corresponds to a user loop.

The author of the language interpreter provides two hints:

1. **Green/red variable classification:** Green variables are those that should be considered part of the program counter of the language interpreter. Red variables are all others. The tracing interpreter adds the values of green variables to the position key, so the loop is only considered closed when these values are the same a second time.
2. **Backward jump identification:** The interpreter author indicates where backward jumps occur (at `can_enter_jit`). The tracing interpreter checks for a closed loop only at these points.

The green/red classification also enables optimization: green variables are checked for fixed values at the beginning of the loop, so operations on them (e.g., bytecode string indexing, program counter arithmetic) can be constant-folded. This optimization removes most of the language interpreter from the trace, leaving code that corresponds closely to the user program loop.

Implementation in PyPy involves three "versions" of the language interpreter embedded in the final executable: as executable machine code (the C translation), as bytecode for the tracing interpreter (for tracing), and as a fallback interpreter (for guard failures). The fallback interpreter runs until a safe point where the C version can resume.

Evaluation on a simple interpreter shows that full optimization with constant folding yields a 2.83× speedup over the pure interpreter. On a Python benchmark, PyPy with the tracing JIT outperforms CPython (6.54× speedup vs. 4.73× for CPython), though Psyco, a manually tuned specializing JIT, remains faster (15.57× speedup).

5.3.5. Specializing Interpreters with Tempo and DyC

Thibault et al. [77] demonstrate static and dynamic program compilation by interpreter specialization using the Tempo specializer for C. Tempo is an offline specializer that performs alias, side-effect, and dependency analyses to keep track of known values across procedures, data structures, and pointers.

For bytecode interpreters, specialization eliminates instruction dispatch by making the program counter statically known. Conditional branches present a challenge: if the branch condition depends on runtime data, the program counter cannot be statically known. The solution is to duplicate the continuation of an if-statement in its branches only when necessary, extracting the interpretation loop into a separate procedure parameterized by the program counter.

Evaluation on three bytecode interpreters shows significant speedups: specialization of the Java Virtual Machine (Hi interpreter) yields up to 4× speedup over the unmodified Hi and 22× speedup over Hi with "quick" instructions disabled. For the Berkeley Packet Filter (BPF), specialization yields 1.7× speedup. For domain-specific languages like PLAN-P and GAL, specialization yields 10-85× speedup, with the specialized PLAN-P bridge achieving 96% of hand-written C performance.

Noel et al. [64] extend this work with template-based run-time specialization. The key idea is to use code templates: fragments of code containing holes to be filled with values calculated during specialization. Templates are automatically generated from ordinary programs and optimized before run time using an existing C compiler. At run time, the specialized code is generated from this precompiled code by copying templates and filling holes.

The run-time specializer first checks a local dynamic-code cache for a previously compiled version with the current static values, reusing it if found. Otherwise, it evaluates static computations, emits machine code by copying templates and filling holes, saves the generated code in the cache, and executes it. This approach introduces little run-time overhead – dumping templates primarily consists of a memory copy operation.

Evaluation on scientific and graphics code shows that run-time specialized programs run up to 10× faster and are nearly as fast as fully optimized programs (80% on average). Break-even points range from 3 to 87 executions, demonstrating that the approach is practical for many applications.

5.3.6. Dynamic PE for Reflective Languages

Sullivan [70] introduces dynamic PE, where PE happens as a side effect of evaluation. At runtime, an expression is evaluated with respect to an environment containing both dynamic bindings (values) and static bindings (types). In addition to producing a value, dynamic PE produces a residual version of the expression based on the types in the environment.

The key insight is that elements of a meta-object protocol (MOP) that are exposed and mutable often change only rarely at runtime. By guardedly treating these aspects as constant, dynamic PE can apply PE techniques to remove reflective overhead. Generic functions and multiple dispatch are used to cache specialized versions: adding specialized methods to a generic function at runtime corresponds to polynomial specialization in compile-time PE.

Sullivan et al. [71] apply these ideas to dynamic native optimization of interpreters using the DynamoRIO framework. The key insight is that a pair (native PC, logical PC) uniquely identifies the current computation point. The interpreter writer supplies the logical PC value through an API, and DynamoRIO identifies immutable program regions and trace-constant addresses.

The system records traces of native instructions, then partially evaluates them with respect to the in-memory representation of the program being interpreted. Dereferences from the bytecode array indexed by the logical PC are statically folded to constants; conditional branches based on now-constant values are removed entirely; and direct increments to the logical PC are identified and tracked, enabling continued PE.

Evaluation on a simple VM (TinyVM) shows that logical tracing recovers performance lost by regular DynamoRIO, and constant propagation further improves performance. The hottest trace exits at the end 97% of the time, compared to only 66% with regular tracing.

5.3.7. Staged Compilation: Compiling Once, Optimizing Everywhere

Chambers et al. [22] propose staged compilation as a way to combine high runtime code quality with low compilation overhead. Compilation is organized as a series of stages, corresponding to separate compilation, library linking, program linking, and runtime execution. Each optimization can be performed at any stage, with earlier stages performing preplanning work that enables the final stage to complete the optimization quickly.

The SCF (Staged Compilation Framework) system automatically constructs staged compilers from the implementation of a single-stage optimizer. The developer writes a series of normal program optimizations as program transformers in a side-effect-free subset of ML. The stager partially evaluates each optimization given a partial description of its input, producing a specialized optimization pipeline. Experimental results show that automatic staging can give order-of-magnitude speedups to optimization pipelines, at a reasonable early-stage compilation time cost.

5.3.8. GraalVM Native Image: Ahead-Of-Time (AOT) Compilation with Heap Snapshotting

Wimmer et al. [84] present GraalVM Native Image, an approach that combines points-to analysis, application initialization at build time, heap snapshotting, and AOT compilation. This yields lightweight, self-contained executables with fast startup and low memory footprint.

The system operates under a closed-world assumption: all Java classes must be known and available at build time. At build time, a points-to analysis determines reachable program elements (classes, methods, fields). Initialization code can run at build time, allocating objects that form the image heap – a pre-initialized part of the heap that is part of the executable and available immediately at application startup.

The points-to analysis and heap snapshotting are executed iteratively until a fixed point is reached. Initialization code can access the current points-to analysis state via API methods (e.g., `isReachable`), enabling data structures to be optimized for exactly the reachable parts of the application. For example, a service loader Feature iterates all service interfaces, checks if each is reachable, and registers reachable service implementations for reflection, triggering subsequent analysis iterations.

The image heap uses references relative to the start of the image heap, allowing memory-mapping without relocation. Multiple processes (or isolates within one process) share the image heap using copy-on-write memory mapping, reducing overall memory footprint. The garbage collector only needs to scan a small, contiguous part of the image heap for root pointers. Evaluation [84] shows that Native Image improves startup performance by up to two orders of magnitude compared to the Java HotSpot VM, while preserving peak performance.

5.3.9. weval: Whole-Program Compilation via Partial Evaluation

Fallin and Bernstein [37] introduce weval, a PE transform that compiles whole program functions ahead-of-time without tracing or profiling. The transform operates on a function body represented as a control-flow graph (CFG) of basic blocks in SSA form, and can apply to almost unmodified interpreters in `C` or C++.

The key innovation is *context specialization*: the interpreter can selectively introduce context via `update_context(pc)` intrinsics. When performing an iterated dataflow analysis for constant propagation, this causes analysis to flow to successor blocks in a new context, avoiding the "meet-over-all-paths" trap that would otherwise conclude that the program counter is not constant.

The algorithm is worklist-driven and clones both basic blocks and SSA values separately for each context. It performs constant propagation as it runs, tracking state per cloned SSA value. Branch folding simplifies conditional branches when inputs are known constants. Contexts for cloning are nested sequences of user-specified values/loop counters, interned to short integer identifiers for efficiency.

The transform also provides state optimizations: `load_register(index)` and `store_register(index, value)` intrinsics allow the interpreter to communicate which memory reads and writes correspond to true SSA locals. `read(index, address)`, `write(index, address, value)`, and `flush()` intrinsics provide a write-back cache abstraction for in-memory state. These intrinsics enable the transform to lift in-memory storage to SSA when possible, reducing memory operations.

Evaluation on SpiderMonkey's JavaScript interpreter shows a 2.17× speedup over the Portable Baseline Interpreter (PBL) with inline caches. State optimizations contribute a 1.37× speedup, eliding 84% of loads and 76% of stores. On PUC-Rio Lua, a minimal port (173 lines of changes) yields a 1.84× speedup.

5.3.10. Basic-Block Versioning and Kernel Bytecode Specialization: YJIT and eBPF

Two further production systems illustrate how widely PE-related techniques are deployed outside research VMs. Chevalier-Boisvert et al.'s YJIT [24] is a JIT compiler shipped inside CRuby, the reference Ruby implementation, based on *lazy basic-block versioning*: rather than compiling a method once for all possible input types, YJIT lazily generates a specialized version of each basic block for the type context observed at that point, deferring compilation of successor blocks until they are actually reached. This is a restricted, online form of PE in which the specialization boundary is the basic block, the static information is the locally observed type context, and the specialization space is bounded by construction, since only reachable, observed contexts are ever compiled. YJIT delivers consistent double-digit percentage speedups

over the CRuby interpreter in production use, including at Shopify's storefront infrastructure [24].

At the kernel level, the extended Berkeley Packet Filter (eBPF) subsystem in Linux compiles user-supplied bytecode programs to native machine code via an in-kernel JIT, after a verifier statically bounds their behavior (e.g., rejecting unbounded loops and out-of-bounds memory access). Wang et al.'s Jitk [81] shows that this bytecode-to-native compilation step can be built on a verified, PE-like pipeline: system-call and packet-filtering policies written in a high-level rule language are compiled down to native instructions with a machine-checked proof of correctness. eBPF's verifier and JIT are rarely described using PE terminology, yet they realize the same pattern documented throughout this survey: an explicit staging boundary (bytecode load time versus packet- or syscall-handling time), a bounded specialization space (enforced by the verifier), and effect discipline (the verifier's restrictions on memory access and control flow).

These two systems, together with Truffle/Graal and PyPy discussed above, and with speculative type-specialization in JavaScript engines such as V8's TurboFan compiler [17], show that techniques closely related to PE are widespread in production compilers and runtimes. What remains comparatively rare is PE in its most general form: a single, explicitly named pass that specializes arbitrary programs with respect to arbitrary static information, independent of a specific language runtime or kernel subsystem.

5.3.11. Summary

VM and JIT systems that leverage PE consistently exhibit the four structural patterns identified in Section 6. *Explicit staging boundaries* appear in Truffle's annotations (`@PEFinal`, `@PEBoundary`) [85], in Deegen's bytecode semantics [87], in PyPy's green/red variable classification [15], and in YJIT's per-block type contexts [24]. *Bounded specialization policies* are evident in Truffle's compilation thresholds and deoptimization, in CompGen's hybrid compilation mode [53], in PyPy's profiling counters, and in eBPF's verifier-enforced program bounds [81]. *Structure-exposing representations* include Graal IR, LLVM IR, Thorin's CPS IR, the SSA CFGs used by weval [37], and eBPF's bytecode IR. *Effect discipline* is present in Truffle's `@PEFinal` fields (interpreter can write, compiled code cannot), in weval's lazy flushing intrinsics, in Tempo's alias and side-effect analyses [77], and in eBPF's verifier-enforced restrictions on memory access and control flow.

5.4. Program Optimization via PE

Beyond DSL compilation and VM implementation, PE has proven effective for a range of program optimization tasks that transcend individual application domains. These tasks share a common goal: automatically specializing programs to eliminate overhead that arises from genericity, abstraction, or configuration variability. This section surveys diverse optimization applications of partial evaluation, including array bounds check elimination, database query optimization [39], program debloating, pattern-match compilation, and simulation acceleration. A recurring theme is that PE provides a principled, semantics-preserving approach to program optimization that complements traditional compiler techniques.

An early example outside the niches above is Sestoft's derivation of a compiler for ML-style pattern matches [69]. He first instruments a naive general matcher so that it collects and exploits extra information, and then shows that partially evaluating the instrumented matcher with respect to a given match produces an efficient specialized matcher, an instance of the first Futamura projection. The instrumentation step is an example of adjusting a program once so that it specializes well (Section 3).

5.4.1. Array Bounds Check Elimination

Array bounds checking is a critical safety feature in languages like Java, but it imposes significant runtime overhead. Popeea, Xu, and Chin [65] propose a precise inference and specialization system for eliminating redundant array bounds checks in imperative programs. Their approach combines forward relational analysis with backward precondition derivation to achieve both precision and practicality.

The key innovation is a three-way classification of checks. A check is *safe* if it is redundant under the program state at its location. It is *partially-safe* if it may become redundant under an extra precondition that can be propagated to callers. It is *unsafe* if no precondition can guarantee its safety. Partially-safe checks are special because they can be propagated across method boundaries, exploiting calling contexts for elimination.

The inference system uses a form of dependent types with size relations to capture symbolic program states. A forward analysis computes postconditions for each method, while a backward analysis derives safety preconditions for each check. For recursive methods, a fix-point analysis computes both the method postcondition and a recursive invariant relating parameters across calls. The integration of forward and backward analysis enables precise interprocedural reasoning that purely forward analyses cannot achieve.

Evaluation on 13 benchmark programs, including SciMark and Linpack, shows that the inference system eliminates 100% of array checks for most programs. The combination of forward postcondition derivation with backward precondition derivation improves check elimination from 84% (with prior forward-only analyses) to 100%.

The flexivariant specialization scheme supports a trade-off between optimization and code size. By weakening configurations, the system can reduce code size with minimal loss in performance. This is particularly important for polyvariant specialization, where creating multiple specialized versions of a method can lead to code explosion.

5.4.2. Polyglot Query Optimization

Modern data processing systems increasingly support polyglot queries that combine relational operators with user-defined functions (UDFs) in languages like Python, JavaScript, and Java. Grulich, Zeuch, and Markl [39] identify a fundamental impedance mismatch between the declarative paradigm of SQL and the imperative paradigm of UDFs, leading to substantial performance overhead, between 3.6× and 124× across commercial and open-source systems.

Babelfish [39] addresses this challenge through a unified IR (the Babelfish-IR) that captures both relational operators and UDFs from different programming languages. The IR represents queries across four abstraction levels: logical query plan, physical query plan, physical operators, and instruction graph. This enables holistic optimizations across operator and language boundaries.

The core optimization is *IR-based operator fusion*, which eliminates function calls and intermediate allocations in three steps. First, *specialization* uses Truffle's PE to specialize operator implementations according to runtime parameters, devirtualizing function calls, and binding data types. Second, *inlining* eliminates function calls between operators within a pipeline, creating a unified instruction graph. Third, *scalar replacement* eliminates intermediate objects (BFRecords) by rewriting field accesses with local variables and direct memory references.

Additional optimizations include *polyglot predication*, which identifies and eliminates predicates across operators using profiling information to derive filter selectivity; *latch-reduction*, which replaces latches with atomic operations for stateful UDFs under contention; and *workload specialization* through PolyglotRopes for efficient text processing and lazy parsing for raw data formats (CSV, Arrow).

Evaluation across TPC-H queries and data science workloads shows that Babelfish outperforms MonetDB and Pandas by up to one order of magnitude and reaches the performance of hand-optimized C++implementations. For text-heavy workloads, PolyglotRopes eliminate intermediate string allocations, yielding speedups of 10× or more over naive string processing. On stateful UDFs with high contention, latch-reduction improves performance by up to 25×.

5.4.3. Compiler-Assisted Application Specialization

Program debloating aims to reduce the attack surface and improve performance of bloated applications by removing unused functionality. Alhanahnah et al. [6] present LMCAS (Lightweight Multi-Stage Compiler-Assisted Application Specialization), a debloating framework based on PE principles.

The key insight is that programs typically consist of two components: *configuration logic*, in which inputs are parsed, and *main logic*, which implements the program's functionalities. The boundary between these components is called the *neck*. LMCAS captures a partial program state by interpreting the configuration logic based on supplied inputs, then applies PE optimizations to generate a specialized program.

A *neck miner* automates neck identification using heuristic and structural analysis. The heuristic analysis finds likely locations (e.g., the end of command-line parsing), while the structural analysis verifies that candidate locations are articulation points in the control-flow graph that dominate all subsequent statements. The neck miner requires only minimal manual intervention, a one-time effort for each program.

Partial interpretation uses KLEE, a symbolic execution engine [20], to execute the program up to the neck with concrete inputs. This captures a precise partial state comprising variables influenced by the supplied inputs and their corresponding values. The state is stored in a dictionary mapping variables to constants.

Constant conversion incorporates the captured partial state into the program. Unlike standard optimizations, this transformation is performed *upstream* of the neck: uses of variables in the configuration logic are replaced with constants based on values captured at the neck. The conversion handles base types, pointers, structs, and strings, using def-use information to replace all occurrences.

Multi-stage simplification applies standard LLVM passes (constant propagation, control-flow simplification) followed by a custom cleanup pass that removes unused functions, global variables, and stack variables. The cleanup pass handles indirect calls by checking function uses before removal, preventing the deletion of functions invoked via function pointers.

Evaluation on 26 programs from Coreutils, Binutils, Diffutils, Nginx, and Tcpdump demonstrates that LMCAS is effective. It runs 1500×, 4.6×, and 1.2× faster than CHISEL, RAZOR, and OCCAM, respectively, and achieves 25% binary size reduction.

5.4.4. Simulation Acceleration for Biochemical Reaction Networks

Stochastic simulation of biochemical reaction networks is computationally intensive, with models often requiring days to simulate even on high-performance resources. Koster et al. [51] apply PE via code generation to accelerate these simulations, demonstrating that specialized simulators can outperform established tools by more than an order of magnitude.

The approach starts with a model defined in a rule-based language (BioNetGen or a custom DSL). A generic simulator implements the stochastic simulation algorithm, specifically the optimized direct method with dependency graphs and reaction ordering. The code generator transforms this generic simulator into a model-specific specialized simulator by partially evaluating the model description.

The generated simulator hardcodes all reactions, storing propensities and population sizes in stack arrays rather than heap-allocated dynamic structures. In the main step function, the index of the next reaction is used to jump directly to the execution of the corresponding reaction, with hard-coded updates to affected reactions. The compiler can inline rate calculation functions, eliminating all indirection.

Profile-guided optimization infers an optimal reaction ordering from a pilot run. Reactions are sorted by firing frequency, reducing reaction selection time. This optimization, combined with PE, targets different phases of the algorithm: PE improves reaction execution, while sorting improves reaction selection. In combination, they achieve speedups of up to 16× over the generic simulator.

The system handles large models by generating multiple source code files and using link-time optimization. This approach reduces compilation time and memory requirements while preserving the optimization potential of whole-program compilation. A key result is that PE is most effective for the *dependency graph*, the hardest part to optimize generically. When the dependency graph is partially evaluated, the compiler can optimize the interaction between reaction execution and dependencies, providing the bulk of the performance improvement.

5.4.5. Cardiac Cell Model Simulation

Cooper et al. [30] apply PE to optimize simulations of cardiac ionic cell models expressed in CellML, an XML-based modeling language for biological systems. These simulations are computationally intensive, with whole-heart simulations requiring days on supercomputers.

The CellML format describes cell models as systems of ordinary differential equations, with mathematics expressed in MathML. The authors convert CellML models to Scheme programs using the Haskell XML Toolbox, specialize them using the PGG partial evaluator, then parse the specialized Scheme to generate optimized CellML or C++code.

PE is well-suited to this domain for several reasons. First, many computations depend only on model parameters, not on state variables, and can be performed once. Second, the mathematics in CellML maps naturally to assignment statements, making specialization straightforward. Third, the simple structure of CellML models (no user-defined recursive functions) avoids difficult binding-time analysis decisions. Fourth, PE provides a solid theoretical basis for correctness, which is crucial in scientific computing where confidence in simulation results is paramount.

The authors also automate *lookup table* generation. Expressions containing only the transmembrane potential and expensive functions (exponential, trigonometric) are replaced with table lookups. PE simplifies the code structure, enabling the lookup table analysis to recognize larger expressions suitable for transformation. This synergy produces an almost 5× speedup over the non-optimized code, compared to 1.7× for lookup tables alone.

5.4.6. Fixed-Point Simulation Acceleration

Edwards [34] explores using program specialization to accelerate SystemC fixed-point simulation. Fixed-point arithmetic is widely used in signal processing applications as a prelude to hardware implementation, but generic simulation libraries are inefficient because they must support arbitrary word sizes, rounding modes, and overflow behaviors.

The Tempo program specializer [28] is applied to fixed-precision arithmetic routines, specializing them for concrete word lengths, rounding modes, and overflow behaviors. The specialization predicates are expressed using Prespec, a semi-graphical front-end that guides the specializer. The specialized code eliminates all decisions derived from these parameters, performing them at compile time rather than during simulation.

A key finding is that the SystemC libraries did not specialize well: their choice of number representation (a multi-word representation with the binary point between words) and use of "mostly static" variables prevented the specializer from fully eliminating overhead. A recoded library using a more efficient representation specialized much better, achieving 3–7× speedups.

The approach is compared to FRIDGE, a dedicated fixed-point code generator. While FRIDGE requires hand-coded code generation algorithms, program specialization allows library designers to write generic algorithms in their favorite language and still gain speed advantages. The flexibility of program specialization is its main advantage: new algorithms do not require changes to the code generator.

5.4.7. Optimizing Generic Programming

Generic programming systems like Scrap Your Boilerplate (SYB) provide powerful abstractions for traversing data structures, but are often an order of magnitude slower than handwritten, non-generic code. Adams et al. [2] present a domain-specific optimization that transforms SYB-style code to be as fast as handwritten code.

The optimization is a specialized form of PE that uses type information to direct inlining and symbolic evaluation. The key insight is that expressions with undesirable types (e.g., `TypeRep`, `TyCon`, `Data` and `Typeable` dictionaries) cause runtime type checks and dictionary dispatches that slow down SYB. By eliminating these types through aggressive inlining and symbolic evaluation, the optimization removes the runtime overhead.

Benchmarks show the optimization improves SYB performance by an order of magnitude, matching handwritten, non-generic code. For MapTree and MapLogic, the specialized SYB code actually ran faster than the handwritten version due to specialization of the function being mapped over the structure. For RenumberIntLogic, the specialized code is 2.2× slower due to GHC not optimizing monadic operations as well as it does in the handwritten version.

5.4.8. Synchronous Concurrent Program Optimization

Zeng [88] demonstrates the application of specialized PE to optimize code generation from DSLs with concurrent semantics. The dissertation addresses three distinct compilation challenges for deterministic concurrent languages (Esterel and SHIM), each requiring a custom PE technique tailored to the language's computational model. This work illustrates a key thesis: general-purpose partial evaluators are insufficient for DSLs because the unique semantics of each language demand model-specific optimization strategies.

The first technique eliminates concurrency overhead in Esterel programs executing on single-threaded processors. A program dependence graph (PDG) representation is restructured by inserting guard variables based on a heuristic topological schedule, minimizing context switches between concurrent threads. The approach first breaks source code into small concurrent pieces, which paradoxically provides the scheduler more choices to form larger atomic blocks, thereby reducing switching overhead. Experimental results show the generated code runs four to six times faster than the Esterel V5 compiler.

The second technique enables separate compilation of Esterel modules involved in communication dependency cycles. The algorithm transforms two-valued programs into three-valued (0, 1, unknown)

equivalents that respond correctly when inputs are undefined. Performing an abstract three-valued simulation with memoization of equivalent states, the compiler generates code that tolerates unknown inputs without manual programmer intervention. While the technique is practical for modest-sized programs, it can produce exponential code growth for large systems.

The third technique unrolls recursion in SHIM programs to enable hardware synthesis. By decomposing functions into extended basic blocks and performing constant propagation and function inlining, the algorithm eliminates recursion when the depth is bounded, replacing dynamic recursive structures with static concurrent implementations. Applied to FFT and pipelined FIFO examples, the technique generates code with an acyclic call graph and bounded resource usage.

A recurring theme across all three techniques is that effective optimization requires deep understanding of the DSL's semantics: synchronous communication in Esterel and rendezvous-style communication in SHIM demand fundamentally different specialization strategies. The work reinforces that specialized PE is not merely a performance optimization but an enabler for compilation problems (separate compilation, recursion elimination) that are otherwise difficult to solve.

5.4.9. Summary

Program optimization via PE spans diverse application domains, from array bounds checking to database queries to biological simulation. Despite their differences, these applications share common themes: the presence of static information that can be exploited (word lengths, model parameters, configuration inputs), the elimination of interpretive or generic overhead, and the importance of semantics preservation. The approaches range from type-based inference [65] to IR-based fusion [39] to neck-based specialization [6] to model-specific DSL compilation [88]. A key insight, which is not specific to program optimization (Section 3), is that effective specialization often requires code to be written with specialization in mind, a modest upfront cost that pays dividends in performance.

6. Structural Patterns for Practical PE

Section 5 surveyed application niches, in which PE has demonstrated practical success. Although these niches differ substantially in their goals, programming models, and implementation techniques, they exhibit four recurring structural patterns. First, *explicit staging boundaries* distinguish computations that may be performed during specialization from those that must remain for execution. Second, *bounded specialization policies* constrain how aggressively specialization may proceed and limit the search space of possible residual programs. Third, *structure-exposing representations* preserve semantic information in forms that make specialization predictable and local. Fourth, *effect discipline* identifies computations that may safely be executed early while preserving observable program behavior.

6.1. Methodology for Pattern Identification

The four patterns were extracted from the surveyed corpus assembled in Section 4 through a qualitative coding process. For each of the corpus works, we recorded the mechanisms that the authors themselves identify as enabling practical specialization: staging annotations, type-level distinctions between static and dynamic values, runtime guards and deoptimization, compilation thresholds, memoization and generalization, restrictions on computational effects, and properties of the program representation over which specialization operates. Recurring mechanisms were then grouped into candidate patterns, and the grouping was iterated: overlapping candidates were merged, and a candidate was retained only if it appeared, as either a primary or a supporting enabling mechanism, in every one of the four niches. This process converged on the four patterns examined below; no additional candidate achieved comparable cross-niche support. The coding was performed by the authors independently and disagreements resolved by discussion.

Table 1 summarizes the outcome, distinguishing niches in which a pattern is the primary enabling mechanism from those in which it plays a supporting role.

The following subsections examine each pattern in detail. For each pattern, we first define it, then present evidence of its presence across niches, and finally discuss its role in enabling practical specialization.

6.2. Explicit Staging Boundaries

A recurring characteristic of successful PE systems is a clear separation between computations

Application Niche	Staging	Effects	Structured IR	Bounded
Multi-Stage Programming	•	•	○	○
Domain-Specific Languages	•	•	•	•
Virtual Machines & JIT	•	•	•	•
Program Optimization	•	•	•	•

• Primary enabling pattern ○ Supporting pattern

Table 1: Mapping application niches to the four structural patterns

that may be performed during specialization and computations that must remain residual. Classical PE attempts to infer this separation automatically through binding-time analysis. Many practical systems instead expose it explicitly through language constructs, type systems, compiler phases, deployment configurations, or toolchain boundaries. Examples include GraalVM's ahead-of-time image build versus runtime execution [84] (Section 5.3) and a container image build stage versus its deployed runtime environment.

Making the staging boundary explicit substantially reduces the uncertainty faced by the specializer. Rather than determining which computations are sufficiently stable to execute early – that is, which values are fixed for every execution of the residual program (Section 2) – the specializer follows boundaries established by the surrounding system. This improves both predictability and engineering reliability while reducing dependence on sophisticated whole-program analysis.

Evidence across niches. This pattern appears across all four application niches with different mechanisms. In multi-stage programming (MSP), Taha and Sheard's MetaML [74] uses brackets `<e>` and escapes `~e` as explicit staging constructs, while Rompf and Odersky's LMS [68] uses the `Rep[T]` type constructor to distinguish staged from unstaged values. Swadi et al. [72] make staging boundaries explicit through monadic combinators that distinguish code-generation time from execution time.

In DSL compilation, Leissa et al. [57] use `@` annotations to mark functions for specialization, putting the programmer in "total control over which parts of the program should be specialized under which conditions". Jovanovic's Yin-Yang [48] uses Scala macros to create an explicit translation boundary between shallow (development) and deep (production) embeddings. Carette et al.'s tagless final approach [21] separates the `Symantics` interface from its implementations, creating a staging boundary between program representation and interpretation.

For the Truffle framework, Wurthinger et al. [85] use `@PEFinal`, `@PEBoundary`, and `transferToInterpreter` as explicit primitives that communicate information from the interpreter to the partial evaluator. Deegen [87] generates a two-tier VM execution engine from explicitly specified bytecode semantics. PyPy's meta-tracing [15] uses explicit hints (`JitDriver`) to mark which interpreter variables constitute the program counter.

In program optimization, Alhanahnah et al. [6] introduce the *neck* concept, a splitting point where configuration logic hands off control to main logic, making the staging boundary explicit through program partitioning. Koster et al. [51] use code generation as the boundary between generic and specialized simulators. Cooper et al. [30] convert CellML models to Scheme programs, using a partial evaluator to create the staging boundary. Zeng [88] employs multiple staging boundaries tailored to DSL semantics: the program dependence graph separates control and data dependencies for scheduling, the schedule itself serves as the staging order for restructuring, and for separate compilation, the boundary between two-valued and three-valued execution defines the specialization frontier. The fixed-point nature of synchronous execution provides a natural staging boundary: all computations within a clock cycle can be specialized, while interactions with the environment remain dynamic.

Role in success. Across these systems, explicit staging boundaries serve a common purpose: they eliminate the need for the specializer to infer which computations are static and which are dynamic. By making this boundary explicit, the system designer can ensure that the specializer has reliable information about what can be evaluated early. As Leissa et al. [57] observe, this "makes PE predictable for the programmer because it eliminates the influence of the imprecision of static analyses that guide other evaluators."

6.3. Bounded Specialization Policies

A second recurring pattern is that successful PE systems constrain the space of specialization decisions. Classical PE exposes numerous interdependent choices, including the static-dynamic split, unfolding strategy, polyvariance, generalization, and resource budgets. In unrestricted software, jointly optimizing these decisions is difficult because their costs and benefits depend on both program structure and workload.

Practical systems therefore bound specialization rather than attempting unrestricted optimization. Such bounds may be supplied by the specialization algorithm itself through memoization, generalization, termination checks, and explicit resource limits [47, 58]. They may also be supplied externally through restricted languages, deployment configurations, or runtime speculation with guarded deoptimization.

Evidence across niches. In MSP, Swadi et al. [72] explicitly address code duplication: "Without appropriate control, unrestricted unfolding may diverge, generate excessive code, or create an impractical number of specialized variants." Their CPS transformation with explicit let-binding bounds code growth. Asai [10] shows that binding-time analysis with a propagation algorithm bounds the number of specialization decisions.

In DSL compilation, Leissa et al. [57] rely on user annotations to bound specialization: "A common problem of PE is to decide which parts of the program should be specialized... a partial evaluator which is too greedy might diverge... For this reason, our approach requires user annotations." Yin-Yang [48] uses compilation guards and code caches to manage dynamic compilation, ensuring that recompilation occurs only when necessary. Carette et al. [21] explicitly address the trade-off: "unfolding too little can degrade the residual code, whereas unfolding too much risks nontermination."

Among the VM/JIT systems surveyed in Section 5.3, Truffle [85] uses compilation thresholds and deoptimization to bound PE: "PE is an expensive operation, therefore waiting for a stable state before initiating PE is important to avoid repeated PE of the same function." CompGen's hybrid mode [53] bounds code growth by generating compilers only for hot methods. PyPy's metatracing [15] uses profiling counters to bound tracing to frequently executed loops. Deegen's baseline JIT [87] prioritizes compilation speed and employs polymorphic IC as its only high-level optimization.

In program optimization, Popeea et al. [65] use flexivariant specialization to trade off optimization against code size: weakening configurations can reduce code size with minimal loss in performance. Alhanahnah et al. [6] use memoization to avoid redundant specialization. Koster et al. [51] use profile-guided optimization to bound the reaction ordering search. Edwards [34] uses specialization predicates to bound the set of word lengths, rounding modes, and overflow behaviors that must be handled. Zeng [88] uses memoization of equivalent simulation states to prevent exponential code explosion in three-valued compilation, and imposes a user-defined limit on function unrolling to bound recursion depth in SHIM programs. This dual strategy, memoization for state-space exploration and explicit limits for recursion depth, illustrates how bounded policies operate at multiple levels within a single system.

Role in success. Bounded specialization policies ensure that PE terminates, generates code of manageable size, and produces predictable performance improvements. As Swadi et al. [72] note, without appropriate control, "unrestricted unfolding may diverge, generate excessive code, or create an impractical number of specialized variants." The mechanisms vary (e.g., memoization, user annotations, compilation thresholds, and profile guidance) but all serve the same purpose: transforming specialization from an open-ended search into a controlled compilation process whose cost and behavior remain predictable.

6.4. Structure-Exposing Representations

Successful PE systems also rely on program representations that preserve the semantic structure needed for specialization. Rather than operating directly on arbitrary source programs, they expose computations through structured representations such as abstract syntax trees, intermediate representations, traces, graphs, bytecode, or domain-specific compiler IRs.

Such representations reduce the amount of analysis required before specialization can begin. Control flow, data dependencies, domain operations, and transformation opportunities become explicit rather than implicit, allowing specialization to operate locally and predictably. Modern compiler infrastructures increasingly follow this philosophy by preserving high-level semantic information until it can be exploited by optimization and code generation.

Evidence across niches. In multi-stage programming, LMS [68] uses the `Rep[T]` type constructor to expose staging boundaries in the type system. BuildIt [19] constructs ASTs through operator overloading and symbolic execution. Swadi et al. [72] use CPS transformation to expose control structure for effective specialization. Allais [7] uses intrinsically typed syntax with De Bruijn indices, where the representation itself guarantees well-scopedness and well-typedness.

In DSL compilation, AnyDSL [57] uses Thorin, a CPS-based IR that is "minimalistic... its only constructs are continuations and primops," reducing the variety of constructs the partial evaluator must handle. Carette et al. [21] represent object terms using combinator functions rather than data constructors, enabling multiple interpretations without modifying the term representation. Yin-Yang [48] uses `Rep[_]` types in LMS-based DSLs to explicitly mark future-stage computations. Brady and Hammond [18] index EDSL representations by their types to eliminate runtime tagging.

Among the VM/JIT systems surveyed in Section 5.3, Truffle [85] uses the Graal IR as a structured representation: a "high-level compiler intermediate representation" that supports deoptimization as a first-class primitive. Deegen [87] uses LLVM IR as the representation for analysis, transformation, and optimization. PyPy [15] uses RPython IR as the input to its tracing JIT. weval [37] operates on SSA-based CFGs, which expose control flow and data dependencies.

In program optimization, Babelfish [39] uses a unified IR spanning four abstraction levels, from logical query plan to instruction graph. Popeea et al. [65] use dependent types with size relations to capture symbolic program states. Alhanahnah et al. [6] use LLVM IR for analysis and transformation. Koster et al. [51] use TOML as a machine-readable model representation. Zeng [88] uses two distinct structured representations: program dependence graphs (PDGs) expose concurrency and data dependencies for scheduling, revealing communication patterns that determine where context-switching overhead can be eliminated; and graph code (GRC) exposes the control-flow structure for three-valued simulation, making reconvergence points explicit so that equivalent simulation states can be identified and shared.

Role in success. Structure-exposing representations enable predictable specialization by making program structure explicit. As Leissa et al. [57] observe, "Thorin's design is minimalistic... This reduces the variety of different constructs the partial evaluator has to handle to a minimum." Similarly, Carette et al.'s [21] combinator representation enables multiple interpretations from a single term representation. Across all niches, the pattern is consistent: successful systems use representations that expose the information the specializer needs, rather than forcing the specializer to recover it through whole-program analysis. We do not claim that such analysis is always expensive: its cost and precision vary with the program and the analysis, and replacing analysis by search has costs of its own (Section 8.3). The claim is only that exposing the structure removes this cost, and the risk of imprecision on dynamic features, from the specializer.

6.5. Effect Discipline

Finally, successful PE systems distinguish computations that may safely be executed during specialization from those whose observable behavior must be preserved for execution. This distinction becomes increasingly important in modern software, where mutation, input/output, exceptions, concurrency, and interactions with external libraries are common.

Effect discipline does not necessarily require a purely functional language. Instead, it requires that the system identify where specialization may safely propagate values, inline computations, or eliminate abstraction, and where it must preserve sequencing and residual execution. Different systems achieve this through different mechanisms, including type-and-effect systems, staging restrictions, runtime guards, explicit effect analyses, or carefully designed execution models.

Evidence across niches. In MSP, Asai and Kameyama [11] show that cross-stage persistence (CSP) must be carefully bounded: unrestricted CSP of higher-type values leads to the absence of a best staging annotation. Swadi et al. [72] use a state monad that manages memoization tables as explicit state, isolating effects during specialization. Biri [13] uses Idris's quantitative types to guarantee that type information is erased at runtime.

In DSL compilation, AnyDSL [57] performs SSA construction before PE, translating mutable variables into functional variables that are non-destructively updated: "Destructive updates... are ignored by the evaluator." Brady and Hammond [18] treat the IO monad as an EDSL, ensuring that I/O operations are constructed rather than

executed during specialization. Yin-Yang [48] uses Scala FX effects to annotate side-effecting methods, preserving effect ordering during optimization.

Among the VM/JIT systems surveyed in Section 5.3, Truffle [85] uses `@PEFinal` fields: "The interpreter is allowed to write the field, but compiled code must not write to it. Every write must be preceded by a call to transferToInterpreter." weval [37] uses lazy flushing intrinsics that "grant the weval tool permission to diverge in semantics in controlled ways (lazy flushing of in-memory state with user-denoted synchronization points)." Hutchins [45] observes that mutable state is the "Achilles Heel of partial evaluators" and proposes treating aspects as functions over immutable classes.

In program optimization, Popeea et al. [65] track side effects through alias and dependency analyses. Edwards [34] provides guidelines for writing effect-disciplined library code that specializes well. Alhanahnah et al. [6] use PE to safely capture program state without executing side effects. Cooper et al. [30] handle singularities by replacing problematic expressions with piecewise statements. Zeng [88] leverages the discipline inherent in deterministic concurrent DSLs, Esterel's synchronous communication and SHIM's rendezvous semantics, to safely perform aggressive transformations like context-switch elimination and recursion unrolling without compromising correctness.

Role in success. Effect discipline ensures that specialization does not alter the observable behavior of the program. By identifying which computations may safely be executed early, successful PE systems can perform aggressive optimization without risking semantic violations. As Brady and Hammond [18] note, "IO operations remain pure, so there is no need to treat them specially" – by treating IO as an EDSL, they isolate effects from specialization. Across all niches, the pattern is consistent: successful systems make effect boundaries explicit rather than relying on the specializer to infer them.

6.6. Discussion: Patterns as Enabling Conditions

The four patterns identified above are enabling conditions rather than sufficient conditions. Their presence does not guarantee success, but their absence correlates strongly with failure or limited adoption.

Table 2 summarizes the mechanisms for achieving each pattern across the surveyed niches. The table reveals that while the specific mechanisms vary widely, the underlying patterns are remarkably consistent.

A key observation is that the most successful systems often exhibit all four patterns. AnyDSL [57], Truffle [85], and Deegen [87] are notable examples: they provide explicit staging boundaries (annotations, bytecode semantics, or generated compilers), bounded policies (user annotations, compilation thresholds, or hybrid compilation), structured representations (CPS IR, Graal IR, or LLVM IR), and effect discipline (SSA conversion, `@PEFinal`, or type-based reasoning). Zeng [88] similarly exhibits all four patterns: the PDG and GRC provide structure-exposing representations; the schedule, three-valued simulation boundary, and recursion-depth limit provide bounded policies; the synchronous semantics and rendezvous communication provide effect discipline; and the data-dependence analysis and schedule ordering provide staging boundaries.

Conversely, systems that lack one or more patterns tend to struggle. Edwards [34] found that the SystemC libraries did not specialize well because they used a number representation that the specializer could not modify and "mostly static" variables that the specializer could not compile away. This underscores the importance of designing code with specialization in mind.

The patterns also interact synergistically. Explicit staging boundaries enable bounded policies by defining the scope of specialization. Structured representations make effect discipline easier to enforce by exposing control flow and data dependencies. Bounded policies prevent the code explosion that would otherwise result from aggressive specialization. And effect discipline enables more aggressive specialization by ensuring that it does not alter program semantics.

These patterns help transform PE from an unpredictable, unconstrained optimization into a reliable, predictable compilation methodology. They explain why PE has succeeded in certain domains, those that provide the necessary structure, while struggling in others.

7. Case Studies in Practical PE

The preceding sections have surveyed representative application niches and identified four recurring structural patterns that enable practical PE. We now examine two concrete systems in greater depth to illustrate how these patterns are realized within

Pattern	Staging	Effects	Structured IR	Bounded
Multi-Stage Prog.	Brackets/escapes	Monads, CSP	Types, CPS	Memoization
DSL Compilation	Annotations	SSA, IO monad	CPS IR, AST	User policies, guards
VM/JIT	Primitive annotations	Deoptimization	Graal/LLVM IR	Thresholds, deopt
Optimization	Neck, partitioning, fixed points	Aliasing, lazy flush, synchronous semantics	LLVM IR, PDG, GRC, dependent types	Profile-guided, memoization, unroll limits

Table 2: Mechanisms for achieving each structural pattern across niches

System	Staging	Effects	Structured IR	Bounded Policies
MetaML	Brackets/escapes	CSP restrictions	Lambda calculus	Type system
LMS	`Rep[T]`	Type-based	Types	Type inference
BuildIt	`static<T>`/`dyn<T>`	Repeated exec.	AST	Static tag memo.
AnyDSL	`@` annotations	SSA	Thorin (CPS)	User annotations
Tagless Final	`Symantics`	Purity	Combinators	Unfolding control
Yin-Yang	Macros	Effects	Rep types	Program slicing
Truffle	`@PEFinal`/`@PEBoundary`	Deoptimization	Graal IR	Compilation thresholds
Deegen	Bytecode semantics	Type-based	LLVM IR	Polymorphic IC
YJIT	Per-block type context	Guard checks/deopt	Basic blocks (LBBV)	Lazy per-block versioning
weval	`update_context`	Lazy flushing	SSA CFG	Worklist algorithm
PyPy	`JitDriver`	Green/red vars	RPython IR	Profiling counters
Babelfish	Unified IR	Lazy flushing	Babelfish-IR	Operator fusion
eBPF/Jitk	Bytecode load-time boundary	Verifier-enforced safety	Bytecode IR	Verifier-bounded program space
Zeng 2007	PDG/GRC, schedule, fixed-point	Synchronous semantics, rendezvous	PDG, GRC	Memoization, unroll limits

Table 3: Mechanisms for achieving structural patterns across representative systems

complete software systems. Rather than introducing additional applications, these case studies provide a detailed view of how explicit staging boundaries, bounded specialization policies, structure-exposing representations, and effect discipline manifest in practice.

GENSYM [83] represents interpreter specialization for software analysis, where PE compiles a generic symbolic executor into a program-specific executor. AnyHLS [90] represents specialization within hardware generation pipelines, where PE removes abstraction before downstream synthesis. Although these systems target very different application domains, both demonstrate the same central observation of this survey: practical PE succeeds because the surrounding system provides the structural patterns identified in Section 6.

7.1. GENSYM: Specializing Symbolic Execution

GENSYM [83] is a symbolic execution compiler for LLVM IR programs that demonstrates how partial evaluation can eliminate interpretation overhead in program analysis. Earlier work on staging symbolic execution [82] with algebraic effects similarly shows how search, path conditions, and symbolic state can be exposed as specialization structure. Traditional symbolic execution engines repeatedly interpret LLVM instructions, dispatching on instruction types while maintaining symbolic program state. Although this interpreter architecture is flexible, repeated dispatch and decoding become a significant source of overhead when exploring large numbers of execution paths.

GENSYM addresses this problem by specializing the symbolic executor with respect to a fixed LLVM IR program. The program under analysis is treated as static input, while symbolic values, path conditions, memory state, and SMT solver interactions remain dynamic. Rather than generating another interpreter, GENSYM emits specialized C++code that directly performs symbolic execution for the target program. Generic instruction dispatch dis-

appears because control flow has already been resolved during specialization. The residual executor therefore embodies the semantics of the analyzed program rather than repeatedly interpreting its LLVM representation.

A key enabling technique is continuation-passing style (CPS) [67], which provides an explicit representation of control flow during symbolic execution. Each execution path is represented by a continuation that can be suspended, resumed, and scheduled independently. This representation naturally supports parallel exploration by allowing path continuations to be distributed across worker threads without replicating heavyweight interpreter state.

7.1.1. Patterns in GENSYM

GENSYM realizes all four structural patterns as follows:

Explicit Staging Boundaries. The fixed LLVM program defines a natural staging boundary that closely follows the first Futamura projection: specializing an interpreter with respect to a fixed program produces a residual executor dedicated to that program. The program under analysis is treated as static input, while symbolic values, path conditions, memory state, and SMT solver interactions remain dynamic. This separation is explicit in the system design: the compiler knows at specialization time which inputs are fixed (the program) and which vary (the symbolic state).

Structure-Exposing Representations. CPS provides an explicit representation of control flow during symbolic execution. Each execution path is represented by a continuation that can be suspended, resumed, and scheduled independently. This representation naturally supports parallel exploration by allowing path continuations to be distributed across worker threads. The CPS transformation therefore contributes not only to execution efficiency but also to the structural organization required for effective specialization. By exposing control flow explicitly, CPS enables the specializer to reason about and optimize execution paths.

Effect Discipline. Symbolic execution isolates effectful interactions through well-defined environment models. The symbolic executor maintains state in a controlled manner: symbolic values, path conditions, memory state, and SMT solver interactions are all managed through explicit interfaces. This discipline ensures that specialization does not inadvertently execute effectful operations that would alter the semantics of the analysis. The environment models provide a clean boundary between the computations that can be performed during specialization and those that must remain for execution.

Bounded Specialization Policies. Specialization is naturally bounded by the finite program being compiled together with the symbolic execution strategy. Unlike general-purpose PE, which may need to handle arbitrary programs, GENSYM specializes only the fixed LLVM program under analysis. The search space for specialization is bounded by the program's size and the symbolic execution strategy, preventing uncontrolled code growth or nontermination. The CPS representation further bounds the state space by representing each execution path as a separate continuation.

7.2. AnyHLS: Hardware Generation via Specialization

AnyHLS [90] demonstrates how PE can serve as the compilation engine within a hardware generation pipeline. Rather than specializing a software interpreter, as in GENSYM, AnyHLS specializes reusable hardware abstractions to generate efficient FPGA implementations. The system enables programmers to express hardware designs using high-level functional abstractions while automatically producing vendor-oriented HLS (high-level synthesis) [25] code suitable for downstream synthesis.

A longstanding challenge in FPGA programming is balancing abstraction with performance. Existing HLS toolchains often require programmers to intermix algorithmic logic with hardware-specific directives, pragmas, memory organizations, and optimization strategies, making implementations difficult to reuse across platforms. AnyHLS separates these concerns by allowing programmers to describe algorithms independently of their hardware realization. Hardware-specific knowledge is encapsulated within reusable libraries that are later specialized during compilation.

PE is the mechanism that makes this separation practical. Many hardware characteristics, including loop bounds, memory layouts, vectorization factors, buffering strategies, and target architectures, are fixed before synthesis and therefore become static inputs to the specializer. During specialization, high-level hardware abstractions are progressively eliminated, producing residual programs that directly encode the selected hardware implementation. The residual artifact is vendor-oriented HLS

code (e.g., OpenCL or C++) that serves as input to existing FPGA synthesis toolchains.

7.2.1. Patterns in AnyHLS

AnyHLS naturally exhibits all four structural patterns as follows:

Explicit Staging Boundaries. The hardware generation workflow defines explicit staging boundaries between design-time specialization and runtime execution. Loop bounds, memory layouts, vectorization factors, buffering strategies, and target architectures are fixed before synthesis and become static inputs to the specializer. Runtime data remains dynamic. This establishes a natural staging boundary between design-time decisions and execution-time computation. The boundary is explicit in the system design: the programmer describes algorithms independently of hardware implementation, and the specializer handles the transition.

Structure-Exposing Representations. Hardware descriptions provide structure-exposing representations through reusable abstractions for pipelines, reductions, finite-state machines, memory hierarchies, and other architectural patterns. These abstractions expose the structure of the hardware design in a form that the specializer can analyze and optimize. The high-level functional abstractions allow programmers to express designs concisely while preserving the structural information needed for specialization.

Effect Discipline. The hardware generation process largely isolates effectful execution from specialization, allowing the specializer to operate primarily on declarative design descriptions rather than arbitrary runtime behavior. Hardware-specific knowledge is encapsulated within reusable libraries, and the specializer operates on these declarative descriptions. This discipline ensures that specialization does not inadvertently execute effectful operations that would corrupt the hardware generation process.

Bounded Specialization Policies. Specialization decisions are inherently bounded by the target architecture, available hardware resources, and compile-time design parameters. The space of possible specializations is constrained by the hardware platform (e.g., FPGA resources, memory hierarchy) and the design parameters (e.g., loop bounds, vectorization factors). This bounding prevents uncontrolled specialization while still allowing significant optimization. Rather than attempting to explore all possible hardware implementations, the specializer operates within the bounds defined by the target architecture and design constraints.

7.3. Cross-Case Synthesis

Viewed together, GENSYM and AnyHLS illustrate that successful PE is not confined to traditional compiler optimization. Although the two systems operate in very different domains, both embed specialization within a larger compilation pipeline that supplies the structural information needed to make PE predictable, scalable, and practically useful.

Table 4 summarizes how each pattern manifests in the two systems.

An important commonality is that neither system eliminates specialization policy; instead, both expose it through explicit staging decisions, constrained optimization choices, and structured compilation workflows. GENSYM's specialization is bounded by the finite program being analyzed, while AnyHLS's specialization is bounded by hardware resources. Both systems make staging boundaries explicit through their system architecture. Both expose structure through carefully designed representations (CPS for GENSYM; hardware abstractions for AnyHLS). And both enforce effect discipline through controlled execution models.

These case studies thus provide concrete validation of the framework developed in Section 6. The structural patterns identified there are realized in complete, working systems that achieve significant performance improvements through practical PE.

8. Toward Agentic Partial Evaluation

The preceding sections argue that successful applications of partial evaluation (PE) do not eliminate specialization policy; rather, they make it explicit, bounded, and structurally constrained. This observation suggests a natural research direction. Instead of hard-coding specialization policies through manually designed heuristics, can policy selection itself be partially automated while preserving the structural constraints that make PE practical?

We use the term *agent* to mean any system – whether a large language model, a learned policy, or a search procedure – that proposes candidate specialization decisions without itself performing or guaranteeing program transformation. We

Pattern	GENSYM	AnyHLS
Staging	Fixed LLVM program/dynamic symbolic state	Design-time parameters/runtime data
Structured IR	CPS continuations expose control flow	Hardware abstractions expose architecture
Effects	Environment models isolate effects	Declarative designs isolate effects
Bounded	Finite program bounds specialization	Hardware resources bound specialization

Table 4: Manifestation of structural patterns in GENSYM and AnyHLS

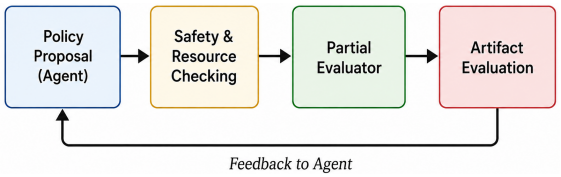


Figure 3: Conceptual workflow of agentic partial evaluation

use the term *agentic partial evaluation* to describe a specialization workflow in which such an agent proposes specialization policies, while deterministic compiler analyses, safety checks, and validation determine whether those proposals are accepted. The agent is therefore not the specializer itself. Instead, it searches over the space of admissible specialization policies, leaving program transformation, correctness, and code generation to conventional compilation techniques.

8.1. Agentic Specialization Loop

Figure 3 illustrates the conceptual workflow. The process consists of four stages:

1. **Policy proposal.** The agent proposes a specialization policy, including decisions such as the static/dynamic partition, unfolding depth, inlining strategy, memoization policy, variant limits, and generalization points.
2. **Constraint checking.** Before specialization begins, deterministic analyses verify that the proposed policy satisfies safety requirements, including effect discipline, termination guarantees, resource budgets, and specialization limits.
3. **Residual artifact generation.** A conventional partial evaluator applies the accepted policy to generate the residual artifact, which may be executable code, a specialized interpreter, generated hardware descriptions, or another compilation artifact.
4. **Evaluation and feedback.** The resulting artifact is evaluated using application-specific metrics such as execution time, throughput, code size, hardware utilization, compilation cost, or analysis performance. These measurements provide feedback for subsequent policy proposals.

Unlike unconstrained program synthesis, agentic PE performs search only within a bounded policy space whose correctness is independently enforced. The agent therefore assists specialization rather than replacing the compiler.

8.2. Case Study: Agentic PE for Pandas Programs

Recent work on agentic PE for Python `pandas` programs [4, 3] provides an initial example of how the proposed workflow can be realized in practice, following the architecture in Figure 3. Such programs frequently manipulate relatively stable structure – schemas, column names, feature-generation pipelines – while operating on datasets that vary at runtime, naturally separating static program structure from dynamic data.

Rather than rewriting programs directly, the agent analyzes a structural summary extracted from the Python AST and proposes specialization decisions such as schema-static expressions, bounded loop-unrolling opportunities, and repeated transformation pipelines worth specializing; it proposes *policies*, not transformed code. Static analyses then recover program structure, check data dependencies, and enforce legality constraints, so correctness does not depend on the agent itself. Once a policy is accepted, the partial evaluator specializes the program, removing generic interpretation overhead and exposing opportunities for downstream compiler optimization while producing an ordinary Python program that existing execution environments can run unmodified. Acceptance is gated by execution-based verification: the system runs both the original and candidate programs on identical inputs and requires the output DataFrames to match exactly, reverting to the original program on any mismatch.

This design was evaluated on three `pandas` preprocessing pipelines – a synthetic feature-engineering workload (*synth*), a NYC taxi-trip feature pipeline (*nyc_taxi*), and a real-world feature-engineering workflow from the Home Credit Default Risk dataset (*homecredit*) – under a five-way ablation isolating the contribution of PE, deterministic rewriting, and agent-guided policy selection [3]. Combining PE and rewriting under agent-selected policies (LLM+PE+rewrite) yielded the largest verified speedups on two of the three benchmarks: 2.715× on *synth* and 1.236× on *homecredit*, compared to 1.659× and 1.208× for the best fixed (non-agentic) policy on the same benchmarks. On *nyc_taxi*, the fixed aggressive policy (PE+rewrite) slightly outperformed the agent-guided one (1.502× vs. 1.467×), because the agent conservatively declined to unroll a loop that the fixed policy unrolled profitably; this illustrates that agent-guided selection is not universally superior to a well-tuned fixed policy, only more adaptable across the benchmarks as a whole. As a contrasting baseline, direct LLM code generation – asking a language model to rewrite the program directly, rather than to select a specialization policy – passed verification in as few as 0 of 5 and at most 5 of 5 trials depending on model and benchmark, confirming that the risk of hallucinated or behavior-changing edits is concentrated in code generation itself, not in policy selection under deterministic execution.

This case study illustrates the central theme of this survey: the contribution of the agent is bounded specialization-policy selection, not unrestricted program synthesis. The surrounding compiler infrastructure still provides the staging boundary, effect discipline, program representation, and validation pipeline that characterize successful PE systems. Agentic PE therefore extends rather than replaces classical PE by automating one historically difficult engineering task – choosing good specialization policies – while preserving the structural constraints that make specialization practical.

8.2.1. Lessons Learned

This early experience suggests two broader lessons for future agentic PE systems. First, language models appear capable of proposing plausible specialization policies, and the measured results above provide initial quantitative support for this hypothesis; however, this remains evidence from a single application domain rather than an established result, and the comparison against direct code generation shows the benefit is specific to constraining the agent to policy selection rather than code synthesis. Second, restricting the agent to policy selection, rather than arbitrary program rewriting, substantially reduces the search space while preserving correctness guarantees, and the resulting feedback loop suggests PE can be viewed as an iterative optimization process rather than a one-shot transformation.

Agentic PE does not alter the central conclusion of this survey: successful specialization still depends on explicit staging boundaries, disciplined treatment of effects, structure-exposing program representations, and bounded specialization policies. The role of an agent is to explore the resulting policy space, not to replace these structural foundations. We analyze the search cost in Section 8.3 and revisit the open challenges this approach raises – search cost, generalization, safety, and reproducibility – in Section 9.

8.3. Cost of Agentic Specialization

Policy search is not free, and an agent is worthwhile only if the speedup it finds recovers the cost of finding it. We make this trade-off explicit. Let t_0 be the execution time of the unspecialized program, s the speedup of the accepted residual program, and N the number of times the residual program is executed after specialization. An agent that tries k candidate policies incurs a search cost of

$$C_{\text{agent}} = k\left(c_{\text{propose}} + c_{\text{check}} + c_{\text{spec}} + c_{\text{eval}}\right),$$

where the four terms are the per-candidate costs of generating a proposal (for a language-model agent, inference), deterministic constraint checking, specialization, and evaluation. In the pandas study above, evaluation includes execution-based verification, so every candidate costs at least one execution of the candidate program, t_0/s at best, in addition to a one-time execution of the original program. Because proposals can be rejected, k is itself uncertain: if each proposal is accepted with probability p, about $1/p$ proposals are needed per accepted policy.

Specializing with the agent pays off against leaving the program unspecialized when $N\,t_0\,(1-1/s) > C_{\text{agent}}$. The more relevant comparison is against a fixed, hand-tuned policy with speedup s_{fixed} and cost C_{fixed}, for which the agent pays off when

$$N\,t_0\left(\frac{1}{s_{\text{fixed}}} - \frac{1}{s_{\text{agent}}}\right) > \Delta C,$$

Benchmark	s_{fixed}	s_{agent}	**Saving/run** (t_0)	N^* ($\Delta C/t_0$)
synth	1.659	2.715	0.234	4.3
homecredit	1.208	1.236	0.019	53
nyc_ taxi	1.502	1.467	−0.016	none

Table 5: Break-even analysis for agent-selected policies, derived from the speedups reported in the pandas study [3]. Extra saving per run is $1/s_{\text{fixed}} - 1/s_{\text{agent}}$ in units of the unspecialized execution time t_0; N^* is the number of executions needed to recover the agent's extra search cost ΔC, expressed in units of t_0.

where $\Delta C = C_{\text{agent}} - C_{\text{fixed}}$ is the agent's extra search cost.

The break-even number of executions is therefore $N^* = \Delta C \,/\, \big(t_0(1/s_{\text{fixed}} - 1/s_{\text{agent}})\big)$ when the bracket is positive; when it is not, the fixed policy wins however often the program is run.

The pandas study did not measure the agent's search cost. Table 5 therefore applies this model to the reported speedups only, and expresses each break-even point in units of the unknown cost ratio $\Delta C/t_0$ [3]. On *synth*, the agent saves an additional $0.23\,t_0$ per execution, so its extra search cost is recovered after about $4.3\,\Delta C/t_0$ executions. On *homecredit*, the additional saving is only $0.019\,t_0$ per execution, so recovering the same extra cost takes about $53\,\Delta C/t_0$ executions, more than ten times as many. On *nyc_ taxi* there is no break-even point. Agentic selection is thus justified for programs that run many times, such as scheduled data pipelines, and whose best policy is not obvious in advance; for one-off scripts, or when a fixed policy is already close to the best, it is not.

Because the search cost was not measured, Table 5 cannot say whether agentic selection actually paid off in the pandas study; it says only how many executions would be needed for a given cost. Measuring k and ΔC, which depend on the model, the prompting strategy, and the hardware, is necessary future work, and reporting them alongside speedups would make agentic systems comparable. Two mechanisms can lower the cost. First, the structural patterns of Section 6 bound the candidate space, which limits k and raises p; by contrast, direct LLM code generation passed verification in as few as 0 of 5 trials on some model and benchmark pairs, so most of its candidates yield nothing. Second, an accepted policy can be reused across executions and, to the extent that policies transfer between inputs (Section 9), across workloads. Finally, replacing analysis by search is not automatically a saving: the whole-program analyses that exposed structure makes unnecessary (Section 6) run once, deterministically, at compile time, whereas agent search must be paid for by every program whose policy is searched.

9. Emerging Niches and Open Problems

The preceding sections examined domains in which partial evaluation (PE) has already demonstrated practical success. A natural question is where similar opportunities are likely to emerge next. Rather than attempting to predict specific applications, our structural perspective suggests a general criterion: PE is most likely to succeed in domains that expose explicit staging boundaries, structure-rich intermediate representations, bounded specialization spaces, and disciplined treatment of effects. Deep learning compilation is the clearest current example of a rapidly evolving compilation area that satisfies these conditions with a mature, evidenced production system; we discuss it below before turning to the open research challenges that remain for PE more broadly, including those raised by agentic PE (Section 8).

9.1. An Emerging Niche: Deep Learning Compilation

Modern deep learning compilers transform neural network models [54] written in flexible programming frameworks into optimized implementations for CPUs, GPUs, and specialized accelerators. Although models are often expressed using dynamic programming environments, deployment pipelines increasingly separate model construction from repeated execution, creating natural opportunities for specialization [23]. Many properties become known before deployment – model topology, tensor shapes, data types, and target hardware – enabling graph capture [9], operator fusion, layout optimization, and code generation. Contemporary systems such as `torch.compile` in PyTorch 2 [9] exemplify this approach: components including `TorchDynamo`, `AOTAutograd`, and `TorchInductor` progressively transform high-level tensor programs into optimized residual programs through multiple compilation stages.

Although these systems are not typically described as partial evaluators, they exhibit the same structural characteristics documented throughout

this survey: the network architecture and deployment configuration serve as static inputs while runtime tensor values remain dynamic; graph capture establishes an explicit staging boundary; guards, graph breaks, and compilation caches bound the specialization space; and tensor intermediate representations [23] expose the program structure needed for transformation. Deep learning compilation can therefore be viewed as an instance of structured specialization rather than as a fundamentally different optimization paradigm, and a domain where agent-assisted policy selection (Section 8) could plausibly help navigate compiler decisions such as fusion aggressiveness or recompilation thresholds.

A more direct instance of this niche is StageML, recent work by two of the present authors on multi-tenant inference serving for large language models that combine Mixture-of-Experts (MoE) backbones with per-tenant Low-Rank Adaptation (LoRA) modules [5]. Unlike `torch.compile`, which exhibits PE-like structure without using PE terminology, this system is framed explicitly as a partial evaluator: it defines a six-stage binding-time lattice (base, adapter, tenant, request, routing, token) that separates values fixed at deployment from values that remain dynamic until token generation, and it applies a binding-time-safety rule to decide which adapter-expert weight combinations can be residualized ahead of routing and token time. A memory-aware planner then materializes only those residuals whose estimated benefit, under observed workload reuse, justifies their memory cost, falling back to dynamic execution otherwise. On Mixtral 8×7B under a Zipfian tenant workload, this residualization removes roughly half of adapter computation from token time and yields a measured 1.1× end-to-end speedup under a constrained memory budget, rising to 2.8× when the residualized weights are fed into an optimized fused-experts backend that assumes static expert weights. This result illustrates the same complementary relationship between PE and low-level kernel optimization observed elsewhere in this survey (Section 5.3): residualization does not replace a tuned kernel, but it can change which computations the kernel sees, unlocking optimizations the kernel could not otherwise apply.

Having examined both established and emerging applications, we conclude by discussing three research challenges that must be addressed to further broaden the practical impact of PE.

9.2. Predictable Specialization Policies

Although successful PE systems restrict the specialization space, selecting an effective specialization policy remains largely heuristic. Decisions such as the static/dynamic partition, unfolding depth, polyvariance, memoization strategy, and generalization points interact in complex ways, making it difficult to predict the resulting specialization cost and runtime benefit [47, 58]. A central research challenge is therefore the development of predictive cost models that estimate the trade-off among runtime improvement, specialization latency, and residual code growth before committing to a policy, rather than only after specialization has completed. Related compiler communities suggest promising directions: Halide's auto-scheduler employs learned cost models to predict profitable schedules, while Ansor searches optimization policies under explicit tuning budgets [1, 89]; similar techniques could guide PE-specific decisions while preserving the deterministic correctness guarantees provided by the underlying specializer.

This challenge is sharpened by agentic PE, where an agent searches the same policy space adaptively. Policy search consumes computational resources, so a proposed policy is only useful if its performance improvement outweighs the added search and specialization cost, meaning evaluation should report specialization overhead alongside end-to-end performance rather than runtime speedup alone (Section 8.3 analyzes this trade-off for the pandas study) [8, 89]. Policies that perform well for one workload may also fail to transfer to different inputs or hardware platforms, so robust evaluation requires diverse benchmark suites rather than optimization for a single instance [1].

9.3. Effect-Aware Specialization

Effectful computation remains one of the principal obstacles to broader adoption of PE. Executing computations prematurely may reorder observable behavior, duplicate side effects, or violate assumptions about mutable state and external interactions [47, 61], so identifying safe specialization boundaries remains an active research problem. Effect systems provide one promising direction, since they explicitly describe the observable behavior of computations alongside their value types [59, 76]; modern effect systems, ownership models, and alias analyses increasingly make such information available to compilers, potentially enabling more aggressive specialization while preserving correctness [55].

A remaining challenge is extending these techniques across foreign-function interfaces, concurrency, and large software ecosystems where complete effect information is often unavailable.

This problem is not new to agentic PE, but agentic PE raises it in a sharper form: an agent may propose specialization policies, but it should never itself determine whether an effectful computation executes during specialization. That decision must remain the responsibility of effect barriers, residualization rules, and compiler analyses defined independently of the agent [61], with the agent's proposals accepted or rejected against those fixed rules rather than trusted directly.

9.4. Evaluation and Reproducibility

Progress in PE also depends on stronger experimental methodology. Historically, PE systems have been difficult to compare because they differ substantially in language subsets, effect models, specialization policies, and implementation assumptions [47]. Agent-assisted specialization increases this challenge further, since search budgets, randomization, hardware platforms, and compiler versions directly influence experimental outcomes, and reproducible evaluation requires reporting these configurations alongside the accepted policies and their measured performance [8, 89].

Future evaluation should emphasize reproducibility alongside performance: benchmark suites should include both controlled microbenchmarks that isolate individual specialization decisions and realistic end-to-end applications drawn from emerging PE niches (for deep learning compilation, MLPerf Inference [66] and TorchBench [40] provide suitable suites), and experimental reports should consistently include specialization time, residual code size, resource budgets, failed specialization attempts, and measurement variance. Establishing such evaluation practices would substantially improve the comparability of future PE research and help move the field toward a more mature, systems-oriented methodology.

10. Conclusion

Partial Evaluation has long been regarded as one of the most elegant ideas in program optimization. Its theoretical foundations are well established, and its ability to remove interpretation overhead and specialize programs with respect to known information has influenced decades of research in compilers, programming languages, and software engineering. Yet its adoption trajectory presents an apparent paradox: despite its conceptual elegance, PE as a self-contained, general-purpose specialization methodology never achieved the broad adoption envisioned in the early literature, even though its underlying principle is embodied in transformations – constant folding and constant propagation chief among them – that are ubiquitous in modern compilers. Section 5.3.10 extends this observation to more recent systems: tracing JIT compilation in PyPy [15], basic-block versioning in YJIT [24], guarded PE in Truffle/Graal [85], verified bytecode-to-native compilation in the Linux kernel's eBPF subsystem [81], and speculative type-specialization in V8 [17] all realize restricted, often unnamed forms of PE.

This survey argues that this apparent paradox is best understood as a question of *systems design* rather than of specialization itself. PE has not failed uniformly; instead, it has repeatedly succeeded – often under other names – in domains that provide the structural information required to make specialization predictable. Across diverse application areas – including multi-stage programming, domain-specific languages, virtual machines, interpreter specialization, configuration specialization, and hardware generation pipelines – we identified four recurring structural patterns: explicit staging boundaries, bounded specialization policies, suitable intermediate representations, and disciplined management of computational effects. These recurring patterns explain why specialization is practical, and in many production systems already pervasive, in environments that supply this structure, while remaining difficult to apply as a single, unconstrained, general-purpose whole-program optimization.

Viewing PE through this systems perspective also clarifies why similar ideas continue to emerge in modern compiler ecosystems. Contemporary deep learning compilers expose explicit representations, constrained optimization spaces, and well-defined compilation pipelines that naturally support structured specialization. Rather than representing an entirely new direction, such systems can be understood as extending the same underlying principles that have historically enabled successful applications of PE.

Building on this perspective, we proposed *agen-*

tic PE as a possible direction for future research, not as a demonstrated solution to the difficulties discussed in Section 3. Rather than replacing classical PE with unconstrained program synthesis, agentic PE treats specialization as bounded policy selection guided by adaptive search while preserving deterministic correctness, termination, and resource guarantees. Whether implemented using learning-based methods or large language models, such agents should operate within the structural constraints that have consistently characterized successful PE systems. The evidence so far comes from a single application domain, a well-tuned fixed policy matched or beat the agent on one of three benchmarks, and the agent's search cost, which has not yet been measured, must be recovered by the speedup it finds (Section 8.3); whether agentic PE improves on fixed policies in general remains an open question.

The broader implication is that the future of PE is unlikely to lie in pursuing fully automatic specialization for arbitrary software. Instead, PE is most likely to advance where systems already expose the structure needed for specialization: explicit staging, controlled effects, suitable intermediate representations, and bounded optimization policies. As compiler infrastructures continue to expand into machine learning, hardware generation, scientific computing, and emerging computational models, these conditions are becoming increasingly common. In that sense, PE should not be viewed as an optimization technique whose opportunity has passed, but as a systems methodology whose relevance continues to grow alongside modern compiler-rich software ecosystems.

Appendix A. Surveyed Corpus

Table A.6 lists the 42 works that satisfy the inclusion criteria of Section 4 and form the surveyed corpus, grouped by the niche of Section 5 in which they are discussed, together with the subsection that discusses each. Other works cited in Section 5, such as tools and analyses on which the surveyed systems build, support the discussion but are not part of the corpus.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process:

During the preparation of this work, the authors used claude for editing. The authors reviewed and edited the output as needed and take full responsibility for the content of the published article.

References

[1] Andrew Adams et al. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4), July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322967. URL `https://doi.org/10.1145/3306346.3322967`.

[2] Michael D. Adams, Andrew Farmer, and José Pedro Magalhães. Optimizing syb is easy! In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 71–82. ACM, 2014. doi: 10.1145/2543728.2543735.

[3] Xavier Adettu and Tian Zhao. Agentic partial evaluation on pandas programs. In *Proceedings of the 49th IEEE Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE Computer Society, 2026. Accepted.

[4] Xavier Adettu and Tian Zhao. Partial evaluation for pandas in python. In *Proceedings of the 41st ACM/SIGAPP Symposium on Applied Computing*, SAC '26, page 1332–1334, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722943. doi: 10.1145/3748522.3779923. URL `https://doi.org/10.1145/3748522.3779923`.

[5] Xavier Adettu and Tian Zhao. StageML: Partial evaluation for multi-tenant MoE-LoRA inference. In *Proceedings of the 2026 Joint International Symposium on Logic-Based Program Synthesis and Transformation and Principles and Practice of Declarative Programming (LOPSTR+PPDP)*, Lecture Notes in Computer Science. Springer, 2026. To appear.

[6] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. Lightweight, multi-stage, compiler-assisted application specialization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021.

[7] Guillaume Allais. Scoped and typed staging by evaluation. In *Proceedings of the*

Work	Section	Topic
Multi-Stage Programming (Section 5.1)		
Taha and Sheard [74]	5.1.1	Foundations: MetaML and Explicit Annotations
Taha [73]	5.1.1	Foundations: MetaML and Explicit Annotations
Rompf and Odersky [68]	5.1.2	Type-Based Staging: Lightweight Modular Staging and BuildIt
Brahmakshatriya and Amarasinghe [19]	5.1.2	Type-Based Staging: Lightweight Modular Staging and BuildIt
Swadi et al. [72]	5.1.3	Monadic Staging: Controlling Code Duplication
Kameyama et al. [49]	5.1.4	Cross-Stage Persistence and Delimited Control
Asai [10]	5.1.5	Binding-Time Analysis and Automatic Staging
Asai and Kameyama [11]	5.1.5	Binding-Time Analysis and Automatic Staging
Taha et al. [75]	5.1.6	Tag Elimination and Jones-Optimality
Allais [7]	5.1.7	Scoped and Typed Staging by Evaluation
Domain-Specific Languages (DSLs) (Section 5.2)		
Leißa et al. [56]	5.2.1	Shallow Embedding and PE
Leißa et al. [57]	5.2.1	Shallow Embedding and PE
Jovanović [48]	5.2.2	Deep Embedding and DSL Compilation
Carette et al. [21]	5.2.3	Tagless Final Interpreters
Biri [13]	5.2.3	Tagless Final Interpreters
Brady and Hammond [18]	5.2.4	PE of DSL Interpreters
Melançon et al. [60]	5.2.5	Generating Compilers from Executable Semantics
Coutts [32]	5.2.6	PE for DSLs in Haskell
Hutchins [45]	5.2.7	PE and Reflection for DSALs
Virtual Machines and JIT Compilation (Section 5.3)		
Würthinger et al. [85]	5.3.1	Truffle/Graal: Practical PE in Production
Latifi et al. [53]	5.3.2	CompGen: The Second Futamura Projection in Practice
Latifi et al. [52]	5.3.2	CompGen: The Second Futamura Projection in Practice
Xu and Kjølstad [87]	5.3.3	Deegen: Automatic JIT-Capable VM Generation
Bolz et al. [15]	5.3.4	Meta-Tracing: PyPy's Tracing JIT Compiler
Thibault et al. [77]	5.3.5	Specializing Interpreters with Tempo and DyC
Noël et al. [64]	5.3.5	Specializing Interpreters with Tempo and DyC
Sullivan [70]	5.3.6	Dynamic PE for Reflective Languages
Sullivan et al. [71]	5.3.6	Dynamic PE for Reflective Languages
Chambers et al. [22]	5.3.7	Staged Compilation: Compiling Once, Optimizing Everywhere
Wimmer et al. [84]	5.3.8	GraalVM Native Image: Ahead-Of-Time (AOT) Compilation with Heap Snapshotting
Fallin and Bernstein [37]	5.3.9	weval: Whole-Program Compilation via Partial Evaluation
Chevalier-Boisvert et al. [24]	5.3.10	Basic-Block Versioning and Kernel Bytecode Specialization: YJIT and eBPF
Wang et al. [81]	5.3.10	Basic-Block Versioning and Kernel Bytecode Specialization: YJIT and eBPF
Botcharnikov [17]	5.3.10	Basic-Block Versioning and Kernel Bytecode Specialization: YJIT and eBPF
Program Optimization via PE (Section 5.4)		
Popeea et al. [65]	5.4.1	Array Bounds Check Elimination
Grulich et al. [39]	5.4.2	Polyglot Query Optimization
Alhanahnah et al. [6]	5.4.3	Compiler-Assisted Application Specialization
Köster et al. [51]	5.4.4	Simulation Acceleration for Biochemical Reaction Networks
Cooper et al. [30]	5.4.5	Cardiac Cell Model Simulation
Edwards [34]	5.4.6	Fixed-Point Simulation Acceleration
Adams et al. [2]	5.4.7	Optimizing Generic Programming
Zeng [88]	5.4.8	Synchronous Concurrent Program Optimization

Table A.6: The 42 works of the surveyed corpus, grouped by niche, with the subsection of Section 5 in which each is discussed.

ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM), 2024.

[8] Jason Ansel et al. Opentuner: an extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 303–316, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328098. doi: 10.1145/2628071.2628092. URL https://doi.org/10.1145/2628071.2628092.

[9] Jason Ansel et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation, 2024. URL https://doi.org/10.1145/3620665.3640366.

[10] Kenichi Asai. Toward introducing binding-time analysis to metaocaml. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 97–102, 2016.

[11] Kenichi Asai and Yukiyoshi Kameyama. Automatic staging via partial evaluation techniques. In *International Symposium on Symbolic Computation in Software Science (SCSS)*, pages 1–13, 2016.

[12] Thomas Ball and James R. Larus. Efficient path profiling. page 46–57. IEEE, 1996. doi: 10.1109/MICRO.1996.566449. URL https://infoscience.epfl.ch/handle/20.500.14299/98712.

[13] Nicolas Biri. Dependent tagless final. In *Proceedings of the ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2022.

[14] Lars Birkedal and Morten Welinder. Handwriting program generator generators. In Manuel Hermenegildo and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '94)*, volume 844 of *Lecture Notes in Computer Science*, pages 198–214. Springer, 1994. doi: 10.1007/3-540-58402-1_15.

[15] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, pages 18–25, 2009.

[16] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, 1991.

[17] Dmitry Botcharnikov. Approaches to optimizing V8 JavaScript engine. *Proceedings of the Institute for System Programming of the RAS*, 27(6):21–32, 2015. doi: 10.15514/ISPRAS-2015-27(6)-2.

[18] Edwin C. Brady and Kevin Hammond. Scrapping your inefficient engine: Using partial evaluation to improve domain-specific language implementation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 297–308, 2010.

[19] Ajay Brahmakshatriya and Saman Amarasinghe. Buildit: A type-based multi-stage programming framework for code generation in c++. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 39–51, 2021.

[20] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX Association, 2008.

[21] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[22] Craig Chambers, Matthai Philipose, Markus Mock, and Susan J. Eggers. Staged compilation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2002.

[23] Tianqi Chen et al. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018. URL https://arxiv.org/abs/1802.04799.

[24] Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. YJIT: A basic block versioning JIT compiler for CRuby. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL)*, pages 25–32. ACM, 2021.

[25] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[26] Charles Consel. New insights into partial evaluation: The SCHISM experiment. In Harald Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 236–246, Nancy, France, 1988. Springer. doi: 10.1007/3-540-19027-9_16.

[27] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 493–501, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915607. doi: 10.1145/158511.158707. URL `https://doi.org/10.1145/158511.158707`.

[28] Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur. A tour of tempo: a program specializer for the c language. *Sci. Comput. Program.*, 52(1–3):341–370, August 2004. ISSN 0167-6423. doi: 10.1016/j.scico.2004.03.011. URL `https://doi.org/10.1016/j.scico.2004.03.011`.

[29] William R. Cook and Ralf Lämmel. Tutorial on online partial evaluation. In *Domain-Specific Languages*, 2011. URL `https://api.semanticscholar.org/CorpusID:548057`.

[30] Jonathan Cooper, Steve McKeever, and Alan Garny. On the application of partial evaluation to the optimisation of cardiac electrophysiological simulations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM, 2006.

[31] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373500. doi: 10.1145/512950.512973. URL `https://doi.org/10.1145/512950.512973`.

[32] Duncan Coutts. Partial evaluation for domain-specific embedded languages in a higher order typed language. Master's thesis, University of Oxford, 2004.

[33] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation: International Seminar, Dagstuhl Castle, Germany, February 12–16, 1996, Selected Papers*, volume 1110 of *Lecture Notes in Computer Science*. Springer, 1996. doi: 10.1007/3-540-61580-6.

[34] Stephen A. Edwards. Using program specialization to speed systemc fixed-point simulation. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. IEEE, 2006.

[35] A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977.

[36] A. P. Ershov. On the essence of compilation. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[37] Chris Fallin and Maxwell Bernstein. Partial evaluation, whole-program compilation. *Proceedings of the ACM on Programming Languages (PLDI)*, 9(PLDI):160:1–160:24, 2025.

[38] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. doi: 10.1023/A:1010095604496. URL `https://doi.org/10.1023/A:1010095604496`.

[39] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. Babelfish: Efficient execution of

polyglot queries. *Proceedings of the VLDB Endowment (PVLDB)*, 15(2):196–210, 2022. doi: 10.14778/3489496.3489501.

[40] Yueming Hao et al. Torchbench: Benchmarking pytorch with high api surface coverage, 04 2023.

[41] John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*. Springer, 1999. doi: 10.1007/3-540-47018-2.

[42] Carsten K. Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991. Glasgow University.

[43] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. volume 27, page 32–43, New York, NY, USA, July 1992. Association for Computing Machinery. doi: 10.1145/143103.143114. URL `https://doi.org/10.1145/143103.143114`.

[44] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es): 196–es, December 1996. ISSN 0360-0300. doi: 10.1145/242224.242477. URL `https://doi.org/10.1145/242224.242477`.

[45] DeLesley Hutchins. Partial evaluation + reflection = domain specific aspect languages. In *Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, 2006.

[46] Neil D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, September 1996. ISSN 0360-0300. doi: 10.1145/243439.243447. URL `https://doi.org/10.1145/243439.243447`.

[47] Neil D. Jones et al. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., USA, 1993. ISBN 0130202495.

[48] Vojin Jovanović. *Uniting Language Embeddings for Fast and Friendly DSLs*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2016.

[49] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Shifting the stage: Staging with delimited control. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 111–120, 2009.

[50] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[51] Till Köster, Tom Warnke, and Adelinde M. Uhrmacher. Generating fast specialized simulators for stochastic reaction networks via partial evaluation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 32 (2):10:1–10:25, 2022. doi: 10.1145/3485465.

[52] Florian Latifi, David Leopoldseder, Christian Wimmer, and Hanspeter Mössenböck. Compgen: Generation of fast compilers in a multilanguage vm. In *Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion)*, 2019.

[53] Florian Latifi, David Leopoldseder, Christian Wimmer, and Hanspeter Mössenböck. Compgen: Generation of fast jit compilers in a multilanguage vm. In *Proceedings of the ACM SIGPLAN International Symposium on Dynamic Languages (DLS)*, 2021.

[54] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[55] Daan Leijen. Koka: Programming with row polymorphic effect types. *Electronic Proceedings in Theoretical Computer Science*, 153: 100–126, June 2014. ISSN 2075-2180. doi: 10.4204/eptcs.153.8. URL `http://dx.doi.org/10.4204/EPTCS.153.8`.

[56] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. Shallow embedding of dsls via online partial evaluation. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 11–20, 2015.

[57] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsene Pérard-Gayot, Richard Membarth,

Philipp Slusallek, Andre Müller, and Bertil Schmidt. Anydsl: A partial evaluation framework for programming high-performance libraries. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2(OOPSLA): 119:1–119:30, 2018.

[58] Michael Leuschel. Homeomorphic embedding for online termination of symbolic methods. *LNCS*, 2566, 12 2002. doi: 10.1007/3-540-36377-7_17.

[59] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 47–57, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912527. doi: 10.1145/73560.73564. URL `https://doi.org/10.1145/73560.73564`.

[60] Olivier Melançon, Marc Feeley, and Manuel Serrano. An executable semantics for faster development of optimizing python compilers. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 15–28, 2023.

[61] Uwe Meyer. Correctness of on-line partial evaluation for a pascal-like language. *Sci. Comput. Program.*, 34(1):55–73, April 1999. ISSN 0167-6423. doi: 10.1016/S0167-6423(98)00015-X. URL `https://doi.org/10.1016/S0167-6423(98)00015-X`.

[62] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1558603204.

[63] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[64] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the International Conference on Computer Languages*, pages 132–142, 1998.

[65] Corneliu Popeea, Dana N. Xu, and Wei-Ngan Chin. A practical and precise inference and specializer for array bound checks elimination. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 177–187. ACM, 2008. doi: 10.1145/1328408.1328434.

[66] Vijay Janapa Reddi et al. Mlperf inference benchmark, 2020. URL `https://doi.org/10.1109/ISCA45697.2020.00045`.

[67] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, 1972.

[68] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 127–136, 2010.

[69] Peter Sestoft. ML pattern match compilation and partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation: International Seminar, Dagstuhl Castle, Germany*, volume 1110 of *Lecture Notes in Computer Science*, pages 446–464. Springer, 1996. doi: 10.1007/3-540-61580-6_22.

[70] Gregory T. Sullivan. Dynamic partial evaluation. In *Proceedings of the Symposium on Programs as Data Objects (PADO)*, pages 238–256, 2001.

[71] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators*, pages 50–57, 2003.

[72] Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 160–169, 2006.

[73] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[74] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, 1997.

[75] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and jones-optimality. In *Proceedings of the Symposium on Programs as Data Objects (PADO)*, pages 257–275, 2001.

[76] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 162–173, 1992. doi: 10.1109/LICS.1992.185530.

[77] Scott Thibault, Charles Consel, Julia L. Lawall, Renaud Marlet, and Gilles Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[78] Peter Thiemann. Cogen in six lines. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 180–189. ACM, 1996. doi: 10.1145/232627.232647.

[79] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2007. ISBN 012088478X.

[80] Todd L. Veldhuizen. C++ templates as partial evaluation. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, number NS-99-1 in BRICS Notes Series, pages 13–18, San Antonio, Texas, USA, 1999. BRICS.

[81] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–47. USENIX Association, 2014.

[82] Guannan Wei, Oliver Bračevac, Shangyin Tan, and Tiark Rompf. Compiling symbolic execution with staging and algebraic effects. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–33, 2020.

[83] Guannan Wei et al. Compiling parallel symbolic execution with continuations. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1316–1328, 2023. doi: 10.1109/ICSE48619.2023.00116.

[84] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanović, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: Application initialization at build time. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 3(OOPSLA):184:1–184:29, 2019.

[85] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 662–676, 2017.

[86] Thomas Würthinger et al. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 187–204, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324724. doi: 10.1145/2509578.2509581. URL `https://doi.org/10.1145/2509578.2509581`.

[87] Haoran Xu and Fredrik Kjølstad. Deegen: A jit-capable vm generator for dynamic languages. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2024. To appear.

[88] Jia Zeng. *Partial Evaluation for Code Generation from Domain-Specific Languages*. PhD thesis, Columbia University, New York, NY, USA, 2007. URL `https://doi.org/10.7916/D8TX3G7H`.

[89] Lianmin Zheng et al. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association. ISBN 978-1-939133-19-9.

[90] M. Akif Özkan et al. Anyhls: High-level synthesis with partial evaluation, 2020.