Dynamo: A Staged Compiler Architecture for Dynamic Program Optimization

Mark Leone and R. Kent Dybvig

Indiana University
Computer Science Department
Technical Report #490

September 1997

Project Summary

Optimizing code at run time is appealing because run-time optimizations can make use of values and invariants that cannot be exploited statically. Dynamic optimization can yield code that is superior to statically optimal code. Recent research has shown that dynamic compilation can dramatically improve the performance of a wide range of applications including network packet demultiplexing, sparse matrix computations, pattern matching, and many forms of mobile code (such as "applets").

Several obstacles have prevented the widespread use of dynamic compilation as a general-purpose optimization technique for stand-alone programs:

- Overhead: Many traditional compilation techniques are too time consuming to perform at run time. Dynamic optimization is profitable only when the time spent optimizing and compiling code is repaid by improved performance.
- Ease of use: Until recently, effective dynamic optimizations could be obtained only manually, by writing code that explicitly generates code at run time. Some progress has been made in building tools and designing languages that better support automatic dynamic optimization, but state-of-the-art systems are still difficult to use.
- Effectiveness: Existing dynamic optimization systems provide a narrow range of optimizations and offer little control over the optimization process. Such systems provide either "lightweight" or "heavyweight" dynamic optimization: they feature a single dynamic compilation model that offers either fast compilation or high-quality code, but not both.

We intend to design and implement a compiler architecture, called Dynamo, that provides effective dynamic optimization with little programmer effort and low run-time overhead.

An important characteristic of Dynamo will be its *staged* compilation model. To reduce the overhead of dynamic compilation, it is necessary to perform some analysis, translation, and optimization prior to execution. To achieve "lightweight" dynamic optimization, most compilation steps are performed statically, resulting in a low-level intermediate representation that admits a few simple dynamic optimizations. "Heavyweight" dynamic optimization employs a high-level intermediate representation and a wide range of optimizations. A staged

compiler architecture supports a broad spectrum of optimization models (including "middleweight" optimizations) by permitting static compilation to be suspended after reaching a high-level, mid-level, or low-level intermediate representation.

Selective dynamic optimization is also critical to obtaining good performance, since the time spent performing unnecessary optimizations contributes to a program's overall execution time. Dynamo will incorporate a suite of automatic program analyses and profiling tools to uncover opportunities for dynamic optimization in ordinary code, and a rich set of optimization directives will give the programmer fine-grained control over dynamic optimization when necessary.

1 Introduction

Compilation is time consuming, but programmers accept its overhead because there is a natural *staging* at work: the one-time cost of optimization is amortized over repeated executions of the entire program or its innermost loops. Other software systems are also staged; for example, an operating system proceeds through a distinct "boot" stage before executing user programs, and user programs often contain distinct stages of execution such as initialization.

Dynamic optimization exploits program staging by postponing certain optimizations until initial stages of execution have been completed. This permits accurate prediction of the program's future behavior and yields code that is superior to statically optimal code. Since the subject program is staged, the overhead of dynamic optimization can be amortized over the execution of the stages that follow. In Section 2 we survey prior research demonstrating that run-time program optimization can dramatically improve the performance of a wide range of applications including network packet demultiplexing, sparse matrix computations, pattern matching, and many forms of mobile code (such as "applets").

Improving the performance of mobile code (as part of "just-in-time" compilation) is perhaps the most obvious application of dynamic optimization. But whereas just-in-time compilation occurs only once per program, when the code is first available, dynamic optimization may occur both as soon as the code is available and at any time thereafter. Benefits are realized in either case when the dynamically optimized portions of the program are sufficiently long-running to compensate for the cost of optimization. This generally requires that the optimized code be executed many times, which can occur even at a relatively fine granularity within a program, e.g., when a dynamically optimized class definition is instantiated many times, a dynamically optimized procedure is called many times, or a dynamically optimized loop body is iterated many times.

The primary obstacle to dynamic program optimization is thus its *overhead*: many traditional compilation techniques are generally too time consuming to perform at run time. The time spent optimizing and compiling code at run time must be more than repaid by improved performance in order to obtain an overall speedup. A simple observation leads to an important technique for reducing this overhead: since the code that will be dynamically optimized is based on statically fixed source code, *preprocessing* can shift much of the effort from run time to compile time. For example, code can be statically compiled to a low-level intermediate representation (such as a machine-code template [20]) that can be quickly optimized at run time. Preprocessing is also used to achieve fast ("just-in-time") compilation

of mobile code written in Java: a Java compiler on the server side preprocesses Java source into Java bytecode, and a client's browser either interprets or compiles the Java bytecode on the fly [18].

A tradeoff exists, however, between the level of preprocessing and the eventual quality of the dynamically generated code: static compilation *restricts* the kinds of optimizations that can be performed dynamically. This occurs because compilation discards information that is useful for both high-level and low-level optimizations. For example, loop transformations are difficult to perform after structured control constructs (such as `for` loops) have been compiled away. As another example, consider the problem of reordering assignments: two assignments that obviously do not conflict (e.g., because one assigns a global and the other a local variable) may appear to conflict after they are compiled to a low-level intermediate representation. To achieve high-quality dynamic optimization, it is sometimes necessary to dynamically compile a high-level intermediate representation or a low-level representation that includes high-level auxiliary information. The development of compact high-level representations and rapid dynamic compilation techniques for high-level code is therefore essential for peak performance.

General-purpose dynamic optimization must also be *selective*: in most cases the bulk of a program should be compiled statically, and only certain regions should be dynamically optimized. Automatically determining *where* to dynamically optimize code is an open research problem. Until recently, selective dynamic optimizations could be obtained only manually, by writing code that explicitly generates code at run time [31, 28, 20, 21, 25, 15, 14]. Some early dynamic optimization systems [11, 7] were not selective at all; they simply postponed all optimization until run time. More recent systems either rely on profiling information to guide *dynamic recompilation* [3, 10] or employ a combination of programmer hints and simple program analysis to determine which portions of a program to dynamically optimize [22, 2, 9].

The question of *how* to dynamically optimize code is also critical, since the overhead of ineffective optimizations harms performance. Ideally, static prediction or profiling information could be used to selectively apply only those dynamic optimizations that are likely to be beneficial, but this is not presently done. Existing systems employ a fixed compilation model: the same compilation techniques are applied to all dynamically optimized regions of a program, regardless of their cost or benefit. Dynamic optimizations can be classified according to their cost and benefit, and range from *lightweight* optimizations, which are fast but typically yield minor improvements, to *heavyweight* optimizations, which are time consuming but often provide substantial performance benefits.

The tradeoff between optimization cost and code quality is illustrated in Figure 1, which depicts the performance of a hypothetical program whose execution time is linear in the size of its input.[1] For small input sizes (less than i), neither form of dynamic optimization yields an overall performance improvement because the dynamically generated code is not executed frequently enough to justify the overhead of creating it. When the input size ranges from i to j, lightweight optimization provides the greatest benefit, whereas heavyweight optimization provides substantial benefit for large inputs (greater than j). Our previous research [22,

[1] This figure assumes that the cost of dynamic optimization, while necessarily dependent upon the input program size, is not dependent upon the input data size, which in our experience is typical though not universal.

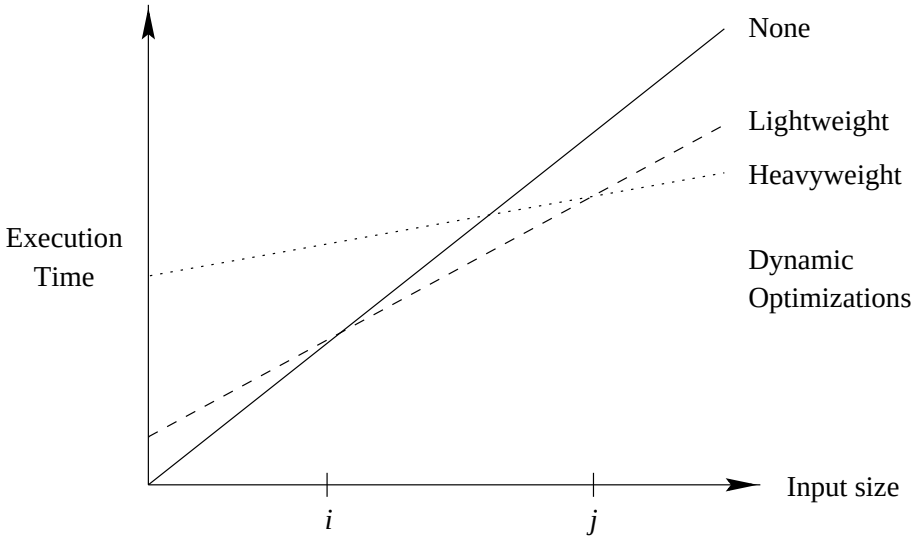

Figure 1: Cost vs. Benefit Tradeoff

23] provides preliminary evidence that some applications benefit only from heavyweight optimizations, whereas others benefit only from lightweight optimizations. Supporting a wide range of dynamic optimization models is therefore desirable.

The remainder of this report describes a compiler architecture for dynamic optimization that addresses all of the above research issues. We first provide motivation for the potential benefits of this technology: Section 2 presents salient examples that demonstrate how dynamic optimization can improve the performance of a wide range of applications. Section 3 details three key aspects of the Dynamo project. First, we propose a *staged* compiler architecture that employs preprocessing to reduce the overhead of dynamic optimization without restricting the range of possible optimizations. Next, we describe how *dynamic optimization directives* further reduce overhead by permitting highly selective optimization. These directives are also the foundation of an experimental test bed for exploring the use of dynamic optimization in real-world applications. Finally, we sketch our plans for incorporating the results of these experiments into automatic program analysis and profiling tools that will largely automate the task of dynamic optimization.

2 The Promise of Dynamic Optimization

This section describes several distinct ways in which dynamic optimization improves performance, and it illustrates the wide range of applications that can benefit.

2.1 Elimination of Interpretive Overhead

The most common use of dynamic optimization is to dynamically compile programs that are constructed at run time or provided as input to a running program. For example, in 1968 Ken Thompson implemented a search algorithm that employed dynamic optimization, based on the observation that a regular expression is a program that specifies the behavior of a finite-

state machine [31]. Previous search algorithms were based on a regular-expression-matching algorithm that was essentially an interpreter. However, repeatedly matching against a fixed regular expression involves duplicated effort, or *interpretive overhead*. Thompson found that compiling a regular expression at run-time into a native-code finite-state machine could amortize this overhead.

Simulation is another application domain that benefits from dynamic optimization. Simulators are general-purpose programs that are parameterized by values that typically remain fixed throughout their execution. For example, a circuit simulator is supplied with a fixed circuit description, and a cache simulator is parameterized by hardware characteristics such as line size, associativity, and coherency policy. These fixed inputs are often tested in the innermost loop of the simulation, resulting in significant interpretive overhead. Dynamic optimization can eliminate this overhead by specializing a simulator to its inputs [8].

Interpreters are commonly used in large-scale systems to permit extensibility without compromising safety or reliability. For example, many Unix operating system kernels contain an extensible *packet filter* procedure that demultiplexes network packets and delivers them to user-level processes. To avoid the overhead of context switching on every packet, a packet filter must be kernel resident. But kernel residence has a distinct disadvantage: it can be difficult for user-level processes to specify precisely the types of packets they wish to receive, because packet selection criteria can be quite complicated. Many useless packets may be delivered as a result, with a consequent degradation of performance.

A commonly adopted solution to this problem is to parameterize a packet filter by a selection predicate that is dynamically constructed by a user-level process [27, 26]. A selection predicate is expressed in the abstract syntax of a "safe" or easily verified programming language, so that it can be trusted by the kernel. But this approach has substantial overhead: the selection predicate is reinterpreted every time a packet is received. Our previous research [22] and related research [16] has demonstrated that dynamic optimization eliminates this overhead: a packet selection predicate can be rapidly compiled into trusted native code. More generally, dynamic optimization permits a kernel to safely and efficiently execute "agents" supplied by user-level processes while avoiding context switches.

2.2 Common-Case Optimizations On Demand

Another frequent application of dynamic optimization is to perform *common-case* optimizations on demand. Optimizing for the common case is an long-standing tenet of systems programmers and compiler writers. This practice typically involves making optimistic assumptions about program inputs and concentrating implementation effort or other resources on improving the performance of the program regions that are most frequently executed.

The primary obstacle to common-case optimization is that it is often difficult to predict statically what the common case will be. One solution is to *partially* or *fully case* a routine, creating specialized code for several different cases. Both of these approaches can cause substantial code growth, and unless these optimizations are automatically applied, significant programmer effort is required. Dynamic optimization yields the benefits of common-case optimizations without either of these drawbacks because the optimization is performed *on demand*: the common case(s) can be determined dynamically, and code space can be reused if necessary.

Pike and colleagues [28] explored the problem of optimizing a `bitblt` procedure for the Blit, a graphics terminal with no special-purpose graphics hardware. `Bitblt` is a general-purpose graphics primitive that is used to copy bitmaps from one region of (screen) memory to another, possibly combining the source bitmap with the destination according to some operation code (such as XOR) that is also supplied as an parameter. Many special cases must be handled by `bitblt`, since bitmaps may be overlapping, clipped, or non-byte aligned, but common cases can be handled quite efficiently. Unfortunately the number of possible cases was found to be too large (over nineteen hundred) to allow full or partial casing. Pike and colleagues used dynamic optimization to generate common-case code on demand and achieved a significant speedup over statically optimized code.

Dynamic optimization was used to perform common-case optimization of interrupt handling in the Synthesis kernel [25]. Rather than performing a full context switch on each interrupt, the Synthesis kernel dynamically synthesizes customized code to save and restore only those portions of the machine state that are used by a particular service routine. For example, it is uncommon for interrupt handlers to issue any floating point instructions, so there is no need to save the state of the floating point unit when servicing most interrupts. Dynamic optimization allows this common case to be exploited, while preserving correct behavior in the more general case.

2.3 Value-Specific Optimizations

Postponing optimization until run-time makes a wide range of *value-specific optimizations* possible [21]. For example, our previous research demonstrated how a general-purpose matrix multiplication routine can be specialized to operate efficiently on sparse matrices using value-specific optimizations [22]. Matrix multiplication is usually implemented as a triply nested loop, where the outer two loops select vectors from the matrices, e.g., a row and a column, and the innermost loop computes their inner product. The vector selected by the outermost loop will be used to compute many inner products, so it is often profitable to create a specialized inner product routine for every such vector. If the vector is sparse, run-time strength reduction can eliminate operations corresponding to its zero elements.

Value-specific optimizations can also be used to create *executable data structures*. Traversing a data structure often requires testing and pointer chasing; this effort is duplicated if the data structure is repeatedly traversed but infrequently modified. The overhead of traversal can be amortized by compiling a data structure into code at run time. For example, the Synthesis kernel employed dynamic optimization to create "self-traversing" buffers and queues [25]. Our prior research has demonstrated that executable data structures can be automatically derived from ordinary code for common list operations, such as membership and association [22].

2.4 Control Flow Optimizations

Another common use of dynamic optimization is the elimination of *dynamic dispatch*. In purely object-oriented languages like Smalltalk and SELF, a method invocation often requires a dynamic type check to determine which method should be used. Compilers for such languages have employed dynamic optimization to eliminate this overhead by creating

customized code to directly invoke methods whose receiver class is known [11, 7]. Operating systems and other interactive systems often employ dynamic dispatching to implement event handling. In the SPIN operating system, dynamic optimization is employed to create a specialized event dispatcher whenever a new event handler is registered [6].

Our prior research has included an investigation of the use of profiling information to improve program control flow through dynamic recompilation [4, 3]. An efficient edge-counting strategy was employed to measure basic block execution frequencies, and this information was used to dynamically reorder blocks to reduce the number of mispredicted branches and instruction cache misses.

3 Staged Compilation

The initial phase of the Dynamo project will involve the design and implemention a compiler architecture that supports a wide range of dynamic optimizations on several distinct intermediate representations. The underlying idea behind this architecture is simple: if the compiler is *staged*, or structured as a sequence of independent compilation phases, dynamic optimization can be achieved simply by postponing the remainder of compilation at a certain stage. Furthermore, the time required to dynamically optimize code can be adjusted by choosing (for each region of code) the stage at which compilation is suspended:

- A region that might benefit from heavyweight dynamic optimization will be partially compiled into a high-level intermediate representation that is suitable for aggressive optimizations such as loop unrolling and register re-allocation.
- A region that would benefit from lightweight dynamic optimizations will be compiled into a mid-level or low-level intermediate representation that can be improved with simple techniques such as constant propagation and peephole optimization.
- Regions that do not benefit from dynamic optimization will be compiled statically into native code.

Although simple in concept, a number of research and design issues must be addressed to implement such a system. Foremost is the question of how to stage the compiler. Most conventional compilers consist of more than one compilation phase, but this staging is introduced primarily to simplify implementation. Staging to support dynamic optimization must take into account several other factors. For example, the most time-consuming analyses and optimizations should be performed in the earliest stages, since they are least likely to be justified at run time. Where possible, optimizations should be unordered, to allow maximum flexibility in choosing when each optimization is performed. Intermediate forms should be compact and amenable to rapid compilation techniques. Each intermediate form should support many different optimizations to permit reordering of optimizations. We present an overview of our proposed compiler architecture in Section 3.1.

An equally important consideration is the speed of the transformations and optimizations employed by the compiler. For dynamic compilation to be effective, it must balance code quality with compilation time. Traditional compilers often rely on optimizations and code generation techniques that are too time consuming to be performed at run time. Section 3.1

also describes our plans to incorporate state-of-the-art rapid compilation techniques into our system.

Automatically determining how to dynamically optimize ordinary code is a challenging problem. Part of the problem is a lack of experience: without in-depth practice at manually making such decisions, it is difficult to devise heuristics to automate the process. In Section 3.2.1 we propose a uniform method for annotating source and intermediate code with dynamic optimization directives, which will provide both the programmer and the compiler with fine-grained control over the dynamic optimization of individual blocks of code. This technique will allow us to implement a system that facilitates experimentation and performance tuning. In Section 3.2.2 we also sketch our plans for the development of automatic program analysis and profiling tools that will enable the compiler to dynamically optimize ordinary code with or without programmer assistance.

3.1 Compiler Architecture

In this section we sketch the initial design of the Dynamo compiler, which will serve as a test bed for experimenting with methods of dynamic optimization and techniques for rapid compilation. We anticipate that this straightforward design will undergo significant refinement as we gain a deeper understanding of the costs and benefits of dynamic optimization in real-world applications.

The compiler will rely on a run-time system that provides automatic storage management (garbage collection) and dynamic reclamation of code space. This affects our selection of source languages, since garbage collection is not feasible for languages that permit unsafe type casts and unrestricted pointer arithmetic. As part of a larger initiative to provide a unified compiler framework for advanced languages and improve their interoperability, we plan to implement compiler front ends for Scheme, ML, and Java. This will allow us to explore language-specific idioms for programmer control over dynamic optimization.

Our emphasis in this report is on the back end of the compiler, which is depicted in Figure 2. After parsing and type checking, a source program will be represented as an abstract syntax tree, which is the input to a static analysis and optimization stage (labeled Alpha). This stage corrects obvious inefficiencies (whether present in the original source or introduced by language-dependent front ends) and performs several program analyses, including control-flow and live-range analysis. Stage Alpha is always performed statically, because its optimizations do not rely on dynamic information.

The resulting code is expressed in a high-level intermediate representation that supports a wide range of analyses and optimizations. Numerous compilation stages operate on this representation; they are depicted by a single box labeled Beta in Figure 2, and for convenience we shall refer to them as a single stage. Control-flow optimizations, such as loop unrolling and procedure inlining, are performed in stage Beta. Data representations are also optimized during this stage; such optimizations are critical to achieving good performance in advanced languages. We anticipate employing techniques similar to those used in the TIL compiler [30] to facilitate such optimizations.

All of the analyses and optimizations performed in stage Beta can benefit significantly from dynamic information. For example, dynamic information is useful because:

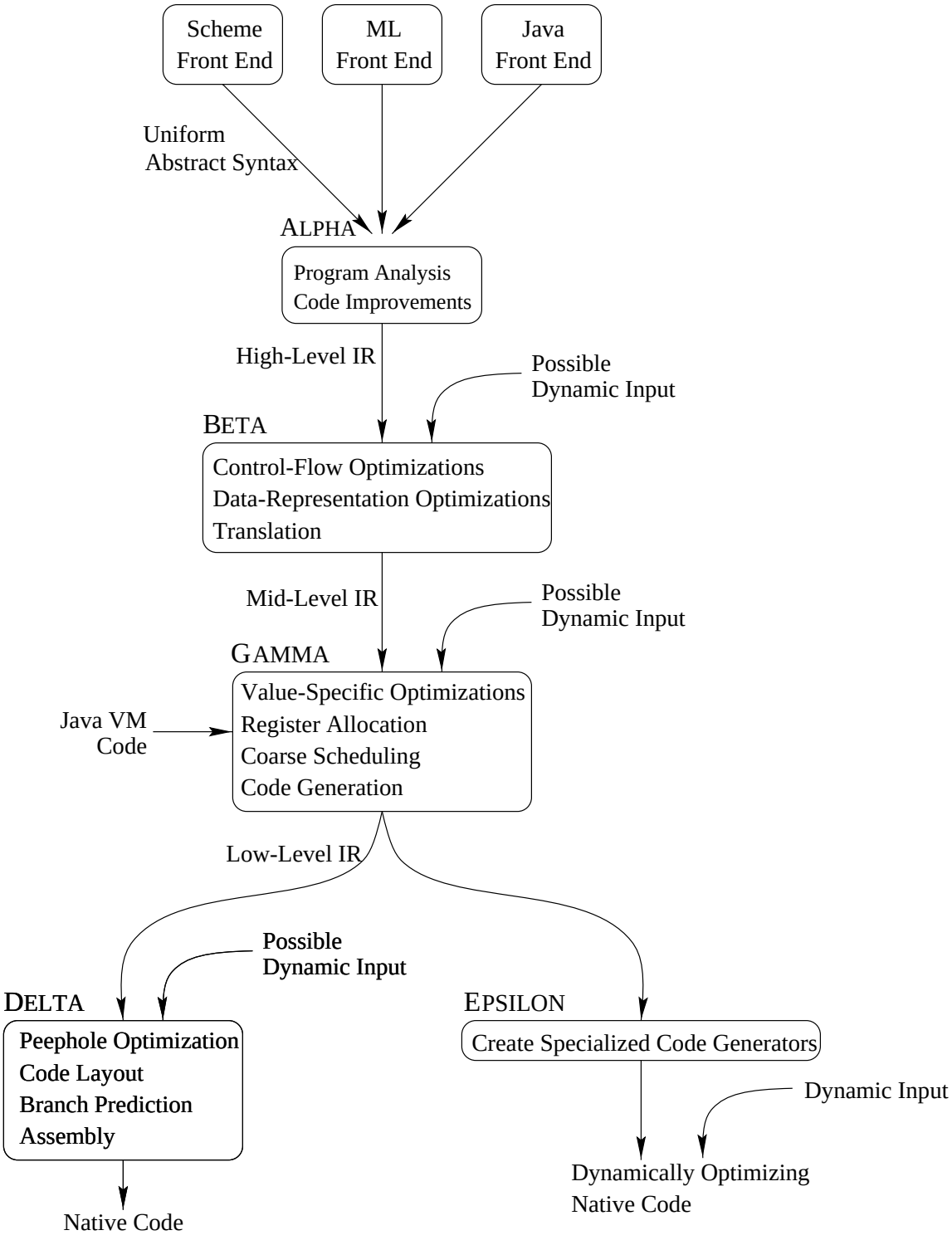

Figure 2: Staged Compiler Architecture

- Dynamic method dispatch and higher-order procedures can make it impossible to determine a program's control flow graph until run time;
- Dynamically allocated data structures can make static loop transformations difficult, e.g., array bounds may not be known at compile time;
- Separate compilation makes it difficult to optimize data representations across module boundaries.

It is therefore possible to postpone part or all of this stage of compilation (and those that follow) until run time (load time or later). To keep the cost of the optimizations performed in stage BETA low, we anticipate incorporating several proven technologies from the *Chez Scheme* compiler, which employs fast linear optimization algorithms to produce optimized programs with minimal compilation overhead.

The next set of compilation stages, GAMMA, operate on a mid-level intermediate representation. This representation is amenable to a variety of machine-independent optimizations, such as code motion and coarse scheduling, because it retains much of the structure of the original program. Abstract syntax trees are used to represent expressions, but most high-level language constructs, such as object/record creation and array indexing, have been compiled away. A number of *value-specific optimizations* [21] are performed in stage GAMMA, ranging from simple constant propagation to aggressive procedure cloning. When this stage is postponed until run time, these optimizations yield code that is specialized to specific run-time values and data structures; for example, a regular-expression matcher can be customized to a particular search pattern. Type-based specialization of polymorphic procedures and generic methods may also be performed in this stage. Stage GAMMA also performs register allocation and code generation, yielding a low-level, machine-dependent intermediate representation.

A key design problem is to keep the overhead imposed by stage GAMMA to a minimum, since the optimizations performed during this stage are the ones most likely to be performed dynamically. We plan to incorporate the results of our prior research in this area, including a fast register allocation algorithm [5] and efficient code generation techniques [12]. We also intend to adopt technology recently developed by researchers concentrating on "just-in-time" compilation of mobile code.[2]

The final compilation stage, labeled DELTA in Figure 2, takes a low-level, machine-dependent intermediate representation as input and performs "lightweight" optimizations such as code layout, constant propagation, and simple peephole optimizations. The emphasis in this stage is rapid generation of optimized native code. When no other dynamic optimizations are performed, DELTA will yield code comparable to that produced by "template-based" run-time code generation systems [29, 6], with comparable cost (20 to 100 cycles per dynamically generated instruction). More expensive low-level optimizations, such as fine-grained instruction scheduling, can also be employed when the compiler (or the programmer) determines that their cost is repaid by improved performance.

[2]Stage GAMMA will be augmented with an interface that permits Java Virtual Machine code [18] to be optimized and compiled on the fly, and we anticipate implementing several network applications (including a simple Web browser) as benchmarks.

Optimization Model	Static Stages	Dynamic Stages
Heavyweight: High cost, aggressive optimization	ALPHA	BETA, GAMMA, DELTA
Mid-weight: Moderate cost, significant optimization	ALPHA, BETA	GAMMA, DELTA
Lightweight: Low cost, simple optimization	ALPHA, BETA, GAMMA	DELTA
Specialized: Trade space for time	ALPHA, BETA, GAMMA, EPSILON	DELTA's optimizations are hard-wired into user code.

Figure 3: Possible Optimization Models for a Program Region

Figure 2 also illustrates an alternate compilation path, labeled EPSILON, that employs compile-time specialization to further reduce the cost of low-level optimization and code generation. Our previous research [24, 22] has demonstrated that this cost can be reduced by an order of magnitude by creating customized code-generators that are "hard wired" to optimize and generate native code for particular procedures or blocks of code. Since this customization is performed statically, its only cost is increased use of code space, which Dynamo can dynamically reclaim.

We also plan to investigate using specialization to reduce the cost of stages BETA and GAMMA: for example, it is possible to reduce the cost of an iterative program analysis by customizing the analysis to a fixed program or control-flow graph.

It is worth emphasizing that these compilation stages are designed to be performed either statically (at compile time) or dynamically (at run time). Also, no global compilation model is assumed: in practice most portions of a program will be statically compiled in full, while the compilation of other regions will be individually suspended at one of the stages described above. Figure 3 illustrates how this can be used to achieve a range of dynamic optimization styles.

3.2 Selective Dynamic Optimization

Dynamic optimization must be *selective* to be effective. Time spent optimizing code at run time contributes directly to the overall execution time of a program, so it is important to determine accurately *where* and *how* to perform dynamic optimizations. The Dynamo compiler must avoid dynamically compiling program regions that do not benefit from run-time optimizations, and it should apply only those optimizations that will significantly improve code quality. Overly conservative strategies are not necessarily the answer, however, because they can result in missed optimization opportunities.

Our previous research [22, 23] has demonstrated that syntactic features of programs (such as loop nests and certain kinds of procedure definitions) provide useful clues that help determine where to perform dynamic optimization. For example, if an inner loop uses a value that is fixed by an outer loop, it can be dynamically optimized once per iteration of the

outer loop. In this project we plan to develop additional program analyses and profile-driven techniques for *identifying optimization candidates*. An overview of these plans is included below, in Section 3.2.2.

After identifying potentially profitable candidates for dynamic optimization, the compiler will perform a *cost-benefit analysis* to refine further the set of optimization candidates. In addition, these analyses will determine *how* to dynamically optimize different regions of code. Each optimization candidate will be annotated with the stage (BETA, GAMMA, etc.) or substage at which its compilation will be suspended, based on the optimization model suggested by cost-benefit analysis. Section 3.2.2 describes the analysis techniques we plan to employ for cost-benefit analysis.

It is foolhardy to automate a process that is not well understood, however. Our intuition can help identify some dynamic optimization candidates, but we need to experiment with a wide range of applications to discover additional opportunities. Furthermore, it will be difficult to implement accurate cost-benefit analysis without quantitative analysis of the performance of both the compiler and the code it generates. We therefore plan to develop a rich set of *dynamic optimization directives* that will supplant the need for program analysis during the initial phase of our investigations. Support for these directives will remain in the system to allow programmers full or partial control over dynamic optimization when desired.

3.2.1 Dynamic Optimization Directives

Initially, the programmer will annotate a program with directives (pragmas) that specify where and how to dynamically optimize it. This will allow us to implement a test bed that supports experimentation with a wide range of subject applications and optimization strategies. We will then extend Dynamo with program analysis and profiling tools that will automatically add such directives to ordinary code, reducing or eliminating the need for programmer guidance. We anticipate that explicit programmer control over dynamic optimization will continue to be useful: our previous research has indicated that domain-specific knowledge is necessary to selectively optimize some applications at run time [22]. We plan to adapt existing visualization tools [32] to provide the programmer with feedback on the costs and benefits of individual optimization directives.

A dynamic optimization directive is a program annotation that may be attached to many different program constructs: procedure definitions, class definitions, loops, blocks, etc. It specifies a number of *actions* that should be dynamically performed by the compiler under certain *conditions*. Actions can provide high-level guidance to the compiler (such as "apply lightweight optimizations to this procedure at run time") or they can specify specific optimizations to be performed, instrumentation to be added, etc. Conditions allow fine-grained control over the circumstances in which actions are taken, which allows the programmer to use dynamic information (such as the value of a variable or the execution frequency of a procedure) to weigh the cost and benefit of performing certain dynamic optimizations.

For example, the following procedure, which is written in pseudo-C-code, computes the sum the elements of a vector `v` of length `n`. It contains two dynamic optimization directives: the first tells the compiler to specialize (clone) this procedure for fixed values of `n` (which may allow the compiler to block the loop and optimize its index arithmetic and bounds checks).

The second directive tells the compiler to unroll the loop completely when it estimates that the resulting code will fit in the instruction cache.

```
double sum(double v[], int n)
#specialize when fixed(n)
{
  int i = 0;
  double result = 0.0;

  #unroll when code_size(sum) < icache_size
  for (; i < n; ++i)
    result += v[i];
  return(result);
}
```

As this example illustrates, the condition in an optimization directive is a full-fledged boolean expression that may reference the dynamic values of variables, invoke procedures, and interact with the compiler and run-time system. This affords great flexibility to the programmer: optimization may be highly selective, based on both static and dynamic context.

Here are some additional examples that demonstrate how dynamic optimization directives might be used in a variety of applications:

- Instrumentation can be statically compiled into a procedure, e.g., to determine the execution frequency of its basic blocks. After it has been invoked a certain number of times, an optimization directive can cause the procedure to be dynamically optimized based on the results of the profiling—with profiling instrumentation removed or left in to gather information for further optimization.

- A general-purpose matrix-multiply routine can be customized to the layout of its arguments when they are sparse. The optimization condition would employ a user-defined procedure to check the condition of the argument matrices.

- A string-comparison procedure can be optimized when one string will be fixed. Doing so will be profitable only if the number of iterations is large, however, so dynamic optimization would be controlled by an optimization directive at its call site, which has access to the contextual information necessary to determine whether this condition will be satisfied.

Dynamic optimization directives provide an important separation of concerns in our research plans. By initially relying on the programmer to indicate where and how code should be optimized at run time, we can concentrate on implementing an infrastructure that supports a wide range of dynamic optimizations. Our design will be evolutionary: by experimenting with real-world applications we expect to discover additional opportunities for dynamic optimization, which can be rapidly incorporated into our compilation system for further experimentation.

3.2.2 Analysis and Profiling for Selective Dynamic Optimization

Automatically determining where and how to optimize code at run time is a difficult task, but it must be addressed in order to make dynamic optimization widely applicable. We intend to develop a set of analysis and profiling techniques that will automatically add dynamic optimization directives to ordinary code; these techniques will also be used to refine existing programmer directives. We also plan to adapt existing visualization and performance analysis tools [32] to provide the programmer with feedback on the costs and benefits of manually directed dynamic optimizations.

Static program analysis and profiling are complementary techniques. Static program analysis has the advantage that it introduces no run-time overhead: the selection of optimization candidates is performed at compile time, based on estimates of the cost and benefit of dynamic optimization. Its disadvantage is a lack of precision: program analysis is conservative and may mispredict the actual dynamic behavior of a program. Profiling provides more precise information about a program's behavior, and it also permits exact information about the cost and benefit of dynamic optimization to be collected. When profile-guided optimization is performed "on the fly," however, the overhead of profiling will contribute at least marginally to the overall execution time of the program even if instrumentation is removed upon recompilation; its cost must therefore be weighed against its benefit.

We plan to employ program analysis and profiling to solve two problems: *candidate selection* and *cost-benefit analysis*. Candidate selection locates program regions that *might* benefit from dynamic optimization. Profiling simplifies candidate selection by identifying "hot spots," or program regions that are frequently executed. Relatively minor improvements in such regions can yield a large overall performance gain, and it is easier to amortize the cost of dynamically optimizing frequently executed code. The compiler will also employ static flow analysis [1] to gain a high-level understanding of a program's control flow. This allows the compiler to estimate execution frequency statically, and it also reveals opportunities for optimization. For example, if the compiler detects that an inner loop uses a variable whose value is fixed in an outer loop, dynamic optimization may yield improvements that are impossible to achieve with traditional loop-invariant removal. Other recent research on static program analyses, such as alias analysis [13] and shape analysis [17], has direct relevance to the problem of candidate selection.

It is relatively straightforward to implement cost-benefit analysis using profiling techniques. The time required to perform certain dynamic optimizations on certain regions of code can be measured, and the improvement (or lack thereof) in the dynamically generated code can quantitatively observed. This technique is highly speculative, since optimizations are attempted without *a priori* knowledge of their costs or benefits; this overhead is likely to be repaid only in long-running programs.

Static estimation of the cost of dynamic optimization has not been previously investigated. We believe that a detailed performance analysis of an early prototype of our compiler will reveal a straightforward relationship between the size and structure of a program and the costs incurred by dynamic optimization and code generation. The experimental framework we propose is well suited to this endeavor: in the initial phases of our investigation, programmer-supplied directives will provide the necessary control over the kinds of dynamic optimizations that are performed.

In contrast, we anticipate difficulty in achieving accurate static estimation of the benefits of dynamic optimization. Because the values that will be used to optimize code will not be available until run time, a "try it and see" optimization strategy cannot be employed at compile time. Instead, each dynamic optimization will require a *static analogue*: given an optimization candidate and information about *which* values will be fixed, the analogue will estimate what kinds of code improvements can be made. In most cases this can be done with a simple dependency or binding-time analysis [19]. Achieving precise results from such an analysis is difficult, because some optimizations (such as constant propagation) enable additional optimizations (such as constant folding). A further complication is that it can be difficult to estimate the actual execution time of code expressed in a high-level intermediate representation; this complicates determining whether certain low-level optimizations will be beneficial. Programmer experience and intuition can be exploited in such matters, and our experimental framework will facilitate programmer involvement in cost-benefit estimation.

References

[1] J. M. Ashley. *A Practical and Flexible Flow Analysis for Higher-Order Languages*. PhD thesis, Indiana University, Bloomington, 1996.

[2] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *PLDI'96 Conference on Programming Language Design and Implementation*, May 1996.

[3] R. G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme*. PhD thesis, Indiana University, Bloomington, Feb 1997.

[4] R. G. Burger and R. K. Dybvig. An infrastructure for profile-driven dynamic recompilation. In preparation.

[5] R. G. Burger, O. Waddell, and R. K. Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 130–138, June 1995.

[6] C. Chambers, S. J. Eggers, J. Auslander, M. Philipose, M. Mock, and P. Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In *WCSSS'96 Workshop on Compiler Support for System Software*, February 1996.

[7] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *PLDI'89 Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

[8] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[9] C. Consel and F. Noël. A general approach to run-time specialization and its application to C. In *POPL'96 Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[10] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *LFP'94 Conference on LISP and Functional Programming*, pages 273–282, June 1994.

[11] L. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk–80 system. In *POPL'84 Symposium on Principles of Programming Languages, Salt Lake City*, pages 297–302, January 1984.

[12] R. K. Dybvig, R. Hieb, and T. Butler. Destination-driven code generation. Technical Report 302, Indiana University Computer Science Department, Feb. 1990.

[13] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI'94 Conference on Programming Language Design and Implementation*, pages 242–256. ACM SIGPLAN, 1994.

[14] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *POPL'96 Symposium on Principles of Programming Languages*, pages 131–144, January 1996.

[15] D. R. Engler and T. A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 263–272. ACM Press, October 1994.

[16] D. R. Engler, D. Wallach, and M. F. Kaashoek. Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533, MIT Laboratory for Computer Science, March 1995.

[17] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL'96 Symposium on Principles of Programming Languages*, pages 1–15, January 1996.

[18] J. Gosling and H. McGilton. The Java language environment: A white paper. Technical report, Sun Microsystems, May 1996.

[19] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[20] D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, November 1991.

[21] D. Keppel, S. J. Eggers, and R. R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[22] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *ACM Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.

[23] M. Leone. *A Principled and Practical Approach to Run-Time Code Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. In preparation.

[24] M. Leone and P. Lee. Lightweight run-time code generation. In *PEPM 94 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.

[25] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Department of Computer Science, Columbia University, 1992.

[26] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Winter 1993 USENIX Conference*, pages 259–269. USENIX Association, January 1993.

[27] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *ACM Symposium on Operating Systems Principles*, pages 39–51. ACM Press, November 1987. An updated version is available as DEC WRL Research Report 87/2.

[28] R. Pike, B. Locanthi, and J. Reiser. Hardware/software trade-offs for bitmap graphics on the Blit. *Software — Practice and Experience*, 15(2):131–151, February 1985.

[29] M. Poletto, D. R. Engler, and M. F. Kaashoek. `tcc`: A template-based compiler for 'C. In *WCSSS'96 Workshop on Compiler Support for System Software*, pages 1–7, February 1996.

[30] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, B. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *PLDI'96 Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[31] K. Thompson. Regular expression search algorithm. *Communications of the Association for Computing Machinery*, 11(6):419–422, June 1968.

[32] O. Waddell. *The Scheme Widget Library User's Manual*. Indiana University, Bloomington, Indiana, 1995.