Partial Dispatch: Optimizing Dynamically-Dispatched Multimethod Calls with Compile-Time Types and Runtime Feedback

Jonathan Bachrach
Artificial Intelligence Laboratory
Massachussetts Institute of Technology
Cambridge, MA 02139
jrb@ai.mit.edu

Glenn Burke
Crysaliz, Inc.
9 Damon Mill Square, Suite 4D
Concord, MA 01742
gsb@pobox.com

1 Abstract

2 Introduction

Dylan is an object-oriented language with a combination of features that present both difficulties and opportunities for efficient compilation. The language is thoroughly object-oriented and supports multimethod dispatch as well as dynamic typing. The semantics of the language are consistently defined in terms of a program runtime model. However, the design goals included providing significant opportunities for compile-time optimizations that are consistent with this runtime model. Additionally, the ability to dynamically add classes and methods is balanced by the ability to seal branches of the class and method hierarchy, thereby declaring them invariant. Optimization proceeds through a pay-as-you-go strategy. Programmers are not required to choose between the extremes of full static dispatch with strict typing and full runtime dispatch with dynamic typing. Instead, type information and constraints can be added incrementally, with resulting incremental improvements in performance, ultimately resulting in performance equivalent to static languages when full type information is available at compile-time.

Additional requirements on Functional Developer include the need to support interactive software development and the production of components using separate compilation. The latter requirement - a real world constraint for commercial development - rules out the possibility of using many whole program analysis techniques. Instead, Functional Developer makes aggressive use of partial type information available at compile-time, and uses novel techniques to combine this with runtime information to produce highly efficient code.

2.1 Generic Functions

Dylan provides polymorphic method invocation through generic function calls ([Keene89], [Shalit96], [Stefik86]). A generic function consists of a set of methods. Each method includes parameter specializers that indicate the types of arguments for which a method is applicable. When a generic function is called, the actual arguments are compared to specializers of all the generic function's methods; the applicable methods are selected and then sorted; finally, the most specific applicable method is invoked, passing along the remaining applicable methods to be used in the next-method chain. [1]

In languages that support multimethods, the types of all the arguments are included in this computation. Needless to say, this multistage process can be quite expensive. The result has been a large amount of research on strategies for reducing the cost, both through compile-time method selection as well as through caching strategies. Dylan's method selection process is complicated by the variety of types that can be used as specializers. Any given object may be a member of several different types, which are ordered through subtyping rules. [2]

- A class type restricts its argument to be an instance of that class.
- A singleton type restricts its argument to be a specific object.
- A subclass type restricts its argument to be a class object that is a subclass of a given class.
- A union type restricts its argument to be an instance of one of a number of other types.
- A limited collection type restricts its argument to be an instance of a collection with additional restrictions on size and collection contents.
- A limited integer type restricts its argument to be within a subset of the range of whole numbers.

Multimethod dispatch has traditionally been optimized in an all or nothing fashion: either the call is statically dispatched (the unique effective method is determined by the compiler) or it is dynamically dispatched, considering all arguments irrespective of compile time information. The Functional Developer compiler supports these approaches and also supports partial static dispatch, whereby the compiler combines inferred type information and sealing declarations to pre-compute some of the logic that would happen at runtime method call time. For example, knowing the argument types and domain sealing constraints may allow the

[1] Dylan's next-method mechanism is a way of passing control to the next-most-specific method, like super.

[2] The algorithm used for ordering classes depends on the type of the actual argument, so is computable for two types which are joint but neither super- nor sub-types of each other.

compiler to statically narrow the number of possibly applicable methods at a call site down to only a few. In that case, the emitted code need only branch on the relevant data distinguishing the few applicable methods, rather than having to choose from among all the methods of a generic function. In fact, if the compiler is able to narrow the set of applicable methods down to one, then the method call can be emitted as a simple function call, or the method body may even be inlined. In this way, Dylan preserves the semantics of a dynamic object-oriented language while recapturing much of the efficiency of a static procedural language.

Dylan supports the development of reusable components using separate compilation. Unfortunately, separate compilation can starve the compiler of information necessary to perform full static dispatch of multimethod calls. A common example of this is when dispatch is performed on abstract types. Dylan's sealing construct provides a way to gain back partial class hierarchy information. For example, subhierarchies of the class hierarchy and the set of methods more specific than a particular signature can be sealed, thus permitting the compiler to perform static dispatch within that hierarchy.

Unfortunately, sealing is not applicable in a number of important cases that turn up in the development of reusable components. The most important case occurs when a particular component defines an interface that client components implement. A classic example that turns up in both our Dylan compiler and graphical user interface toolkit is where an abstract backend is defined in a backend interface component, methods specialized on this abstract backend class are defined in core components, a particular backend is defined in a concrete backend component, and then top-level methods are defined in a backend specific glue component which uses all of the previously mentioned components (see figure 1). The problem is that the concrete backend is unknown when compiling the core code component leading to full dynamic dispatch on all methods specializing on the backend. Sealing doesn't help because the backend interface is intentionally open to allow the graceful addition of backends either at compile-time or through dynamic loading. Furthermore, performing whole program class hierarchy analysis at compile-time [Dean96] is not applicable because it is at odds with separate compilation and dynamic loading. First, components would have to compile in implementation details about used components (i.e., their class hierarchies) and thus would have to be recompiled even if implementation details of these used components change (aka fragile base class problem). Second, and even worse, in Dylan, where methods can be added in using components (and in Cecil where classes can be added in using components), in order to utilize whole program class hierarchy analysis, components would have to compile in implementation details about using components. Third, this sort of whole program analysis would rule out the dynamic loading of components as this could potentially invalidate assumptions derived at compile-time. Fourth, this analysis implies that sources for a given component are made available for recompilation in the field.

Delaying whole-program optimizations until link-time is unacceptable for Functional Developer for a number of reasons. First, it slows down start up time by delaying the bulk of the optimizations until link time. Although an incremental implementation is conceivable, it is difficult to support our "pay as you go" philosophy. Second, it relies on non-standard linker technology complicating the support of and integration with other languages and platforms. Third, it can prevent object code sharing by specializing each component.

3 Overview

We present our engine-node paradigm for constructing decision trees at run time, with which we develop a number of optimizations. First, we present our basic shared decision tree approach, which incrementally augments a shared (one per generic-function) decision tree only as needed, where unnecessary dispatch steps are replaced by type checks. Next, we expand the paradigm to call site caches - decision trees that are local to a call site (or shared among "similar" call sites), which take advantage of type information injected into the runtime by the compiler to reduce or eliminate decision steps, and which may also benefit from "lighter weight" decisions due to the more limited size of the decision tree. We then show how we can make use of disjointness information and Dylan's differentiation of abstract and concrete classes to prune the decision steps further still, and discuss the runtime dependency tracking needed for this to work in the dynamic Dylan environment. We show how our per-call-site decision trees can be profiled, and how we can use this information to make implementation decisions in a future compilation. Lastly, we demonstrate how our decision trees can be used to eliminate dispatch from certain self-recursive functions. Finally, we compare our work to others and then discuss our future directions.

4 Runtime Dispatch with Discrimination Trees

For those cases in which function calls cannot be fully statically optimized, Functional Developer uses an adaptive dynamic dispatch mechanism. Generic function dispatch is performed by a program - a decision tree - dynamically constructed of building blocks we call engine nodes. An engine node gets invoked to perform some action, for instance to check whether a particular argument is of some type and perform alternate actions depending on the result. Typically the action is to invoke another engine node, and ultimately the method most applicable in the generic function call. The MEP (for Method Entry Point) calling sequence is designed so that shared code for generic function dispatch and next-method invocation can efficiently access arguments and pass them along. Both engine-nodes and methods have an MEP slot, which contains the address of the code to handle the invocation. For this calling sequence, only a fixed number of arguments can ever be passed by any particular function. If the function takes any optional arguments (i.e., has `#rest` and/or `#key` arguments), they have been formed up into a stack allocated vector, which is then treated as a single argument. A Dylan function

```
define generic futz (a, b, #rest more);
```

will consist of engine nodes and methods whose entry points expect three arguments, corresponding to parameters `a`, `b`, and `more`. (Dylan's definition of congruency requires all methods of a generic function to have the same number of required arguments, unlike some other languages. Optional arguments can be passed by keywords and `#rest` arguments.)

The arguments are passed in registers and/or stack; the engine node or method being invoked is placed in the function virtual register; an "extra argument" is placed in the extra argument register; and the address in the MEP slot is jumped to in order to perform the invocation. At the start

of dispatch, the extra argument is the generic function; as long as engine nodes are being invoked, it will be either this or a parent engine node from which the generic function can be derived. (This is how the generic function can be recovered from the dispatching state without the need for explicit pointers to it from the individual engine nodes, as will happen when a dispatch miss occurs as described below.) For a method, the extra argument is theoretically customized to the method; in practice, if it is needed, it is information used for the execution of next-method calls within the method. [3]

5 Dispatch Engine Design

Engine nodes are implemented as Dylan objects, all subclasses of the (internal) class `<engine-node>`. They are organized in a type hierarchy that describes their functionality, and permits them to be programmatically understood and manipulated. For instance, ¡discriminator¿ is the (abstract) class of all engine-nodes that examine a particular argument in order to make some decision. All discriminators store the argument number they examine, and information about the number of arguments the function takes. Other abstract types of engine node are `<terminal-engine-node>`, which would be used when discrimination is complete, and `<encapsulating-engine-node>`, which is used when some action (e.g. relating to profiling, metering, or tracing) needs to be inserted into the discrimination program, but doesn't otherwise affect discrimination.

The discrimination program of a generic function is incrementally constructed out of engine nodes as needed. If it encounters a set of arguments it does not handle - which indicates that either the discrimination program needs to be augmented or the generic function is not applicable to that set of arguments - a dispatch miss is said to occur. The internal Dylan function handle-missed-dispatch is invoked. This function recovers some of the dispatch state and the generic function involved from the extra-argument register mentioned earlier. The generic function is locked to prevent race conditions, and additional discrimination programs composed of engine nodes and methods is generated to dispatch this call. The dispatch is then continued, or a "no applicable method" error is signaled. This is also the point at which ambiguous method ordering might be detected.

Some examples of `<terminal-engine-node>` types are:

`<absent-engine-node>` Executing this type of engine node causes a dispatch miss to occur. An absent engine node is placed anywhere an engine node can be placed, to indicate a control branch has not yet been computed. For instance, in an `<if-type-discriminator>` described below, one branch of the "if" might not be computed - it would contain an absent- engine-node. The initial discrimination program of a generic function is an absent-engine-node. For efficiency, we only create a single instance of `<absent-engine-node>`, and use it repeatedly. This saves both the space of multiple instances, and permits more efficient checking for the absent-engine-node when a discrimination program is being modified.

`<slot-accessing-engine-node>` This is the common superclass of a number of classes of engine node which are used to perform slot access. We do not use the slot method to do the access, but instead use a `<slot-accessing-engine-` for the exact kind of access and offset called for. This enables some optimizations. First, the object that represents the method for a slot accessor does not require an MEP slot as it does not take part in this protocol, making the representation for slot functions smaller. We also share slot-accessing-engine-node objects: the very same one can be used for different slots in different generic functions that are accessed the same and lie at the same offset. Finally, because of multiple inheritance, the "same" slot may be allocated in different locations in different classes that inherit it from a common superior class. So we can use different slot-accessing-engine-nodes to access the same slot depending on the exact class of the argument, as determined by some preceding discriminator.

Some examples of discriminator types:

`<by-class-discriminator>` This is the general-case discriminator for specializers of type `<class>`. It is keyed by the object-class of the argument. Our implementation uses multiple subtypes of `<by-class-discriminator>` that correspond to different table lookup strategies depending on the number of keys in the table. This discriminator fetches the object-class of the specified argument, looks that up in its table, and tail-recursively invokes the resulting engine node or method; if no entry was found, the absent-engine-node is used.

Currently we utilize three lookup strategies corresponding to the degree of polymorphism, which we call monomorphic, polymorphic, and megamorphic. The monomorphic by-class discriminator is a one entry table and is made to be as fast and compact as possible. The polymorphic by-class discriminator is used for a relatively small number of entries and employs a linear lookup strategy. [4] Finally, the megamorphic by-class discriminator handles larger numbers of entries and is implemented as a hash table.

`<if-type-discriminator>` This discriminator has three additional slots: type, then, and else. When invoked, it looks to see if the specified argument is of type type, and then tail-recursively invokes the engine-node or method in then or else depending on the answer. An `<if-type-discriminator>` is normally used when, at a particular argument position, there are just two different specializers remaining to be differentiated to further select methods - `<object>` and something else, which is the type to be checked. if-type discriminators are also used to construct decision trees for differentiating among sets of specializers which the discrimination code does not understand directly, such as some kinds of limited types. [5]

`<typecheck-discriminator>` This discriminator has two additional slots: type and next. If the specified argument is of type type, it tail- recursively invokes the next engine-node or method. If the argument is not of that type, it invokes the absent engine node.

`<singleton-discriminator>` Singleton discriminators store a table-like association of keys with resultant engine nodes, plus a default. Consider

[3] And as will be seen later, reused for providing precomputed discrimination state in recursive partial dispatch.

[4] Currently we switch from polymorphic to megamorphic by-class discriminators when the number of concrete classes exceeds four.

[5] using partial dispatch later, we generalize this to one type which needs to be checked, and another which does not.

- M1: `define method G (x) ... end;`
- M2: `define method G (x == #"foo") ... end;`
- M3: `define method G (x == 3) ... end;`

The discrimination we need to occur is: if the argument is 3, invoke M3; else, if the argument is `#"foo"` invoke M2; else invoke M1. So this could be implemented as a singleton discriminator with keys of 3 and `#"foo"` and a default engine-node of M1.

In practice, we always precede a singleton-discriminator by a by-class-discriminator. This means that each singleton- discriminator contains only elements of the same direct class, permitting specialization and optimization of the lookup strategy.

`todo:`

- Microbenchmark results go here.

5.1 Thread Safety and Synchronization

Engine-node discrimination programs are designed to be able to operate in a multi-threading environment without requiring any synchronization other than at dispatch miss time. handle-missed-dispatch and any other defining (e.g. add-method) or redefining operations use a general object-locking protocol to lock the generic function when they do make any modifications. The side effects that may be performed on engine-nodes are strictly limited to those that will not interfere with ongoing execution, such as a single store of a new engine-node in a slot. For instance, say that a dispatch miss occurs, and the size or representation of a `<by-class-discriminator>` needs to be changed because a new entry is being added. We

1. Create a new table of appropriate size
2. Copy all existing key/value pairs to it
3. Add the new entry to it
4. Generally make it completely set up and ready to use
5. Drop the new one into the place where the old one was - a single write.

If another thread just happens to be looking through the old table, it can just continue on its merry way, because no damaging modifications have been performed to it or any of the discrimination program it points to. If that thread takes a dispatch miss, it will wait until we have finished because of the lock on the generic function. Then (once it can hold the lock) it will either add some more discrimination program, or it might find that what it needed to execute is now already there, and just go around and try again.

5.2 Call Site Caching

Call Site Caching is a general term for utilizing information saved on the argument usage at a particular call site to help predict or otherwise improve performance. Usually it refers to something simple like remembering the types of the arguments used on the last call and the resulting method. Before doing dispatch, a check is performed to see if the arguments are the same. If so, the dispatch is skipped. Otherwise, the dispatch is performed, and the new argument/method set is stored in the cache.

Implementation of this as described has a number of pitfalls for Functional Developer, including multithread synchronization issues (the overhead of storing back multiple pieces of data atomically) and the difficulty of determining whether a type set is 'the same' in the face of singleton and other complex specializers.

Our engine-node paradigm, however, lends itself towards this. Let us presume that when we call a generic function from a piece of code, instead of invoking the discrimination program that is stored in the generic function (and used by all callers to it), we start a new discrimination program just for that call site. This discrimination program will only be constructed for the kinds of arguments used at that call site. Furthermore, it will be tuned to that call site automatically, because the kinds of arguments used at a particular call site will generally vary less than those used with the function as a whole. Consequently we would expect improved performance from any kind of micro-caching of results done at the individual engine node level, plus from any advantage of having fewer discrimination choices at some of the discrimination steps.

For instance, in order to improve the performance of linear by-class discrimination based on the class frequencies, a very simple but thread safe re-ordering mechanism is employed. A start index slot is added to the polymorphic by-class linear discriminator which specifies the index to start scanning. The scanning is then modified to wrap around and stop right before it reaches its start index. The start index is updating upon matching. This mechanism is cheap and thread safe because it atomically updates only one slot and does not require locking. It does though have the disadvantage that the order of the keys does not approach the order of their observed frequencies as would a move-to-front approach.

`todo:`

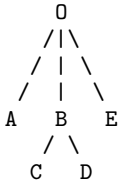
- Speed up results using re-ordering go here.

6 Partial Dispatch

A truly powerful addition to call site caching is informing the discrimination program generator of argument type information determined by the compiler. The whole class hierarchy can be consulted by waiting until execution time. This enables it to prune out methods that are provably not applicable, reducing the number of alternatives at discrimination steps, and even eliminating discrimination steps. The "partial dispatch" could reduce to just a single applicable method; in this case, even though the compiler might not have figured it out, what might have been a full generic function dispatch is now just an indirection to a direct method call.

Call-site specific compile-time inferred types can now lead to much more specialized dispatch trees often only dispatching on one or no arguments before jumping to the most specific method. First, methods are ruled out as impossible candidates because their specializers are disjoint with corresponding compile-time inferred types.

Consider the following class hierarchy



and the following methods on the generic function f:

- 1 - f(A, A)
- 2 - f(C, E)
- 3 - f(D, E)

Suppose that a particular call-site has inferred types B and E for the two arguments. Because of disjointness, method 1 would be ruled out leaving methods 2 and 3 each of which have E as their second specializer. The entire second dispatch step can now be eliminated because the inferred type on this position, E, guarantees that all second arguments will always be instances of E.

The definition of disjointness we use to make these inferences is more powerful than what is used by the compiler to do static method selection, because it is based on the actual, current state of the class hierarchy rather than only immutable characteristics of the hierarchy. For the above example, we find it useful to assert that A and B are disjoint, even though that assumption could be invalidated later by an operation as simple as the creation of a class with A and B as common superclasses. The runtime dependency tracking needed to let us use this power is discussed later.

Concrete-subtype? is a mechanism for leveraging inferred types to yield even more precise types by pushing down through abstract classes until concrete classes are reached. The insight is that abstract classes are not directly instantiable, and thus there can never be any direct instances of them at runtime. For classes C1 and C2, we say that C1 is a concrete subtype of C2 if all concrete subclasses of C1 (which may include C1) are subclasses of C2. So if `<c-back-end>` is the only subclass of `<back-end>`, and `<back-end>` is abstract and `<c-back-end>` concrete, `<back-end>` is a concrete subtype of `<c-back-end>`. Consider also the lattice, with `A`, `B`, and `C` abstract:

```
  A
 / \
B   C
 \ /
  D
```

We can similarly deduce that `A`, `B`, and `C` are all concrete subtypes of `D`. Even more unusual is deducing that `B` and `C` are mutually concrete subtypes. (They are all "concrete type equivalent" - they have the same set of concrete subclasses.) For the more complicated hierarchy

```
    A
   / \
  B   C
 / \ /
D   E
```

in which `D` and `E` are concrete and `A`, `B`, and `C` abstract, we can see that `C` is a concrete-subtype of `B` (if there are any indirect instances of `C`, they are going to be instances of `E` and are therefore instances of `B`), even though the reverse no longer holds.

In practice, we only perform concrete-subtype? on two classes if the classes are joint. Otherwise, we can get quasi-false positives where unrelated classes with no concrete subclasses are noted - correctly - as being subtypes.

6.1 Dependency Tracking

In order to maintain correct partial-dispatch call-site caches, information derived from the current class-hierarchy must be tracked such that when it becomes invalid, such as during dynamic class creation or class redefinition [6] dependent call-sites caches are recalculated. Disjointness can become invalid when two classes that were disjoint suddenly become joint because a common subclass is added. Similarly, concrete-subtype? can become invalid when subclasses are added (e.g., another back-end is loaded).

During the creation of a partial-dispatch tree, disjointness between all specializers and corresponding compile-time inferred types is used to proactively prune out inapplicable methods from further consideration. The disjointness predicate, which operates on arbitrary Dylan type objects, eventually devolves to determining the disjointness relationships of pairs of classes. When a pair of classes is encountered, is considered to be disjoint, and is a candidate for future jointness (i.e. creation of a future common subclass is possible based on sealing restrictions), the generic function is noted as being a "subclass dependent generic" of both classes. In this way, whenever a subclass (direct or indirect) is added to either of these classes, all call sites on that generic will be decached and forced to be recomputed. Dependencies on potentially ephemeral results of concrete-subtype? are tracked similarly. concrete- subtype? eventually devolves to computing the relation on two classes; if such a pair of classes are found, they are concrete-subtype? even though not subtype?, and that relationship is potentially changeable (in view of, e.g., sealing declarations), then the generic is tracked on the subclass dependent generics of the first class, as the relationship can only be changed by addition of new subclasses of that class.

The space overhead is a slot in each class and a sequence element for every subclass dependent generic. The measured overhead is quite reasonable even for large programs. For example, in the Functional Developer compiler, the number of subclass dependent generics totaled 492.

`todo:`

- Need data total number of generics and classes.

6.2 Profile Feedback and Sharing

`todo:`

- SHARING

The proliferation of call-site decision trees increases the importance of sharing of equivalent decision trees, if that does not detract from the micro-caching optimizations which they permit.

Within the same decision tree, there may be multiple equivalent subtrees. We do not address decision subtree sharing in this paper, except to observe that - in our incrementally generated decision trees - the structural equivalence of subtrees is not sufficient to determine that they can be shared; some knowledge about the state from the preceding decisions is needed. (Two subtrees can be structurally equivalent, but it may be only because they are incomplete

[6] The Dylan language does not support most forms of redefinition. The intent of this is that redefinition is a development activity, and consequently need not be supported by Dylan, only by Dylan development environments. This doesn't preclude a Dylan implementation from implementing incremental development support in the Dylan runtime proper, but it does mean we don't need to worry about that level of assault on our type hierarchy here.

- if they were incrementally extended, they would diverge.) We also note that Dujardin ([Dujardin96]) provides a model we believe could be used to track this state efficiently and so address this issue.

The selective sharing of decision trees among multiple call sites can be guided by a number of factors. Intuitively, decision trees that are most similar should be shared: those which handle the same types of arguments most frequently and with similar degrees of polymorphism, and which require the same decisions to be made - i.e., they have similar compile-time-inferred type information. We consider a number of ways these sharing decisions can be made.

In the runtime alone, it can be arranged for all call sites with the same compile-time-inferred type information to use the same decision tree. We experimented with this to good effect, using just simple type equivalence for the inferred type information as the matching criterion. (A more general and strictly correct mechanism would be to upgrade the type information to Dujardin's "pole types" ([Dujardin96]) before matching.) It is also possible to decide - at runtime - that a decision tree has grown too large ("gone megamorphic") and should no longer be maintained as a separate call site decision tree, but instead just use one default megamorphic decision tree.

`todo:`

- Sharing space savings results go here.

Profiling information can be used to refine these decisions. For instance, call sites which are known to be infrequently used, or known to be megamorphic, could be flagged by the compiler so that they automatically shared rather than maintained as separate call site decision trees from the start. As described below, we are able to able to profile individual call sites, and can quantify not only their frequency of use but the degree of polymorphism, the distribution of calls to different methods, and the average call cost. This information could be used to help match, at compile time, call sites which are sufficiently similar that they could be shared without performance loss.

`todo:`

- Megamorphic space savings results go here.

6.3 PROFILE FEEDBACK

Call-site dispatch trees (like Polymorphic Inline Caches) provide a natural framework for collecting runtime call statistics that can be fed back into the compiler to inform future optimization decisions.

- A profiling engine-node - one which just ticks an internal counter and then passes control on to another engine node - can be used near the root of the discrimination program to meter uses of that call site.

- A profiling engine-node can be inserted just in front of a method to meter the uses of a particular method.

Partial dispatch and call-site caches have a runtime and compile-time overhead that can be mitigated by avoiding their expense in low frequency and megamorphic call-sites. Call-site dispatch-tree caches can be instrumented to collect both their call-site frequency, degree of polymorphism, and even their callee distribution.

Partial dispatch is able to tighten the type information inferred at compile-time turning most call-sites into monomorphic decision trees and reducing the number of multimethod dispatch decision steps. While this is an improvement over standard dynamic dispatch, because there is not a compiler present in the runtime, this refined type information can not be used to better optimize downstream code. Arguably the biggest win comes when inlining a specialized copy of the chosen method. This is especially true when either the chosen method's body is sufficiently small (e.g., integer arithmetic) or without inlining would cons a closure. Of secondary importance, fed back profile information can be used to trigger such optimizations as customization, splitting, and dead branch elimination ([Holzle91],) which can leverage and amplify this tighter type information.

Various compression tricks can be utilized to reduce the space overhead of storing inferred types in a call-site cache. A bit mask can be used to code special cases of types. The two most important cases are the frequently occurring ¡object¿ case and the type declaration case. If a specializer is the same across all methods, which we call a type declaration (because it isn't used for discrimination between methods just adds type information), and this can be determined at compile-time, then a bit can be used in place of the type itself. Another trick is to fuzz out types such as singletons to be the class of their respective singleton objects. In practice, we only do this when the singleton object is not a class, as singleton classes play a big role in Dylan. Estimated space saving results.

Even though a compiler is not present in the runtime, profile information can be collected, dumped, and then read back into the compiler in order to produce a more efficient application upon the next compilation. Results for profile feedback.

7 Related Work

7.1 Multimethod Dispatch Caches

There has been considerable work in the area of multimethod dispatch caches. The bulk of the techniques involve shared caches stored in generic functions. Moon ([Moon86]) presents an early mechanism that precomputes shared dispatch caches. Later shared multimethod dispatch cache approaches for CLOS ([Dussud90] and PCL [Kiczales90]) involved both hierarchical dispatch tables and lazy cache creation.

There are several related multimethod dispatch tree approaches. Quinnec ([Queinnec95]) presents a technique involving selectable dispatch tree components. He discusses the possibility of dispatch tree nodes customized to the method structure. He presents only a couple types of tree nodes and does not present experimental results. He suggests a new type of tree node that utilizes a fast subclass algorithm that he also details.

Dujardin ([Dujardin96]) details an approach to organizing and compressing dispatch trees. His approach utilizes subclass tests (as in [Queinnec95]) at the nodes which he find useful for exploiting the regularity of the methods. Although his compression technique shows promise, unfortunately, he only compares his space savings against worst case and does not take into account the cost of the subtype tests.

Chen et al ([Chen94], [Chen95]) present an approach to organizing and compressing multimethod dispatch based on automaton techniques. They present very few actual results and again present only space saving against worst case and do not analyze the actual runtime costs. The major contribution is their automaton perspective which maps fairly well onto dispatch tree approaches.

Many researchers have worked on the problem of compressing dispatch tables. The basic idea is to exploit regularity in the class hierarchy and the set of generic methods to reduce the space consumption of dispatch caches.

7.2 Call-site cache techniques

Deutsch and Schiffman ([Deutsch84]) are considered to be the first to present a call-site cache technique. Their technique involves a per call-site cache of the last argument class and corresponding taken method. A quick check is performed in the prologue of every method comparing the argument class against the last seen class. If they match, then the method is executed, otherwise, a generic dispatch is invoked and the cache is updated. Updating the cache involves patching the call instruction at the call-site and storing the argument class in the new method taken.

This one-entry call-site cache can be extended to better mirror the call-site polymorphism. Holzle et al. ([Holzle91]) introduced the Polymorphic Inline Cache (PIC), which records the n last most recently received argument classes and their corresponding methods. The cache is linearly scanned during lookup and a more expensive dispatch is employed on a cache miss. The PIC provides the facility for record class/method distributions and can be reordered to improve the hit to miss ratio.

The Cecil group ([Chambers93], [Chambers95], [Dean96]) have shown how to extend the PIC to the multimethod context. Their implementation tests all argument positions for each entry unlike our decision tree cache design which checks each argument position only once.

7.3 Type Feedback

Holzle and Ungar's SELF system ([Holzle91],) was the first system to feedback call-site type information to improve subsequent compilations. Their system employed PICs to collect call-site statistics such as number of hits and class distribution to both decide when a call-site warranted further optimization and how to optimize in order to favor the most likely classes. They describe standard optimizations such as inlining and more advanced optimizations such as splitting, uncommon branch elimination, and customization (ref). Their SELF system actually employs an in resident compiler that is triggered when "hot spots" are detected, and thus acts as an on-line adaptive system. The Cecil group ([Chambers95], [Dean96]) extended their work to multimethods. Their system feeds back types in an off-line fashion, but nonetheless, they present remarkable gains from incorporating profile information into subsequent compilations.

Our work shows how to feed back call-site statistics in order to construct more economical caches.

7.4 Link-Time Optimizations

Another approach for performing whole program optimization in the context of separate compilation is link-time optimization. Fernandez ([Fernandez95]) describes a link-time optimization system for Modula-3. It performs data- driven simplification of expressions, conservative static method dispatch, inlining, and specialization. The system performs these optimizations on an intermediate code format that retains the information needed for advanced object- oriented optimizations. Their conservative static method dispatch avoids converting calls when any method could be overridden by a subtype. Our call-site based concrete-subtype mechanism will convert more calls based on actual concrete classes.

The Zuse Translation System (ref) reduces the runtime overhead of software encapsulation mechanisms defined in the programming language Zuse. The system is targeted specifically for Zuse and the Sun-3 platform. It is able to operate on both intermediate code and native object code files.

The Apple Object Pascal linker (ref) replaced dynamic calls with direct calls if an argument type had only one implementation. This is very similar to the idea of concrete-subtype?.

Other link-time optimization system ([Srivastava94]) are very low-level and are usually platform specific. Some of the object-oriented idioms might be transparent enough to allow these low-level systems to operate, but most of the type information needed for object-oriented link-time optimizations would be not be available.

Link-time optimizations address similar problems to our work, but in our view, have several shortcomings. The major problems with link-time optimizations are that they require non standard tools, impose an impediment to incremental development and a "pay as go" runtime, and prevent the delivery of code shareable components. The advantages are that more sophisticated optimizations can be performed, such as inlining and specialization, and indirect calls can be avoided. Our approach performs those optimizations in sealed domains and performs partial dispatch elsewhere. We feel this hybrid approach better matches our constraints and gives best of both worlds performance.

8 Future Work

8.1 Alternatives to the Entry Point Design

The implementation of function and engine-node invocation described here has all been in terms of jumping to addresses (or invoking primitive procedures) fetched out of slots of objects. These are computed gotos. Often these entry points are shared code - part of the Dylan "runtime kernel" - which do their work by examining the object which has been left in the function register for just this purpose.

On some hardware architectures this might be anywhere from slightly to substantially less efficient, especially if the processor is highly reliant on branch prediction. It is important to note however that our implementation as described can be transformed to one without computed gotos with a fairly simple conceptual modification: wherever one stores an object which may be invoked, instead store space for a piece of code which branches/jumps to it. For instance, instead of jumping indirect through the XEP of a function, we jump to the XEP location in a function, which contains executable code. Furthermore, the object itself may not need to be loaded into the function register, as it might be able to be reconstructed within the executing function by pointer manipulation of the PC.

The details of how engine-node and function-entry code would be shared depends on the requirements and limitations of the architecture (e.g., does the code have to use only relative branches and be position independent, what's the size of the allowed branches, etc.), but it's easy to imagine sharing common code within "spaces" from which Dylan objects containing executable instructions are allocated and connecting things together as necessary with glue code and/or branch islands - techniques well used in dynamic linking.

9 Conclusions

We presented an approach to gaining back complete class hierarchy information by delaying the construction of dispatch caches until the whole class hierarchy is available at run-time. Run-time call-site caches can then be constructed as specialized decision trees built from disjointness and concrete-subtype operations on actual arguments combined with compile-time inferred types injected into the run-time. Unnecessary decision steps can be avoided and often run-time dispatch can be completely eliminated. We consider this to be a nice half-way house between full static compilation and dynamic compilation which mitigates the runtime expense of separately compiled components while satisfying our implementation constraints of code shareable components, multi-threaded runtime, incremental development, "pay as you go philosophy", and interoperability with standard tools.

10 Appendix

The following represents a set of early results using different dispatch strategies running the Functional Developer compiler. More complete results will be available at camera-ready time. In the following table, the size is the size in words occupied by all decision trees and support structure in the Functional Developer environment after compilation of a large sample program. The cost is average number of decision steps per call. The Cache Hit Fraction is the proportion of times that a "micro-cache" hits,

Size Cost Cache Hit Fraction Partial Dispatch 182159 1.22 0.72 Simple Call-Site Cache 207900 1.61 0.83 Shared Generic Cache 135712 1.69 0.70 Shared Partial Dispatch 110577 1.23 0.68

11 Acknowledgements

Andrew Shalit was indispensible for helping create a smooth presentation. Keith Playford was involved in numerous fruitful discussions and gave many helpful comments on the paper. Steve Rowley carefully read a draft of this paper and produced very detailed comments. This paper also benefitted from informative discussions with Craig Chambers about multi-method dispatch.

12 Bibliography

[Amiel94] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-methods dispatch using compressed dispatch tables. In Proc. OOPSLA, 1994.

[Amiel96] E. Amiel, E. Dujardin, and E. Simon. Optimizing multi-methods dispatch using compressed dispatch tables. Technical report, INRIA, 1996.

[Chambers93] Craig Chambers. The Cecil Language – Specification and Rationale. Technical Report CSE-TR-93-03-05, University of Washington, 1993.

[Chen94] W. Chen, V. Turau, and W. Klas. Efficient dynamic look-up strategy for multi-methods. In Proc. ECOOP, 1994.

[Chen95] W. Chen and V. Turau. Multiple dispatching based on automata. TAPOS, 1(1), 1995.

[Dean95] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In SIGPLAN Conference on Programming Language Design and Implementation, 1995.

[Dean96] J. Dean, Whole-Program Optimization of Object-Oriented Languages. Dissertation at University of Washington, 1996.

[Deutsch84] L. Peter Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. Proceedings of the 11th Symposium on the Principles of Programming Languages, Salt Lake City, UT, 1984.

[Driesen95] K. Driesen, U. Holzle, and J. Vitek, Message Dispatch on Pipelined Processors. In Proc. ECOOP '95, Aarhus, Denmark, August 1995. Springer-Verlag.

[Dussud90] P. Dussud, TICLOS: An Implementation of CLOS for the Explorer Family, in Proc. Conf. OOPSLA, 1990.

[Dujardin96] E. Dujardin, Efficient dispatch of multi-methods in constant time with dispatch trees. Technical Report 2892, INRIA, 1996.

[Fernandez95] M. Fernandez, Simple and Effective Link-Time Optimization of Modula-3 programs. In Proc, SIGPLAN '95, La Jolla, CA, USA.

[Garret94] Charles D. Garret, Jeffrey Dean, David Grove, and Craig Chambers. Measurement and Application of Dynamic Receiver Class Distributions. Technical report CSE-TR-94-03-05. University of Washington, February 1994.

[Grove95] D. Grove. The Impact of Interprocedural Class Analysis on Optimization. In Proc. CASCON '95, pages 195-203, Toronto, Canada, October 1995.

[Holzle91] Urs Holzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In ECOOP'91 Conference Proceedings, Geneva, 1991. Published as Springer Verlag Lecture Notes in Computer Science 512, Springer Verlag, Berline, 1991.

[Ingals86] D. Ingalls, A Simple Technique for Handling Multiple Polymorphism. In Proc. Conf. OOPSLA, 1986.

[Keene89] Keene, Sonya E. Object-Oriented Programming in Common Lisp, Addison-Wesley 1989.

[Kiczales90] Gregor Kiczales and Luis Rodriguez, Efficient Method Dispatch in PCL, in proc. ACM POPL, 1990.

[Moon86] D. Moon, Object-Oriented Programming with Flavors", Proc. OOPSLA '86, pp. 1-8.

[Stefik86] Stefik, Mark and Danniel G. Bobrow. "Object-Oriented Programming: Themes and Variations," AI Magazine 6:4, Winter 1986.

[Shalit96] Shalit, Andrew, The Dylan Reference Manual, Addison Wesley Developers Press, 1996.

[Srivastava94] A. Srivastava and D. Wall. Link-time optimization of address calculation on a 64-bit architecture. In SIGPLAN Conference on Programming Language Design and Implementation, pages 49-60, 1994.

[Queinnec95] Christian Queinnec, Fast and Compact Dispatching for Dynamic Object-Oriented Languages, INRIA Technical Report, 1995. Submitted for publication, available by ftp on ftp.inria.fr as /INRIA/Projects/icsla/Papers/dispatch.ps.