Caching Function Calls Using Precise Dependencies

Allan Heydon Roy Levin Yuan Yu

Compaq Computer Corporation
Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301, USA
caheydon@yahoo.com, levin@pa.dec.com, yuanyu@pa.dec.com

Abstract

This paper describes the implementation of a purely functional programming language for building software systems. In this language, external tools like compilers and linkers are invoked by function calls. Because some function calls are extremely expensive, it is obviously important to reuse the results of previous function calls whenever possible. Caching a function call requires the language interpreter to record all values on which the function call depends. For optimal caching, it is important to record precise dependencies that are both *dynamic* and *fine-grained*. The paper sketches how we compute such dependencies, describes the implementation of an efficient function cache, and evaluates our implementation's performance.

1 Introduction

We consider the problem of implementing a pure functional language in which some function calls are expected to be extremely costly. This problem arises in the context of the Vesta software configuration management system, a system for managing and building potentially large-scale software [6, 13]. As an integrated version control and build system, Vesta provides several advantages over traditional version control systems like RCS and CVS, and over the build program Make. These advantages include strong support for parallel development by multiple developers, support for easily specifying build customizations, and guarantees that all builds produce correct results and are reproducible at any time in the future.

In Vesta, the instructions for building software artifacts take the form of programs written in a functional *system modeling language*. Conceptually, these programs describe how to build a system from scratch. When evaluated, such programs may call functions that invoke external tools such as compilers and linkers, the net results of which are typically returned as part of the evaluation result. Invocations of external tools account for the vast majority of the time spent building a software artifact, so it is obviously crucial to reuse results from previous builds whenever possible. However, tool invocations occur only at the leaves of an evaluation's function call graph. In a large build, testing whether it is safe to reuse a previous result at each of these leaves might require thousands or tens of thousands of checks, adversely affecting incremental build performance.

To produce a system whose incremental build performance scales to large software, we claim that it is therefore important to reuse larger units of work than individual tool invocations. The use of a functional language meshes well with this goal because function calls make convenient units of caching for later reuse. By using a cached function result whenever it is safe to do so, unnecessary recompilations and other work can be avoided. But when is it safe to reuse a cached result? Only when the evaluation context at a candidate call site agrees with those parts of the context on which some previous call to the same function depended.

When detecting dependencies, it is of course essential not to omit any, or the cache would be unsound, sometimes returning incorrect results. However, it is also important not to err by introducing overly broad dependencies, or the cache will be ineffective, sometimes failing to return a result when it should. For the cache to be most effective, each function call's dependencies must be recorded as precisely as possible. For example, consider the following simple function:

```
f(x, y, z) {
  return (if x > 0 then y else z);
}
```

Because of the conditional expression, the arguments on which this function depends vary *dynamically* from call to call. For example, in the call $f(1, 2, 3)$, the result depends only on the values of x and y; the value of z is irrelevant. Hence, the subsequent invocation $f(1, 2, 7)$ in which the values of x and y are identical should produce a cache hit on the first call.

In this particular example, the observant reader will notice that the exact *value* of x is also unimportant. What matters is simply whether or not x is positive. Hence, to get the most accurate caching, dependencies should take the form of *predicates* on values, not the exact values themselves. For now, we will assume that all dependencies are recorded on values, but we describe how to represent more general dependencies in Section 4 below.

The caching problem is further complicated when the language includes *composite* value types. For example, let $[a = 1, b = 2]$ denote a record with two fields named a and b, whose values are 1 and 2, respectively. For such a record r, let the expression $r.a$ denote the selection of the a field. Now consider a slight modification to our previous example:

```
f(x, y, z) {
  return (if x > 0 then y.a else z);
}
```

In this case, the result of the call $f(1, [a = 2, b = 5], 3)$ depends only on x and $y.a$. The subsequent call $f(1, [a = 2, b = 9], 7)$ should produce a cache hit. Recording a dependency on the entire record y would cause the second call to get a false cache miss. This example demonstrates that the dependencies calculated with respect to composite values should be as *fine-grained* as possible.

From these two examples, we conclude that the Vesta interpreter must compute fine-grain dependencies dynamically (that is, during an evaluation). The challenges in accurately implementing a caching scheme based on dynamic, fine-grain dependencies are two-fold. First, algorithms must be developed for representing, computing, and propagating dynamic, fine-grained dependencies. We sketch the technique used by the Vesta interpreter in Section 4. Second, due to the dynamic nature of the dependencies, it is impossible to compute a single cache key at a function call site before the function has been evaluated. To handle this problem, we divide the dependencies into two groups: the *primary dependencies*, on which the function is known to depend before the call is performed, and the *secondary dependencies*, on which the function dynamically depends. In Section 3, we describe how we organize the function cache so as to perform lookups efficiently.

As mentioned above, the Vesta interpreter caches both "higher level" user-defined functions and those that directly invoke external tools. By caching user-defined functions, the interpreter can get cache hits on larger units of work than individual tool invocations; for example, on the construction of an entire library archive. These higher-level cache hits make the cost of an incremental build proportional to the amount of work to be performed, not to the size of the system being built. As a result, Vesta's incremental build performance scales well to large software systems, a trait that is not shared by other software construction tools such as Make [4]. We demonstrate the effectiveness of Vesta's higher-level caching in Section 5.

2 System Modeling Language

Before describing our caching and dependency calculation algorithms, we first sketch the main features of Vesta's system modeling language.

Vesta's build language is a full-fledged programming language that is functional (that is, side-effect free), modular, dynamically typed, and lexically scoped. Its value space contains booleans, integers, text strings, lists, closures, and bindings. The first four data types are the familiar ones from Algol-like languages and LISP. Closures are first-class, higher-order functions that can be nested. Bindings are described below. The language contains about 60 built-in functions for arithmetic and boolean operations, for basic manipulations of texts, lists, and bindings, and for invoking external tools. Overall, these core language facilities were designed to be as basic and "methodology neutral" as possible so that support for particular styles of system construction or organization could be programmed in the language, rather than being built-in to the language or interpreter. The complete language syntax and semantics are described elsewhere [5].

For the purposes of this paper, two features of the modeling language are noteworthy: the *binding* type and the *runtool* primitive.

Bindings are like records, except that the list of fields in a binding can be dynamically extended at runtime. Conceptually, bindings are functions mapping texts to values. Bindings are constructed using a square bracket notation and may be nested, as in $[f = 1, g = [a = 2, b = 3]]$. The field f of a binding b is selected by writing b/f; it is a checked runtime error to select a nonexistent field. The boolean expression $b!f$ is true if and only if the field f is defined in b.

Bindings are used extensively in Vesta system models. They are convenient for representing both build customizations (for example, $[debug =$ "-g", $opt =$ "-O2"]) and nested file directories, in which the "leaves" of the binding associate file names with file contents. (In fact, it was this latter usage that inspired the choice of "/" for the binding selection operator.) In our system models, both build customizations and file directories are stored together in one composite binding called the *environment*. The use of an environment binding allows build customizations affecting the entire build or selected parts of the build to be specified once at a high level. For the environment to have this effect, the environment binding is passed as a parameter to every function call. Almost all functions depend on selected parts of the environment, so recording fine-grain dependencies on this binding is crucial.

The language includes several primitives for combining bindings. One of these is the binary overlay (+) operator, defined by:

$$(b_1 + b_2)/n = \text{if } b_2!n \text{ then } b_2/n \text{ else } b_1/n$$

This operator merges two bindings, giving precedence to the second binding wherever both define the same name. It can be used to override a default set of build options with user-supplied customizations. For example, the following statement augments the build options to specify that compilation should produce debugging symbols:

```
options = options + [ debug = "-g" ];
```

There are two classes of function calls in the Vesta modeling language: calls of the built-in *runtool* primitive, which is used to invoke external tools, and calls of user-defined functions.

Calls of the runtool primitive are treated specially. The techniques used to implement it are beyond the scope of this paper, but we sketch them briefly here. The runtool implementation is a cooperative process between the Vesta interpreter and the Vesta repository, which stores all of the versioned sources, tools, and libraries in the system. For maximum extensibility, the Vesta interpreter has no built-in knowledge of any of the tools it invokes. Given a command line and an evaluation environment as arguments, the runtool primitive simply invokes the tool given in the command line on the given arguments. The environment contains a binding that defines the file system in which the tool is to be run. Vesta arranges that the tool (including any child processes it forks) has access to only the files in this file system, and that all its file references are detected by the repository and reported back to the interpreter. The interpreter then records these references as dependencies on the environment.

Although runtool caching is certainly important for incremental build performance, this paper focuses on effectively caching user-defined functions, which often represent larger units of work in the function call graph.

3 Efficient Caching

The interpreter and function cache use *fingerprints* to represent the cache keys. A fingerprint is a fixed-size hash of an arbitrary byte sequence [3, 10]. Fingerprints come with a mathematical guarantee bounding the probability of a collision; by choosing long enough fingerprints, the probability of a collision can be made vanishingly small[1]. As a result, fingerprints can be used as a basis for equality tests, since we can safely assume that $FP(a) = FP(b) \iff a = b$. Two operations supported on fingerprints are extending a fingerprint by more bytes and extending a fingerprint by another fingerprint. In the latter case, we write $fp_1 \oplus fp_2$ to denote the result of extending fp_1 by fp_2. The $\oplus$ operation is non-commutative.

As described previously, the dependencies for each function call are divided into two groups, primary and secondary. The primary dependencies are determined at the call site, before the function is evaluated. The primary dependencies normally include the body of the function being invoked and the values of the function's arguments that are of scalar types.[2] From the primary dependencies, the interpreter computes a *primary cache key* by fingerprinting the body of the function being invoked, then extending it by the fingerprints of the relevant argument values.

The secondary cache key also uses fingerprints, but it takes a different form, namely, a set of name-value pairs in which no name occurs more than once. The secondary key represents the names on which the function dynamically depends, as well as the values of those names in the evaluation context. Since the only operation required on the values when performing a cache lookup is testing for equality, the values can be represented in the cache by fingerprints, thereby saving both space and time. The fingerprints of file values are supplied by the Vesta repository[3]; the fingerprints of all other values are computed by the interpreter.

To make the concepts of primary and secondary keys more concrete, consider the call of a user-defined *compile* function for compiling a single C source file:

```
compile("test.c", env);
```

Here, *env* is a binding that contains a representation of a filesystem in which the compilation is performed. The primary key for this call is the fingerprint of the *compile* function's body combined with the fingerprint of the literal `test.c`. The secondary key will contain the names of all parts of the *env* binding referenced during the function evaluation, together with the fingerprints of the corresponding values. For example, the names in the secondary key might include:

```
env/usr/lib/cmplrs/cc
env/test.c
env/usr/include/stdio.h
env/options/debug
```

[1] For safety, Vesta uses 128-bit fingerprints, making the probability of a collision much less than 1 in 2^{80}.

[2] Including argument values among the primary dependencies is a heuristic; strictly speaking, until the function is evaluated, we do not know whether it will depend on these arguments. The interpreter accepts pragmas to modify this heuristic; see Section 4.6.2.

[3] Whenever a file changes, the repository computes a new fingerprint for it. The fingerprint of a small file is computed from the file's contents; the fingerprint of a large file is computed by fingerprinting the file's unique identifier. The threshold for distinguishing small files from large files is a configuration parameter that defaults to 1 megabyte.

Once the primary key has been computed, the question that immediately arises is how to perform a cache lookup. Due to the dynamic nature of the secondary dependencies, there is no way to know *a priori* what the dependencies will be without evaluating the function. But obviously, if we had to evaluate a function before we could look up its value in the cache, the cache would be useless.

The way around this chicken-and-egg problem is for the function cache to group entries by primary key. The interpreter and cache then cooperate to search through the appropriate group for an entry whose secondary dependencies match the current context. Using this idea, the lookup operation becomes a four-step process:

1. The interpreter computes a primary key *pk* and communicates it to the function cache.

2. The function cache examines all of the cache entries with primary key *pk*, and returns to the interpreter the *union* of all names in their associated secondary keys.

3. The interpreter computes (the fingerprints of) the values associated with those names in the current evaluation context, and sends them to the function cache.

4. The cache again examines the entries with primary key *pk*, and checks to see if any have secondary keys matching the values in the current context. If so, a cache hit occurs, the cached function result is returned, and the interpreter skips the function evaluation.

If the evaluation of a function is deterministic, and if the dependencies recorded for all evaluations of that function are complete, then the cached result returned in the event of a cache hit will be the same as the result that would be produced by actually evaluating the function. Of course, it is quite possible to invoke external tools that produce nondeterministic results (such as the Unix *date* program). In practice, however, all of the tools we have used (mainly compilers and linkers) are sufficiently deterministic to produce reliable caching behavior. For example, the standard C compiler writes a timestamp into each object file it generates, but this nondeterminism does not prevent a previously-written object file from being safely reused at a later time.

We now consider the question of how best to organize entries in the cache so as to efficiently implement steps 2 and 4 of the lookup algorithm. As mentioned, cache entries sharing the same primary key are grouped together. In practice, however, there might be hundreds of such entries. For example, if the primary key corresponds to compiling a particular source file, there will be a new cache entry created each time a different version of the file is compiled. Moreover, it is not atypical for cache entries themselves to have hundreds of name-value pairs in their secondary keys. Considering the compilation example again, each file that is accessed during the compilation is a secondary dependency. Hence, even if the entries with a given primary key could be easily enumerated, a naive implementation of the lookup algorithm might require tens of thousands of fingerprint comparisons.

To avoid this problem, cache entries are organized in a two-level hierarchy. First, all entries with the same primary key are grouped together. Then the entries in each group are partitioned in such a way that only a subset of the entries in each group need be examined on any lookup.

To explain how this partitioning is done, we introduce some notation. For any cache entry e, let $e.pk$ denote e's

primary key, let *e.names* denote the set of names in e's secondary key, and for any name $n \in e.names$, let $e.val(n)$ denote the fingerprint value associated with n in e's secondary key. Now define the following:

$$\begin{aligned} Entries(pk) &= \{e \mid e.pk = pk\} \\ AllNames(pk) &= \bigcup_{e \in Entries(pk)} e.names \\ CommonNames(pk) &= \bigcap_{e \in Entries(pk)} e.names \end{aligned}$$

$$CFP(e) = \bigoplus_{n \in CommonNames(e.pk)} e.val(n)$$

The set $CommonNames(pk)$ thus consists of those names that occur in every cache entry with the given primary key. The names in $AllNames(pk) \setminus CommonNames(pk)$ occur in some cache entries with the given primary key but not others; we call them *uncommon*. The value $CFP(e)$, e's *common fingerprint*, is the result of combining the fingerprints of all secondary values of e corresponding to $e.pk$'s common names. Due to the non-commutative nature of the $\oplus$ operation, it is important to enumerate the names n in a well-defined order. To this end, the function cache maintains a canonical ordering for each pk of the names in $AllNames(pk)$; it uses that ordering when computing $CFP(e)$.

Given these definitions, we can now describe how the cache implements steps 2 and 4 of the lookup algorithm. For each primary key pk, the cache maintains $Entries(pk)$, $AllNames(pk)$, and $CommonNames(pk)$ (the last of which is represented by a bit vector with respect to $AllNames(pk)$). In step 2 of the lookup algorithm, the cache simply returns $AllNames(pk)$ for the supplied primary key. To efficiently perform step 4 of the lookup algorithm, the cache computes $CFP(e)$ for every entry e. It then groups the entries $Entries(pk)$ into equivalence classes according to their common fingerprints.

As an example, consider Figure 1, which shows the secondary names and values of four cache entries sharing a primary key. Each column is a different cache entry, and the circled letters correspond to different fingerprint values. The absence of a fingerprint in row i and column j means that name i is not in the secondary dependency set of the entry for column j. The entries shown could correspond to invocations of a C compiler on a source file named *test.c*.

In this example, only the names *env/usr/lib/cmplrs/cc* and *env/test.c* are referenced by all four cache entries, so those are the only two names in $CommonNames(pk)$. Figure 2 shows the $CFP(e)$ fingerprints that might be computed for these entries; notice that $CFP(e_3) = CFP(e_4)$ because the values associated with the common names agree on those two entries. The entries are then arranged in a hierarchy as shown in Figure 3; the top two levels of the hierarchy are implemented using hash tables. All of the entries sharing the same primary key and common fingerprint are said to belong to the same *cfp-group*.

In step 4 of the lookup operation, the cache is given a pk and the fingerprints $f_1, f_2, \ldots, f_k$ of the values corresponding to the names $AllNames(pk)$ at the call site. Call these fingerprints f_i the *call site fingerprints*. To perform the lookup, the cache first combines those call site fingerprints associated with $CommonNames(pk)$, thereby producing a common fingerprint *cfp*. Next, the cache does a hash table lookup to see if pk has *cfp* as one of its associated common fingerprints. If not, the cache reports a miss. Otherwise, the cache examines the entries in the identified cfp-group. In testing for a hit, only the uncommon names need be examined, since by virtue of being in the correct cfp-group, all entries being considered are known to have matching values for the common names.

	Entries			
Secondary Names	1	2	3	4
`env/usr/lib/cmplrs/cc`	A	A	A	A
`env/test.c`	B	D	E	E
`env/usr/include/stdio.h`	C	C		
`env/defs.h`			F	G

Figure 1: The secondary names and values of four cache entries sharing the same primary key.

	Entries			
Secondary Names	1	2	3	4
`env/usr/lib/cmplrs/cc`	A	A	A	A
`env/test.c`	B	D	E	E
CFP(e)	X	Y	Z	Z
`env/usr/include/stdio.h`	C	C		
`env/defs.h`			F	G

Figure 2: The common fingerprints computed for the cache entries of Figure 1. Here, $X = A \oplus B$, $Y = A \oplus D$, and $Z = A \oplus E$.

This lookup algorithm results in two major cost savings compared to the brute-force algorithm. First, only those entries in the identified cfp-group need be examined. In practice, the cfp-groups tend to be quite small (often containing only one entry), so this produces a major saving. Second, when examining the entries in a cfp-group, only the values associated with the uncommon names need be examined. In our experience, the fraction of names that are uncommon tends to be quite low, averaging only a few percent. These two effects thus drastically reduce the number of fingerprint comparisons required to perform a cache lookup.

When a new entry is added to the set $Entries(pk)$, the sets $AllNames(pk)$ and $CommonNames(pk)$ can change. Since changes to the latter would require common fingerprints to be recomputed and cache entries to be reorganized, newly added entries are kept in two side buffers (one for entries that have all of the common names, and one for those

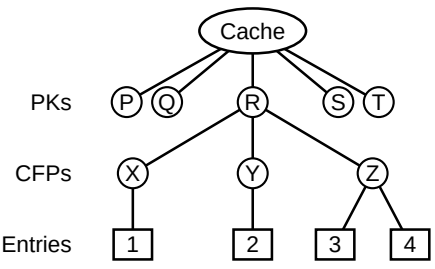

Figure 3: The hierarchical arrangement of the cache entries first by primary key, and then by common fingerprint for the entries of Figure 2.

$e ::=$	
$\mid a$	literal ($a \in \mathit{Literal}$)
$\mid x$	variable ($x \in \mathit{Id}$)
$\mid \lambda x.e$	lambda
$\mid \underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3$	conditional
$\mid [n_1 = e_1, n_2 = e_2, ..., n_k = e_k]$	binding constructor
$\mid e/n$	binding selection
$\mid e!n$	binding domain test
$\mid e_1 + e_2$	binding overlay
$\mid \underline{\text{let}}\ x = e_1\ \underline{\text{in}}\ e_2$	let construct
$\mid e_1(e_2)$	function application

Table 1: The syntax for a subset of our system modeling language.

that do not). The lookup algorithm must consult these side buffers in addition to checking for a hit in $\mathit{Entries}(pk)$. Once a large enough number of new entries are collected, they are merged into $\mathit{Entries}(pk)$ together, thereby amortizing the cost of recomputing all of the common fingerprints. The details of this update process are beyond the scope of this paper.

In Vesta, the function cache is implemented by a persistent, fault-tolerant server process. The Vesta interpreter communicates with the cache over a local network via RPC. This design has two obvious advantages. First, the cache process can be run on a powerful server machine for better performance. Second, since developers at the same site share the same function cache, they can benefit from each other's builds.

4 Computing Dependencies

In this section we consider how to calculate dependencies for the Vesta language. The interpreter computes dependencies dynamically during an evaluation. This section gives the mathematical rules we have developed for computing dependencies. It also states (but does not prove) a correctness theorem.

To describe the key ideas used in our dependency calculation, we define a subset of the Vesta modeling language [5]. Table 1 gives the subset language's syntax. Here, *Literal* is the set of literals, and *Id* is the set of identifiers. This subset has been chosen to include the core parts of the Vesta language that present the most significant challenges to effective dependency analysis. It omits the language's primitive functions (of which there are approximately 60), its iteration construct, its provisions for importing one system model from another, and its support for binding program variables to versioned directories and files in the Vesta repository.

Every expression is evaluated in some *evaluation context*, which is a mapping from variable names to values. Let $\mathit{Eval}(e, c)$ denote the result of evaluating the expression e in the context c, and let $\mathit{Dpnd}(e, c)$ denote the dependency information resulting from evaluating e in c. Since we use standard call-by-value evaluation, the rules for $\mathit{Eval}(e, c)$ are straightforward, so we will not describe them here. The remainder of this section describes the rules for computing $\mathit{Dpnd}(e, c)$.

4.1 Relation to Caching

As outlined in the previous section, when faced with a function invocation $f(e_1, e_2, \ldots, e_n)$, the interpreter first computes a primary key from the function's body and zero or more of the argument values. In response to this primary key, the function cache returns a set of secondary names. The cache treats the secondary names as meaningless strings, but to the interpreter they represent dependencies. The interpreter evaluates each of the secondary names in the current context and sends the resulting list of value fingerprints to the cache. The cache then tests for a hit as previously discussed. In the event of a cache hit, the interpreter uses the cached result value.

In the event of a cache miss, the interpreter proceeds to evaluate the function on the given arguments. As it does so, it represents each runtime value by a pair containing the true value and the dependencies detected while computing that value. Value-dependency pairs are also stored for any sub-values nested inside composite values such as lists or bindings. Once it has finished evaluating the function, the interpreter collects up the dependencies in the function's result value. Any dependencies that are not already part of the primary key are considered part of the secondary key. The interpreter then calls the function cache to create a new cache entry with the computed primary key, secondary key, and result value. It is important to note that the cached result value, like all of the interpreter's runtime values, itself is a pair that includes dependencies. Whenever a later evaluation gets a hit on this entry, both the value and its dependencies will be needed so that the dependency analysis for subsequent uses of the value can proceed correctly.

For caching to be correct, the cache entries created by the interpreter must satisfy a theorem, a formal statement of which is given in Section 4.5 below. To understand the intuition behind the theorem, imagine that the cache contains an entry with primary key pk that resulted from evaluating the expression e in a context c_1. The secondary names associated with this cache entry will be the set of dependencies $\mathit{Dpnd}(e, c_1)$.

When attempting to evaluate e in another context c_2, the cache will produce a hit on the cached entry only if the values computed in c_2 for the dependencies $\mathit{Dpnd}(e, c_1)$ match the values stored in the cache. Note that the values associated with the dependencies in the cache are precisely the values computed in c_1 for $\mathit{Dpnd}(e, c_1)$.

We therefore define *equivalence* between two contexts with respect to a set of dependencies d as follows:

$$\mathit{Equiv}(c_1, c_2, d) = (\forall p \in d : \mathit{Value}(p, c_1) = \mathit{Value}(p, c_2)).$$

In this definition, $\mathit{Value}(p, c)$ denotes the result of "evaluating" the dependency p in the context c. It corresponds to the fingerprint associated with each secondary name in the cache. (We define $\mathit{Value}(p, c)$ for the particular dependencies created by the interpreter in the next section.)

Type	Dependency on the component's ...	Computed Value
V	...complete value	$\mathit{Value}(V : \mathit{path}, c) = \mathit{Eval}(\mathit{path}, c)$
X	...existence	$\mathit{Value}(X : \mathit{path}/\mathit{id}, c) = \mathit{Eval}(\mathit{path}!\mathit{id}, c)$
D	...domain	$\mathit{Value}(D : \mathit{path}, c) = \{n \mid \mathit{Eval}(\mathit{path}!n, c)\}$
T	...type	$\mathit{Value}(T : \mathit{path}, c) = \mathit{Eval}(\mathit{typeof}(\mathit{path}), c)$
L	...length (number of subcomponents)	$\mathit{Value}(L : \mathit{path}, c) = \mathit{Eval}(\mathit{length}(\mathit{path}), c)$
E	...expression (closures only)	$\mathit{Value}(E : \mathit{path}, c) = \mathit{Eval}(\mathit{path}, c).\mathit{body}$

Table 2: The meanings of the six dependency types and the rules used by the interpreter to evaluate each type of path in a context c.

Clearly, the cache hit will be correct if and only if performing the evaluation of e in c_2 would produce the same result as the one stored in the cache, that is, $\mathit{Eval}(e, c_2) = \mathit{Eval}(e, c_1)$. Therefore, to prove our caching correct, we must show that

$$\mathit{Equiv}(c_1, c_2, d) \Longrightarrow \mathit{Eval}(e, c_2) = \mathit{Eval}(e, c_1).$$

The next section describes how the interpreter computes the dependencies so as to satisfy this requirement.

4.2 Representing Dependencies

As mentioned in Section 1, dependencies in general are predicates on the evaluation context. In a practical implementation, however, allowing for arbitrary predicates would be costly in both space and time. We therefore use a small, fixed collection of predicates, encoded as *dependency paths*. It is these dependency paths that are passed to the function cache as the names in a cache entry's secondary key. The function $\mathit{Dpnd}(e, c)$ evaluates to a set of such paths; Section 4.3 below describes the rules for computing it.

The syntax of a dependency path is given by the following grammar:

$$\begin{array}{rcl} \mathit{dpath} & ::= & t : \mathit{path} \\ t & ::= & V \mid X \mid D \mid T \mid L \mid E \\ \mathit{path} & ::= & \epsilon \mid \mathit{id}/\mathit{path} \end{array}$$

A dependency path takes the form $t : \mathit{path}$, where t denotes the dependency type, and *path* specifies the component of the evaluation context on which the evaluation depends. We use ϵ to denote an empty path; a path of the form id/ϵ is equivalent to the path *id*.

The meanings of each of the interpreter's six dependency types are given in Table 2, along with the rules it uses to compute $\mathit{Value}(t : \mathit{path}, c)$, the value of the dependency path $t : \mathit{path}$ in the context c. The V (value) dependency type is the strongest, and hence subsumes the other dependency types. In this table, the language's primitive *typeof* and *length* functions compute the dynamic type of a value and the length of a list or binding, respectively. The notation $\mathit{cl}.\mathit{body}$ denotes the closure cl's body component.

4.3 Dependency Calculation Rules

We now give the mathematical rules for calculating dependencies. We first define $D(e, c, p)$ where p is a dependency path; again, $D(e, c, p)$ evaluates to a set of dependency paths. We then define $\mathit{Dpnd}(e, c) = D(e, c, V : \epsilon)$. Intuitively, $D(e, c, p)$ is the dependency for just the portion of e's value that is selected by p. The definition of $D(e, c, p)$ now proceeds by cases on the program structure:

- If $e = a$ ($a \in \mathit{Literal}$), then $D(e, c, t : p) = \emptyset$. Evaluating a constant has no dependency on the context.
- If $e = x$ ($x \in \mathit{Id}$), then $D(e, c, t : p) = \{t : x/p\}$. Evaluating a variable x depends only on the dependency path extended on the left by x.
- If $e = \lambda x.e_1$, only the following two cases arise:

  $$\begin{array}{l} D(e, c, V : \epsilon) = \mathit{FVs}(e) \\ D(e, c, E : \epsilon) = \{\} \end{array}$$

  where $\mathit{FVs}(e)$ denotes the set of e's free variables. In both cases, the path must be empty. If the type of the path is V, the lambda expression depends on the whole closure value, that is, the set of e's free variables. If the type of the path is E, it depends on only the lambda expression of the closure value, namely, the expression e. Since e is incorporated into the primary key of every function call whose body contains e, it is correct not to record any dependencies in this case.
- If $e = \underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3$, then we have the following rule:

  $$\frac{\begin{array}{c} d_1 = D(e_1, c, V : \epsilon) \\ v_1 = \mathit{Eval}(e_1, c) \\ d_2 = \text{ if } v_1 \text{ then } D(e_2, c, t : p) \text{ else } D(e_3, c, t : p) \end{array}}{D(e, c, t : p) = d_1 \cup d_2}$$

  This rule states that the dependency of a conditional is the union of the dependencies of the guard e_1 and the dependency of either e_2 or e_3, depending on the value of e_1. We use an empty path in computing the guard's dependencies because the guard evaluates to a boolean value that has no components.
- If $e = [n_1 = e_1, n_2 = e_2, ..., n_k = e_k]$, there are two cases to consider. If p is empty, it means that we depend on the entire binding. If $p = t : n_i/p_1$, it means that we depend only on the value of field n_i. The following rules cover the two cases:

  $$\begin{array}{l} D(e, c, t : \epsilon) = \cup_{i=1}^{k} D(e_i, c, V : \epsilon) \\ D(e, c, t : n_i/p_1) = D(e_i, c, t : p_1) \end{array}$$

- If $e = e_1/n$, then $D(e, c, t : p) = D(e_1, c, t : n/p)$. For binding selection, we recursively call D with the path extended on the left by n.
- If $e = e_1!n$, then $D(e, c, t : p) = D(e_1, c, X : n/p)$. For the binding domain test, we recursively call D with the path extended on the left by n. Note that the new dependency path has type X, regardless of the type t.
- If $e = e_1 + e_2$, then there are two cases to consider: If p is empty, it means that we depend on the entire binding. If $p = t : n/p_1$, it means that we depend

only on the binding that supplies the n field. In the case that the n field comes from e_1, we must add the dependency that n is not defined in e_2. The following rules cover the two cases:

$$
\begin{aligned}
&D(e, c, t{:}\epsilon) = \\
&\quad D(e_1, c, V{:}\epsilon) \cup D(e_2, c, V{:}\epsilon) \\
&D(e, c, t{:}n/p_1) = \\
&\quad \text{if } \mathit{Eval}(e_2!n, c) \\
&\quad \text{then } D(e_2, c, t{:}n/p_1) \\
&\quad \text{else } D(e_2, c, X{:}n) \cup D(e_1, c, t{:}n/p_1)
\end{aligned}
$$

- If $e = \underline{\text{let}}\ x = e_1\ \underline{\text{in}}\ e_2$, then

$$
\frac{\begin{array}{c} c_1 = c \circ \{x \rightarrow \mathit{Eval}(e_1, c)\} \\ d_2 = D(e_2, c_1, t{:}p) \\ d_{2a} = \{p' \mid p' \in d_2 \wedge \mathit{head}(p') \neq x\} \\ d_{2b} = \{t'{:}p' \mid t'{:}x/p' \in d_2\} \end{array}}{D(e, c, t{:}p) = d_{2a} \bigcup \cup_{p' \in d_{2b}} D(e_1, c, p')}
$$

where $\circ$ denotes the operation for extending a context, and $\mathit{head}(p)$ denotes the first element of p's path. We first augment the evaluation context with x mapped to $\mathit{Eval}(e_1, c)$ and compute d_2 as the dependency of e_2 in the augmented context. We then divide the dependency paths in d_2 into two sets d_{2a} and d_{2b}. The set d_{2a} contains paths unrelated to x. So, d_{2a} must be included in the result. The set d_{2b} contains paths starting with x. So, we need to recursively compute $D(e_1, c, p')$ for each path p' in d_{2b}.

- If $e = e_1(e_2)$, then

$$
\frac{\begin{array}{c} \mathit{Eval}(e_1, c) = < \lambda x.e_3, c_3 > \\ d_1 = D(e_1, c, E : \epsilon) \\ c_1 = c_3 \circ \{x \rightarrow \mathit{Eval}(e_2, c)\} \\ d_3 = D(e_3, c_1, t{:}p) \\ d_{3a} = \{p' \mid p' \in d_3 \wedge \mathit{head}(p') \neq x\} \\ d_{3b} = \{t'{:}p' \mid t'{:}x/p' \in d_3\} \end{array}}{\begin{array}{l} D(e, c, t{:}p) = d_1 \bigcup \\ \quad (\cup_{p' \in d_{3a}} D(e_1, c, p')) \bigcup \\ \quad (\cup_{p' \in d_{3b}} D(e_2, c, p')) \end{array}}
$$

This rule is similar to the rule for the let construct, where e_1 in the let expression is like the argument e_2 here, and e_2 in the let expression is like the closure body e_3 here.

4.4 Example

We now present a simple example to demonstrate the above dependency rules. We compute the dependencies for the expression:

$$e = \underline{\text{let}}\ x = [r{=}[s{=}y], t{=}z]\ \underline{\text{in}}\ x/r/s$$

in the context c. Obviously, $\mathit{Eval}(e, c) = c(y)$. Here is the start of the dependency calculation:

$$
\begin{aligned}
& \mathit{Dpnd}(e, c) \\
\equiv\ & \{ \text{ definition of } \mathit{Dpnd} \ \} \\
& D(e, c, V{:}\epsilon)
\end{aligned}
$$

Since the expression e is a let construct, the let rule applies. The main step in calculating the dependencies for the let construct involves calculating the dependency set named d_2 in that rule, where $e_1 = [r{=}[s{=}y], t{=}z]$ and $e_2 = x/r/s$. Here we derive the value for d_2, using c_1 to denote the augmented context $c \circ \{x \rightarrow \mathit{Eval}(e_1, c)\}$:

$$
\begin{aligned}
& D(x/r/s, c_1, V{:}\epsilon) \\
\equiv\ & \{ \text{ binding selection rule } \} \\
& D(x/r, c_1, V{:}s) \\
\equiv\ & \{ \text{ binding selection rule } \} \\
& D(x, c_1, V{:}r/s) \\
\equiv\ & \{ \text{ variable rule } \} \\
& \{\ V{:}x/r/s\ \}
\end{aligned}
$$

From the dependency set d_2, we compute the partitioned sets d_{2a} and d_{2b}:

$$
\begin{aligned}
& d_{2a} = \emptyset \\
& d_{2b} = \{\ V{:}r/s\ \}
\end{aligned}
$$

We can now continue computing $\mathit{Dpnd}(e, c)$:

$$
\begin{aligned}
& D(e, c, V{:}\epsilon) \\
\equiv\ & \{ \text{ let rule } \} \\
& \emptyset \cup D([r{=}[s{=}y], t{=}z], c, V{:}r/s) \\
\equiv\ & \{ \text{ binding constructor rule } \} \\
& D([s{=}y], c, V{:}s) \\
\equiv\ & \{ \text{ binding constructor rule } \} \\
& D(y, c, V{:}\epsilon) \\
\equiv\ & \{ \text{ variable rule } \} \\
& \{\ V{:}y\ \}
\end{aligned}
$$

Hence, the evaluation of e in c depends only on the value of y, as we would expect.

4.5 Correctness

The following theorem states the correctness of the dependency calculation rules.

Theorem 1 (Caching Correctness) *If the expression e evaluates to a value v in the context c_1, then we can compute $\mathit{Dpnd}(e, c_1)$, and, if every path in $\mathit{Dpnd}(e, c_1)$ evaluates to the same value in contexts c_1 and c_2, then e also evaluates to v in c_2. Formally,*

$$
\begin{aligned}
&(\exists v : \mathit{Eval}(e, c_1) = v) \Longrightarrow \\
&\quad (\exists d : \mathit{Dpnd}(e, c_1) = d) \\
&\quad \wedge\ (\mathit{Equiv}(c_1, c_2, \mathit{Dpnd}(e, c_1)) \Longrightarrow \\
&\qquad \mathit{Eval}(e, c_2) = \mathit{Eval}(e, c_1))).
\end{aligned}
$$

Before implementing the dependency algorithm, we formalized the above evaluation and dependency rules for this subset of the Vesta language in the Nqthm theorem prover [2], and mechanically checked the correctness theorem. The proof is beyond the scope of this paper. It took several iterations of running the prover and correcting our rules before the mechanical proof succeeded. This proof effort revealed a couple of subtle errors in earlier versions of the rules. We then implemented the rules in the Vesta interpreter. Although we did not mechanically check the rules for the complete Vesta language, the subset we did verify covers the most complex aspects of the language, and so the mechanical verification was quite useful.

4.6 Practical Considerations

A number of practical issues arise in the implementation of the dependency calculation. Below, we briefly describe three of the more interesting ones.

4.6.1 Runtool Caching

As a practical system modeling language, the Vesta language provides a number of primitive functions. Except for the runtool primitive, all of the other primitives can be handled using mathematical rules similar the ones developed in Section 4.3. The runtool primitive must be treated specially because any dynamic dependencies that result from file system references must be recorded during tool invocations.

Currently, we support two different kinds of file references: *lookup* (looking up a name in a binding representing a filesystem directory) and *list* (listing the entries in such a filesystem binding). When a lookup succeeds, we record a value (V) dependency on the file or directory it returns. When a lookup fails, we record an existence (X) dependency asserting the nonexistence of such an object. Finally, for a list reference on a directory, we record a domain (D) dependency on the *names* in that directory.

4.6.2 Primary Vs. Secondary Key

As mentioned earlier, we use the fingerprint of a closure's body as the basis for its primary key. Including none of the arguments in the primary key would produce many cache entries with the same primary key. That would cause cache lookups to take longer, since there would be more entries to search through for any given primary key—one for each time the same function was invoked. Conversely, folding all of the arguments into the primary key, as we have explained, would produce cache entries that are too coarse-grained.

We have chosen a course between these two extremes. We use a heuristic that folds the values of the simple arguments (booleans, integers, and texts) into the primary key, but does fine-grained dependency analysis on the composite arguments (bindings, lists, and closures).

The Vesta language includes pragmas for overriding this heuristic. None of the Vesta models written by end users have required these pragmas. Their use has been limited to the specialized "bridges" written by wizard users. These bridges encapsulate the invocation of compilers and linkers, providing a less primitive interface than the one offered by runtool. The typical bridge model contains less than 10 such pragmas. Although only a small number of pragmas are required, their effect on performance can be noticeable, especially as the number of cache entries increases. Importantly, pragmas impact performance only; their use cannot produce incorrect caching behavior such as false cache hits.

4.6.3 Coarse-Grained Entries

When one function calls another, many dependencies of the callee typically become dependencies of the caller. For example, the function responsible for compiling a library archive will depend on the *union* of all of the header files on which the compilations of the constituent source files depend. Hence, if no special measures were taken, the root function of an evaluation would have an extremely large number of dependencies. This situation would make cache lookup time proportional to the size of the system being built, and therefore would prevent evaluation from scaling well.

To prevent too many dependencies from being propagated up to the root of an evaluation, we automatically identify some functions as special. In Vesta, these are typically the functions responsible for building large components of a system, such as entire library archives. In addition to the normal fine-grained cache entries created for evaluations of the special functions, the interpreter also creates cache entries for these functions whose dependencies are more coarse-grained. These special functions thus serve as cutoff points in the function call graph, above which overly fine-grained and bulky dependencies do not propagate.

Test	Source Lines	Source Files	Runtool Calls
Hello	10	1	2
Interpreter	53,304	103	117
Release	119,602	255	333

Table 3: Properties of three source collections.

5 Performance

In this section, we demonstrate that Vesta's dependency and caching algorithms perform well, and that it is important to cache user-defined function calls in addition to runtool calls.

Table 3 summarizes three collections of source code used for our measurements. The *Hello* test contains a single 10-line "hello world" program. The *Interpreter* collection consists of the code for the Vesta interpreter. The *Release* collection contains the sources for a complete Vesta release.

We performed incremental builds of these three collections using Vesta and Make. In both cases, the builder was run on a Digital AlphaStation 500 5/333, with a 333 MHz Alpha 21064 CPU and 192MB of memory. The server processes (the function cache and repository for the Vesta build, and the NFS server for the Make build) were run on a Digital AlphaStation 400 4/233, with a 233 MHz Alpha 21064 CPU and 192MB of memory. These machines are now two generations old, so we expect the absolute performance in both the Vesta and Make cases would be substantially better on modern hardware.

Figure 4 is a graph comparing the performance of Vesta and Make on incremental builds of the three source collections. In each case, we modified a single source file, and then measured the elapsed time to rebuild the system. Since only a single source file was modified in each test, both Vesta and Make performed exactly the same tool executions; the differences shown in their performance are due entirely to the time required by each system to decide what sources to recompile. Vesta is much faster than Make on the Interpreter and Release tests because Vesta gets a high-level cache hit on the call for building the (unmodified) libraries, whereas Make has to *stat* hundreds of source, header, and object files to ascertain that the libraries are up-to-date.

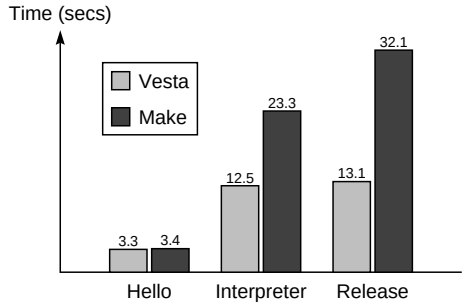

Figure 4: Incremental build performance of Vesta and Make on the three source collections of Table 3.

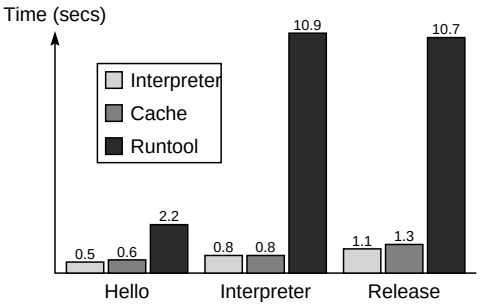

Figure 5: Elapsed time spent by various Vesta components performing the incremental builds of Figure 4.

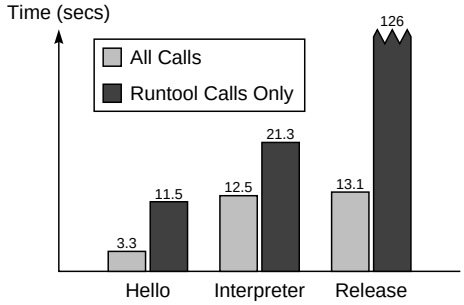

Figure 6: Elapsed time spent performing Vesta incremental builds when all calls are cached vs. when only runtool calls are cached.

Figure 5 shows the overhead of performing the dependency calculations and cache lookups in each of the incremental Vesta builds of Figure 4. The time spent in the interpreter computing dependencies and in the cache performing lookups is minimal. On average, the two phases of the cache lookup operation together take approximately 70 ms, and adding a new cache entry takes approximately 22 ms. The vast majority of the time is spent performing runtool calls, that is, invoking the compiler and linker.

Since the tool invocations under Make take the same amount of elapsed time as under Vesta, we conclude from Figures 4 and 5 that in the Release test, Make spends fully two-thirds of its time deciding which file to recompile. By contrast, Vesta spends less than 20% of its time outside the tool executions. We expect that as the software being built gets larger, the differences will become even more extreme.

Finally, Figure 6 shows why it is so important to cache calls of user-defined functions. The figure compares the Vesta incremental build times of Figure 4 in which all function calls are cached to incremental builds in which only runtool calls are cached. The latter numbers were produced by running the interpreter in a mode in which it did fine-grain dependency analysis on runtool calls, but absolutely no other dependency analysis. When only runtool calls are cached, the incremental build performance suffers greatly, especially on the larger Release test. The main reason the runtool-only case is so much slower is that many more cache lookups are required in that case. When all calls are cached, Vesta's incremental build performance is proportional to the scope of the change rather than the size of the system being built.

Although we have not implemented a version of the interpreter that computes either coarse-grain or static dependencies, our experience using the system suggests that so many false cache misses on user-defined functions would occur that the system's performance would sink to the levels shown for the runtool-only caching case in Figure 6. (In fact, the performance would be even worse if coarse-grained dependencies were recorded for runtool calls because then even some tool invocations would be unnecessarily repeated.) This effect is due to the wide-spread use of the *environment* binding in our models. Recording a coarse-grained dependency on the environment would cause false cache misses throughout the entire call graph whenever any part of the environment was changed. This reasoning led us to conclude that recording dynamic, fine-grain dependencies was critically important if we wanted Vesta to exhibit good incremental build performance.

6 Related Work

Memoization is a well-known technique for caching the results of function calls [7, 8]. Memoization is most effective when applied to recursive programs, particularly those that use dynamic programming. For example, adding memoization to the straightforward recursive computation of the nth Fibonacci number changes the program's time complexity from exponential to linear. However, in standard memoization, all of a function's arguments are incorporated into the cache key, and there are no dynamic dependencies. This technique would produce cache entries that are far too coarse-grained for our needs, resulting in costly and unnecessary cache misses. Pugh and Teitelbaum describe how to better structure recursive computations so as to improve the effectiveness of memoization when the arguments vary from call to call in certain ways [9].

Program slicing is a technique for determining the parts of a program that may contribute to some values of interest [12, 14]. Computing the slice of a program affecting the value of a variable v at a particular statement S involves determining all statements and variables on which v's value at S depends. The algorithms in the literature do not admit composite values, so the dependencies computed by traditional program slicing techniques are coarse-grained.

The work most closely related to ours is a paper by Abadi, Lampson, and Lévy, which uses a labelled λ-calculus to compute both dynamic and fine-grained dependencies [1]. Their approach is quite different from ours. It associates labels with all expressions in a function body, and then develops rules for keeping the labels of only those expressions that are evaluated during a call. As a result, it records only one kind of dependency, analogous to our value dependencies. Also, their calculus supports only the selection operation on records. Computing dependencies for the binding operators ! and + complicates the problem significantly.

7 Conclusions

When interpreting a functional language in which some function calls may be extremely expensive, it is imperative for good performance to perform only those function calls that are absolutely necessary. One obvious approach is to cache the results of each function call, and to perform a cache lookup before executing any candidate call. In the event of a cache hit, the cached result can be used directly.

As we have shown, accurately caching function calls requires the interpreter to record precise dependencies that are both dynamic and fine-grained. Otherwise, cache entries are created that can produce false cache misses, causing unnecessary work to be performed. We have described

a technique for recording and propagating such precise dependencies, and we have described a cache organization that supports efficient lookup in the presence of such dependencies.

The techniques described in this paper are an integral part of the Vesta software configuration management system. Vesta is now in daily use by a group of over 100 Compaq engineers designing the next-generation Alpha processor [11]. They use Vesta to build simulators of the chip from its RTL models. Despite the significant differences between such hardware design tasks and standard software development, Vesta has worked quite well for them. In fact, they estimate that their use of Vesta instead of CVS (for version control) and Make (for building) have put them 3 to 6 months ahead of schedule on the first phase of their project. Analysis of the function call graphs produced from their incremental builds shows that our precise dependency analysis and caching is as effective for them as in our own use and performance studies.

Acknowledgments

We wish to thank Martín Abadi, Jim Horning, Butler Lampson, and Tim Mann for helpful discussions related to this work, Tim Mann and Marc Najork for their comments on earlier drafts of the paper, and the anonymous PLDI referees for their many helpful suggestions.

References

[1] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 83-91. Association for Computing Machinery, May 1996.

[2] Robert S. Boyer and J Strother Moore. A Computational Logic Handbook. Academic Press. 1988.

[3] Andrei Broder. Some applications of Rabin's fingerprinting method. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[4] S. I. Feldman. Make — A program for maintaining computer programs. *Software — Practice and Experience*, 9(4):255–265, April 1979.

[5] Allan Heydon, Jim Horning, Roy Levin, Timothy Mann, Yuan Yu. The Vesta-2 Software Description Language. Compaq Systems Research Center Technical Note 1997-005c, June, 1998.

[6] Allan Heydon, Roy Levin, Timothy Mann, Yuan Yu. The Vesta Approach to Software Configuration Management. Compaq Systems Research Center Technical Note 1999-001, June 22, 1999.

[7] John Hughes. Lazy Memo-Functions. *Lecture Notes in Computer Science*, 201:129–146. Springer-Verlag, Berlin, Sept. 1985.

[8] D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218:19–22, April, 1968.

[9] William Pugh and Tim Teitelbaum. Incremental Computation via Function Caching. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages* (POPL '89), pages 315–328, January, 1989.

[10] M. O. Rabin. Fingerprinting by random polynomials. Report TR–15–81, Department of Computer Science, Harvard University, 1981.

[11] Matt Reilly. Designing an Alpha Microprocessor. *IEEE Computer*, 32(1):27–34, Jan. 1999.

[12] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[13] Vesta Home Page. `http://www.research.compaq.com/-SRC/vesta/`.

[14] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.