Jason W. Kim and Terrance E. Boult
EECS Dept. Lehigh University
Room 304 Packard Lab. 19 Memorial Dr. W.
Bethlehem, PA. 18015 USA
{jwk2|tboult}@eecs.lehigh.edu

Abstract

We introduce a novel optimization engine for META4, a new object oriented language currently under development. It uses Static Single Assignment (henceforth SSA) form coupled with certain reasonable, albeit very uncommon language features not usually found in existing systems. This reduces the code footprint and increased the optimizer's "reuse" factor. This engine performs the following optimizations; *Dead Code Elimination* (DCE), *Common Subexpression Elimination* (CSE) and *Constant Propagation* (CP) at *both runtime and compile time* with linear complexity time requirement. CP is essentially free, whether the values are really source-code constants or specific values generated at runtime. CP runs along side with the other optimization passes, thus allowing the efficient runtime *specialization* of the code during any point of the program's lifetime.

1. Introduction

A recurring theme in this work is that powerful expensive analysis and optimization facilities are not *necessary* for generating *good code*. Rather, by using information ignored by previous work, we have built a facility that produces good code with simple linear time algorithms. This report will focus on the *optimization* parts of the system. More detailed reports on META4**?**; **?** as well as the compiler are under development.

Section 0.2 will introduce the scope of the optimization algorithms presented in this work. Section 1. will dicuss some of the important definitions and concepts related to the META4 programming language and the optimizer used by the algorithms presented herein. The special features of META4 allow for linear (or near-linear) complexity optimization, along with an essentially free run-time constant propagation that is implemented by a simple, compact engine. The code size for the entire optimization engine (without the header and support routines) is less than 250 lines of semi-dense C^{++} code.

Section 2. will discuss in more detail the background information on the optimization algorithms, which are presented in section 3. (and discussed with more detail in **?**). Sections 4. and 5. end with discussion of related work, lessons learned and future directions.

1.1 Optimization

Optimizations that do not *directly* modify control flow, such as Dead Code Elimination (DCE), Constant Propagation (CP), Common Subexpression Elimination (CSE), and even *Runtime Constant Propagation* (RCP) can be implemented in a two pass algorithm — each optimization[1] has a prepass algorithm that generates the needed data structures. They share a single *postpass algorithm* which actually carries out all actions.

Let N be the size of the program[2] and let $h(p)$ be the cost of the hash function for keys of some constant length p. Then, as we will see in 3.6, the prepass is at most $O(h(p)N)$ and the postpass is $O(N)$. If $h(p)$ is a constant, this reduces the complexity to $O(N)$, i.e. linear time.[3]

META4 implements Partial Evaluation (henceforth PE)**?**; **?**; **?** as CP at execution time. The programmer is able to generate and invoke specialized versions of *any* code segment with specific run time values treated as constants and propagated through (and re-optimized) at *any time* during the lifetime of the program. META4 supports (R)CP as well as unrestricted Run Time Code Generation.

This work will also show that CP, even at runtime does not explicitly require a separate prepass but can run as a part of the other optimizations featured here. The programmer only needs to specify that a particular *instance* is a *known* value and instruct the *optimizer* to generate a specialized form of the routine(s) that use this value. To the best of the author's knowledge, this is the first work that *combines* these optimizations along with runtime CP that is executed by a *single, unified engine* in linear (or near linear) time.

2. META4: Language Design Compiler Basics

Before the actual optimization engine can be discussed in detail, it is first necessary to briefly mention the some of the important attributes of the META4**?** language which *actively* contribute to the optimization process.

The first definition addresses the issue of *storage*. A unit of storage in META4 is known as an instance; and is identified by a pair of numbers. The first number identifies a unique instance which correspond to a textual variable name in the source code, and the second specifies the Static Single Assignment**?** index for that instance. The SSA index guarantees that an instance has one unique point of creation. Storage or data members of classes do exist in META4, but they are not directly accessible. Rather we introduce a single concept, property that generalizes and expands on data members and member functions.

A property is a behavioral trait of a class that is typed, named, inheritable, specializable, generalizable. It may require arguments, and may also return values as well. A crude analog would be a virtual member function of a class in C^{++} or Java.[4] Note that within META4, the *only way* an instance can be accessed from a class is through a property**?**. Furthermore the property is the only mechanism for communicating with an object in META4. Operations that are generally considered as primitives (such as add, subtract etc..) are also properties (of the integer class). The unification of storage and function invocation allows META4 language to separate storage from inheritance. Its definition also specifies the types and data-flow directions of all parameters. In addition a property may be declared as a *predictable* property which will be described later.

An invocation is an abstract wrapper for the binding of a *property* with its actual arguments. There are two main types of invocation, a concrete statement which binds arguments to the property being invoked and a metastatement which is a subtype of invocation used for encapsulating control flow change. A nonempty list of properties serves as the "continuation." For example, a "loop" metastatement contains two properties, a "test" continuation and a "body" continuation. The latter is conditionally (re)executed pending the result from the "test". The two subclasses of invocation, metastatement and statement, are combined to form structured control-flow**?** equivalent of the program.

All code segments, a.k.a. implementations, are sequences of invocations. An implementation defines the body of a property. It is an array of invocations that also contains information on the instances used within that implementation. If a property can accept as arguments multiple *types* for the same parameter, then it has more than one implementation. As long as the actual type for that parameter is known at compile time[5], a new implementation can be generated that better meets the argument specifications. This can potentially aid in producing better (*i.e.*, faster executing) code by avoiding dynamic dispatch for invocations as well as through RCP. This also directly supports *polyvariant specialization* and *polyvariant division* **?** as a property can have an implementation that are generated for specific argument *values* as well as specific *types*.

2.1 Synergistic Cooperation Between Programmer, Compiler and the Language

In META4, certain keywords are provided, so that the programmer can use them to inform the compiler of information that would be very useful and difficult to obtain automatically. These are analogous to certain annotations used in various PE engines for the language C**?**; **?**; **?**. Similar ideas are used here to help the optimizer in improving the code. However, in this work, “annotations” are not added onto an existing language, but are an integral part of the programming process that has been designed from ground up to support principled use of Run Time Code Generation (RTCG). This entices the programmer to work with the language and the compiler in producing good code.

The programmer specifies the dataflow to and from a property. Thus the compiler knows exactly which parameters import data to the property (i.e. are *incoming* parameters), which parameters return data (i.e. are *outgoing* parameters), and which parameters do both (i.e. are *in/out* parameters). Also, many “primitive” functions, such as integer operations have predictable results (*i.e.*, given the same input, they always produce the same output, and have no side-effects.). This predictability can be exploited in numerous optimizations. Many user defined properties are also predictable in the same manner as “primitives,” but automatically detecting this is generally difficult. However, in META4, due to the explicit data flow, SSA and careful definition of primitives, guaranteeing the correctness of predictability is easy. It is impossible for a programmer to mark a nonpredictable property as predictable. Regardless, when the programmer correctly marks a property as predictable, both the programmer and the compiler benefit from this explicit presence of semantic information. Therefore, we encourage the programmer to specify these as *predictable properties*.

These keywords, along with the nature of properties and *predictable properties* serve to shortcut some of the analysis that a compiler would need to carry out. They play a critical role in optimization as well as during RTCG and PE.

3. Optimization in META4

Previously reported optimization algorithms **?**; **?**; **?** operate on intermediate forms of “primitive” operations, and sometimes group optimization algorithms together to form more expensive[6] and more powerful combined optimization **?**; **?** that allows them to find and optimize complex structures not amenable to repeated passes of separate optimizations.

The linear complexity algorithms presented here are arguably weaker than the more expensive combined approaches in **?**; **?** but make up for it by allowing for the direct manipulation of user defined *properties*, not just “primitives.” For purposes of the optimization algorithms presented here, there are no differences between "primitive" *properties* and user defined *properties*. The only difference is that "primitives" have direct machine code form, while user defined properties are usually composed of these "primitives".

These aid compilation in three ways. First, common subexpression elimination can now eliminate common calls to complex user defined properties. Second, the compiler knows exactly which parameters to a property will change, and which will not, that is data-flow is made explicit to both programmer and compiler, and is strongly enforced. Third, SSA adds additional data-flow information to the compiler, i.e. a tighter knowledge of what changes where.

Rather than using a powerful and expensive analysis engine which tries to guess which part is known, specializable, or not known, the knowledge of the programmer is exploited, and by supplying enough tools and fast, simple yet effective analysis routines, the programmer can specialize (manually or otherwise) during any phase of its life cycle (even at runtime of the program itself) any algorithm that can be specialized. In addition, the compiler itself is available to the programmer as a class and can be specialized, increasing the efficiency for appropriate tasks.

During the optimization pass (whenever that may be), invocations of *predictable* properties that have all incoming parameters KNOWN can be marked DEAD and once the invocation is activated, all outgoing parameters can be marked KNOWN. This operation can occur at any time during the lifetime of the program. Since values generated at runtime can be marked as *known* and be propagated, this is a valid form of runtime specialization of the program.

In reiteration, the specialization options available are runtime constant propagation and programmer controlled inlining/unrolling of statements/loops[7].

4. Algorithms and Implementation

The overall design of the optimization engine in the META4 compiler is extremely simple yet powerful in its capabilities. Each application of the optimization engine is composed of one or two passes. DCE and CSE require a prepass to generate the information required in the second pass.

There is a main driver routine called *traverse*$(impl, op)$ which traverses through the intermediate representation $impl$.[8] The argument *op* is a routine which processes an invocation. *traverse*$()$ invokes *op* on each invocation in $impl$. The second pass is merely the application of *traverse*$()$ to the implementation $impl$ with a special *op* called *postpass_apply*$()$. The following sections 3.2, and 3.3 explain the prepass algorithms. The postpass algorithm is explained in section 3.5.

4.1 Helper Routines

In the SSA world, an instance which is created by assignment has a predecessor, which is the right hand argument. e.g. assuming $x_i = y_j$, then the *predecessor*(x_i) is y_j. SSA enforces that x_i contains the exact same value as y_j.

The routine *isdead*$(i, impl)$ returns TRUE if the argument i is DEAD within $impl$. A variable v is DEAD if it already has a predecessor and its only use is to construct other variables. If a variable is only used as a copy for another, and it has a predecessor, then the variable need not be created, and the predecessor can be used in its place.

4.2 Dead Code Elimination (DCE)

In this SSA world, *dead code* is any invocation which modifies non-external variables. These variables also must not be used as incoming parameters to other invocations. *Dead conditional branch elimination* is handled as part of the postpass. As *traverse*() processes each invocation, the special postpass operator *postpass_apply*() handles any cleanup duties required for metastatements, such as killing off any dead branches and aligning the SSA indices of variables. The actual DCE prepass algorithm is called *prepdce*() and is used as an operator argument to *traverse*(). *prepdce*() actually marks statements LIVE, if they use or modify an argument (e.g. external parameter) to the property that $impl$ implements. All statements not marked LIVE will be considered for removal during the postpass.

4.3 Common Subexpression Elimination (CSE)

Assume some predictable property fp and within some implementation $impl$, the set of chronologically ordered list of invocations of fp in $impl$ called S. There exists a subset of S called S_{equiv} with one or more invocations s_k such that for all *incoming* parameters p_i in fp and for all s_a, s_b in S_{equiv}, $nth-arg-of(s_a, i)$ *equals* $nth-arg-of(s_b, i)$. Trivially, each invocation in S can be its own S_{equiv}, but the goal here is to partition S into its largest equivalent subsets, given the current state of information available to the compiler. All invocations in S_{equiv} are considered to be *equivalent*. Thus all but the chronologically first invocation s_1, in S_{equiv} can be marked DEAD and the future uses of the outgoing parameters for the rest of the statements in S_{equiv} can be replaced by the respective outgoing arguments of s_1.

When *traverse*(*generate_S*) below is applied to an implementation $impl$, it generates the set S for each predictable property invoked in $impl$.

```
generate_S(statement $s$, $impl$)
  $prop$ = invoked_property($s$)
  if prop is a predictable, then  add $s$ to the end of $S$ for $prop$.
```

Prepcse(), the second prepass for CSE partitions each S generated above into a list of S_{equiv}.

```
prepcse(implementation $impl$):
  for_each entry $S$ in $impl.S\_list$
    if (partition($S$, $list\_of\_S\_equiv$))
      for_each entry $f$ in $list\_of\_S\_equiv$ DO
        statement $base$ is first invocation in $f$
        for_each remaining statement $s_j$ in $f$ { mark $s_j$ DEAD
          for_each outgoing parameter $arg_i$ of $s_j$
            predecessor($arg_i$) = args($base$, $i$) }
```

In *prepcse*(), list_of_S_equiv is a hash table containing a list of invocations that partitions a particular S. The hash key used to access list_of_S_equiv is an array of instances[9] comprised of the incoming parameters of the invocation.

The following routine actually partitions S into non intersecting sets of invocations S_{equiv}, and returns TRUE if at least one S_{equiv} has more than one invocation in it. Of course, *h*() is the hash function for an instance list.

```
partition(invocation_idx_list $S$, $list\_of\_S\_equiv$)
```

```
  integer $c$ = 0;  for_each invocation $pi$ in $S$ {
    $hkey$ = concatenation of all incoming params of $pi$.
    if $hkey$ is already in $list\_of\_S\_equiv$ then c = c + 1;
    insert $pi$ into the end of $list\_of\_S\_equiv[h(hkey)]$ }
  return $S$.size() > c;
```

4.4 Breaking the Single Assignment Chain (BSAC)

Prep_bsac() finds all chains of non-useful (*i.e.*, non-overlapping) generation of variables and marks them DEAD and sets the predecessor for the killed variables. (Runtime) Constant propagation will work, even after the SSA chain has been broken, as any SSA chains that had conflicting usage frontiers are NOT broken. The only other requirement is that the meta-level information for all related entities be available when creating a new implementation of a property.

```
prep_bsac(statement $s$, $impl$)
  if $s$ is not DEAD or LIVE; then $prop$ = invoked_property($s$)
    if is_assignment($prop$) and last use of right hand arg of $s$ is $s$ then
      mark $s$ DEAD; mark the left hand argument of $s$ DEAD;
      set the left hand argument's predecessor as the right
        hand side of $s$
```

4.5 The Post Pass Algorithm and the driver

The *postpass_apply*() routine uses the information generated in the prepass and actually carries out the optimization/specialization as needed. The *activate*() call within *postpass_apply*() invokes the property bound to the current statement and updates the StorageTag[10] records of all outgoing parameters. These new *known values* can then be used as arguments to other properties.

```
postpass_apply(statement $s$, $impl$)
  if $s$ is not DEAD or LIVE then prop = invoked_property($s$);
    for_each argument instance $arg$
     if isdead($arg$) then replace arg with predecessor($arg$)
      if all incoming args are KNOWN and is_predictable($prop$)
        activate($s$)* and mark $s$ DEAD; mark all outgoing params KNOWN;
```

This next routine is the main driving engine: The argument *op* is any of the routines above that have the correct interface. The job of the *op.pre_process*() and *op.post_process*() is to handle any setup and cleanup required for processing metastatements. The only operator that has a defined pre_process() and post_process() is *postpass_apply*(). Specifically, it kills any dead branches. Deciding to kill a branch is as simple as considering the number of live invocations in the implementation of that branch. If it is zero, then the branch can be killed. At runtime, for loops and conditionals, if the test variable is KNOWN, than the appropriate branch can be killed.

```
traverse($impl$, $op$) : for each invocation $s$ in $impl$
  if is_metastatement($s$) then op.pre_process($s$);
    for_each continuation $c$ in $s$ traverse($c.impl$, $op$)
    op.post_process($s$)
  else  op($s$, $impl$)
```

4.6 Costs of the Passes

Each operator argument to *traverse*$()$ is executed once per invocation. Assuming C is the cost of the operator and N the size of the program, the cost of the optimization is $O(CN)$. C is either 1 in the case of *generate_S*$()$, or is proportional to the number of *parameters* p to a property, which can be considered a constant. The routine *prepcse*$()$ is executed once per pass, not once per invocation like the other operators. *prepcse*$()$ processes each candidate invocation exactly once, and the total number of candidate invocations is trivially less than N. Therefore its complexity is $O(h(p)N)$ where $h(p)$ is the hash predictable for the constant length p. It is arguable that for an optimistic view of the hashing cost, this reduces the overall complexity to $O(N)$.[11]

5. Related Work

In **?**, Click and Cooper report that that an optimization framework simultaneously combining constant propagation, global value numbering (a form of CSE) and dead code elimination is possible. The cost is quadratic in the size of the program, but this cost is mitigated by the fact that the combined framework can detect and optimize away some program structures undetected in separate, iterated applications of its parts. There is an explicit tradeoff with the optimization engine presented here. While being linear in complexity, the CSE engine presented in section 3.3 is arguably weaker than the *global value numbering* based expression matchers presented in **?** and reworked in **?**; **?** since it can not directly detect equivalent compound structures of "primitives." (*e.g.*, two loops that calculate the same value). However, this lack is mitigated by the fact that additional programmer-controlled information is available as *language features* to the compiler. Specifically, it is possible to group certain invocation sets as a user defined "predictable property" which will allow equivalent invocations of the property to be weeded out by the CSE engine; thus encouraging programmers to work *with* the compiler in generating good code.

Along with a sacrifice in power of the CSE engine, another feature here is that constant propagation is essentially free. That is, there needn't be a separate pass of the engine for propagating them. All that needs to be done is a certain meta-level instance (that correspond to a real instance) be marked KNOWN at some point during the lifetime of the program, and the next time the postpass is run, the constant propagation proceeds side by side, similar in spirit to what the more complex combined optimization **?**; **?**; **?**; **?** frameworks do. The only requirement here is that along with the actual machine code of the routine being specialized, the meta-level information (specifically the implementation for the property and all other related structures) must be available to the compiler at specialization time. Of course, the *activate*$()$ call would not be made unless ALL parameters are KNOWN at that time.

The need for guessing when to carry out inter-procedural analysis **?**; **?** is neatly avoided here as well. The only time inter-procedural analysis is performed is at the beck and call of the programmer, that is when the programmer hints that an invocation is *inlined* at its call site or when there are metastatements within an implementation (metastatements are always "inlined"). This is in keeping with the philosophy espoused here, which is that the programmer can provide necessary hints so that the compiler can do its job efficiently and productively, and these hints are provided as necessary language features explicit in the program, not from an outside analysis facility. However, this does not preclude the future addition of an analysis engine should the need arise.

6. Conclusion

As software increases in size and complexity, it is vitally important that it be organized, designed and produced in a reusable, updatable manner. This inevitably leads to abstractions that negatively impact performance. Yet by allowing the programmer to *remove* such abstraction layers by *specialization* when needed, it is possible to recapture some of the inherent performance lost to the abstraction layers.

The SELF programming language **?** and compiler feature automated runtime profiling, which is currently missing from the algorithms presented here and would likely be a useful addition. Also, the current lack of certain control-flow related optimizations such as strength reduction and invariant code motion (especially the latter) will have to be addressed in the future.

In principle, the metastatement abstraction is powerful enough to encapsulate language level parallel constructs. The author hopes to extend META4 to handle parallel and distributed computation in the future.

While available literature is filled with separate research efforts regarding specialization **?**; **?**, optimization **?**; **?**, runtime code generation **?**; **?**; **?**; **?** and object-oriented programming, there are precious few that explore the intersection of these research areas. Despite the lack of a formal partial evaluation framework, the author hypothesizes that the combination of the approaches presented in this work (*e.g*, the language features, the RTCG/PE facilities and the efficient optimization engine) is powerful enough for all *principled* uses of specialization and RTCG in an object-oriented development system. Showing evidence to such is left for future research.

Notes

1. except for (R)CP, which do not require a prepass.

2. SSA does add additional bulk to the code size. However the bloat is a constant factor because the added code is a simple rewrite of assignments and control-flow joins. In practice, the expansion is a small constant.

3. Unfortunately, hash functions are very difficult to analyze *exactly*, but experience indicates that a good hash function *can* be considered a constant for most kinds of inputs of fixed size.

4. The renaming is necessary, as these “properties” in META4 are not strictly equivalent to member functions in other languages such as Java or C^{++}**?**.

5. Whenever that may be.

6. *i.e.*, quadratic or $Nlog(N)$**?** complexity, as opposed to linear

7. The inlining/unrolling algorithms are omitted for space reasons

8. As defined in 1., the “intermediate form” worked on by the optimizer is the implementation, which is an array of statements and metastatements.

9. Since a pair of numbers denote an instance, an instance_list is also a list of numbers.

10. When an instance i is assigned a StorageTag s, this implies that i now has a *value* associated with it. This value does *not necessarily* imply a *physical address.* The name is somewhat misleading but its use is retained for historical reasons.

11. For those that are interested, a more detailed outline of the linearity of these algorithms are presented in **?**.

References

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Technques and Tools*. Addison-Wesley, 1986.

Lloyd Allison. *A pratical introduction to denotational semantics.* Cambridge University Press, 1986.

Bowen Alpern, Mark N. Wegman, and F. Kennth Zadeck. Detecting Equality of Variables in Programs. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

Andew M. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

Marc M. Brandis. *Optimizing Compilers For Structured Programming Languages.* PhD thesis, ETH Zurich, 1995.

Marc M. Brandis and H. Mossenbock. Single-Pass Generation of Static Single Assignment Form for Structured Languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, November 1994.

Cliff Click and Keith D. Cooper. Combining Analysies, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

Charles Consel and Francois Noel. A General Approach for Runtime Specialization and its Application to C. Technical report, University of Rennes / IRISA, Campus Universitaire de Beaulieu 35042 Rennes Cedex, France, 1995.

Ron K. Cytron, Jeanne Ferrante, Barry K. Rosen, and Mark N. Wegman. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. ‘C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation. Technical Memo, 1995.

Brian Grant et al. Annotation Directed Run-Time Specialization in C. In *In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 163–178, Amsterdam, The Netherlands, June 1997.

Brian Grant et al. An Evaluation of Staged Run-Time Optimzations in DyC. In *Preedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 293–304. ACM, 1999.

Robert Harper. Introduction to Standard ML. Technical report, School of Computer Science, Carnegie Mellon University, 1993.

Urs Hörzle. *Adapative Optimization for SELF: Reconciling High Performance with Exploratory Programming.* Ph.D. dissertation, Stanford University, August 1994.

Neil D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):481–503, September 1996.

David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimisations. Technical Report UW-CSE-93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

Jason W. Kim. Efficient Run-Time Compilation Using Static Single Assignment: Appendices. Technical Report LU-CS-99-12-01A, Department of Electrical Engineering and Computer Science. Lehigh University, 1999. Online at `http://www.eecs.lehigh.edu/~jwk2/`.

Jason W. Kim. The META4 Programming Language. Technical Report LU-CS-1999-12-02, Department of Electrical Engineering and Computer Science. Lehigh University, 1999. Online at `http://www.eecs.lehigh.edu/~jwk2/`.

Mark Leone and Peter Lee. Lightweight Run-Time Code Generation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.

Renaud Marlet, Charles Consel, and Phillipe Boinot. Efficient Incremental Run-Time Specialization for Free. In *Preedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 281–292. ACM, 1999.

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

Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global Value Numbers and Redundant Computations. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

Eric Ruf. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, Department of Computer Science, 1993.

Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.