Evaluating Runtime-Compiled Value-Specific Optimizations

David Keppel, Susan J. Eggers and Robert R. Henry

Technical Report 93-11-02

Abstract

Traditional compiler optimizations are either data-independent or optimize around common data values while retaining correct behavior for uncommon values. This paper examines *value-specific* data-dependent optimizations (VSO), where code is optimized at runtime around particular input values. Because VSO optimizes for the specific case, the resulting code is more efficient. However, since optimization is performed at runtime, the performance improvement must more than pay for the runtime compile costs. We describe two VSO implementation techniques and compare the performance of applications that have been implemented using both VSO and static code. The results demonstrate that VSO produces better code and often for reasonable input sizes. The machine-independent implementations showed speedups of up to 1.5 over static C code, and the machine-dependent versions showed speedups of up to 4.3 over static assembly code.

1 Introduction

Traditional compiler optimizations are performed statically and are either data-independent, or optimize for common data values while retaining correct behavior for uncommon values. Here we examine data-dependent optimizations that are performed at runtime and are based on the actual values of key variables. These *runtime-compiled, value-specific optimizations* (VSO) rely on input variables whose values are constant over some region of the program's execution and are used to produce better code. For example, a runtime optimizer can assign these "runtime constants" to instruction immediates, thereby saving loads and stores and freeing up additional registers; similarly, dynamically unreachable code can be eliminated, as can any tests that would have branched around it. In general, VSO can use all information available to a static compiler, plus information present only at runtime. Thus, runtime-compiled, value-specific optimizations can generate better code than is possible with static compilation.

The first two authors may be reached as at the University of Washington, Department of Computer Science and Engineering, FR-35; Seattle, Washington 98195 or as {pardo,eggers}@cs.washington.edu. The third author may be reached at Tera Computer Company, 400 N. 34th Street, Suite 300; Seattle, Washington 98103 or as rrh@tera.com.

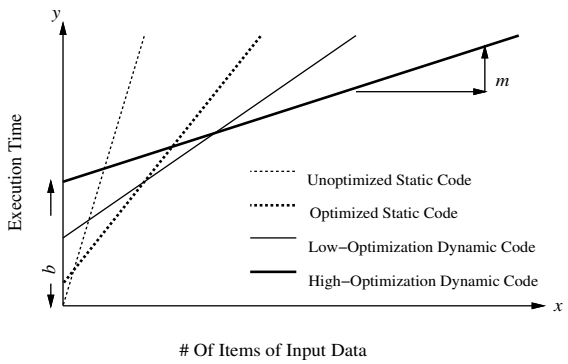

Figure 1: Startup and Asymptotic Costs

Since the optimizer is invoked at runtime, the generated code must run enough faster that it pays for the time spent optimizing. The payback depends on both the runtime cost of optimizing, and how much faster and how many times the runtime-generated code executes. Thus, it is important to perform only the optimizations that have the highest payback for the time invested, and to perform them using only values that are stable long enough that the optimized code gets executed repeatedly. If, as in Figure 1, the general cost of executing some code fragment is $y = mx + b$, the dynamically-compiled code will typically have a larger startup cost, b, caused by the cost of optimizing at runtime, but a smaller incremental (asymptotic) cost m, due to better optimizations. Thus, for sufficiently large inputs, x, the VSO'd code will be executed enough times that the overall cost, y, will be lower than for statically-optimized code.

VSO is partial evaluation [Par91] applied at program

runtime. However, all data values are available at runtime; applying partial evaluation blindly can cause the partial evaluator to slowly consume all input values, instead of producing code optimized around the key values. Thus, it is necessary to control partial evaluation and compilation, so that they do little work, yet still produce substantially better code; the application can then run the improved code on the remaining data. Ultimately, we would like VSO to be performed automatically, as is often done with static partial evaluation. However, we first need to demonstrate that optimizing at runtime provides performance improvements at modest cost. Thus, in these experiments we use partial evaluation applied manually.

The goal of this research is to experimentally examine the costs and benefits of using VSO and to determine where specific optimizations can be successfully applied. Our benchmarks for the experiments are five applications for which we expect speedups from VSO. The experiments consider the performance of four implementations of each program: traditional, statically compiled code; VSO performed using an application-specific compiler and a general-purpose retargetable code generator; hand-written static assembly code; and VSO performed using an application-specific compiler with an application-specific and machine-dependent code generator. For each VSO implementation we identified code that was both a bottleneck in the application and was executed repeatedly with some values held constant. We then supplied a small application-specific dynamic compiler that was invoked at runtime, once key input values were known. The dynamic compiler generated an optimized version of the bottleneck code.

We examine several aspects of runtime-compiled, value-specific optimizations. First, we identify bottlenecks in our workload that are amenable to VSO. Second, we report the costs of compiling at runtime, the speedups achieved when executing VSO'd code, and the *breakeven point* where VSO becomes profitable. Our results show that most programs in our workload achieved asymptotic speedups in the range of 1.06 to 1.50 over well-optimized static C code. In addition, despite the cost of optimizing at runtime, the speedups were usually achieved with reasonable input sizes. Third, although VSO has usually been performed with machine-dependent code [Tho68, PLR85, Loc87, MP89, KEH91, CK93], we show the above speedups using retargetable code generation, linking and instruction cache flushing. Fourth, we describe what optimizations are actually performed and show that runtime-compiled VSO can enable optimizations that can't be performed statically. Thus, VSO can sometimes produce code that is faster than even hand-crafted static assembly. Finally, we show that by using very definite runtime information, VSO with a fast code generator produces code of the same calibre as that produced by a slower, high-quality code generator.

In this paper we use runtime code generation (RTCG) to produce code that is better than the best possible static code. Some previous studies have used RTCG to dynamically improve unoptimized code produced by a fast compiler for an edit-compile-debug environment [Han73, Bro76]. In those cases, however, the best dynamically-generated code was only as good as the best statically-generated code. Other studies have used dynamic type information to eliminate the high overheads of type dispatch in dynamically-typed languages [Mit70, DB77, GW78, DS84, CUL89]. With VSO, types are known statically; yet we can still optimize using the specific values of variables.[1]

The rest of this paper is organized as follows: Section 2 briefly describes why VSO fell from favor and why it is again useful. Section 3 describes the workload and the bottleneck of each application where we applied VSO. Section 4 describes the methodology and experimental framework. Section 5 presents and analyzes the results: the speedup of the VSO'd applications as a whole; the breakeven point at which VSO became profitable; the cost of generating code at runtime; and the optimizations that were seen in practice. Finally, we conclude in Section 6 with a summary of the contributions of the paper.

2 Historical Perspective

In early systems, memory was tight and most programs were written in assembly. VSO was used where performance was important; implemented as ad hoc self-modifying code, these sequences were often smaller and faster [KEH91].

Changing technology reduced the demand for VSO. Memories grew, reducing the I/O component of memory access times. The proliferation of architectures and implementations, coupled with software maintainance costs, meant portability became an issue. Portability was achieved using high-level languages, and most lacked constructs for expressing VSO. Compilers encouraged a distinction between compile-time and run-time, although in principal it is easy to perform compilation at runtime. Finally, there were still many opportunities for improving static code, so development time was best spent improving static optimizations.

Technology and workload changes make VSO attractive once again. First, advances in software methodology have solved some portability problems; for example, machine dependencies have been encapsulated with retargetable code generators [Sta89, FH91] and portable interfaces for instruction cache flushing [Kep91]. Moreover, these tools are now available as library services rather than being bundled in compilers and so on. Sec-

[1] We note that there is a fuzzy distinction between type and value; for example, VSO can also use range information.

ond, static optimizers have improved, so additional static optimizations are progressively harder to discover; in addition, with tighter code, small (runtime) optimizations have a higher percentage payoff. Third, memory access times are once again a bottleneck; VSO uses processor cycles to eliminate memory references. Finally, the VSO breakeven point depends on input size but not on machine speed. Thus, as machines get faster and are used to process larger data sets, there is a greater likelihood that runs of a given application will reach the VSO breakeven point.

3 Workload

Our workload was chosen to satisfy several criteria. First, we selected applications with wide applicability. The second critera was ease of implementation; since each application has four implementations, it was important to use applications that could be built in a modest amount of time. Finally, VSO, like other optimization techniques, is only sometimes applicable; since this study examines the costs and benefits of applying VSO, we selected programs where we expected it to yield speedups.

We identified potential applications as those with one or more variables that were used repeatedly in a critcal inner loop and whose values changed only occasionally at runtime. We inspected the bottleneck code to estimate important features of the application's behavior: what optimizations, and thus what speedups, might be produced using VSO; how often the key variables changed, and thus how many times the VSO'd code would be used. The code generation cost was usually estimated using the size of the statically-generated code, although VSO'd code can be larger or smaller than its static counterpart.

The remainder of this section describes the applications that make up our workload and the routines where we applied VSO.

Sorting The application sorts a collection of records. The order of field comparisons is selected by the user, and thus is known only at program runtime. The sorting routine calls a record comparison predicate to determine the relative order of a given pair of records. The predicate compares individual fields to determine the order. The static code must use a predicate that can be parameterized for any possible sort order. VSO can generate a predicate that is specific to the user's sort order.

Cache Simulation A uniprocessor cache simulator reads a list of memory addresses and performs table (cache) lookup to simulate some cache configuration. The simulation produces cache behavior metrics, such as cache hit ratios. The core table lookup must be flexible, to simulate a variety of cache sizes, associativities, etc. VSO builds a cache lookup function that is optimized to the particular line size and associativity[2] of the cache being simulated [PHH88].

Debugging Debuggers implement conditional breakpoints, which halt a program if certain program variables satisfy a user-specified condition. The debugger detects that a conditional breakpoint has been reached, evaluates the conditional expression, and, if the program is continuing, resumes execution. Breakpoints are generally detected by patching the program text with a trap or jump to code that implements the conditional breakpoint. The code that resumes execution also needs to simulate the instruction that was displaced by the patch. Static code alternatives, therefore, must use an interpreter to evaluate the breakpoint conditional and simulate the displaced instruction. VSO code evaluates the conditional directly and often executes the displaced instruction directly [Kes90].

Data Decompression The compression algorithm is optimized for fast decompression using table lookup. The table maps bytes in the compressed input data stream to strings of bytes in the decompressed output stream. The static code reads a byte from the compressed input stream and uses it as a table index; it then reads the length and address of the output string and dispatches to an unrolled copy loop. VSO generates a decompressor inner loop that corresponds to the lookup table used for the particular file; it uses each input byte to dispatch directly to an unrolled copy that outputs the appropriate string.

Bitblt The bitblt (bit block transfer) algorithm is typically used as a graphics kernel operation, and is an example where VSO has been used successfully in the past [Loc87, PLR85]. The `bitblt` function has several parameters (source and destination alignment; transfer operation) that alter control flow in the innermost loop of a loop nest. These parameters are constant across any given call; therefore, control flow is the same for all iterations in any given invocation. The VSO version of `bitblt` compiles a new version of the function each time it is called. Past studies have achieved VSO speedup by generating partially-unrolled inner loops without inner loop control flow. We extensively optimize the static

[2] Note that VSO optimizes on the user-specified associativity and line size, but not on other configuration parameters. Careful simulator design allowed us to build a fast simulator in which table lookup computations are a principal bottleneck. The design has not sacrificed generality: reported performance is for a simulator that computes metrics for each memory line, and that supports different line sizes and associativity in each cache, multi-level caches, both unified and split I/D caches, write-through and writeback memory update policies, and invalidations in lower-level caches caused by a loss of inclusion in higher-level caches.

code version so that it, too, has most of these optimizations. Here, VSO does a still better job of unrolling the inner loop, and also eliminates redundant assignments.

4 Experimental Framework

This section discusses our experimental methodology: the optimization and code generation strategies, the benchmark measurement techniques, and the metrics used to evaluate and compare the implementations. Each application was implemented four ways: as static code, written in C; with VSO performed using an application-specific compiler and a general-purpose retargetable code generator (called an *IR compiler* below); as static assembly code; and with VSO performed using an application-specific compiler and an application-specific and machine-dependent code generator (a *template compiler*). All experiments were run on a SPARCstation-2.

The static C code represents a traditional implementation of each program using a retargetable code generator. The IR-based VSO counterparts show code quality that can be achieved using retargetable compiler technology. The static assembly code represents an upper bound on performance with static code. The VSO implementations using a template compiler provide a measure of the best possible performance.

4.1 Static Code

The static code versions of each application were extensively hand-optimized. For example, key loops were unrolled by hand in order to turn our state-of-the-practice static compiler, GCC [Sta89], into a state-of-the-art compiler. We feel this yields code close to the best code a static compiler *could* produce, and represents what real programmers do when faced with the challenge of speeding up their programs. Comparing VSO'd code to the best static code also yields conservative results; VSO would show better speedups if compared to less-optimized code.

The static assembly code was similarly tuned. For example, Figure 2 shows prototypical assembly code that implements a record comparison predicate. The routine iterates through a list of <*type*, *offset*> tuples. For each *type*, the routine dispatches to a code fragment that performs the given field comparison. Here, the assembly code version has been implemented as a threaded-code interpreter, with *type* fields encoded as pointers to the corresponding code fragments, in order to reduce the dispatch costs.

4.2 Template Compiler

Each template compiler is specific to both the application and the target machine. The application programmer writes machine code sequences with "holes" called *templates*. The template compiler is invoked at runtime and generates code by concatenating templates in a data-dependent way. For example, if the template compiler unrolls a loop by N, it concatenates N copies of the inner loop template and fills the "holes" with constants, addresses and register numbers. Code generation is one-pass and fast, typically tens of instructions per generated instruction. Since the code sequences are hand-written, code quality is good unless there are major variations in the structure of the generated code. However, portability is poor, since retargeting is done by rewriting machine code sequences and, in some cases, higher-level

```
cmp:                ! call cmp(p0,p1)    iteration
    mov vec,r1      ! tuple list           1
L0: ld (r1),r2      ! get <type>           12
    ld (r1+4),r3    ! get <offset>         12
    add #8,r1       ! ptr++                12
    j *r2           ! dispatch             12

INT:
    ld (p0+r3),r4   ! int compare          12
    ld (p1+r3),r5   ! ...                  12
    sub r5,r4,r0    ! ...                  12
    bz L0           ! continue if =        12
    ret             ! return <, >           2
CHAR:
    ldb (p0+r3),r4 ! char compare
    ldb (p1+r3),r5 ! ...
    sub r5,r4,r0    ! ...
    bz L0           ! continue if =
    ret             ! return <, >

... ! other compare types

DONE:
    mov #0,r0       ! records are =
    ret
```

Figure 2: A prototypical static code function that compares two records by comparing fields in an order described by a list of <*type*, *offset*> tuples, stored in global variable `vec`. *type* is encoded as a pointer to a corresponding code fragment, `INT`, `CHAR`, etc., with `DONE` indicating no more fields to compare. Registers `p0` and `p1` hold pointers to the records being compared; register `r0` holds the return value, which is negative if `p1` is less than `p2`, zero if equal, and positive if greater. The numbers on the right show instructions that are executed, and in which iteration, when comparing two records using the tuple list `{INT,24}, {INT,16}`, where the first pair of fields have identical values and the second pair of fields are different; eighteen instructions are executed.

```
t_int:
    .s0 3,20     ! 3 holes; 20 bytes
    .s1 [0],INT1 ! actual value
    .s1 [1],INT1 ! actual value
    .s1 [2],PCR2 ! pc-relative

    ld (p0+[0]),r4
    ld (p1+[1]),r5
    sub r5,r4,r0
    bz [2]
    ret
```

Figure 3: A prototypical template that compares two integer fields. There are similar templates for other field types, the function prologue, and so on. The notation `[N]` indicates a "hole" that is filled during function compilation with a field offset for loads and the address of the next comparison fragment for the branch. `.s0` and `.s1` are symbol table entries. `.s0` holds the number of holes and the size of the template. The other symbols describe each hole; the first two holes are filled with integer values and the third is filled with a pointer that is encoded as a pc-relative value. As in Figure 2, `p0` and `p1` are pointers to the records being compared.

parts of the compiler. Template compilers have been used by most systems that perform VSO [Tho68, PLR85, Hol87, Loc87, KL89, MP89, Kes90, CK93].

Figure 3 depicts a prototypical template for comparing two integer fields in a record; it corresponds to the `INT` fragment in Figure 2. The template is a code fragment with "holes", plus symbol table information used to patch the holes during compilation.[3] The template compiler in Figure 4 uses one template for each field type, plus a template for the function prologue and an epilogue template for the case of equal records. The compiler traverses the tuple list, building a function with one code fragment per tuple. Figure 5 shows its output when the tuple vector describes a sort on two integer fields. This code is optimized to the specific values in the sort vector, and has no fetches or control flow operations that depend on the user-specified sort order: all operations depend directly on record values. Consequently, it executes half as many instructions as the comparison example of Figure 2.

4.3 IR Compiler

Our other VSO implementation technique eliminates the machine-dependent template compiler and instead uses a

[3] Our assembler lacks a notation for holes and the corresponding symbol table information. We thus encode symbol tables by hand. Since our experiments use each symbol table just two ways, each is implemented as two code fragments.

```
d=tcp(codebuf,t_entry) // prologue
for i=1..vec.n         // unroll
  x=vec[i].offset      //   get <offset>
  switch(vec[i].type){ //   dispatch
  case INT:
    d=tcp(d,t_int,x,x,d+tsz(t_int))
  case CHAR:
    d=tcp(d,t_char,x,x,d+tsz(t_char))
   ...
  case DONE:
    d=tcp(d,t_done)
    iflush(codebuf)
    return(codebuf)
  }
  i=i+1
}
```

Figure 4: A template compiler, which builds a sort predicate by concatenating templates in a data-dependent way. The compiler first generates a function prologue, then builds a function body with one fragment for each field to compare. Loop unrolling, fetching the offset, and type dispatch are all performed here during code generation, instead of being performed each time the predicate is invoked, as in Figure 2. The routine `tcp` copies a template and fills the template holes with the remaining arguments, for example, the field offset, `x`, and the address of the following field comparison fragment.

```
codebuf:          !            iterations
M0: ld (p0+24),r4 !               1
    ld (p1+24),r5 !               1
    sub r5,r4,r0  !               1
    bz M1         !               1
    ret
M1: ld (p0+16),r4 !                2
    ld (p1+16),r5 !                2
    sub r5,r4,r0  !                2
    bz M2         !                2
    ret           !                2
M2: mov #0,r0
    ret
```

Figure 5: A dynamically-compiled value-specific record comparison routine. The routine first compares two integer fields at offset 24, then two integer fields at offset 16. The VSO implementation is fully unrolled and has no dispatching. The numbers to the right correspond to the iteration numbers shown in Figure 2.

retargetable code generator that generates code from an intermediate representation (IR). The application programmer still writes an application-specific compiler, but its output is a machine-independent IR. Executable machine code is generated from a directed acyclic graph IR using a retargetable code generator. IR code generators are primarily used by dynamic compilers, which are already IR-based [Han73, CUL89, VP89]; a few systems that are not inherently IR-based also use IR code generators [PHH88].

The IR compiler has a number of advantages over a template compiler. First, a portable IR frees the application programmer from writing machine-dependent code. Second, simple optimizers can rewrite the IR, which also frees the programmer from worrying about detailed optimizations. Third, the code generator can work on larger code sections than a template compiler; template compilers use fixed code sequences and cannot generally perform optimizations across template boundaries, much as local code generators do not optimize across basic block boundaries.[4]

However, the IR compiler is slower than a template compiler. The IR compiler needs multiple passes to build the IR, perform optimizations, generate code and link. Also, where the template compiler can statically precompute fixed subexpressions, the IR compiler generates a portable IR at runtime and, therefore, must defer machine-dependent computations until application runtime.

The application-specific IR compiler is similar to the template compiler. At program runtime, statically-compiled IR fragments are combined by copying and then filling holes using symbol table information. Holes that represent constants are set to point at IR fragments that represent constants; holes that represent unbound variables are set to point to IR fragments that represent variable references. As with the template compiler, the IR compiler performs machine-independent optimizations such as loop unrolling, type dispatch, and fetching and inlining the offset.

For a particular application, much of the IR detail is the same from invocation to invocation. Thus, our IR compilers build IR from "precompiled" IR fragments, analogous to templates in a template compiler. Each application combines IR fragments to create a complete IR that is passed to the code generator. Using IR fragments can save time, because IR nodes do not need to be manipulated individually.

We could reduce the programmer effort by using a language that allows IR fragments to be specified directly. The fragment in Figure 6 shows what a fragment might look like. The static compiler would translate each fragment to IR and symbol table information which would be saved away for later use [FST91]. At runtime the fragments would be combined as with our IR compiler. Since current languages don't provide constructs for building fragments, we built the IR fragments by hand, avoiding the need to write or extend a compiler.

```
  FRAGMENT
irf_int (char *p0, char *p1) {
  int off;
} {
  int diff;

  diff = *(int*)(p1+off) - *(int*)(p0+off);
  if (diff != 0) {
    return (diff);
  }
}
```

Figure 6: An IR fragment written in a hypothetical high-level language. This fragment corresponds to the assembly template in Figure 3. IR fragments are dynamically inlined into an "empty" function fragment to build an IR tree that represents a predicate like that in Figure 5. In this example, `p0` and `p1` are set to reference the corresponding parameters in the blank function fragment. `off` is a runtime constant filled in during runtime compilation.

We could further reduce the progammer effort by translating whole functions to IR and allowing the application programmer to parameterize the IR in stages. Although this is closer to our goal of automated VSO, it also has a higher runtime cost, because all optimization opportunities are discovered at runtime, once parameter values are known. We speculate that, given a function and a list of parameters and possible values for each parameter, the static compiler could discover optimization opportunities statically. It could thus derive the IR fragments and the IR compiler, and would have runtime behavior equivalent to our IR compiler.

4.4 IR Code Generation Costs

We lack a code generator that reads an IR and produces machine code directly. Thus, for our experiments we simulate a retargetable code generator by building the IR, converting it to C and then calling a retargetable C compiler. We estimate that generating machine code directly from IR takes one thousand instructions per generated instruction. This is conservative, since locally-optimal code generators can produce assembly code in a few hundred instructions per generated instruction [FH91].

[4] For example, the code in Figure 5 is good but still suboptimal. M0 must contain a branch, so its template has a branch. The same template generates M1, so M1 has an unneeded branch on zero.

Our code generation cost estimates use a fast and simple code generator rather than a slower code generator with procedural optimizations, because, as we show in Section 5.3, a simple code generator can produce good code with VSO. Good code is possible, because VSO promotes variables to constants, which has the effect of enabling loop unrolling and thus increasing basic block size, reducing register pressure, and so on. Thus, the code generator is often left only with simpler machine mapping tasks. Delay-slot filling and generating machine code instead of assembly code should add only tens of instructions per instruction. Although procedural register allocation is expensive, our applications typically need only a few registers; a code generator that performs simple allocation, again at tens of instructions per instruction, appears sufficient. Further, most of the applications are relatively insensitive to code generation costs and optimization so, for example, doubling the estimated cost yields similar results (see Section 5.2).

4.5 Metrics and Measurement

The metrics are (1) the dynamic compile time,[5] (2) the asymptotic speedup and (3) the breakeven point at which the total running time of two implementations is equal. The *asymptotic speedup* is the performance improvement for very large inputs; alternatively, it can be viewed as the speedup on processing each data item after breakeven has been reached. The *breakeven point* is the input size at which statically-optimized code and VSO'd code have the same cost. For smaller inputs, the VSO'd code is slower; for larger inputs, faster. To determine the asymptotic speedup and breakeven point, we measured the running time with a variety of input sizes.

We tried various tools to perform measurements. Unfortunately, all had problems, such as limited resolution, intrusion and changes in the code generation strategy. To minimize the side-effects, all measurements were made using elapsed time (wallclock) timers and the short-running phases were placed inside of a loop that executes the phase repeatedly, increasing the running time relative to the timer accuracy. Note that repeating a phase artificially improves cache hit ratios and, therefore, artificially reduces running times.

5 Results

Tables 1 and 2 summarize the important measurements for the workloads. Table 1 shows the asymptotic speedup and the breakeven point for each implementation of each application. The column *Asymptotic Speedup* shows speedups relative to the static C code, showing relative performance of the various strategies[6]. The column *Breakeven Point* shows IR breakeven computed relative to static C and template breakeven computed relative to static assembly, comparing static and dynamic code produced with the same code generation technology. Table 2 summarizes the VSO costs using both the IR and template compilers. The IR compiler costs are broken down into IR build, code generation and linking costs, with code generation cost estimated. VSO with a template compiler is one-pass and cannot be subdivided.

In this section we consider four general aspects of VSO: (1) the speedups for applications as a whole and the input sizes for which VSO provides speedups; (2) the costs of performing VSO; (3) the tradeoff between runtime compile costs and runtime-generated code quality, and (4) the optimizations that were achieved in practice using VSO. We note that speedups, build time and breakeven are all data-dependent and it is easy to construct pathological cases with especially good and bad behavior. However in practice, we observe typically small data-dependent variations.

5.1 Breakeven & Speedup

Table 1 shows asymptotic speedups for applications as a whole, even when substantial time is spent in code that is not improved with VSO. We measured overall speedups, because they better reflect the overall impact of VSO optimizations. Raw speedups of just the VSO'd code can ignore important aspects of an application's overall behavior, such as time spent executing statically-optimized code and cache and I/O effects. For example, sorting divides its time between record comparison and the basic sorting algorithm but only record comparison is VSO'd. The sorting application using VSO IR `-O1` had a 1.40 asymptotic speedup; the VSO'd predicate alone had a 1.90 asymptotic speedup. Likewise, the cache simulator reads a compact binary-format input and is thus an order of magnitude faster than simulators that read text on each run; yet I/O is still about 25% of the running time.[7]

Most applications show good speedups with reasonable input sizes. For example, sorting with VSO pays off when more than about 1,200 records are being sorted. Thus, statically compiled code should be used for sorting directory listings with tens or hundreds of entries, but

[5] An idle processor could perform VSO in parallel with other program computations. However, we report results for a uniprocessor, where VSO costs cannot be overlapped.

[6] The speedup of template over assembly is computed by dividing the template's asymptotic speedup by the assembly's asymptotic speedup. For example, the relative speedup of the assembly implementations of the debugger is 5.4/1.25, which is 4.3.

[7] We note that studying the raw speedups of just the VSO'd routines is also interesting. These speedups would more closely reflect optimizations due solely to VSO. Also, VSO'd code such as the sort predicate can be used in other domains such as graph traversal, where the overhead of the non-VSO'd code is different; raw speedups would help in predicting the performance of the other applications.

Application	Implementation	Asymptotic Speedup (m)	Breakeven Point
Sorting	static C	1.00	–
	VSO IR `-O0`	0.72	never
	VSO IR `-O1`	1.40	1250 records
	VSO IR `-O2`	1.44	1200 records
	static assembly	1.20	–
	VSO template	1.65	10 records
Cache Simulation Direct-Mapped	static C	1.00	–
	VSO IR `-O0`	0.80	never
	VSO IR `-O1`	1.06	1000 refs
	VSO IR `-O2`	1.06	1000 refs
	static assembly	1.09	–
	VSO template	1.14	75 refs
	traditional	0.15	never
Cache Simulation 4-Way Associative	static C	1.00	–
	VSO IR `-O0`	0.80	never
	VSO IR `-O1`	1.10	1000 refs
	VSO IR `-O2`	1.10	1000 refs
	static assembly	1.07	–
	VSO template	1.14	150 refs
	traditional	0.20	never
Debugger	static C	1.00	–
	VSO IR `-O0`	1.34	2000 crossings
	VSO IR `-O1`	1.50	1500 crossings
	VSO IR `-O2`	1.50	1500 crossings
	static assembly	1.25	–
	VSO template	5.40	700 crossings
	traditional	0.0036	never
Decompression	static C	1.00	–
	VSO IR `-O0`	0.53	never
	VSO IR `-O1`	0.97	never
	VSO IR `-O2`	0.97	never
	VSO IR `cc -OX`	1.13	1.19Mb
	static assembly	1.28	–
	VSO template	1.56	24Kb
Bitblt	static C	1.00	–
	VSO IR `-O1`	1.25	megapixels

Table 1: Asymptotic speedups and breakeven points for the five applications. Template and static assembly code speedups are computed relative to static C, showing overall code quality. IR breakeven points are computed relative to static code and template breakeven points are computed relative to static assembly, showing breakeven compared to the best static code with the same code generation strategy. The notation `-On` describes the code generation strategy, with higher numbers indicating more optimization. Entries marked "–" are base cases for computing breakeven. The cache simulator entries marked "traditional" read text input for each run, instead of predecoding to a compact format. The debugger entry marked "traditional" reflects a conventional implementation with high IPC overhead.

Application	Implementation	Compile Time (ms) build	cgen	link	total	% of RT at breakeven
Sorting	VSO IR	5	10	8	23	29%
	template				0.033	27%
Cache Simulation	VSO IR	2	1	7	10	6%
Direct-Mapped	template				0.02	5%
Cache Simulation	VSO IR	10	3	10	23	9%
4-Way Assoc.	template				0.05	7%
Debugger	static C				4	–
	VSO IR	6	2	6	17	40%
	static assembly				5	–
	VSO template				17	83%
Decompression	VSO IR	63	45	11	119	12%
	template				3	18%

Table 2: Startup costs for VSO implementations. The code generation cost for the IR compiler (*cgen*) is estimated. The remaining costs are measured using wallclock timers. Debugger startup costs include the time to download from the debugger to debuggee. The rightmost column is the percentage of time spent during a run in which the breakeven point is just reached. The VSO IR breakeven point is computed using `-O1` speedups. The figures for template compilers reflect breakeven compared to static assembly code. Entries marked "–" are base cases for computing breakeven points. The right column (*% of RT at breakeven*) is the percentage of execution time spent in code generation for program runs that reached breakeven.

VSO'd code is appropriate for sorting larger collections of records from a database.

For cache simulation, the asymptotic speedup is smaller, only 5-10%, but the breakeven point is at only 1,000 references. Since memory reference traces are typically millions of references, VSO "always" pays. Many other cache simulators read text input that is then converted [HLL+93]. Our simulators read trace input in a compact, binary form that is shared from run to run of a simulator. The binary form reduces both the I/O and per-run conversion costs. The *traditional* cache simulator entries in Table 1 show the slowdown for a conventional cache simulator, highlighting both the value of the binary form and the quality of our static-code implementations.

The debugger payoff depends on how the conditional breakpoint is being used. The data in Table 1 shows behavior for a simple conditional expression. Using a more complicated expression would tend to improve the advantage of VSO'd code; setting a breakpoint in a more complicated loop would tend to diminish VSO's asymptotic speedup, but not the breakeven point. Predicting debugger breakeven is harder than with other applications, because it is difficult to predict how many times the VSO'd code will be used. However, we note that the worst VSO times are tiny compared to the user response time. Thus, using VSO does not affect the observed time to set a breakpoint.

Previous conditional breakpoint studies have compared VSO with implementations that require several protection domain crossings on each invocation of the conditional expression [Kes90]. Even with faster IPC mechanisms, crossing costs would still dominate. Here, we make the fairest comparison by implementing both static and VSO'd versions so that the conditional expression executes in the debuggee, without domain crossings. Thus, we report just the speedup due to VSO. In Table 1, the debugger implementation marked *traditional* represents a debugger that requires a domain crossing each time the conditional expression is invoked.

For decompression, GCC performs global common subexpression elimination on the VSO'd code, which reduces code size but increases the running time by adding an extra branch to most iterations. Since most iterations are short, the extra branches hurt performance substantially and thus VSO delivers no speedup. We were unable to make GCC optimize for time instead of space, which would have produced code more representative of code from a fast local code generator. However, the host C compiler at a low optimization (marked `cc -OX`) does not optimize for space and produces code representative of code from a local code generator. Since the code lacks the space-saving optimization, VSO shows speedups. For other applications, GCC was unable to apply the space-saving optimization, and thus code quality was unhurt.

For `bitblt`, the build time, speedup, and breakeven all depend on the size, shape and alignment of the source and destination regions. Although the VSO'd code shows asymptotic speedups, the IR build time is

more than the static code running time on a typical screen region. Here, VSO would pay only for atypically large regions or if the VSO'd `bitblt` could be cached for later reuse.[8] VSO'd `bitblt` was profitable in other experiments [PLR85, Loc87], because they used less-optimized static code and their compiler was hand-coded and machine-dependent and thus fast; yet even there, small regions used static code.

For all applications, the static assembly code implementations are faster than the static C code implementations, showing that there is still room for some static optimization. Note, however, that VSO'd code is often faster than the hand-crafted assembly code (sorting, debugging, associative cache simulation); thus, VSO exposes optimizations that cannot be performed statically.

5.2 Code Generation Costs

VSO startup costs must be repaid before VSO is profitable. For implementations based on the IR compiler, the cost is composed of the times to build the IR (*build* in Table 2), generate machine code (*cgen*) using a 20 MIPS processor,[9] and link the code into the running program (*link*). Cache flush times were negligible [Kep91]. For the debugger, there is an additional cost (reflected in the *total* column) to move code and data from the debugger to the debuggee.

Build, code generation and link costs have similar ratios for all programs, with code generation time typically the smallest. We believe, however, that the build and link times can be reduced substantially, because both the IR and the linker are more general than is needed for VSO. VSO IR manipulation is more complicated than with a traditional IR, because the VSO IR supports holes and hole filling operations; our IR is even more general, supporting hole deletion, which is unused in practice. Similarly, the linker [HO91] links files, rather than code generated in-core, so there is overhead for opening and reading files, converting disk formats to core formats, etc. (For example, 60% of the linker time is typically spent in the `open` and `read` system calls.) Thus, we expect that a linker designed for in-core linking could easily be an order of magnitude faster. Improving build and link times would substantially reduce the startup cost and thus move the breakeven point to smaller inputs.

Even fast retargetable code generators are slow compared to the template compilers, because template compilers produce machine code directly, are one-pass and are specialized to both the application and the target machine. Specializing the retargetable code generator and IR to the application [Hen87] and target machine would improve the template compilers' performance.

Our reported template compiler costs are conservative. We focused on code quality and ease of development, and ignored compiler speed. For example, templates are small, aligned and their size is known statically; each template could be copied at a cost of just a few instructions. However, our compilers copy templates using calls to a general-purpose copy routine that checks for alignment and overlap and is thus an order of magnitude more expensive than necessary. Similarly, template holes are often patched using calls to small out-of-line routines instead of using inlined code.

The right column of Table 2 shows that programs with similar breakeven points may spend vastly different amounts of time performing code generation. For example, at breakeven, code generation is less than 10% of the time for simulation but is 40% to 80% of the debugger's time. Yet both applications reach breakeven with a small number of invocations of the VSO'd code. Simulation has a low breakeven because its compile costs are low. On the other hand, debugging has high enough speedups that its relatively high compile costs are paid back quickly.

5.3 Code Generation Strategies

Our estimated code generation costs are based on the cost of a fast, locally-optimal code generator. The goal of `-O0` optimization is "to produce code quickly" [Sta89]. Code is generated on a statement-by-statement basis and has many redundant loads and stores. It is thus worse than that produced by a fast, locally-optimal code generator. `-O1` optimization performs common optimizations, such as CSE, jump optimization, register allocation, and delay slot filling. The `-O2` optimization level is more expensive and performs more sophisticated optimizations, such as strength reduction and induction variable elimination.

Intuitively, better optimization should produce more efficient code at greater time cost. However, all three code generation strategies produced similar code except that `-O0` had more loads, stores and unfilled branch delay slots. The advanced optimizations of `-O2` were of little use when applied after VSO, because VSO had already applied them in the application-specific compiler. VSO "promotes" variables to constants and then uses the constants to perform optimizations such as constant inlining, loop unrolling and dead code elimination. As shown in Table 1, sophisticated optimizers rarely produce better code.

For some applications, VSO can expose additional opportunities for optimization. For example, the sorting predicate could have been dynamically inlined in the

[8] Although the bitblt source is small, static replication and specialization causes bitblt to grow to about 25 kilobytes of machine code. Some versions of bitblt can merge regions with different bit depths, color operations, and so on [May92]. Since case-by-case specialization leads to combinatorial space explosion, advanced static optimizations are probably impossible with these versions of bitblt.

[9] The estimated code generation costs are computed by multiplying the estimated instruction count by the processor's speed.

Application	Loop Unrolling	Dead Code Elimination	Constant Folding	Constant Inlining
Sorting	√	√		√
Cache Simulation Direct-Mapped			√	√
Cache Simulation 4-Way Associative	√		√	√
Debugger	√	√		√
Decompression				√
Bitblt	√			

Figure 7: Optimizations enabled due to VSO's better runtime information.

sorting routine. For large data sets, the added compile cost would be paid for by the reduction in procedure call overhead (fewer jumps, better register allocation). However, since one of our goals was to keep the implementations simple, we did not study these additional optimizations.

5.4 Optimizations

Figure 7 describes the optimizations seen in practice using VSO. Loop unrolling eliminates both branches and operations needed to compute loop termination. For all applications except `bitblt`, loops were unrolled completely, eliminating all looping overhead. Dead code elimination discards both code that is *dynamically* unreachable and branches around that code. Constant folding takes advantage of runtime variables that are constant over the lifetime of the VSO'd code, folding their values with static constants and other runtime constants. Constant inlining promotes runtime constants from memory variables to instruction immediates, reducing both the number of memory references and the number of instructions executed.

6 Conclusions

In this paper we have studied the benefits of applying runtime-compiled, value-specific optimizations to generate code optimized for specific data values. VSO can be applied to code that has variables that are changed rarely and are used in an important inner loop. We have shown that for some types of applications, VSO more than compensates for the runtime compile costs and produces good speedups over highly tuned static code and at reasonable input sizes. For our workload, the machine-dependent approach used in previous work showed speedups of up to 4.3 over well-optimized assembly code. We also demonstrated that VSO produces speedups with retargetable compiler technology, as high as 1.5. Occasionally, applying VSO did not produce speedups. This occurred when the runtime compiler optimized for space over execution time, when the code generation strategy was too simple, or when the asymptotic speedup was too small to support the cost of compiling for typical inputs.

Counter to what might be expected, increasing levels of runtime compiler optimizations did not produce more efficient code. Higher levels of optimization were of little use when applied after VSO, because VSO performs the optimizations as a consequence of promoting program variables to input constants. Thus, a simple and fast code generator can produce high quality runtime-generated code. Even using IR build and link techniques that were more general (and thus slower) than needed and conservative cost estimates for IR code generation, runtime compiler costs were typically small enough to reach the breakeven point at realistic input sizes. Sometimes, the code produced using IR-compiled VSO (always, for template-compiled VSO) was better than hand-crafted assembly code, showing that VSO exposes optimizations that cannot be performed statically.

This paper advances the study of VSO in several respects. We showed that VSO is a general technique that optimizes on the current values of heavily used, rarely changed variables, and that doing so can produce code better than the best static code. We demonstrated that VSO can use retargetable code generation technology, not just machine-dependent code, and still be profitable. We developed a cost model that incorporates compile cost and the size of the input data, and used that model to decide when to apply VSO. Finally, we showed that with VSO, a fast and simple code generator can produce code comparable to code from a slower, optimizing code generator.

7 Acknowledgements

Craig Chambers, Jim Larus and David Wall reviewed earlier versions of this paper and greatly improved its presentation. Thanks also to Robert Bedicheck for explaining the internals of gdb, from which our debugger is derived [Sta87]. This work was supported by NSF PYI Award #MIP-9058-439, NSF #CDA-8619-663 and Sun Microsystems.

References

[Bro76] P. J. Brown. Throw-away Compiling. *Software – Practice and Experience*, 6:423–434, 1976.

[CK93] R. F. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report UWCSE 93-06-06, University of Washington, 1993.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. *OOPSLA '89 Proceedings*, pages 49–70, October 1989.

[DB77] E. J. Van Dyke and K. A. Van Bree. A Dynamic Incremental Compiler for an Interpretive Language. *Hewlett-Packard Journal*, pages 17–24, July 1977.

[DS84] P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. *11th Annual Symposium on Principles of Programming Languages*, pages 297–302, January 1984.

[FH91] C. W. Fraser and R. R. Henry. Hard-Coding Bottom-Up Code Generation Tables to Save Time and Space. *Software – Practice and Experience*, 21(1):1–12, January 1991.

[FST91] A. Fyfe, I. Soleimanipour, and V. Tatkar. Compiling from Saved State: Fast Incremental Compilation with Traditional Unix Compilers. *USENIX*, pages 161–171, Winter 1991.

[GW78] L. J. Guibas and D. K. Wyatt. Compilation and Delayed Evaluation in APL. *Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 1–8, 1978.

[Han73] G. J. Hansen. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs.* PhD thesis, Carnegie-Mellon University, March 1973.

[Hen87] R. R. Henry. Code Generation by Table Lookup. Technical Report 87-07-07, University of Washington Computer Science, 1987.

[HLL+93] M. D. Hill, J. R. Larus, A. R. Leback, M. Talluri, and D. A. Wood. Wisconsin Architectural Research Tool Set. *Computer Architecture News*, 21(4), August 1993.

[HO91] W. Wilson Ho and Ronald A Olsson. An Approach to Genuine Dynamic Linking. *Software-Practice and Experience*, 21(4):375–390, April 1991.

[Hol87] G. J. Holzmann. PICO - A Picture Editor. *AT&T Technical Journal*, 66(2):2–13, March 1987.

[KEH91] D. Keppel, S. J. Eggers, and R. R. Henry. A Case for Runtime Code Generation. Technical Report UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, November 1991.

[Kep91] D. Keppel. A Portable Interface for On-The-Fly Instruction Space Modification. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–95, April 1991.

[Kes90] P. Kessler. Fast Breakpoints: Design and Implementation. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 78–84, June 1990.

[KL89] P. J. Koopman, Jr. and P. Lee. A Fresh Look at Combinator Graph Reduction (Or, Having a TIGRE by the Tail). *ACM SIGPLAN 1989 Symposium on Programming Language Design and Implementation*, pages 110–119, July 1989.

[Loc87] B. N. Locanthi. Fast BitBlt With asm() and CPP. *European Unix Users Group Conference Proceedings (EUUG)*, September 1987.

[May92] T. May. Personal Communication, May 1992.

[Mit70] J. G. Mitchell. The Design and Construction of Flexible and Efficient Interactive Programming Systems. Technical Report Ph.D. Dissertation, Carnegie-Mellon University, 1970.

[MP89] H. Massalin and C. Pu. Threads and Input/Output in the Synthesis Kernel. *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[Par91] Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation. *SIGPLAN Notices*, 26(9), September 1991.

[PHH88] S. Przybylski, M. Horowitz, and J. Hennessy. Performance Tradeoffs in Cache Design. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 290–298, May 1988.

[PLR85] R. Pike, B. N. Locanthi, and J. F. Reiser. Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software - Practice and Experience*, 15(2):131–151, February 1985.

[Sta87] R. M. Stallman. *GDB Manual: The GNU Source-Level Debugger.* Free Software Foundation, Cambridge, Massachusetts, January 1987.

[Sta89] R. M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Cambridge, Massachusetts, September 1989.

[Tho68] K. Thompson. Regular Expression Search Algorithm. *Communications of the Association for Computing Machinery*, 11(6):419–422, June 1968.

[VP89] S. Vegdahl and U. F. Pleban. The Runtime Environment for Screme, a Scheme Implementation on the 88000. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 172–182, April 1989.