

Profile-Guided Function Specialization in the MIRV Compiler

David A. Greene
Advanced Computer Architecture Lab
The University of Michigan
greened@eecs.umich.edu

Advisor: Trevor Mudge

Submitted for the Preliminary Examination

September 15, 1998

Abstract

Today's complex software systems are written in a modular, re-usable fashion. Because of the desire to optimize programmer time, generic software modules are developed and assembled into more complex systems. These modules are often highly parameterized. Parameters set at execution time often remain constant throughout the program run and across executions. Furthermore, input data sets are often redundant. Because programs exhibit re-use of values, it is possible to specialize programs to particular frequently-occurring values in an attempt to increase performance by eliminating unnecessary overhead. This paper examines a technique to automatically specialize programs written in C. Value profile information is gathered through the use of an instrumenting compiler. The compiler utilizes this information to decide where and on what values to specialize. Preliminary results are presented that show some promising performance gains. To further explore the potential for specialization, more extensive experiments are needed using large programs written in a modular, re-usable style.

1 Introduction

Much of the growth in computer performance continues to be absorbed in more sophisticated software structures that support modularization for re-use and ease of management. These attributes are desirable from a programmer productivity viewpoint, but come at the cost of efficiency. The sheer size of many of today's software systems requires the use of design techniques such as object-oriented programming. These techniques have been developed

to organize the interaction of the various components in the software architecture and to increase the opportunity for code re-use.

Code re-use implies that complex software is written in a generic fashion that allows code modules to be assembled into more complex systems. Because these modules are written to be re-usable, they are usually highly parameterized. The generic nature of the code implies that there is some interpretive aspect to the code that examines the parameter values and acts accordingly. This interpretive aspect usually results in a slowdown of program performance because interpreting the parameters is overhead computation.

This paper examines how generic code can be *specialized* to frequently occurring parameters in order to reduce the interpretive overhead. When generic code is specialized, runtime parameters become constants which can be propagated through the code, allowing a wealth of code transformations to be applied in order to more efficiently implement the algorithm. The system presented here is novel in that it is completely automatic, relying solely on profile information gathered through representative executions of the subject program, and it attempts to apply specialization to generic code, rather than specifically targeted language constructs as in previous studies.

Specialization works because programs are written in a generic, re-usable fashion. They are written this way because program maintenance is simplified. Programmer time is already expensive. The programmer must not only write the actual code, he or she must also debug it, optimize it and prove correctness through quality assurance testing. Specialization helps the optimization phase. To specialize a program, the designer has the option of manually writing specialized code to be invoked when certain values are observed. This is often impractical for several reasons, including time-to-market considerations. Alternatively, a specializer can be used. A dynamic compiler, for example, can be used to automatically generate specialized code at run-time, improving performance. However, the annotations used by current dynamic compilers provide an additional source of errors to the programmer.

An improperly placed annotation (for example, specifying a dereferenced pointer as constant when it is not) can cause inefficient or incorrect program behavior. These errors can be particularly difficult to correct because the code is generated behind the scenes. Most likely, special debuggers and other tools will be necessary to track down these kinds of problems.

For these reasons, we believe a completely automatic method of specialization is required. By augmenting a compiler with the ability to profile values and specialize code automatically, an additional source of errors is eliminated. Once the compiler is shown to be correct, the software designer need not worry about introducing errors through incorrect program annotations. Value profiling provides enough information to the compiler to allow it to decide which values are run-time constants (in the sense that they appear often in the program run) and which code is worth specializing.

Section 2 describes previous work related to specialization. In Section 3 we present our rationale for using value profiling to guide specialization and discuss the implementation of the profiler. The specialization process is described in Section 4. In addition to presenting a specialization technique based on profile information, we also explore how this technique compares to previous efforts with dynamic compilation. We describe the benchmarks used in this study in Section 5. Results of our experiments appear in Section 6. Section 7 concludes with examinations of the tradeoffs in each technique and discussion of the possibility of using profile information as an alternative to dynamic compilation. Future research directions are discussed in Section 8.

2 Previous Work

The work presented here is based on previous work in the area of *partial evaluation* [JGS93]. Partial evaluation attempts to improve program performance by specializing a program to (part of) its input. By choosing the correct subset of input values to specialize against, substantial improvement in performance is possible, especially in programs with a high amount of interpretive overhead.

Originally, partial evaluation was developed as a way to automatically produce compilers and compiler generators [JGS93]. As one specializes an interpreter given the program to interpret, the bytecodes are translated directly to a native implementation. Thus specializing an interpreter to a program is one way to compile the program to native machine code. Traditionally, partial evaluation has been most extensively studied with respect to functional languages [CD93].

Partial evaluation of the C language was studied by Andersen in [And94]. In this work, he discusses the difficulties in specializing C code, especially dealing with implementation dependencies, complex data structures and pointers. Because of C's use of pointers, alias analysis and the ability to treat aggregate objects as partially static are major concerns. The main thrust of this work was to produce *generating extensions* of C programs. A generating extension, given partial input to the subject program, generates code to implement a specialized version of the program. A generating extension is produced by annotating the subject programs with *binding time information* that indicates which parts of the program evaluate constants and which computations must be delayed until runtime. The generating extension essentially “fills in” the constant computations with the actual input values given.

Partial evaluation as a compiler transformation is a relatively new area of study. Much recent work has concentrated on runtime dynamic compilation. The DyC compiler [?, GMP⁺97b, GMP⁺97a] makes use of programmer annotations that indicate which parts of the program should be specialized and which variables are runtime constants. The Tempo [Con98] specializer takes a similar approach, though the granularity of specialization is at the function level rather than the more flexible “specialize anywhere” approach of DyC.

A somewhat different approach is taken by the ‘C group at MIT [EHK96, PEK97, PHEK97]. ‘C is a language developed for dynamic compilation. It is based on ANSI C and contains operators and a runtime environment to describe the creation and compilation of code fragments at run time. Like DyC and Tempo, ‘C requires the programmer to specify

where and on what values to specialize. In 'C, however, the programmer explicitly codes the invocation of the compilation routines, and invokes the compiled code through a function pointer. In DyC and Tempo, the compilation and dispatch code is generated automatically by the compiler. The programmer need only specify where and on which values to specialize. Dispatch is handled behind the scenes.

Profile information was used in a limited fashion to optimize method dispatch for programs written in the Self language [HU94]. Because Self uses an expensive dispatch method to invoke an overloaded method, performance gains can be realized by directly calling (or inlining) the appropriate method when the type of the receiver object is known.

A similar approach has been used to eliminate virtual function call overhead in C++ [GDFC95, AH96] and Cecil [DCG95, GDFC95]. Because the overhead of a C++ virtual function call is not as extreme as in Self, the major benefit of this optimization is the ability to inline and further optimize such calls.

Procedure cloning [CHK93] and inlining [ASG97] have been used to allow interprocedural optimization. By examining call sites with constant parameters, the procedure invoked can be cloned or inlined and the constant parameters propagated through the body. Inlining of called procedures into the clone body is critical, because as constants become available, the inlined code itself can be optimized. The propagation of constant parameters can result in substantial opportunities for optimization.

Before performing the actual cloning and specialization, the compiler described in [CHK93] performed an *Important Variable* analysis to determine which variables in a function would benefit most from being constant. Important Variables are those that (1) determine control flow, (2) determine array size or (3) are used in array index operations. Call sites were examined and any constant arguments were matched up with the Important Variables. A heuristic was employed to determine whether it was beneficial to clone the function and propagate the call site constants.

Procedure cloning as described in [CHK93] is quite similar to the study presented in this paper. However, this study uses profiling in an attempt to characterize the runtime behavior of the program. By using profile information, the compiler can treat certain parameters as constant, even though they do not appear (statically) constant in the source program.

Dynamic compilation and specialization through profiling work because programs are written in a generic fashion. At run time, the code is fed various parameters (either through arguments to the program or through the input dataset) that control the behavior of the program. Furthermore, input datasets are often redundant. For example, a compression program by definition operates on an input set that is expected to contain a high degree of redundancy. Image processing programs also usually work on redundant data; images often contain large areas of similar color, shading and so forth.

The generic nature of programs and input set redundancy cause the machine to manipulate the same or similar data items repeatedly. This phenomenon also forms the basis for work in hardware value prediction [LWS96] and instruction reuse [SS97]. Values can be predicted and instructions reused because programs contain runtime constants that are not known at compile time. Such hardware schemes and partial evaluation are two sides of the same coin. Both work (at least in part) because programs are generic and input sets are redundant.

We have implemented value profiling and function specialization in the MIRV compiler, an experimental research compiler being developed at the University of Michigan. All programs were compiled by the MIRV compiler targeted to a Pentium Pro-class x86 processor. There are several notes that must be made concerning the compiler technology. The MIRV compiler is still in a state of development and thus lacks the level of stability required for measurement of aggressively optimized code. The larger benchmarks could not be compiled with optimizations due to faults in the optimization filters. In addition, the compiler does not perform global register allocation. This results in code that is inferior to that produced

by gcc when its register allocator is enabled.

3 Value Profiling

In order to capture the run-time behavior of programs, we use value profiling [CFE97] to track values to function parameters and global variables at the invocation of each function in the program.

3.1 Why Value Profiling?

Classic partial evaluation specializes a program with respect to part of its input. This input is fed to the specializer at compile time, where the program is transformed to exploit these constant values. This method requires that either the input set be rather generic or the specialized program be very specific and not applicable to a general range of input.

Value profiling provides a way to statically model the behavior of a dynamic compiler. A dynamic compiler has the advantage of exploiting the values produced in each program run independent of any other run. The effect is similar to profiling and running on the same input set, with the additional overhead of the run-time compilation. Profiling eliminates the problem of specializing on a generic input (possibly seeing no benefit whatsoever) by providing actual values seen during profile runs.

Profiling also captures the variation of behavior within a run. A particular piece of code may use variables that have the same value repeatedly, but that value may change as the program runs (i.e., the value is semi-stable). In this case, the code could be specialized to two or more different values, reflecting the fact that there are really several different contexts in which the code is invoked. Specializing the code across many constant values is known as *polyvariant specialization* in the partial evaluation literature.

Value profiling can also be used to get a sense of which values remain constant over different input sets. The stability of data values across input sets was observed by Calder, Feller and Eustace [CFE97] and Gabbay and Mendelson [GM97]. If the value profiler

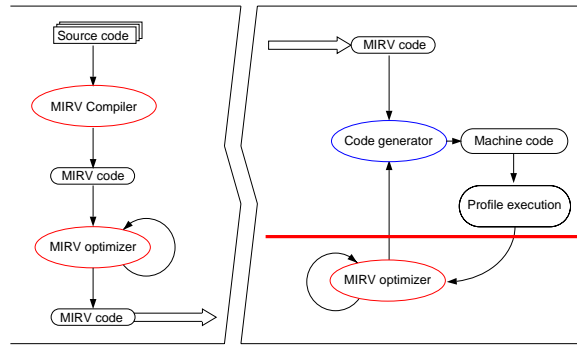


Figure 1: MIRV Optimization Model

maintains a cumulative database of profile runs, parameters and globals that maintain the same values across input sets will tend to remain in the profile, while values that change will be removed. Cumulative profiling eliminates the problem of specializing on *too much* input, which can render the program unusable for generic tasks.

Cumulative value profiling allows the specialized nature of the program to change over time. As programs are run, users provide various input sets, use different features, etc., all of which can change the run-time constant nature of the program. If a cumulative profile is maintained, the program can be periodically recompiled and re-specialized to the new operating environment.

Figure 1 presents the profile-feedback optimization model being explored with MIRV. In this model, a background thread handles program profiling and invokes an optimizer periodically to improve program performance. This is similar to Digital's FX!32 [HH97], which profiles and optimizes Intel x86 binaries to run on the Alpha microprocessor. The model fills a point on an optimizing continuum from static optimization to full dynamic compilation.

The driving force behind using value profiling, however, was to make the specialization process completely automatic. Current specializers (including dynamic compilers) are manual in the sense that the programmer provides input values or specifies which values are runtime constants, and the compiler handles the transformations or dynamic compilation

(they are automatic in the sense that the compilation is performed behind-the-scenes). Our goal is to produce a truly automatic specializer, in the sense that not a single line of source code need be changed to support specialization.

3.2 Implementation

When compiling a program that is to be profiled, the user passes an optimization flag indicating that value profiling should be performed. The compiler instruments each function in the compilation unit to record the value of each parameter and relevant global variable in a value profile database. A global variable is relevant if it is referenced in the function body. Each function has a profile database associated with it. Upon entry to the function body, each parameter and relevant global value is placed into a tuple which is sent to the database to be recorded. The compiler also inserts a call to `atexit` in the `main` function to invoke a profile database output routine when the program exits. The entire value profile for the program is maintained in memory throughout the run of the program. While this can use a substantial amount of memory, we have not found this to be a problem in practice, even for the larger benchmarks tested. If memory usage is a concern, the user can limit the size of the profile database or simply compile some source files without value profiling enabled.

Our profiler is an implementation of the *Top N Value* (TNV) profiler described by Calder, Feller and Eustace [CFE97]. The profiler tracks value tuples, where each tuple is a set of parameter and global values to each function.

As in the Calder, et. al. profiler, we use a least-frequently-used (LFU) replacement policy. Each time a particular tuple is examined, its occurrence count in the profile database is incremented. If the tuple does not exist in the database, it is added. If the profile database for a particular function grows above a certain limit (passed as an option to the compiler), the entry with the lowest occurrence count is replaced.

In addition to using LFU replacement, the Calder, et. al. profiler periodically clears the bottom half of the value profile database (the half with the lower occurrence counts). This

is to prevent the situation where two frequently occurring values battle for a single database entry (profiling `XYXYXYXY`, for example). Our profiler invokes this clear operation (independently on each function database) periodically when a *clear interval* is reached. A clear only occurs when the number of tuples examined since the last clear exceeds this number. The clear interval is twice the occurrence count of the middle profile entry. This allows new values to migrate to the top half of the profile before a clear is invoked. To avoid clearing too frequently, the clear interval is kept to at least 2000 tuple examinations before any clear operation can occur. This clearing strategy is identical to that used in [CFE97].

In addition to recording the values of all parameters and globals, it is useful to record the values of subsets of the tuple. This allows some parameters to be regarded as constants while others are free to take on various values. Profiling tuple subsets allows a function to be specialized with respect to some of its parameters while others remain free. Moreover, a single function can be specialized to different subsets depending on the program state when it was invoked. In the partial evaluation literature this is known as *polyvariant division* (a *division* is a classification of variables as constant or non-constant for the purposes of specialization). Previous studies have shown that both polyvariant specialization and polyvariant division are useful techniques to increase the speedup of specialized code [GMP⁺97a].

In order to allow the greatest number of divisions, the profiler computes the power set of the parameter/global tuple. Each subset is represented by a tuple with the removed elements replaced by “unspecified” values. An unspecified value matches only another unspecified value for the purposes of tuple comparison and recording.

Because the power set of a set contains an exponential number of subsets, this is an expensive operation. The profiler has been written to use efficient data structures and memoized algorithms to compute the power set, but it is still a fundamentally expensive undertaking. For functions with large numbers of parameters (as in the Spec95 go bench-

mark), profiling is expensive. The go benchmark training run (described in Section 6), for example, executes for over one hour on a 200 MHz Intel Pentium Pro processor.

Our value profiling implementation successfully finds all frequently occurring values in the smaller benchmarks studied, and finds a (sometimes surprisingly) large number of such values in the larger benchmarks. Section 6 provides some measurements of the profiler’s effectiveness.

3.3 Issues

There are several subtle issues associated with value profiling targeted toward specialization. During specialization, function inlining may introduce more relevant global variables that have no profile information. This is unfortunate, because these globals too may have stable values in the context of the specialized function. A more advanced profiler could recognize that functions invoked by the profiled function may reference other global variables, and appropriate profiling code could be inserted.

Profiling integer and floating point values is straightforward. Pointer values present an interesting problem: should the profiler track the pointer value, the value being pointed to, or both? Our profiler tracks only the pointer value (the address) because this is straightforward¹. Profiling the contents pointed to would require a much more complex profile map to track what values in the tuple correspond to which addresses. Aggregates add to this complexity. Profiling the contents of an array, for example, would be excessive. Current dynamic compiler implementations overcome this problem by providing annotations that tell the compiler to assume a constant address points to constant data [GMP⁺97a, Con98]². Profiling only pointer addresses creates other problems for specialization that we describe in Section 4.

¹Usually, specializing on addresses is not beneficial, since dynamic memory allocation implies that addresses are not similar across program executions. Local variable addresses, however, can more often be specialized.

²In this case the pointer is *fully static*. If the contents are annotated as non-constant, the pointer is *partially static* [And94].

Once profiling is complete, the database is read by the compiler during the specialization phase. This is described in the following section.

4 Function Specialization

Once a program has been profiled and the function argument value frequencies recorded, the program is re-compiled. During this phase the compiler reads the profile database and maps the most frequently occurring value to the corresponding function arguments and global variables. These variables are then treated as constants for the purposes of specialization.

4.1 Why Specialization?

As mentioned earlier, programs are written in a generic fashion. They are highly parameterized, and often these parameters take on the same value over and over again as the program is run and re-run. A typical example is a program such as a compiler, which takes many run-time arguments to describe the types of optimization to perform. Usually the user has a favorite set of optimization flags that are used often. By specializing to these settings, efficiency can be gained.

Programs also operate on redundant data. The example of data compression was given earlier. If the data is compressed at all, it means that the input set contains redundant data. It may be possible to specialize the data transformations on commonly occurring values to speed up the compression.

Many programs simply interpret data. A bytecode emulator is one example. The bytecode array remains constant throughout the run of the program. The algorithm spends much time fetching, decoding and dispatching opcodes. Looping constructs imply that these operations occur over the same data repeatedly. By specializing the interpreter to the bytecode, this overhead can be eliminated.

Finally, programs sometimes employ algorithms that operate on the same data repeat-

```
int fib(int n)
{
    if (n == 0 || n == 1) {
        return(1);
    }
    else {
        return(fib(n-1) + fib(n-2));
    }
}
```

Figure 2: Recursive Fibonacci Implementation

edly due to the nature of the computation. A well-known example is a recursive implementation of the Fibonacci sequence.

Specialization relieves the programmer of the burden of writing special-case code for common situations. Because less code is written, there is less chance of error and program development time and cost is reduced. Specialization trades off offline processor time (to profile and compile the specialized program) to save online processor time and the more expensive part of software development: programmer time.

4.2 Specialization Example

As mentioned earlier, the Fibonacci sequence presents an opportunity for specialization because a recursive implementation repeatedly computes the same values. It is important to note that the Fibonacci sequence is more efficiently coded using a memoized algorithm. The recursive implementation was chosen because it is straightforward to specialize and shows most of the opportunities that arise during the specialization process. In Section 6 we examine both recursive and memoized versions of the Fibonacci algorithm specialized to profile information.

Figure 2 shows a recursive implementation of the Fibonacci sequence. The argument `n` determines the control-flow of the procedure, making it a prime candidate for specialization (recall the Important Variable analysis in [CHK93]). It is also used in the arguments to the recursive calls, meaning the calls can be inlined, allowing further specialization.

```
int fib_3(int n = 3)
{
    return(fib(2) + fib(1));
}
```

Figure 3: Specialization to $n = 3$

```
int fib_3(int n = 3)
{
    return(fib(1)+fib(0) + 1);
}
```

Figure 4: Specialization to $n = 3$, Inlined

```
int fib_3(int n = 3)
{
    return(3);
}
```

Figure 5: Specialization to $n = 3$, Final Version

Figure 3 shows the function body after specialization to the value 3. The `if` statement has disappeared because we know the condition is false. In addition, the recursive call arguments are constant-valued, allowing us to inline and specialize the calls. Figure 4 is the result.

We have omitted the inlined control flow from Figure 4 because it too can be specialized away. The final version of our specialized function appears in Figure 5, where constant folding has been applied. The function now returns a single constant value with no computation necessary.

If we specialize the Fibonacci sequence on enough values, we essentially create an algorithm that memoizes the most common values of the Fibonacci sequence, with no extra effort on the part of the programmer. For example, if we were to specialize to the value of `fib(10)`, execution of `fib(20)` would be much faster, because the call tree has been halved in height. Note that specialization to `fib(N)` implies specialization to `fib(N-1)`, `fib(N-2)`...`fib(0)`, but only `fib(N)` and `fib(N-1)` will ever be encountered when the specialized version is ex-

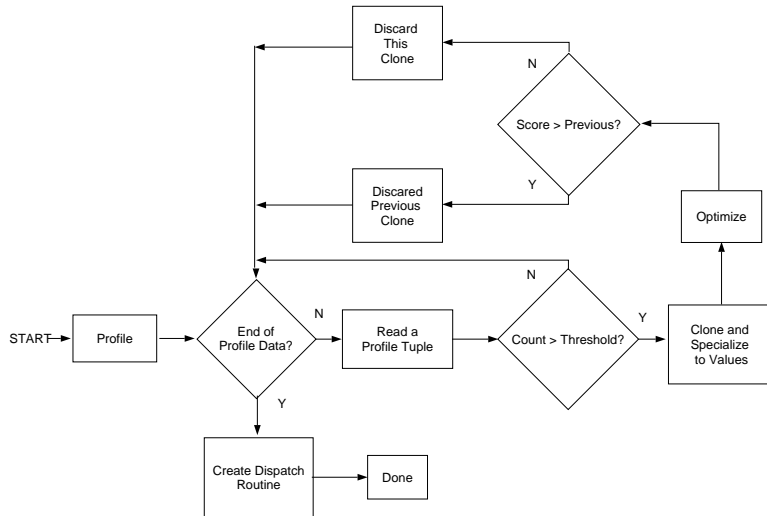


Figure 6: Function Specialization Process

ecuted, since they return constant values. The remaining specializations occur because computing $\text{fib}(N)$ and $\text{fib}(N-1)$ requires the computation of the shorter sequences. These specializations can be discarded if the compiler can recognize that they are not needed. Our implementation currently does not have this capability.

4.3 Implementation

In order to perform specialization, the compiler must make two decisions. It must decide which parameter values should be considered constant for the purpose of specialization and it must determine the values of these constants. The profile database provides information to answer both questions.

Figure 6 presents a flowchart of the specializer operation. The first task of the specializer is to examine the profile database for each function and decide which parameters and global variables should be considered constant. Because we wish to support polyvariant division, many different subsets may appear in the profile database. This can cause conflicts in the specializer because there may be subsets that overlap one another. For example, a function with arguments a and b may be called many times with $a = 3$ and $b = 4$. Tuples $(3, 4)$, $(-, 4)$ and $(3, -)$ (where $'-'$ is an unspecified value) may all appear in the profile database.

At run time, if `a = 3` and `b = 4`, only one of three specializations (one for each tuple) can be chosen for execution, because the dispatch routine compares arguments to specialization tuples to decide which function to invoke³. Thus the other two are not relevant to the final specialized program. We describe the resolution of this conflict below.

The specializer examines each tuple in the database. If the occurrence count for a tuple is above some threshold (provided by the user on the command line), the corresponding function body is cloned and specialization begins.

The specialization process itself is quite straightforward. Because specialization is simply the application of constant propagation, constant folding and any further transformations enabled by these operations, support already exists in the compiler to perform the majority of the specialization work. To enable the use of existing transformation code, the specializer inserts constant assignments to each specified value in the profile tuple. For example, our example function would have the statements `a = 3; b = 4;` inserted at the top of the function body. These assignments create the start point for constant propagation and folding. Constant propagation and folding proceed as in any optimizing compiler. The compiler then performs function inlining, loop unrolling, strength reduction and dead-code elimination. The cycle is then repeated a second time to allow further constant propagation, folding and optimization.

While the optimizing passes transform the code, they keep track of various statistics. The constant propagation filter, for example, counts the number of times propagation is performed. These statistics are registered with a global manager so that any filter may examine the statistics of any other filter.

These statistics are used to resolve the tuple conflicts described above. After each tuple is examined and specialization is performed, a score is calculated for the specialized versions based on the number of code transformations performed. Each unspecified value in a tuple

³If the dispatch routine sees tuple (3, 4), after comparing the first argument to 3 it would have to decide whether to call a routine specialized on (3, -) or continue comparing arguments, hoping for a more exact match. We wish to avoid this decision at run-time to reduce dispatch time.

is treated as a wildcard, allowing it to match any value in any tuple. The tuple's score is compared against the score of a previous tuple that matches it (there should only be one). The tuple with the higher score is preferred. The other tuple and its specialized clone are discarded.

This greedy algorithm could be tweaked to obtain better performance of the specialized code. Let us return to the previous conflict example. Say that tuple (3, 4) has a score of 30 while tuple (3, -) has a score of 28. It may be advantageous to prefer the second tuple over the first, because fewer values are specified. This reduces dispatch overhead, which may offset the lower specialization score.

This algorithm is rather slow because every possible specialization tuple results in the specialization of a function. Because the optimization score is already a simplistic estimate of the potential execution improvement, the specializer could estimate the benefits of specialization through code analysis. Important Variable analysis could be extended to provide a more accurate estimate of the specialization benefit. Only those tuples with specified values for the Important Variables need be examined further.

Once specialization has been completed and the final tuple set chosen, the original function body is converted into a dispatch function (the original body is saved so that it may be referenced when necessary). The dispatch function is essentially a large if-else tree that compares the parameters and relevant global variables to determine which specialized function should be invoked. Figure 7 shows a dispatch function for the Fibonacci example.

Note that the dispatch function contains a call to the original `fib` implementation if the argument does not match any of the specialization tuples. In addition, note that the most common values of `n` have their value tested first. This is because ordering comparison trees to test the most common values first lowers the dynamic branch count, potentially increasing performance [YUW98]. In this case, branch ordering actually hurts performance because the most common profiled versions of `fib` are never invoked (they are covered by

```
int fib(int n)
{
    if (n == 0) {
        return(fib_0(n));
    }
    else if (n == 1) {
        return(fib_1(n));
    }
    else if (n == 2) {
        return(fib_2(n));
    }
    else if (n == 3) {
        return(fib_3(n));
    }
    else if (n == 3) {
        return(fib_4(n));
    }
    else if (n == 5) {
        return(fib_5(n));
    }
    else {
        return(fib_orig(n));
    }
}
```

Figure 7: fib Dispatch Function

```
int fib(int n)
{
    if (n == 0) {
        return(1);
    }
    else if (n == 1) {
        return(1);
    }
    else if (n == 2) {
        return(2);
    }
    else if (n == 3) {
        return(3);
    }
    else if (n == 4) {
        return(5);
    }
    else if (n == 5) {
        return(8);
    }
    else {
        return(fib_orig(n));
    }
}
```

Figure 8: `fib` Inlined Dispatch Function

`fib_5` and `fib_4`, since these functions never make recursive calls to compute the shorter sequences). The maximum number of branches must be evaluated before these versions can be called. This problem could be solved by further profiling and optimizing the program.

A final optimization pass inlines the original function and all specialized versions into the dispatch function. This avoids the extra function call overhead apparent in Figure 7. Figure 8 shows the final dispatch function for `fib`.

4.4 Issues

Specialization of programs written in C is problematic due to the use of pointers. Specializing on pointer and aggregate arguments and global variables is critical, since pointers and aggregates are often passed as function arguments or used as global variables. The

main problem here is profiling the value information efficiently. To solve this problem, we propose using the Important Variable analysis to direct the profiler. Only the Important Variables (which can be expanded to include “Important Aggregate Elements”) need be examined. This coupled with the subset pruning described in Section 3 may be a practical value profiling mechanism.

A classic specialization example involves optimization of an interpreter. If we assume that the interpreted bytecode array is constant throughout the run of the program, we can specialize on it and eliminate the fetch and decode overhead of the interpreter. However, profiling the contents of a bytecode array is not feasible. This is one example where a dynamic compiler has a distinct advantage over profile-based specialization. Programmer annotations indicate to the dynamic compiler that the array should be treated as a constant. Furthermore, the dispatch mechanism need only check the address of the array passed to the interpreter routine (i.e., there is no need to check every element before dispatching). In addition, there is no need to perform loads from the bytecode array when specialized code is run. The user provides an explicit annotation indicating that if the address remains constant, so do the contents at the address and throughout the array [GMP⁺97a], which avoids the overhead of performing a load. This, of course, presents serious problems if the programmer provides incorrect annotations, as the compiler will assume pointed-to memory is constant when it is not!

5 Benchmarks

In order to evaluate the effectiveness of our profile-based specializer, we optimized a few small programs, some of them from the ‘C benchmark suite [PEK97, PHEK97]. We chose programs from the ‘C suite because some of these programs are also used by the DyC and Tempo groups. This provides a basis for comparison.

In addition to these small kernels, we also profile and specialize the compress benchmark from the Spec92 suite and the go benchmark from the Spec95 suite. As we elaborate in

```

hashElement *hash(int value, int scatter, int norm, int size)
{
    int hash = ((value*scatter)/norm) % size;
    hashElement *he;
    hashElement *front;

    hash = abs(hash);
    he = table.array[hash];
    front = he;

    he = find(he, value);

    if (he == 0) {
        hashElement *new = (hashElement *)malloc(sizeof(hashElement));
        hashElement *temp = front;

        new->key = value;
        new->next = front;
        table.array[hash] = new;
        return(0);
    }
    else {
        return(he);
    }
}

```

Figure 9: hash Benchmark

Section 6, we encountered difficulties working with these benchmarks. Thus results for these two benchmarks cannot be considered conclusive. The present state of the compiler is the main reason more benchmarks could not be examined. We plan to extend the study in the near future to encompass the entire Spec92 and Spec95 integer benchmark suites.

5.1 Benchmark Descriptions

Our first benchmark is the Fibonacci implementation described in Section 4. We include the **fib** benchmark because of the obvious applicability of specialization. Thus it serves as a reasonable test of the correctness of the specialization implementation.

The second benchmark is taken from the ‘C suite. It is an implementation of a hash table using multiplication-based hashing. The source of the **hash** function appears in Figure 9.

The hash function is a target for specialization because `scatter`, `norm` and `size` all remain constant throughout the program run. By specializing through profiling, the assumption is that the user will rarely change these values (as part of a larger program, the hash function would not be accessible to the user at all, and would most likely remain at least semi-stable between program runs).

The benchmark has been changed slightly to accommodate the profiling scheme. Instead of sending `scatter` and `norm` as members in a `struct`, they have been provided as separate arguments. A profiler that can examine individual aggregate parameter fields would be able to handle the benchmark as originally written. We briefly discussed the implementation of such a profiler in Section 4. In addition, we do not pass the address of the hash table. Instead, the table is a global `struct`. This prevents the specializer from specializing on this address, which is allocated at run-time. We found that allowing specialization to the address hurt performance.

By combining `fib` and `hash`, we can obtain a more realistic implementation of a Fibonacci sequence: `fib_memoized`. A memoized version of `fib` will run much faster than even a specialized recursive version, because the memoization mechanism is essentially an on-line value profiler. It does not suffer the constraints of a limited value profile. In the memoized version, the user would like fast lookup of values. Thus a hash table may be employed. By specializing the hash function, we may increase the performance of even a memoized Fibonacci implementation. This benchmark may show that specialization can be applied even to performance-tuned algorithms.

The fourth benchmark, `exp`, also comes from the 'C suite. The rationale for using this benchmark comes from computer graphics [PHEK97]. In certain algorithms, an exponentiation function may be applied to a data set. The function remains the same throughout the application. By specializing on the exponent, the compiler can fully unroll the loop and eliminate some control flow. The source appears in Figure 10. The benchmark has been

```

long exp(int base, int exp)
{
    int bit;
    long result;

    if (1 & exp) {
        result = base;
    }
    else {
        result = 1;
    }

    for(bit = 2; bit <= exp, bit <<= 1) {
        base *= base;
        if (bit & exp) {
            result *= base;
        }
    }

    return(result);
}

```

Figure 10: exp Benchmark

changed to return a `long` instead of a `double` due to problems with floating point support in the compiler.

Our fifth benchmark is `sort`. It is based on a 'C sorting benchmark. We use quicksort instead of the heapsort used by the 'C group. When sorting a container, it is necessary to swap elements repeatedly to put them in their correct positions. If these objects are large, this necessitates a block copy. One option is to invoke the C `memcpy` routine. This involves function call and potential loop overhead.

Figure 11 presents an alternate method for copying large objects. This swap routine essentially embeds `memcpy`, eliminating the function call overhead⁴. The looping overhead remains, however. This overhead is significant, as the actual copying work involves only 14 instructions⁵. By specializing the swap routine on the size of the objects being swapped,

⁴As in the 'C benchmark, our `swap` routine is restricted to only work with objects that are a multiple of `sizeof(long)` (4 bytes on a Pentium Pro).

⁵The overhead would be even less if the compiler performed proper register allocation.

```
void swap(void *a, void *b, int size)
{
    int i;
    long temp;
    int bound = size / 4;

    for(i = 0; i < bound; i++) {
        temp = ((long *)a)[i];
        ((long *)a)[i] = ((long *)b)[i];
        ((long *)b)[i] = temp;
    }
}
```

Figure 11: swap Routine

the loop can be completely unrolled, eliminating the loop overhead and creating constant array indices.

The final two benchmarks are **compress92** and **go**, described above. These two benchmarks are quite a bit larger and contain much more complex code than the other simple kernels. It is because of this complexity that the optimizer currently fails to run on these benchmarks. However, it is critical to get some idea of how specialization carries over from small and somewhat contrived examples to more realistic programs.

6 Results

To measure the speedup obtained by profiling and specializing code, we profiled and specialized our benchmarks as described in Sections 3 and 4. Both **go** and **compress92** were successfully profiled. Because MIRV could not compile them with optimizations, we manually specialized the code, performing the same actions the specializer would have performed. Every attempt was made to adhere to the optimization rules in use by the MIRV compiler with respect to pointer alias analysis and other dataflow concerns.

For the **compress92** benchmark, this manual process was not as painful as we expected. The **go** benchmark is much larger, and manually specializing functions with every frequently occurring tuple was infeasible. Thus the results for **go** reflect application of only a small

Table 1: Benchmark Training Runs

Benchmark	Profile Input	Profile Time (sec.)	Time No Profiling (sec.)
fib	n = 20	0.77	0.01
hash	scatter = 1024 norm = 17 table size = 1048576 30,000 random keys	92.07	0.29
fib_memoized	scatter = 1024 norm = 17 table size = 1048576 n = 20	0.27	0.04
exp	e = 13 10,000 random numbers	1.08	0.06
sort	10,000 16-byte random items	71.58	0.18
compress92	input.short	5.90	0.07
go	Board Size = 19 Intelligence = 9	1 hour	1.97

number of specialization tuples. However, we attempted to pick the functions we thought would benefit most for specialization. Thus the results in some sense reflect the best-case outcome of a full specialization pass if our assumptions about specialization benefits are correct.

Table 1 describes the datasets for which each benchmark was profiled. The second column describes the profile dataset. The program was run with this input and a value profile was obtained. This profile was used by the specializer to optimize the program.

For the hash, sort and exp benchmarks, the random numbers were provided in input files to allow repeatability. Separate input files were used for profiling and execution timing. Input files were completely read into a buffer at program start to avoid I/O overhead during execution of the computation kernels. The hash benchmark sent each input to the `hash` routine twice, to ensure at least one hit.

Inlining into specialized functions was disabled for all benchmarks except `fib` and `fib_memoized`

Table 2: Benchmark Results

Benchmark	Input Set	Time Original (sec.)	Time Specialized (sec.)	Speedup	Specialized Invocations
fib	n = 40	47.25	30.07	1.57	50%
hash	scatter = 1024 norm = 17 table size = 1048576 1,000,000 random keys	21.52	9.01	2.39	100%
fib_memoized	scatter = 1024 norm = 17 table size = 1048576 n = 50,000	0.18	0.10	1.80	100%
exp	e = 13 5,000,000 random numbers	25.37	25.45	1.00	100%
sort	1,000,000 12-byte random items	90.84	82.03	1.11	100%
compress92	MIRV C front-end and filters	122.47	125.47	0.98	36%
go	Board Size = 50 Intelligence = 21	337.27	347.49	0.97	28%

to allow a fair comparison to the original, non-inlined versions. Any inlined code would not have benefited from constant values.

Profiling runs were executed on a 200 MHz Pentium Pro processor with 256 MB of memory. For the small benchmarks, hash and sort are particularly hard-hit by the value profiling. The sort swap routine is called over 40,000 times, resulting in a significant slowdown. In Section 8 we discuss some ways to reduce profile time.

Table 2 presents information about the success of the specializer. All programs were executed on the same Pentium Pro machine. Execution time was measured with the UNIX `time` command. Times reported are the best user time obtained over five runs. The final column in the table gives the percentage of specialized function invocations with respect to

the total number of function invocations of functions that were specialized. These numbers were obtained by enabling a flag in the specializer indicating that it should insert code in the dispatch routine to track the number of times the dispatcher was called and the number of times it dispatched to a specialized function⁶. These numbers give some idea of the accuracy of the value profile.

When specializing, we specified value count thresholds high enough to prevent clearly undesirable specialization. This is a current limitation of our system. In the near future we will provide better heuristics to automatically limit unwanted specialization.

It should be noted that the input sets to the ‘C benchmarks are radically different than the inputs given in [PHEK97]. With the inputs as given in the ‘C report, our benchmarks ran far too quickly to provide any meaningful results. We attribute this to hardware differences as well as the lack of runtime code generation with profile-guided specialization. Also, benchmarks such as hash and sort provide greater speedup as the number of items processed increases. This is because the savings gained through specialization is accumulated as more processing occurs. Thus meaningful speedup comparisons with current dynamic compilers are not possible at this time. As the MIRV compiler matures and access to dynamic compilation systems becomes available⁷, meaningful comparison studies will be performed.

As expected, **fib**, **hash**, **fib_memoized** and **sort** show some benefit from specialization. This is expected, as these benchmarks were designed to highlight the advantages of specialization through runtime compilation. What is significant is that benefits can be obtained through profiling and static compilation, avoiding the complexity of a dynamic implementation.

The **exp** benchmark shows slightly reduced performance. The only computation eliminated by specialization was the loop overhead, which was not significant enough to overcome the dispatch overhead. Even so, the degradation is so small as to be insignificant.

⁶The times reported in column 4 do not include execution of this profiling code.

⁷Tempo has recently been released to the public but time did not permit a comparison study of that system.

Polyvariant specialization is critical for the **fib** benchmark. There are many arguments to **fib** that appear over and over again. This is precisely why memoization is so successful. The key is that there are many of these values, not just one. Polyvariant specialization is necessary to exploit this behavior.

Likewise, during examination of the profile databases for various benchmarks, we observed that there were several different tuples that appeared often in the value profile. By setting the reuse threshold slightly lower the compiler could specialize on these different tuples, making use of polyvariant division. However, the contrived nature of these benchmarks means that one tuple tended to dominate the others and led to the most gain from specialization. Specializing on anything more would have increased dispatch time to provide minimal (if any) gain.

Unfortunately, the gains observed by the small kernels do not translate over to the larger benchmarks. Both **go** and **compress92** suffer longer execution times with specialization than without. There are several reasons for this. The **compress92** benchmark reads its input from a file and writes its output to a file. While the benchmark is most likely IO-limited, this is not a factor when measuring user time. However, it is a significant factor for specialization, because the main compression loop cannot be specialized. Because the bytes read are not passed through a function call, the compiler cannot take advantage of any frequently occurring values. The limited granularity of specialization results in many lost opportunities.

Surprisingly, however, there is a significant opportunity for specialization in the **compress92 output** routine. Our profiler indicated that 16 tuples appeared as arguments 1,000 times or more. The tuples were composed of two global values: the size of the data to output and an offset used to determine which parts of a buffer to output. Some control flow was eliminated and many bit shift and mask operations became constant. However, this benefit did not outweigh the dispatch overhead caused by specializing on many tuples, since

specialized functions were invoked only 36% of the time⁸. Moreover, additional profiling of the benchmarks revealed that only 9% of the execution time was spend in the `output` routine.

The `go` benchmark provided much less opportunity for specialization. We had hoped that `go`'s extensive use of complex array accesses could be specialized to constant indices, eliminating addressing calculations. Indeed, the routines specialized did in fact show that indices were quite stable. However, these indices were always the last index in the array calculation, meaning that only one scale and add operation could potentially be saved. Exacerbating this was our use of the x86 as a target platform. The x86 architecture includes many addressing modes capable of accessing complex aggregates in a single operation. Furthermore, there is little or no additional penalty for using these modes. Thus specializing an array index calculation is only beneficial if the array has many dimensions and the index requires scaling code to be emitted. Architectures that only provide base+offset addressing can benefit from index specialization, because it always saves a scaling operation.

We had hoped that elimination of control flow would help significantly. Unfortunately, specialization of `go` did not remove any control flow. This was due to the restricted profiling of pointer arguments. Most of the control flow in the functions examined is determined by array contents, which were not profiled.

In addition, the profiling and specialization we were able to do with `go` did not cover the values used by the program. The specialized functions were only executed 28% of the time. Even running the specialized version on the profiling input only resulted in 23% utilization of the specialized code. This agrees somewhat with the results in [CFE97]: only nine procedures in `go` had at least one parameter with a value that occurred 50% of the time or more. Only one had a parameter with a value re-use of 70% or more⁹. It seems that `go` does not exhibit much parameter value re-use. The fact that executing the profile input

⁸The `compress92` dispatch routine was not inlined due to complications with the code.

⁹Note that the input set used in [CFE97] was different than our profile input.

resulted in *less* utilization of specialized code indicates that go's value re-use is dependent on the board size and intelligence level of play.

Most critically, go suffers the same problem as compress92: the functions specialized are not where the execution time is spent. Only 8% of the execution time is spent in the 5 functions that were specialized. In fact, go's execution time is spread fairly evenly across many functions, making this a difficult benchmark to specialize on a function-level granularity.

Even with all of these problems in the larger benchmarks, the performance degradation is quite small. The dispatch overhead is very small because only one or two parameters need be compared to decide which clone should be called.

Given that we present our scheme as an alternative to dynamic compilation, it is important to explore the differences and disadvantages of our scheme as compared to current dynamic compilers.

1. Current dynamic compilers rely on programmer annotations to guide specialization.

Thus there is no need to extensively profile and perform analyses to decide what is a runtime constant. However, we do not believe this is a major difficulty, given a more efficient profiler implementation.

2. We do not currently specialize on the contents of aggregates or values pointed to by pointer arguments. This is a big loss in most large programs, because aggregate structures are used to organize data, and these structures are usually passed by reference.
3. Programmer annotations can also be used to specify where to specialize, giving the developer great flexibility to target fine-grained program points such as loops. We are currently limited to the granularity of functions.
4. A dynamic compiler can use accurate information about the current program run. It thus has more opportunities for specialization.

A dynamic compiler adapts automatically as the pattern of program usage changes. By employing cumulative profiling over long usage periods, the same effect can be observed. By periodically re-specializing the code, it can be adapted to perform well under the new operating conditions. Essentially, a continuous, cumulative profiler coupled with background optimization can achieve the same effect as a dynamic compiler, albeit at a somewhat coarser granularity.

7 Conclusion

In this paper we have presented some preliminary results of using value profile information to drive function cloning and specialization. Some of the microbenchmarks realized speedups of 1.1–2.3, but these did not translate to the go and compress92 benchmarks. Even with the microbenchmarks, we found the benefits of specialization to be highly program-dependent.

Even so, we believe value profiling and static (re)compilation is a viable alternative to dynamic compilation. Clearly, the value profiler must make use of execution time profiles. The results for go and compress92 demonstrate the need to target specialization to those portions of the program that are responsible for the majority of the execution time. The current profiler can be easily modified to track call frequencies. Profiling the time spent in each routine will require more work.

As with any compiler transformation, there are some programs that benefit from specialization and others that do not. As currently implemented, our scheme does not benefit the compress92, go and exp benchmarks. The small size and lack of procedures is the main problem encountered with compress92. The go benchmark simply does not allow enough specialization in key areas of the program. The loop overhead saved in exp was not enough to offset the dispatch overhead introduced by specialization.

Other benchmarks performed much better, particularly the hash benchmark. Hashing and sorting are common operations, so we were quite pleased with the results for hash and sort. In the near future we will explore whether these gains can be observed in realistic

programs.

8 Future Work

Because the current profiler is quite slow, it is necessary to increase its efficiency. The majority of the profile time is due to the power set computation. One option to overcome this problem is to profile only fully specified tuples and compute the power sets during the specialization phase. This is certainly a viable option, but it sacrifices profile accuracy. Two fully specified tuples often share subsets. For example, tuple (2, 3) and tuple (2, 4) share the subset (2, -). Thus the tuple (2, -) is used more often than either of the fully specified tuples. We would like the value profile database to reflect that, both to provide an accurate count to any specialization heuristic and to prevent the (2, -) subset from being lost due to replacement of the fully specified tuples.

There are various methods that could be employed to prune the subset generation of parameter and global tuples. By not generating all subsets and losing some degree of accuracy, we can speed up the profiling a great deal. The current dynamic compilers avoid this problem altogether through the use of user annotations to guide the specializer.

In addition, profile information about where the time is being spent in the code should be used to target the value profiler. Only these regions of code that execute frequently should be profiled and specialized. In fact, one study of profile-based optimization indicates that optimizing as little as 20% of the static code of large programs can result in enormous performance gains [AdJPS98].

In the immediate future (after fixing various compiler problems), we plan to add cumulative profiling support and increase the speed of the profiler as described above. Concurrently, we will add the capability to specialize on pointer targets and aggregates, making use of Important Variable analysis to direct the profiler. Following this, a study comparing dynamic compilation to cumulative profiling and static re-optimization will be in order.

A comparison of value prediction, instruction reuse and specialization may uncover

interesting characteristics of target benchmarks. By specializing a program and running it on a simulator, one could discover the source of the value predictability and instruction reusability encountered in earlier studies.

Finally, we would like to specialize code that has been designed in a generic fashion. Library code presents a wealth of such code. Currently, the compiler is not able to generate code for the C standard library, but we plan to specialize such library code to evaluate the effectiveness of our implementation.

9 Acknowledgments

Much work has gone into the development of the MIRV compiler and it is impossible to attribute every contribution to the correct author. Kris Flautner was responsible for much of the early MIRV intermediate language design, as well as the design of the dataflow infrastructure. Dave Helder and Charles Lefurgy wrote many of the optimization filters, including the dataflow analysis and transformation infrastructures. Pedro Marron wrote the original x86 code generator, while Matt Postiff, Dave Helder and Charles Lefurgy contributed greatly to the debugging and optimization effort on the code generator.

Many thanks go to the Edison Design Group, Inc. for their generous donation of the C++ front end. This has saved us many months of design and debugging of a C parser.

This work was supported by DARPA contract DABT63-97-C-0047. The development and test facility was provided through an Intel Technology for Education 2000 grant.

References

- [AdJPS98] Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable cross-module optimization. *ACM SIGPLAN Notices*, 33(5):301–312, May 1998.
- [AH96] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In Pierre Cointe, editor, *ECOOP'96—Object-Oriented Programming, 10th European Conference*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166, Linz, Austria, 8–12 July 1996. Springer.
- [And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

- [ASG97] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 134–145, New York, June15–18 1997. ACM Press.
- [CD93] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1992*, pages 493–501, New York, NY, USA, 1993. ACM Press.
- [CFE97] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 259–269, Los Alamitos, December1–3 1997. IEEE Computer Society.
- [CHK93] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, April 1993.
- [Con98] C. Consel. *Tempo Specializer Documentation*. IRISA/INRIA Compose Group: <http://www.irisa.fr/compose/tempo/doc>, 1998.
- [DCG95] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. *ACM SIGPLAN Notices*, 30(6):93–102, June 1995.
- [EHK96] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’96)*, pages 131–144, St. Petersburg, Florida, January 21–24, 1996. ACM Press.
- [GDFC95] David Grove, Jeffrey Dean, Charles Farrett, and Craig Chambers. Profile-guided receiver class prediction. In *ACM SIGPLAN ’95 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 108–123, Austin, TX, 1995.
- [GM97] Freddy Gabbay and Avi Mendelson. Can program profiling support value prediction? In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 270–280, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.
- [GMP⁺97a] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 163–178, New York, June12–13 1997. ACM Press.
- [GMP⁺97b] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. Technical Report TR-97-03-03, University of Washington, Department of Computer Science and Engineering, March 1997.
- [HH97] R. J. Hookway and M. A. Herdeg. DIGITAL FX!32: Combining emulation and binary translation. *Digital Technical Journal of Digital Equipment Corporation*, 9(1):3–12, 1997.

- [HU94] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with runtime type feedback. *ACM SIGPLAN Notices*, 29(6):326–336, June 1994.
- [JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).
- [LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.
- [PEK97] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. *ACM SIGPLAN Notices*, 32(5):109–121, May 1997.
- [PHEK97] Massimiliano Poletto, Wilson Hsieh, Dawson Engler, and M. Frans Kaashoek. ‘c and tcc: A language and compiler for dynamic code generation. 1997.
- [SS97] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 194–205, New York, June2–4 1997. ACM Press.
- [YUW98] Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Improving performance by branch reordering. *ACM SIGPLAN Notices*, 33(5):130–141, May 1998.