Partial Evaluation in an Optimizing Prolog Compiler

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

For the Degree of Ph.D

in the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

by

Srinivasa Bharadwaj Yadavalli

Fall 2000

Permission To Use

In presenting this thesis in partial fulfillment of the requirements for a Post-graduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5A9

Abstract

Specialization of programs and meta-programs written in high-level languages has been an active area of research for some time. Specialization contributes to improvement in program performance. We begin with a hypothesis that partial evaluation provides a framework for several traditional back-end optimizations. The present work proposes a new compiler back-end optimization technique based on specialization of low-level RISC-like machine code. Partial evaluation is used to specialize the low-level code. Berkeley Abstract Machine (BAM) code generated during compilation of Prolog is used as the candidate low-level language to test the hypothesis. A partial evaluator of BAM code was designed and implemented to demonstrate the proposed optimization technique and to study its design issues.

The major contributions of the present work are as follows: It demonstrates a new low-level compiler back-end optimization technique. This technique provides a framework for several conventional optimizations apart from providing opportunity for machine-specific optimizations. It presents a study of various issues and solutions to several problems encountered during design and implementation of a low-level language partial evaluator that is designed to be a back-end phase in a real-world Prolog compiler. We also present an implementation-independent denotational semantics of BAM code – a low-level language. This provides a vehicle for showing the correctness of instruction transformations.

We believe this work to provide the first concrete step towards usage of partial evaluation on low-level code as a compiler back-end optimization technique in real-world compilers.

Acknowledgements

The period I worked on this thesis was very unique in my life – frustrating yet immensely enriching. Being the only researcher in the department working on compiler optimizations for all these years has been a challenge. Several individuals have "schemed" together to provide this experience for me. It was a time when I could passionately learn and experiment with my favorite area of research.

Eric Neufeld gave me the freedom and supported my work unequivocally. I will always be grateful for Eric's understanding and help during these years. Tony Kusalik, with his keen attention to detail, helped me sharpen the accuracy and presentation. This proved invaluable and I believe will only make me a sharper researcher in the future. I wish to express my sincere thanks to Tony. I also wish to express my gratitude for the financial support Eric and Tony have provided during this period.

I deeply value Carl Mccrosky's help and guidance with regards to denotational semantics. Paul Tremblay was a very great resource for compiler related research material and advise.

Peter Van Roy was very patient and prompt in his replies to every question I posed related to Aquarius Prolog compiler and BAM. This helped me sustain the passion in my work.

My dream of earning a doctorate would not have been realized without the sacrifices and the unflinching support of my life partner and the only love of my life – Nirmala. The very joy of being in the presence of our kids Aditya and Anindita gave me the much needed balance and perspective during these years. I can only promise a life-time full of love and affection to Nirmala, Aditya and Anindita. My parents have given the freedom to pursue my interests while ensuring that I know right from wrong. My humblest *namaskaramulu* (salutations) to them.

I thank NSERC, Canada and IRIS for the funding provided for a significant time of my thesis work.

I also thank Digital Equipment Corporation/Compaq for providing me with an opportunity to work in an exciting group during the later years of this thesis work.

Dedication

To the lotus feet of God

Whose benevolence manifests

as affection of my parents –

Sri. Venkata Ramana Rao and Smt. Annapurna,

as unconditional love and support of my wife –

Nirmala

and as pure love of my children –

Aditya and Anindita.

Table Of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Specializing high-level language programs for some subset of program data that is known to be constant across independent invocations of the program has been an active research area for several years now. Such specialization is commonly referred to as *partial evaluation*. Complete program data is available to the program at execution-time. In comparison, any constant data available at compile-time is partial. Hence the term *partial*. The term *evaluation* seems to originate from early usage of this technique on programs written in functional programming languages like Lisp [44, 49]. Subsequently, partial evaluation of programs written in logic programming languages like Prolog [58, 69], imperative languages like C [5] and object-oriented languages like C++ [22] was also studied. Partial evaluation research was done with such goals as reducing the level of abstraction in meta-programs for execution efficiency, generating compilers and generating compiler-compilers via self-applicable partial evaluators [43], to name a few. Various techniques have been employed to achieve these goals. Using minimal user annotations to guide the partial evaluation process [33], automating discovery of the constant portion of a given program and not relying on user annotations [58] are some of the more effective ones. Partial evaluation was also used to generate specialized programs that are more efficient than the original [6, 42].

Jones [40] discusses several interesting open problems regarding issues such as program control, data and correctness that play an important role during partial evaluation. Several of these have been worked on since. However, the question "Can

partial evaluation yield efficient low-level machine code?" which he poses seems unanswered as far as we know. Efficient low-level code generation is a requirement in several system-related tools like compilers, binary translators, emulators. This thesis investigates efficient code generation using partial evaluation in a compiler back-end. Before deciding whether "efficient" code may be generated or not, we need to understand the techniques needed to implement a low-level language partial evaluator. Further, it is well-known that several conventional back-end optimizations are inter-related. The benefits of performing partial evaluation as a back-end with relation to conventional optimizations needs to be understood. Hence, this thesis investigates low-level language partial evaluation techniques and the potential benefits/relationship of partial evaluation with traditional back-end optimizations. The candidate low-level language may either be machine-level language or an intermediate abstract machine-level language. We begin with the hypothesis that by performing partial evaluation several conventional compiler optimizations such as constant propagation, dead-code elimination, and loop-unrolling are automatically performed. This hypothesis is tested by building a framework within which a partial evaluator of a real world abstract machine code is designed and implemented. Several traditional optimizations are shown to result from the partial evaluation. The framework provides a basis to study the issues involved in designing and implementing a low-level language partial evaluator.

In general, partial evaluation is done with the knowledge of two distinct pieces of information, viz., candidate program unit to be specialized and the invariant data for which the program unit is being specialized. Partial evaluation of high-level languages benefits from the inherent higher-level program abstraction and structure. For example, a function is the program unit with well-known structure and behaviour that facilitate partial evaluation. It has zero or more arguments of which some may be input arguments and some output arguments. The function argument and return value variables hold potential program invariants. The programming language model defines the behaviour of a function call. For example, the control flow returns to the calling function after returning from the callee and the callee does not alter the

return address. In the world of low-level code, neither a structure nor a well-defined behaviour of program units may be expected. Hence the first step is to correctly identify program units and invariant code variables in any given low-level code to prepare for partial evaluation. These form the input for a partial evaluator.

Partial evaluation research has largely focused on high-level programming languages. The present work describes partial evaluation of Berkeley Abstract machine (BAM) code generated during compilation of Prolog sources by the Aquarius Prolog compiler. The partial evaluation phase is intended to fit non-intrusively into the existing phases of the Aquarius Prolog compiler. Further, the BAM code partial evaluator is designed not to depend on any user annotations: it is an *automatic* partial evaluator. There is no investigation into use of partial evaluation on low-level languages in the context of compiler optimizations in general to the best of our knowledge. More specifically, this is the first such attempt in the context of Prolog compilation as far as we know. Nonetheless, the techniques described herein are applicable during partial evaluation of any low-level language code.

The present work is in the context of a Prolog compilation model that translates Prolog source to abstract machine instructions that are in turn compiled to native executable code. Prolog is a dynamic-typed language. Hence, the abstract machine code generated during Prolog program compilation contains one code stream for each basic abstract machine data type a Prolog variable can assume at run-time. It also contains run-time type-checks that dispatch execution flow to the appropriate code stream depending on the type of the variable. In other words, the abstract machine code is generic enough to facilitate execution of code corresponding to data-types that would be known at run-time. The motivation for partial evaluation of abstract machine code is to eliminate any generic code and to specialize it for its data-types.

In an effort to improve code performance, several new ideas are being studied and implemented in the research community. Various types of *profile-directed* schemes have been recently shown to hold promise [14, 17]. We view the profile-directed schemes as partial evaluation schemes. A profile is a record of some invariant run-time behaviour of the program that is used to optimize the executable. For example,

a profile might record the number of calls made to call-site in the executable or a library. Thus the profile provides invariants that are used to specialize the executable. The profile collection and subsequent specialization may occur after one or more runs of the program. Alternately, some of the more recent research attempts to perform profile collection and specialization at run-time. This technique is often referred to as *Dynamic optimization* [23,31]. Several of the techniques described here are directly applicable in the context of such efforts to improve code performance.

The thesis is laid out as follows. A brief introduction to Prolog, Berkeley Abstract Machine and the compilation model of the Aquarius Prolog compiler is given in Chapter 2. The denotational semantics of BAM instructions are presented in Chapter 3. This implementation-independent specification facilitates proof of instruction specialization. It also provides a precise definition of instructions for the implementation of the partial evaluator. Program specialization is introduced in Chapter 4. The correctness of all possible instruction transformations is shown and opportunities for program specialization are detailed with the help of examples in this chapter. The design and implementation of the BAM partial evaluator is described in the Chapter 5. Various data structures that extend the BAM to facilitate analysis of run-time information are described along with the analysis algorithms. Chapter 6 describes partial evaluation of each individual BAM instruction and all the issues involved in maintaining the correctness of program state during partial evaluation. In Chapter 7 the results are summarized and the conclusions of the work are discussed. The relevance of the work in the context of current research and future work are discussed in Chapter 9.

Chapter 2

Prolog and Berkeley Abstract Machine

This thesis deals with optimizing Berkeley Abstract Machine (BAM) code using program specialization during Prolog compilation. Section 2.1 presents an overview of Prolog. Section 2.2 details the BAM. An implementation-independent and complete denotational semantics specification of BAM presented in Chapter 3 allows us to show the correctness of the specializations in Chapter 3.

2.1 The Prolog Programming Language

Prolog is a dynamic-typed logic programming language. In other words, the program must be executed to compute the types of Prolog data items [60]. This section briefly presents the language syntax and execution model.

2.1.1 Syntax

2.1.1.1 Data Representation in Prolog

We use the `typewriter` font to represent language tokens (or terminals [2]) while describing Prolog syntax. All Prolog programs in this thesis also appear in this font. The sanserif font is used for meta-language constructs. *Italics* are used when a new term is being defined or described for the first time. A similar convention is followed while describing BAM code. However, the syntactic conventions followed in BAM denotational semantics specification is different. Corresponding syntactic conventions are discussed in more detail at appropriate places.

Prolog has a single data type known as *term*. A term is one of the following:

- a *constant* symbol that stands for an individual entity all through the program. Prolog constants are either *atoms*, *integers* or *floating-point numbers*. An atom is an alpha-numeric string whose first character is lower-case. Any character may be part of such a string if it is enclosed within single quotes "'". Any sequence of characters from the set {+,-,*,/,\,^,<,>,=,`,~,:,.,?,@,#,$,&} is also an atom. `abc`, `var1`, `'Prolog'`, `'80x86 Architecture'`, `1024`, `3.141` are Prolog constants.

- a *variable* symbol that stands for a distinct but as yet unidentified entity. It is represented by an alpha-numeric string whose first character is either upper-case or "_". If a variable is only referred to once in a Prolog construct's scope (Section 2.1.1.2), it does not need to be named and may be written as an *anonymous variable*. A variable whose first character is "_" is an anonymous variable. `A`, `Var`, `_list` are some Prolog variables.

- a *compound term* that stands for a collection of entities. This allows grouping of data elements similar to structures in C and records in Pascal. It consists of a structure name known as a *functor* and constituent entities known as *components*. A functor symbol is an atom and the components (or arguments) are themselves terms. A compound term with no components is an atom. The number of arguments of a compound term is the *arity* of the functor and the compound term is uniquely represented as functor/arity. `capital('India', 'New_Delhi')` is an instance of a compound term with arity 2 and we write the compound term as `capital/2`.

  A Prolog *list* of terms is a special kind of compound term. An empty list is denoted by the atom []. A non-empty list is a compound term with "." as functor and two arguments, viz., the first element of the list, called the *head* of the list and the rest of the list, called the *tail* of the list. Thus a list of the two terms `a` and `b` is `.(a, .(b, [])) ` and is conveniently represented in

short-hand notation as `[a, b]`. A list is also be represented as [*head*|*tail*]. Hence the list `[a, b]` is also represented as `[a|[b]]` or `[a|[b|[]]]`.

2.1.1.2 Prolog Program Constructs

Prolog program constructs are a subset of first-order logic known as *Horn clause logic* [48]. However, the terminology used is various places in this thesis to describe Prolog constructs follows the traditional Prolog terminology [16] rather than than that of predicate calculus [48]. A Prolog program consists of set of *clauses* that represent a consequent of a conjunction or disjunction of a (possibly empty) set of antecedents. The sequence of two or more antecedents separated by commas(",") represent their conjunction. Let `conseq` represent a consequent and `antec` represent a sequence of antecedents. The representation

```
conseq :- antec.
```

is interpreted as "conseq is true if antec are true". The symbol ":-" is read as "if". A clause is terminated by a period(".). For example, the clause

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

can represent "X is the sibling of Y if Z is the parent of X and Z is the parent of Y". A clause is also known as a *rule*. The consequent is known as the *head* of the clause, and the antecedent as the *body* of the clause. The above clause is said to *define* the *predicate* `sibling/2`. A predicate definition may consist of more than one clause indicating several choices to satisfy the relationship. For example,

```
p(X) :- q(X,a).
p(X) :- q(X,b).
```

defines the predicate `p(X)` to be true if either `q(X,a)` or `q(X,b)` is true. A disjunction of two or more clause bodies represents the definition of those clauses using only one clause. The above example may be written as

```
p(X) :- q(X,a);q(X,b).
```

where ";" stands for the disjunction.

A clause with no body is known as a *fact* or a *unit clause*. It represents a relationship of zero or more Prolog terms that is a tautology. For example, the relationship between `knife` and `knives` may be represented by the fact

```
plural(knife, knives).
```

The scope of a Prolog variable is restricted to the clause it occurs in.

2.1.1.3 The Goal Clause

Prolog program execution involves verifying whether a *goal* or a *query* clause is true or false in the program context. In interpreted Prolog it is common to represent a goal that finds the siblings of `john` as `?- sibling(john,X)`. But in compiled Prolog, the goal clause provides the entry point to a compiled Prolog program. This is similar to the (default) entry point `main()` to a C program which otherwise is a listing of several function definitions. Thus the compiler either uses a reserved keyword to identify the goal clause head or adopts some other mechanism to identify the goal clause for a given program. The compiler used for the present work considers the first clause in the program to be the goal clause and it expects the clause to be of arity 0.

2.1.2 Execution model

Given a goal clause, Prolog program execution is based on SLD-resolution [48]. A typical operational semantics of Prolog execution are given by the Algorithm 1. This algorithm does not address the presence of negation, built-ins and similar advanced clause body Prolog constructs nor does it handle the cut operator [63] that prunes Algorithm 1's naïve depth-first clause traversal.

GStack, the goal stack, keeps track of the goals still to be satisfied. The set of goals in **GStack** represents the *resolvant*. Since Prolog execution attempts to satisfy body goals left-to-right in their order of listing in the body, the goals in the list in

Algorithm 1 Prolog Execution Semantics

execute_prolog(G: goal):boolean

Let **GStack** be a stack of the pairs (*goal, index*)
Let **ChStack** be a stack of the tuples (*index,goal,bindlist*)

```
1: push (G,1) onto GStack
2: while GStack ≠ empty do
3:    (G_cur, i) = pop(GStack)
4:    V := List of unbound variables in G_cur
      /* H_1, H_2, ..., H_n are the clause heads with */
      /* same functor and arity as G_cur. */
5:    n := number of clauses with same functor/arity as G_cur.
6:    while ((G_cur does not unify with H_i) ⋀ (i ≤ n)) do
7:       i := i+1
8:    end while
      /* The clause with head H_i has a body B_1, B_2, ..., B_m */
      /* with m varying for different i */
9:    if (i < n) then
10:      push (i + 1, G_cur, V) onto ChStack
11:      push ((B_1,1),(B_2,1), ... ,(B_m,1)) onto GStack
12:   else
13:      if (ChStack == empty) then
14:         return false /* No more choices */
15:      else
16:         (i, G, V) := pop ChStack
17:         restore variable bindings from V to goal G
18:         push (G, i) onto GStack
19:      end if
20:   end if
21: end while
22: return true
```

Step 11 are pushed onto **GStack** such that B_1 is at the top of the stack. **ChStack**, the choice stack, keeps track of the next possible choice to unify a current goal with.

The algorithm assumes variables of the clause whose head is H_i are uniquely named to avoid duplication with the variable names of G_{cur} and the variables already built in the program before performing the pattern-matching operation called *unification* of G_{cur} and H_i at Step 6. Unification equates (unifies) two identical constants, or a constant and a variable, or a variable and a compound term in which the variable does not occur, or two compound terms. For example, the terms `f(A, s(s(0)), c)` and `f(a, s(B), C)` unify to produce the substitution: `A` with `a`, `B` with `s(0)` and `C` with `c`. A formal specification of unification algorithm is given by Lloyd [48]. If the current goal fails to unify with any program clause, execution attempts to re-satisfy the previously successful goal (Steps 16-18). Execution upon successful unification is known as *forward execution* and upon its failure is known as *backtracking*.

The algorithm starts with a Prolog goal term and indicates the success of the goal in the context of a given program. Bindings of the variables in the goal G, if any, resulting from the function execution give the computed answer.

Several built-in arithmetic, input-output, term inspection and control manipulation operators make the language practical. Clocksin and Mellish [16] provide a complete description of Prolog. Sterling and Shapiro [63] and O'Keefe [55] provide advanced material about programming in Prolog.

2.2 Berkeley Abstract Machine

Prolog was initially implemented as an interpreter. David H. D. Warren developed the first Prolog compiler in 1977 and an improved execution model for compiled Prolog, the Warren Abstract Machine(WAM) [3, 28, 70], in 1983.

The Berkeley Abstract Machine(BAM) [67] retains the fundamental features of the WAM but defines a finer-grained instruction set that facilitates compiler optimizations and maps more directly to general purpose processor architectures [68].

Aquarius, an optimizing Prolog compiler to BAM [35,67] was also a part of the BAM project. A global flow analysis (GFA) phase [67] in the Aquarius Prolog compiler derives information used for optimized BAM code generation, exploiting the finer-grained instruction set. An overview of the BAM architecture and its instruction set follows.

We use lowercase alphabet with `typewriter` font for BAM instructions.

2.2.1 Memory Areas and Data Structures

The memory areas of BAM (Figure 2.1) are similar to those of the WAM.

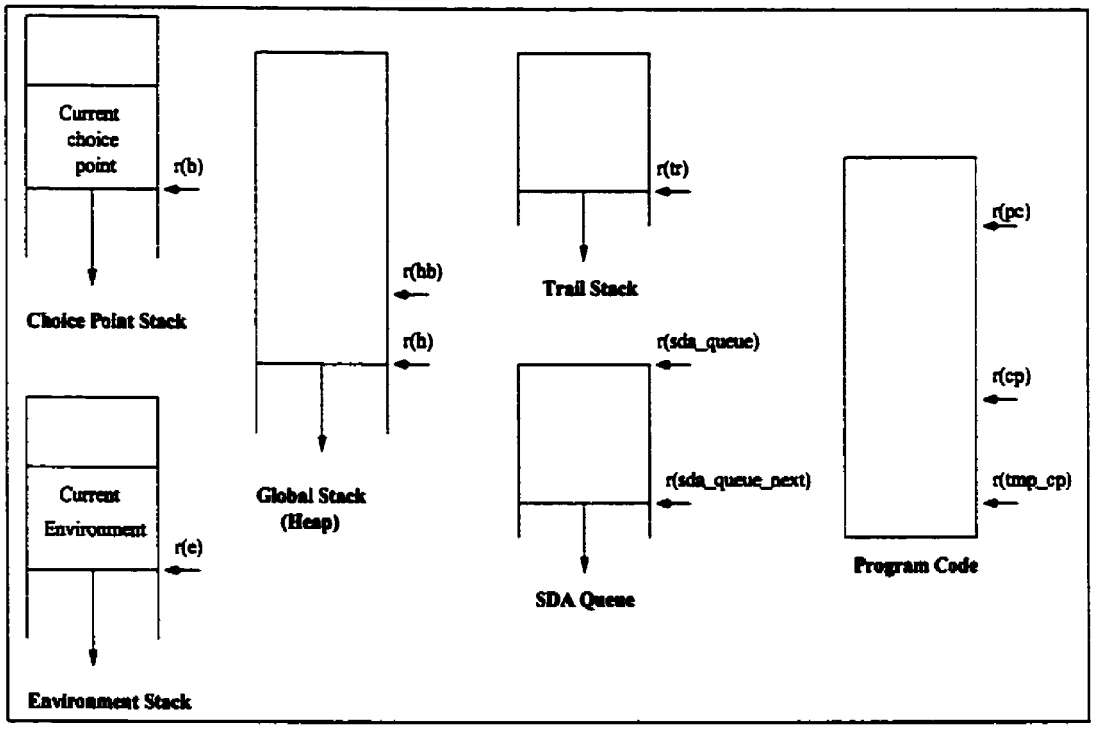

Figure 2.1: BAM Memory Areas

The BAM has six memory areas: *the environment stack*, *the choice point stack*, *the global stack (heap)*, *the trail*, *the SDA queue* and *the program code area*. During forward execution, a Prolog variable can be assigned only one value and the variable may not be re-assigned. For efficiency reasons, the Aquarius Prolog compiler recognizes a Prolog extension known as *stepped destructive assignment* [35] that is supported by the memory area SDA queue. However, the present work does not

consider Prolog programs with stepped destructive assignment. Hence neither this memory region nor its associated registers are discussed further.

Algorithm 1 presents the Prolog execution algorithm. We introduce the BAM memory areas by referring to corresponding data structures of the algorithm. Unifications result in variables being bound to data values as the execution progresses. The algorithm does not specify any data structure to store these data values. The *heap* stores these data values built at run-time. Further, in the algorithm variable binding information of the current predicate is saved (Step 4) in a variable V to be restored at the time of re-trial (Step 17) of the current goal. The *trail* stack stores this information.

The environment stack and choicepoint stack provide the functionality of **GStack**, the goal stack and **ChStack** the choice stack. The stack items stored on these stacks are known as *environments* and *choicepoints* respectively. The structure of these run-time entities is as follows.

2.2.1.1 Environment

Information needed during procedure execution of an imperative language program is maintained in an *activation record* [2]. An environment is similar to an activation record, both in content and intent. A clause is referred to as a *procedure* and a predicate call is also referred to *procedure call.* An environment stores values of the variables that occur across the clause. It also contains a pointer to the call return location (which is an instruction) and the previous environment on the stack.

2.2.1.2 Choicepoint

A choicepoint records the next available code location to be tried if a predicate call fails. Since the abstract machine state needs to be restored for the re-trial, the following information is stored prior to trying an execution path in the choicepoint:

- values of the current procedure variables to facilitate their restoration.

- current heap top such that all the data values built on the heap during the failed path can be discarded.

- current trail top such that currently all unbound variables can be restored to their unbound state.

Further, the return address of the current procedure call and the previous choicepoint address on the choicepoint stack are also stored.

The environment stack and choicepoint stack may either be implemented in a single memory space, known as the *local* stack or separately as shown in Figure 2.1. The symbol table is arranged as a hash table whose form and hashing function are not specified as part of the BAM semantics.

2.2.2 Data Types and Registers

A BAM data entity is called a *dataword*. BAM supports two dataword formats: tagged and untagged words. Untagged datawords represent machine integers and memory addresses. Tagged datawords contain a *tag* representing the data type and a *value* representing the data value with a tag-dependent interpretation. The tag and value components of a dataword will henceforth be referred to as *datatag* and *datavalue* respectively to avoid any confusion that may arise from the usage of the general terms "tag" and "value". A tagged dataword is written as `T^V` where `T` is the datatag and `V` is the datavalue. Table 2.1 shows datatags and their corresponding datavalue interpretations. An unbound variable is represented as a dataword on the heap with a datatag `tvar` and a self-referential address as datavalue. A compound term is represented by a dataword with a datatag `tstr` and address of a fixed number of contiguous heap cells that store its functor and arguments. A list term is represented by a dataword with a datatag `tlst` and an address of two contiguous heap cells indicating its head and tail. The other datatags and their datavalues are evident from the table.

BAM has three types of registers — *state*, *argument* and *permanent* registers (Table 2.2). Eight state registers contain untagged datawords that point to various

Table 2.1: Datatags in BAM	
Tag	Datatype represented
`tvar`	An unbound variable or a general pointer
`tstr`	Pointer to a structure – a compound term with a functor and fixed number of arguments.
`tlst`	Pointer to a cons cell – a compound term consisting of two parts, a head and a tail.
`tatm`	An atom.
`tpos`	A nonnegative integer.
`tneg`	A negative integer.
`tint`	A integer.
`tflt`	A floating-point number.

Table 2.2: BAM registers	
Registers	Description
`r(pc)`	Program counter
`r(e)`	Pointer to current environment on environment stack
`r(b)`	Pointer to top-most choice point on choice point stack
`r(h)`	Pointer top of the heap (i.e., global stack)
`r(cp)`	Continuation pointer (return address)
`r(tmp_cp)`	Continuation pointer to interface with assembly code
`r(tr)`	Top of trail
`r(hb)`	Heap backtrack point - top of heap when current choice point was created.
`r(0)`, ..., `r(N)`	argument and temporary registers
`p(0)`, ..., `p(N)`	permanent variables

BAM memory areas and execution information. Of these registers, only the heap register `r(h)` and the backtrack pointer `r(b)` are explicitly visible in the instructions generated by the Aquarius compiler. The rest are implicitly manipulated by the various BAM instructions.

The *argument* registers correspond to the arguments of the current predicate call. They are represented as `r(0)`, `r(1)`,...,`r(N)` where `N` is arbitrarily finite. They may contain either tagged or untagged datawords. The *permanent* registers correspond to the variables of the current predicate call whose scope spans over the current clause. They are represented as `p(0)`, `p(1)`,...,`p(M)` where `M` is arbitrarily finite. They may contain either tagged or untagged datawords. To the use of argument and permanent registers, consider the clause,

```
p(X,Y,Z) :- q(Y,X), r(Z,Y), t(Y,X).
```

The three variables `X`, `Y` and `Z` are in the scope of `p/3`. At the procedure entry their values are in argument registers `r(0)`, `r(1)` and `r(2)` in accordance to their argument positions. They are stored in three permanent registers `p(0)`, `p(1)` and `p(2)` in the environment which is created upon procedure entry. Similarly, the variables `Y` and `X` of the predicate `q(Y,X)` correspond to the argument registers `r(0)`, `r(1)`. Hence before a call to `q/2` is made, the registers `r(0)` and `r(1)` are correctly set using the permanent registers `p(1)` and `p(0)` respectively. Based on this basic calling convention, several optimizations exist to reduce the number of permanent variables based on their occurrence in the clause [3, 13, 70].

2.2.2.1 Addressing Modes

The following are the *addressable* entities of BAM along with the values they evaluate to:

- atomic terms referred to as directly addressable entities. These evaluate to themselves and are also called to as *immediate values.*
- registers referred to as directly addressable entities. These evaluate to their contents.
- entities of the form `T^X`, where `X` is an addressable entity. These evaluate to a dataword with datatag `T` and datavalue `X`.
- entities of the form `[X]` referred to as *indirect addressables.* These evaluate to the contents of `X`, where `X` is an addressable.
- entities of the form `X+N` referred to as *offset addressables.* These evaluate to the address `N` locations beyond `X` in the memory.

2.2.3 Instruction Set

The BAM instruction set may be divided into four categories [29]:

- **Procedural Control Flow Instructions**: These instructions provide unconditional flow of control. Table 2.3 summarizes these instructions.

- **Conditional control flow instructions**: These instructions provide clause selection and backtracking mechanisms. Table 2.4 summarizes these instructions.

- **Unification Instructions**: These instructions implement term unification. Table 2.5 summarizes these instructions.

- **Arithmetic Instructions**: These instructions perform the binary operations `add`, `sub`, `mul`, `div`, `mod`, `and`, `or` and `xor` and the unary operations logical shift left (`sll`), arithmetic shift right (`sra`), and bit-complement (`not`). Further, instructions that convert between integer and floating point and between tagged and untagged values are also provided. As these instructions are similar to those of a general purpose RISC processor, they are not detailed here.

- **Pragma instructions**: `pragma` instructions embed information that may be used in the Aquarius Prolog compiler back-end for better translation of the target machine to assembly language. A summary of these instructions is given in Table 2.6.

2.2.4 Compilation of Prolog to BAM Code

Here we outline how Aquarius compiles Prolog programs into BAM code. This is similar to that of a WAM-based Prolog compiler. The GFA phase of the Aquarius compiler however is not discussed.

The first clause of the Prolog program text is deemed to be the program entry point. Its arity needs to be zero. In other words, the body of the entry point predicate is the query to the program. Consider the following Prolog program.

```
main :- foo(a,B,c).
```

Table 2.3: Procedural Control Flow Instructions of BAM

Instruction	Description
`procedure(P)`	Marks the entry point to procedure `P`.
`entry(P,N)`	Marks an acceptable point where memory overflow check and garbage collection can occur.
`allocate(N)`	Create an environment of size `N` on the local stack.
`deallocate(N)`	Remove the top-most environment, of size `N`, from the environment stack.
`call(P)`	Call the procedure `P`.
`return`	Return from a procedure call.
`label(L)`	Marks `L` as a branch target.
`jump(L)`	Jump unconditionally to label `L`.
`jump_ind(X)`	Jump to address in `X`
`simple_call(P)`	Non-nestable call used to interface with routines written in BAM assembly language of the VLSI realization of BAM and of no relevance in the present work.
`simple_return`	Non-nestable return used for routines written in BAM assembly language and hence of no relevance in the present work.

Table 2.4: Conditional Control Flow Instructions of BAM

Instruction	Description
`hash(V,T)`	Look up value `V` in a hash table, `T`.
`switch(V,T,L1,L2,L3)`	Branch to `L1`, `L2`, `L3` depending on the tag of `V` being `tvar`, `T` or any other value respectively.
`test(E,T,X,L)`	Branch to `L` if tag of `X` is equal to or not equal to `T` depending on whether `E` is `eq` or `ne`, respectively.
`jump(T,C,A,B,L)`	Conditional branch to `L` if numeric comparison `C` between `A` and `B` holds; data types of `A` and `B` need be consistent with `T`.
`cut(V)`	Removes latest choice point from local stack. `V` contains the address of previous choice point.
`choice(1/N,Rs,L)`	Create a choice point containing the registers listed in `Rs` and set the retry address to `L`. $N > 1$
`choice(I/N,Rs,L)`	(1<I<N) Restore argument registers listed in `Rs` from the current choice point and modify the retry address to `L`.
`choice(N/N,Rs,L)`	Restore the argument registers listed in `Rs` from the current choice point and pop the current choice point from the local stack. `N` is a positive integer.
`trail(X)`	Push address of `X` onto trail stack if trail condition $X < r(h)$ holds.
`fail`	Restore trailed variables and jump to retry address in current choice point.

Table 2.5: Unification Instructions

Instruction	Description
`deref(S,D)`	Dereference `S` and store result in `D`; `F` indicates mode of `S`, if known.
`equal(S1,S2,L)`	Branch to `L` if `S1` and `S2` are not equal; else fall through.
`unify(V1,V2,F1,F2,L)`	General Unification of `V1` and `V2` branch to `L` on failure. Trailing is done by this instruction. `F1,F2` $\in$ {`?`, `var`, `nonvar`}. `var` and `nonvar` indicate whether `V1` and `V2` are known to be variables or nonvariabes. `?` indicates nothing is known about them.
`unify_atomic(V,A,L)`	Unify `V` with atom `A` and branch to `L` if it fails. No trailing is done by this instruction.
`move(S,D)`	Move S to D.
`push(S,R,N)`	Push `S` onto the stack with stack pointer `R` and increment `R` by `N`.
`adda(S,O,D)`	Add offset `O` to the tagged pointer in `S` and store result in `D`.
`pad(N)`	Add `N` words to the heap pointer.

Table 2.6: Embedded information (pragmas)

Instruction	Description
`pragma(align(X,N))`	The contents of location `X` are a multiple of `N`.
`pragma(tag(X,Tag))`	The contents of location `X` have a tag `Tag`.
`pragma(push(term(N)))`	A term of size `N` is about to be created on the heap.
`pragma(push(cons))`	A cons cell is about to be created on the heap.
`pragma(push(structure(N)))`	A structure of arity `N` is about to be created on the heap.
`pragma(push(variable))`	An unbound variable is about to be created on the heap.
`pragma(hash_length)`	A hash table of length `hash_length` is about to be created.

```
foo(X,Y,Z) :- q(Y,X), s(Z,Y), t(Y,X).
```

The basic compilation scheme of the above program is illustrated as follows.

- Body goal compilation:

  1. Generate code to create an environment that holds permanent variables used across the clause if there are more than one body goals.

     The clause `main/0` has no permanent variables. So no environment need be created during its execution. Hence no corresponding BAM code is generated while compiling the clause `main/0`.

     The predicate `foo/3` has a body with more than one predicate call. Hence, compilation of the body of `foo/3` begins by generating code that creates an environment. The environment stores the values of the three permanent variables `X`, `Y` and `Z` in permanent registers `p(0)`, `p(1)` and `p(2)` respectively.

  2. Generate code to load the argument registers with corresponding argument values to set up for a procedure call.

     Thus code to load the argument registers `r(0)`, `r(1)` and `r(2)` with `tatm^a`, `tvar^r(h)` and `tatm^c` is generated during compilation of the body of `main/0` to set up for a call to `foo/3`.

     Compilation of the body of `foo/3` next generates code to load the argument registers of `q/2`, viz., `r(0)` and `r(1)` from `p(1)` and `p(0)` respectively to set up the ensuing call to `q/2`.

     Each body predicate is compiled similarly. However, instead of generating a call to the last predicate in the body, i.e., `t/2`, code to deallocate the current environment followed by a jump to `t/2` is emitted. This may be done since there are no more body predicates to use the permanent register values in the environment. This technique is often referred to as *last call optimization* and allows all arbitrarily deep tail-recursive predicates to run with a constant number of environments.

- Head compilation: The head of a clause is compiled to BAM code that unifies the argument registers with the non-variable arguments, if any. Multiple clauses of a predicate definition are compiled using choicepoint instructions. The example program does not result in choicepoint instructions. An example of compiling to choicepoint instructions is given in Section 4.3.2.

Summarizing, Figure 2.2 shows the stylized form of BAM code for the two clauses listed above .

```
main/0 : load tatm^a r(0)
         load tvar^r(h) r(1)
         load tatm^c r(2)
         call foo/3

foo/3  : /* head argument unification not done */
         /* as no non-variable arguments exist */
         allocate environment with 3 variables
         move r(0) to p(1)
         move r(1) to p(0)
         move r(2) to p(2)
         /* Set argument registers to call q/2 */
         move r(1) to r(0)
         move p(0) to r(1)
         call q/2
         move p(2) to r(0)
         move p(1) to r(1)
         call s/2
         move p(1) to r(0)
         move p(0) to r(1)
         deallocate environment with 3 variables
         jump to t/2
```

Figure 2.2: A simple compiled BAM code format

With this background, we present a complete denotational semantics specification of all BAM instructions generated by the Aquarius Prolog compiler in the next chapter. We also present the BAM execution model. The semantics specification provides the basis to show the correctness of various instruction specializations. It further provides an implementation-independent specification for the partial executor that is implemented.

Chapter 3

Denotational Semantics of BAM

This chapter presents the denotational semantics of BAM. These semantics are implementation-independent and provide a basis for the implementation of a BAM partial evaluator as well as for proof of correctness of the transformations performed by the BAM partial evaluator.

3.1 Notational Conventions

An overview of the terminology, primitive domains and the operators used in the definition of the denotational semantics is provided. **bold** letters are used for domain names; `typewriter` font for syntactic constructs, domain tags and BAM instruction opcodes and operands. We continue to use *italics* while defining or introducing new terminology in running text. We also use *italics* to represent a set element in set notation and to represent function or operator names and their arguments in mathematical notation. The context shall make the meaning unambiguous. The *CALLIGRAPHIC* font is used for valuation functions.

The notational conventions used closely follow those of Schmidt [59]. The natural number domain **N**, rational number domain **Q**, truth value domain **B**, and character domain **C** are the primitive domains used to build the semantic domains of BAM. The character domain **C** is defined as follows.

$$\mathbf{C} = \{x \mid x \text{ is an ASCII character}\}.$$

Given a domain **D**, the *power set* of **D** i.e., the set of all subsets of **D** is denoted

as **P(D)**.

3.2 Domain Constructors

The following conventional domain constructors, along with their corresponding assembly and disassembly operators, are used.

The *product* of n domains, $\mathbf{A_1}, \mathbf{A_2}, \ldots, \mathbf{A_n}$ where $n > 1$, is defined as

$$\mathbf{A_1} \times \mathbf{A_2} \times \ldots \times \mathbf{A_n} = \{(x_1, x_2, \ldots, x_n) \mid x_1 \in \mathbf{A_1}, \ldots, x_n \in \mathbf{A_n}\}.$$

The *disassembly operator* of a product domain, denoted $\downarrow i$, maps an element of the domain to its ith element.

$$(x_1, x_2, \ldots, x_n)\downarrow i = x_i.$$

The union of two or more disjoint domains is known as *disjoint union*. We denote the disjoint union of two domains **A** and **C** as $\mathbf{A} + \mathbf{C}$ which is defined as

$$\mathbf{A} + \mathbf{C} = \{(\mathbf{zero}, x) \mid x \in \mathbf{A}\} \cup \{(\mathbf{one}, y) \mid y \in \mathbf{C}\}.$$

Entities **zero** and **one** "flag" members of **A** and **C** and are referred to as *domain tags*. The entities x and y are referred to as *value* components of an element of the disjoint domain $\mathbf{A} + \mathbf{C}$.

The assembly operators *in***A** and *in***C** are defined as

$$\forall x \in \mathbf{A},\ \mathit{in}\mathbf{A}(x) =(\ \mathbf{zero}, x) \text{ and } \forall y \in \mathbf{C},\ \mathit{in}\mathbf{C}(y) = (\mathbf{one}, y).$$

To define the disassembly operator of any $p \in \mathbf{A} + \mathbf{C}$, we quote Schmidt [59] to avoid any confusion this notation might result due to its uniqueness.

> To remove the tag from an element $p \in \mathbf{A} + \mathbf{B}$, we can simply say $p\downarrow 2$, but will instead resort to a better structured operation called *cases*. For any $p \in \mathbf{A} + \mathbf{C}$, the value of
>
> cases p of
>
> $\quad \mathrm{is}\mathbf{A}(x) \rightarrow x$
>
> $\quad [] \mathrm{is}\mathbf{C}(y) \rightarrow y$
>
> end
>
> is "x" when $p = (\mathbf{zero}, x)$ and "y" when $p = (\mathbf{one}, y)$. The *cases* operation makes good use of the tag on the sum element; it checks the tag

before removing it and using the value. Do not be confused by the is**A** and is**C** phrases. They are not new operations. You should read the phrase is$\mathbf{A}(x) \rightarrow x$ as saying "if p is an element whose tag component is zero and whose value component is x, then the answer is x". As an example, for

$f(m) =$ cases m of
is$\mathbf{N}(n) \rightarrow n+1$
[] is$\mathbf{B}(b) \rightarrow 0$
end

f(in**N**(2)) = f(**zero**,2) = 2+1 = 3, but f(in**B**(true)) = f(**one**,true) = 0.

The disjoint union operation may be extended to more than two domains. Correspondingly the definition for assembly and disassembly operators may be extended. We assume the definition of these operators on each disjoint domain and thus are not specified explicitly whenever one is constructed.

3.3 Functions and Function Domains

A function is most commonly represented as an equation. For example, the function between the domains **N** and **N** that doubles a natural number may be defined as

$$double(x) = 2^*x$$

In general, a function f between a domain **A** and a codomain **C** is represented (or defined as) as

$$f(x) = e$$

where $x \in \mathbf{A}$ and e is an expression that may contain occurrences of x and that evaluates to an element in **C**. The notation $f : \mathbf{A} \rightarrow \mathbf{C}$ is used to refer to the function f.

The function f is alternately represented using *typed lambda notation* as $\lambda x.e$. This form is known as *abstraction*. In equational representation, the abstractions

are given names such as f. Using this notation a function need not be named.

The application of a function $f : \mathbf{A} \rightarrow \mathbf{C}$ to $a \in \mathbf{A}$ is denoted by $f\ a$ when unambiguous and as $f(a)$ otherwise. The result of function application is $c \in \mathbf{C}$, such that f maps a to c.

Following is the description of another domain constructor known as *function space builder*. For domains $\mathbf{A}$ and $\mathbf{C}$, a *function space* is a domain of all functions from domain $\mathbf{A}$ to codomain $\mathbf{C}$ and is denoted by $\mathbf{A} \rightarrow \mathbf{C}$. Function application is the disassembly operator of this domain building operation, i.e.,

$$somefunction(someargs) : (\mathbf{A} \rightarrow \mathbf{C}) \times \mathbf{A} \rightarrow \mathbf{C}.$$

It may be noted here that function application is represented as an infix operator for clarity purposes. The following is (the more familiar) prefix representation of the same:

$$()(somefunction, someargs) : (\mathbf{A} \rightarrow \mathbf{C}) \times \mathbf{A} \rightarrow \mathbf{C}.$$

where $somefunction \in \mathbf{A} \rightarrow \mathbf{C}$ and $someargs \in \mathbf{A}$. The function application produces $c \in \mathbf{C}$, where $somefunction$ maps a to c.

Given a function space $fs : \mathbf{A} \rightarrow \mathbf{C}$ we designate $[a \mapsto c]fs$ to represent the function which is exactly the same as fs except that it maps the value $a \in \mathbf{A}$ to $c \in \mathbf{C}$.

We now define the notation used in the context of finite sequences of elements of a domain. Let $\mathbf{D}$ be any domain. $\mathbf{D}^*$ denotes a domain of all finite sequences of elements of $\mathbf{D}$. If $d \in \mathbf{D}^*$ then either d is the empty sequence nil or $d = d_1 :: d_2 :: \ldots :: d_n :: nil$, where $n > 0$ and $d_i \in \mathbf{D}$ such that $1 \leq i \leq n$. The ith element in a sequence d is represented by $elem\ i\ d$; the first element (head) by $hd\ d$ and the remainder (tail) by $tl\ d$. We assume the standard operator $cons$, represented as $::$, maps $d_1 \in \mathbf{D}$ and sequence $d \in \mathbf{D}^*$ to a sequence $d' = d_1 :: d \in \mathbf{D}^*$. Given a non-nil finite sequence $d = d_1 :: d_2 :: \ldots :: d_n :: nil$, any sequence $d_i :: d_2 : i+1 \ldots :: d_n :: nil$, such that $1 \leq i \leq n$ is referred to as a *suffix* of d.

An undefined element is represented by $\perp$. Given a domain $\mathbf{A}$ we represent the corresponding *lifted* domain as $\mathbf{A}_\perp$ to represent $\mathbf{A} \cup \{\perp\}$.

The *if-then-else* conditional expression is represented as

$$x \rightarrow y \; [] \; z$$

and is read as "if x then y else z". The expression evaluates to y if x is true or to y otherwise. The *if-then-else* conditional is in no way related to the *cases* operator defined earlier. The symbol [] happens to be part of the syntax of these. For clarity, the expression is sometimes written on multiple lines. The infix operator == is used to test equality while the infix operator = is used for definition of an expression.

3.4 Semantic Domains

The semantic domain **String** consists of character strings:

$$\textbf{String} = \mathbf{C}^*$$

A Prolog atom is a character sequence that uniquely denotes some entity, as defined in Section 2.1.1.1, in the problem domain. Let **Atom** denote the domain of Prolog atoms. Thus,

$$\textbf{Atom} = \{x \mid x \in \textbf{String} \text{ and } x \text{ satisfies the syntactic conditions specified in Section 2.1.1.1}\}.$$

A functor designates the name and arity of a structure. The domain of functors **Funct** is

$$\textbf{Funct} = \{\mathtt{f/n} \mid \mathtt{f} \; \textit{in} \; \textbf{Atom} \text{ and } \mathtt{n} \in \mathbf{N}\}$$

For example `p(a,b)` has functor `p/2` with name `p` and arity 2.

The computer memory is characterised by an address and its contents. An address is a natural number:

$$\textbf{Address} = \mathbf{N} + \mathbf{IV},$$

where **IV** is a domain consisting of initial values of various memory areas of BAM. These are identified when specifying the domains corresponding to the memory areas.

The content of a BAM memory location is a dataword. A dataword in BAM is one of the two types: tagged or untagged. Let **Untagged** and **Tagged** denote the domain of untagged and tagged datawords, respectively. An untagged dataword denotes a memory address (a pointer) or an integer value. Thus,

$$\mathbf{Untagged} = \mathbf{Address}.$$

It may be noted that an element of the domain **Untagged** is distinguished as an address or an integer by its use only. Further, the domain element does not restrict the address to be of any one of the possibly several memory areas of an architecture such as heap, stack or code area.

A tagged dataword consists of a tag and a value. Tags indicate the data type represented. There are two pointer types – atomic and pointer. The three atomic tag domains are defined as

$$\mathbf{T_a} = \{\texttt{tatm}\}$$

$$\mathbf{T_i} = \{\texttt{tint, tpos, tneg}\}$$

$$\mathbf{T_f} = \{\texttt{tflt}\}$$

The domain of pointer tag types, is defined as

$$\mathbf{T_p} = \{\texttt{tvar, tlst, tstr}\},$$

The tag domain T is defined as

$$\mathbf{T} = \mathbf{T_a} + \mathbf{T_i} + \mathbf{T_f} + \mathbf{T_p}.$$

The tags and corresponding value interpretation of a tagged dataword is given in Table 2.1 of Section 2.2.2.

The atomic tagged dataword domains are defined as

$$\mathbf{DW_a} = \mathbf{T_a} \times (\mathbf{Atom} + \mathbf{Funct})$$

$$\mathbf{DW_i} = \mathbf{T_i} \times \mathbf{N}$$

$$\mathbf{DW_f} = \mathbf{T_f} \times \mathbf{Q}$$

The pointer tagged dataword domain is defined as

$$\mathbf{DW_p} = \mathbf{T_p} \times \mathbf{Address}$$

Thus the tagged dataword domain is defined as

$$\mathbf{Tagged} = \mathbf{DW_a} + \mathbf{DW_i} + \mathbf{DW_f} + \mathbf{DW_p}.$$

DW, the domain of datawords is defined as

$$\mathbf{DW} = \mathbf{Tagged} + \mathbf{Untagged}.$$

BAM registers are classified as state registers, argument registers and permanent registers. Each register contains a dataword.

3.4.1 State Register Domain

$\mathbf{R_s}$, the domain of BAM state registers is

$$\mathbf{R_s} = \{\texttt{pc},\texttt{e},\texttt{b},\texttt{h},\texttt{cp},\texttt{tr},\texttt{hb}\}.$$

The function domain **StRegVal** maps state registers to their contents and is defined as

$$\mathbf{StRegVal} = \mathbf{R_s} \rightarrow \mathbf{DW}$$

An operator *initstatereg* $\in$ **StRegVal** initializes state registers. The domain **Untagged** is expanded to include the initial values of the state registers. These initial values point to respective initialized but empty memory areas. Thus the domain

$$\mathbf{Untagged} = \{init\_pc, init\_e, init\_b, init\_h, init\_tr\}$$

contains initial values of state registers `pc`,`e`,`b`,`h`, and `tr`, respectively. The initial value of cp along with specification of *initstatereg* is discussed in Section 3.4.8.

The operator *getstreg* that maps $r \in \mathbf{R_s}$ and $s \in$ **StRegVal** to the current content of r is defined as

$$getstreg : \mathbf{R_s} \rightarrow \mathbf{StRegVal} \rightarrow \mathbf{DW}$$

$$getstreg = \lambda r.\lambda s.(s\ r)$$

The operator *setstreg* that maps $r \in \mathbf{R_s}$, its new content $d \in \mathbf{DW}$ and register state $s \in$ **StRegVal** to a new register state s' that contains the updated value of r as d, is defined as

$$setstreg : \mathbf{R_s} \rightarrow \mathbf{DW} \rightarrow \mathbf{StRegVal} \rightarrow \mathbf{StRegVal}$$

$$setstreg = \lambda r.\lambda d.\lambda s.[r \mapsto d]s$$

3.4.2 Argument Register Domain

The BAM architecture assumes an arbitrary but finite number of argument registers. The argument register domain, $\mathbf{R_a}$, is defined as

$$\mathbf{R_a} = \{\texttt{r0},\ \texttt{r1},\ \texttt{r2},\ldots,\texttt{r}n\} \text{ for } n \geq 0.$$

The function domain **ArgRegVal** that maps the argument registers to their contents is

$$\mathbf{ArgRegVal} = \mathbf{R_a} \rightarrow \mathbf{DW}_\perp$$

Let $\perp$ signify that no argument registers are mapped yet. This is represented by lifting **ArgRegVal**. The operators of **ArgRegVal** are defined slightly differently from those of **StRegVal** owing to the availability of an arbitrary number of argument registers.

The initialization operator of **ArgRegVal** is

$$initargreg : \mathbf{ArgRegVal}_\perp$$

$$initargreg = \perp$$

The operator to access an argument register value is defined as

$$getargreg : \mathbf{R_a} \rightarrow \mathbf{ArgRegVal}_\perp \rightarrow \mathbf{DW}_\perp$$

$$getargreg = \lambda r.\underline{\lambda} v.(v\ r)$$

The underlined lambda specifies that the curried function $\underline{\lambda} v.(v\ r)$ maps $\perp \in \mathbf{ArgRegVal}_\perp$ to $\perp \in \mathbf{DW}$. This signifies the undefined value resulting from a register value access from an uninitialized register value mapping. This is in accordance with the usual interpretation. Further the definition of function space **ArgRegVal**: $\mathbf{R_a} \rightarrow \mathbf{DW}_\perp$ implies that for a function $v \in \mathbf{ArgRegVal}$ and $r \in \mathbf{R_a}$ the function application $(v\ r)$ either results in a value in **DW** mapped to r if such a mapping exists, or results in $\perp$ otherwise.

The operator to update a register value is defined as

$$setargreg : \mathbf{R_a} \rightarrow \mathbf{DW} \rightarrow \mathbf{ArgRegVal}_\perp \rightarrow \mathbf{ArgRegVal}$$

$$setargreg = \lambda r.\lambda d.\lambda s.[r \mapsto d]s$$

The domain of registers, **Registers**, is

$$\mathbf{Registers} = \mathbf{R_s} + \mathbf{R_a}.$$

A domain **RegState** is

$$\mathbf{RegState} = \mathbf{StRegVal} \times \mathbf{ArgRegVal}_\perp.$$

Permanent registers are not included since they are accessible only via the environment stack.

The initialization operator of **RegState** is defined as

$$initregstate : \mathbf{RegState}$$

$$initregstate = (initstatereg, initargreg)$$

getregvalue that accesses values of a register $r \in$ **Registers** given $s \in$ **RegState** is defined as

$$\begin{aligned}
&\textit{getregvalue} : \textbf{Registers} \rightarrow \textbf{RegState} \rightarrow \textbf{DW}_{\perp} \\
&\textit{getregvalue} = \lambda r.\lambda s.(\text{cases } r \text{ of} \\
&\qquad\qquad \text{is}\mathbf{R_s}(v) \rightarrow \textit{getstreg } v\ s{\downarrow}1 \\
&\qquad\qquad [] \ \text{is}\mathbf{R_a}(v) \rightarrow \textit{getargreg } v\ s{\downarrow}2 \\
&\qquad \text{end})
\end{aligned}$$

setregvalue that maps a register $r \in$ **Registers**, a value $d \in$ **DW** and a register state $s \in$ **RegState** to a new register state $s' \in$ **RegState** that has the value of r updated to d, is defined as

$$\begin{aligned}
&\textit{setregvalue} : \textbf{Registers} \rightarrow \textbf{DW} \rightarrow \textbf{RegState} \rightarrow \textbf{RegState} \\
&\textit{setregvalue} = \lambda r.\lambda d.\lambda s.(\text{cases } r \text{ of} \\
&\qquad\qquad \text{is}\mathbf{R_s}(v) \rightarrow ((\textit{setstreg } v\ d\ s{\downarrow}1), s{\downarrow}2) \\
&\qquad\qquad [] \ \text{is}\mathbf{R_a}(v) \rightarrow (s{\downarrow}1, (\textit{setargreg } v\ d\ s{\downarrow}2)) \\
&\qquad \text{end})
\end{aligned}$$

The environment and choicepoint stacks are considered to be separate stacks in the memory throughout our discussion, as shown in Figure 2.1.

3.4.3 Permanent Register Domain

Permanent registers are stored in the environment stack whose semantic domain is specified in Section 3.4.4. The BAM architecture assumes the availability of an arbitrary (but finite) number of permanent registers.

The permanent register domain, $\mathbf{R_p}$, is defined as

$$\mathbf{R_p} = \{\texttt{p0},\ \texttt{p1},\ \texttt{p2},\ldots \texttt{p}n\} \text{ for } n \geq 0.$$

The function domain **PermRegVal** maps permanent registers to their contents

$$\textbf{PermRegVal} = \mathbf{R_p} \rightarrow \textbf{DW}_{\perp}$$

The initialization operator for the domain **PermRegVal** is

$$\textit{initpermstate} : \mathbf{N} \rightarrow \textbf{PermRegVal}_{\perp}$$

$$initpermstate(n) = \begin{cases} \perp & \text{if } n = 0 \\ \{(\mathbf{p}x, \perp), \forall x \mid 0 \leq x < n\} & \text{if } n > 0 \end{cases}$$

where $n \in \mathbf{N}$ is the number of permanent registers to be created. Since permanent registers exist in an environment, specification of the access and updating operators of these registers is given in Section 3.4.4 where the semantic domains of environment and environment stack are specified.

3.4.4 Environment and Environment Stack Domain

An environment contains (a finite number of) permanent register values, address of the previous active environment and return address for the current predicate. Thus the environment domain **Env** is defined as

$$\mathbf{Env} = \mathbf{PermRegVal}_{\perp} \times \mathbf{Address} \times \mathbf{Address}.$$

Consequently the environment stack domain **EnvStack** is defined as

$$\mathbf{EnvStack} = (\mathbf{Address} \times \mathbf{Env}_{\perp})^*.$$

The access and update operators on the domain **PermRegVal** are defined as

$$getpermreg : \mathbf{R_p} \rightarrow \mathbf{EnvStack} \rightarrow \mathbf{DW}_{\perp}$$

$$getpermreg = \lambda r.\underline{\lambda} e.(((hd\ e)\downarrow 2)\downarrow 1\ r)$$

and

$$setpermreg : \mathbf{R_p} \rightarrow \mathbf{DW} \rightarrow \mathbf{EnvStack} \rightarrow \mathbf{EnvStack}$$

$$setpermreg = \lambda r.\lambda v.\lambda e.(cons\ ([r \mapsto v]((hd\ e)\downarrow 2)\downarrow 1)\ (tl\ e))$$

3.4.5 Choicepoint and Choicepoint Stack Domain

A choicepoint contains a finite number of argument register values, pointers to the top of the heap, trail and environment stack when the current choicepoint was created; value of the register cp when the current choicepoint was created; address of the previous choicepoint; and address of the next choice associated with the current choicepoint. Thus the choicepoint domain **ChoicePt** is defined as

$$\textbf{ChoicePt} = \textbf{DW}^* \times \textbf{Address} \times \textbf{Address} \times \textbf{Address} \times \textbf{Address} \times \textbf{Address} \times \textbf{Address}.$$

Consequently the choicepoint stack domain **ChPtStack** is defined as

$$\textbf{ChPtStack} = (\textbf{Address} \times \textbf{ChoicePt}_\perp)^*.$$

Let *init_b* $\in$ **Address** be the initial location of the choicepoint stack.

An operator *chpsuffix* which maps *addr* $\in$ **Address** and *chpt* $\in$ **ChPtStack** to *chpt'* $\in$ **ChPtStack** such that *chpt'* is a suffix of *chpt* and the head of *chpt'* is a mapping of *addr* to its corresponding choicepoint, is defined as

$$\textit{chpsuffix}:\textbf{Address} \rightarrow \textbf{ChPtStack} \rightarrow \textbf{ChPtStack}_\perp$$

$$\textit{chpsuffix} = \lambda a.\lambda s.(((\textit{hd}\ s) == (\textit{init\_b}, \perp)) \rightarrow \perp$$

$$[] (((\textit{hd}\ s)\downarrow 1 == a) \rightarrow s$$

$$[] \textit{chpsuffix}\ a\ (\textit{tl}\ s)))$$

This operator may be seen as a multiple pop operation on the choicepoint stack that pops the stack elements until the top of the stack has the given address.

3.4.6 Heap Domain

The heap contains datawords. Thus the domain **Heap** is defined as

$$\textbf{Heap} = (\textbf{Address} \times \textbf{DW}_\perp)^*$$

Let *init_h* $\in$ **Address** be the initial location of the heap.

A heap access operator *getheapvalue* that maps $a \in$ **Address**, $h \in$ **Heap** to dataword $d \in \textbf{DW}_\perp$, which is the content of a in h, is

$$\textit{getheapvalue}:\textbf{Address} \rightarrow \textbf{Heap} \rightarrow \textbf{DW}_\perp$$

$$\textit{getheapvalue} = \lambda a.\lambda h.(((\textit{hd}\ h) == (\textit{init\_h}, \perp)) \rightarrow \perp$$

$$[] (((\textit{hd}\ h)\downarrow 1 == a) \rightarrow (\textit{hd}\ h)\downarrow 2$$

$$[] (\textit{getheapvalue}\ a\ (\textit{tl}\ h))))$$

3.4.7 Trail Domain

The trail contains argument registers that need be unbound upon backtracking. Thus the argument registers and the heap addresses of the variable they represent

are stored on the trail stack. The domain **Trail** is

$$\textbf{Trail} = (\textbf{Address} \times (\textbf{R}_a \times \textbf{DW}))^*$$

Let *init_tr* $\in$ **Address** be the initial location of the trail.

The operator *trsuffix* maps *addr* $\in$ **Address** and *tr* $\in$ **Trail** to *tr'* $\in$ **Trail** such that *tr'* is a suffix of *tr* and the head of *tr'* is a mapping of *addr* to its corresponding register value pair. It is defined as

$$\begin{aligned}
&\textit{trsuffix}{:}\textbf{Address} \rightarrow \textbf{Trail} \rightarrow \textbf{Trail}_\perp \\
&\textit{trsuffix} = \lambda a.\lambda s.(((hd\ s) == (init_tr, \perp)) \rightarrow \perp \\
&\quad [] ((hd\ s)\downarrow 1 == a \rightarrow s \\
&\quad\quad [] trsuffix\ a\ (tl\ s)))
\end{aligned}$$

This operator may be seen as a multiple pop operation on the trail stack which pops the trail until the top of the stack has the given address.

3.4.8 Memory State Domain

The internal state of BAM at any given time of execution may be characterized by the collective state of its registers, heap, environment stack, choicepoint stack and trail. Hence **Memory** is

$$\textbf{Memory} = \textbf{RegState} \times \textbf{Heap} \times \textbf{EnvStack} \times \textbf{ChPtStack} \times \textbf{Trail}$$

BAM instructions use only a subset of the addressing modes as defined in Section 2.2.2.1. Thus an addressable entity is an immediate value and is one of the following.

- immediate value i.e., a dataword
- argument register
- permanent register
- `r(h)`, `r(b)`
- [R], where R is an addressable entity
- [R+N], where R is an addressable entity and N $\in$ **N**

Thus the domain **Adrable** is defined as

Adrable: $\mathbf{DW}+\mathbf{R_a}+\mathbf{R_p}+\mathbf{HB}+\mathbf{Indirect}+\mathbf{Offset}$

where $\mathbf{HB} = \{\mathtt{r(h)}, \mathtt{r(b)}\}$

$\mathbf{Indirect} = \{x \mid x = [\mathrm{R}]\}$, where R is an addressable entity

$\mathbf{Offset} = \{x \mid x = [\mathrm{R+N}]\}$,

where R is an addressable entity and $\mathrm{N} \in \mathbf{N}$

getvalue that accesses the value of addressable entity is defined as

getvalue: $\mathbf{Adrable} \rightarrow \mathbf{Memory} \rightarrow \mathbf{DW}$

$getvalue = \lambda a.\lambda m.$cases a of

$\mathbf{isDW}(d) \rightarrow d$

$[]\ \mathbf{isR_a}(d) \rightarrow getargreg\ d\ s{\downarrow}2$

$[]\ \mathbf{isR_p}(d) \rightarrow getpermreg\ d\ e$

$[]\ \mathbf{isHB}(d) \rightarrow getstreg\ d\ s{\downarrow}1$

$[]\ \mathbf{isIndirect}([d]) \rightarrow getheapvalue\ (getvalue\ d\ m)\ h$

$[]\ \mathbf{isOffset}([d+n]) \rightarrow getheapvalue\ ((getvalue\ d\ m)+n)\ h$

end

where $m = (s,h,e,c,t)$.

setvalue that updates the value of addressable entity is defined as

setvalue: $\mathbf{Adrable} \rightarrow \mathbf{DW} \rightarrow \mathbf{Memory} \rightarrow \mathbf{Memory}$

$setvalue = \lambda a.\lambda v.\lambda m.$cases a of

$\mathbf{isDW}(d) \rightarrow (s,[d \mapsto v]h,e,c,t)$

$[]\ \mathbf{isR_a}(d) \rightarrow ((s{\downarrow}1,(setargreg\ d\ v\ s{\downarrow}2)),h,e,c,t)$

$[]\ \mathbf{isR_p}(d) \rightarrow (s,h,(setpermreg\ d\ v\ e),c,t)$

$[]\ \mathbf{isHB}(d) \rightarrow (((setstreg\ d\ v\ s{\downarrow}1),\ s{\downarrow}2),h,e,c,t)$

$[]\ \mathbf{isIndirect}(d) \rightarrow setvalue\ (getvalue\ d\ m)\ v\ m$

$[]\ \mathbf{isOffset}([d+n]) \rightarrow setvalue\ ((getvalue\ d\ m)+n)\ m$

end

where $m = (s,h,e,c,t)$.

The access and update operators specify the addressing modes allowed in BAM.

The input and output to the program are considered to be a sequences of char-

acter strings. Thus,

Input=String*

Output=String*

The instruction set is denoted by **Instr**. The program code is defined as follows:

$$\textbf{ProgCode} = \textbf{Address} \rightarrow \textbf{Instr}$$

These initial state register values are described as follows:

- pc: The address of the first executable instruction in the program $p \in \textbf{ProgCode}$ denoted by $init\_pc \in \textbf{DW}$.
- e: The address of the first location in the environment stack denoted by $init\_e \in \textbf{DW}$. $(init\_e, \perp)$ denotes the initialized environment stack.
- b: The address of the first location in the choicepoint stack denoted by $init\_b \in \textbf{DW}$. $(init\_b, \perp)$ denotes the initialized choicepoint stack.
- h: The address of the first location in the heap denoted by $init\_h \in \textbf{DW}$. $(init\_h, \perp)$ denotes the initialized heap.
- tr: The address of the first location in the trail stack denoted by $init\_tr \in \textbf{DW}$. $(init\_tr, \perp)$ denotes the initialized trail.
- hb: Its initial value is $init\_h \in \textbf{DW}$.

3.4.9 BAM Code Execution

A BAM code execution is deemed to have terminated if pc is mapped to one of the special values *success* and *failure* or to a value that points to an address outside the address area of **ProgCode**. The execution is said to have successfully terminated if pc maps to *success*. It is deemed to be a failure in other cases. Correspondingly the initial value of the continuation pointer is mapped to *success* to indicate no code need be executed upon the return of the first call.

We digress to modify the domain **DW** to include these special values and correspondingly define the necessary disassembly operators.

$\mathbf{DW} = \mathbf{Tagged}+\mathbf{Untagged}+\mathbf{Spl}$

where $\mathbf{Spl}=\{success, failure\}$

The operator *gettag* on a domain element of **DW** is defined as

$gettag : \mathbf{DW} \rightarrow \mathbf{T}_\perp$

$gettag = \lambda s.$ cases s of

$\mathbf{isTagged}(d) =$ cases d of

$\mathbf{isDW_a}(da) = da\downarrow 1$

$[]\ \mathbf{isDW_i}(di) = di\downarrow 1$

$[]\ \mathbf{isDW_f}(df) = df\downarrow 1$

$[]\ \mathbf{isDW_p}(dp) = dp\downarrow 1$

end

$[]\ \mathbf{isUntagged}(d) = \perp$

$[]\ \mathbf{isSpl}(d) = \perp$

end

The operator *getdataval* on a domain element of **DW** is defined as

$getdataval : \mathbf{DW} \rightarrow \mathbf{DW}_\perp$

$getdataval = \lambda s.$ cases s of

$\mathbf{isTagged}(d) =$ cases d of

$\mathbf{isDW_a}(da) =$ cases $da\downarrow 2$ of

$\mathbf{isAtom}(dat) = dat$

$[]\ \mathbf{isFunct}(daf) = daf$

end

$[]\ \mathbf{isDW_i}(di) = di\downarrow 2$

$[]\ \mathbf{isDW_f}(df) = df\downarrow 2$

$[]\ \mathbf{isDW_p}(dp) = dp\downarrow 2$

end

$[]\ \mathbf{isUntagged}(d) = d$

$[]\ \mathbf{isSpl}(d) = d$

end

The initialization operator of the domain **StRegVal**, *initstatereg* left unspec-

ified in Section 3.4.1, is defined as a mapping of the state registers $\mathbf{R_s}$ to their corresponding initial values.

$initstatereg = \mathbf{R_s} \rightarrow \mathbf{StRegVal}$

$initstatereg = ((\mathtt{pc}, init\_pc), (\mathtt{e}, init\_e), (\mathtt{b}, init\_b), (\mathtt{h}, init\_h),$
$(\mathtt{cp}, success), (\mathtt{tr}, init\_tr), (\mathtt{hb}, init\_hb))$

An operator *initpgmcode* that maps a sequence of instructions to its corresponding function in **ProgCode** is assumed to be defined.

The memory initialization operator *initmem* is defined as follows.

$initmem$: **Memory**

$initmem = (initstatereg, (init\_h, \perp), (init\_e, \perp), (init\_b, \perp), (init\_tr, \perp))$.

An element of the domain **ProgState**, defined as

ProgState : **ProgCode** × **Memory** × **Input** × **Output**,

represents the BAM state along with the current position in the code, the input consumed and the output produced till that point of program execution. The operator that initializes a new program state is defined as

$initprogstate$: $\mathbf{Instr}^* \rightarrow \mathbf{Input} \rightarrow \mathbf{ProgState}$.

$initprogstate = \lambda p.\lambda i.(((initpgmcode\ p), initstatereg, initmem, i, nil))$

Now we return to definition of operators relevant to program execution and termination. The termination test operator for BAM on a given $s \in \mathbf{ProgState}$ is defined as

$terminate$: $\mathbf{ProgState} \rightarrow \mathbf{B}$

$terminate((code, mem, in, out)) =$
$(getregvalue\ \mathtt{pc}\ s) == success \rightarrow true$
$[]\ ((getregvalue\ \mathtt{pc}\ s) == failure \rightarrow true$
$[]\ ((getregvalue\ \mathtt{pc}\ s) == \perp \rightarrow true$
$[]\ false))$
where $mem = (s, h, se, sc, tr)$

An operator that performs the "fetch" operation upon a given program state is defined as

$fetchinstr$: $\mathbf{ProgState} \rightarrow \mathbf{Instr}_\perp$

$$fetchinstr((code, mem, in, out)) = code\ (getregvalue\ \mathtt{pc}\ s)$$

where $mem = (s, h, se, sc, tr)$

Conventionally, instructions that are targets of a branch or that start a procedure are attributed a label. A separate instruction `label` instruction designates such entry points in BAM code. For example, consider the following pseudo-code of a typical RISC architecture:

```
          ....
          move r1, r2
          compare 0, r2, r3
          jump_on_not_zero 'zlbl'
          ....
          ....
'zlbl'  : move r4, r2
          ....
```

Using the style of BAM code, this code segment is written as

```
          ....
          move(r1,r2).
          compare(0, r2, r3).
          jump_on_not_zero('zlbl').
          ....
          ....
          label('zlbl').
          move(r4, r2).
          ....
```

The label `fail` in the instructions does not correspond to a program label. A transfer of BAM execution to `fail` results in execution of global failure whose semantics are given by the valuation function for the instruction `fail.`, viz., $\mathcal{I}[\![\mathtt{fail}]\!]$ in the following section.

An address look-up operator for a given label or procedure instruction *instr*, on a *code* is defined as

$$fetchaddr : \mathbf{Instr} \rightarrow \mathbf{ProgCode} \rightarrow \mathbf{DW}_{\perp}$$

$$fetchaddr(instr, code) = (instr == \texttt{fail}) \rightarrow \mathcal{I}[\![\texttt{fail}]\!] \; [] \; addr$$

where $(code\ addr) = instr$

3.5 Valuation Functions

The semantics of BAM execution model is provided by a valuation function $\mathcal{B}$ defined on an instruction sequence and an input.

$$\mathcal{B} : \mathbf{Instr}^* \rightarrow \mathbf{Input} \rightarrow \mathbf{ProgState}$$

$$\mathcal{B} = \lambda p.\lambda i.\mathcal{S}(initprogstate\ p\ i)$$

The valuation function $\mathcal{S}$ maps a program state p to a new program state p' by evaluating a sequence of instructions whose first instruction address is in the register pc.

$$\mathcal{S} : \mathbf{ProgState} \rightarrow \mathbf{ProgState}$$

$$\mathcal{S} = \lambda p.terminate\ p \rightarrow p$$

$$[] \; \mathcal{S}\ (\mathcal{I}[\![fetchinstr\ p]\!]\ p)$$

BAM instructions that are syntactic constructs are distinguished by enclosing them in $[\![\ ]\!]$. No other semantics are attributed to the usage of this notation.

The valuation function $\mathcal{I}$ maps an instruction i and a program state p to a new program state p' by evaluating i with respect to program state p.

$$\mathcal{I} : \mathbf{Instr} \rightarrow \mathbf{ProgState} \rightarrow \mathbf{ProgState}$$

The specification of $\mathcal{I}$ for each of the BAM instructions follows. We assume the definition of an operator *incr* that maps a register $r \in \mathbf{Registers}$, a value $d \in \mathbf{Z}$ and a register state $s \in \mathbf{RegState}$ to a new register state s' where the value of the register r is incremented by d. Further, the components of an element of a product domain are explicitly specified in a *where* expression to simplify complex compositions. If the compositions are simple, the disassembly operator $\downarrow$ is used.

3.5.1 Procedure Control Flow Instructions

$$\mathcal{I}[\![\texttt{procedure}(f)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
$$\text{where } mem = (s, h, se, sc, tr),\ f \in \textbf{Funct},$$
$$mem' = (s', h, se, sc, tr),$$
$$s' = incr\ \text{in}\mathbf{R}_s(\texttt{pc})\ 1\ s$$

$$\mathcal{I}[\![\texttt{entry}(f, n)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
$$\text{where } mem = (s, h, se, sc, tr),\ f \in \textbf{Funct} \text{ and } n \in \mathbf{N},$$
$$mem' = (s', h, se, sc, tr),$$
$$s' = incr\ \text{in}\mathbf{R}_s(\texttt{pc})\ 1\ s$$

$$\mathcal{I}[\![\texttt{allocate}(n)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
$$\text{where } mem = (s, h, se, sc, tr),\ n \in \mathbf{N},$$
$$mem' = (s'', h, se', sc, tr),$$
$$se' = cons\ ((addr, (p, et, ct))\ se),$$
$$addr = (getregvalue\ \text{in}\mathbf{R}_s(\texttt{e})\ s) + 1,$$
$$p = initpermstate\ n,$$
$$et = (getregvalue\ \text{in}\mathbf{R}_s(\texttt{e})\ s),$$
$$ct = (getregvalue\ \text{in}\mathbf{R}_s(\texttt{cp})\ s),$$
$$s' = incr\ \text{in}\mathbf{R}_s(\texttt{e})\ (n + 2)\ s,$$
$$s'' = incr\ \text{in}\mathbf{R}_s(\texttt{pc})\ 1\ s'$$

$$\mathcal{I}[\![\texttt{deallocate}(n)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
$$\text{where } mem = (s, h, se, sc, tr),\ n \in \mathbf{N},$$
$$mem' = (s'', h, se', sc, tr),$$
$$se' = tl\ s,$$
$$s' = incr\ \text{in}\mathbf{R}_s(\texttt{e})\ (-n - 2)\ s,$$
$$s'' = incr\ \text{in}\mathbf{R}_s(\texttt{pc})\ 1\ s'$$

$$\mathcal{I}[\![\texttt{call}(p)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$

where $mem = (s, h, se, sc, tr)$,

$mem' = (s'', h, se, sc, tr)$,

$s' = setregvalue\ \text{in}\mathbf{R}_s(\text{cp})\ ((getregvalue\ \text{in}\mathbf{R}_s(\text{pc})\ s) + 1)\ s$,

$s'' = setregvalue\ \text{in}\mathbf{R}_s(\text{pc})\ (fetchaddr\ \texttt{procedure}(p)\ code)\ s'$

$$\mathcal{I}[\![\texttt{return}]\!]((code, mem, in, out)) = ((code, mem', in, out))$$

where $mem = (s, h, se, sc, tr)$,

$mem' = (s', h, se, sc, tr)$,

$s' = setregvalue\ \text{in}\mathbf{R}_s(\text{pc})\ (getregvalue\ \text{in}\mathbf{R}_s(\text{cp})\ s)\ s$

$$\mathcal{I}[\![\texttt{label}(l)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$

where $mem = (s, h, se, sc, tr)$, $l \in \textbf{String}$,

$mem' = (s', h, se, sc, tr)$,

$s' = incr\ \text{in}\mathbf{R}_s(\text{pc})\ 1\ s$

$$\mathcal{I}[\![\texttt{jump}(l)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$

where $mem = (s, h, se, sc, tr)$, $l \in \textbf{String}$

$mem' = (s', h, se, sc, tr)$,

$s' = setregvalue\ \text{in}\mathbf{R}_s(\text{pc})\ (fetchaddr\ \texttt{label}(l)\ code)\ s$

3.5.2 Conditional Control Flow Instructions

$$\mathcal{I}[\![\texttt{switch}(t, x, l1, l2, l3)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$

where $mem = (s, h, se, sc, tr)$, $t \in \mathbf{T}$,

$x \in \textbf{Adrable}$ and $l1, l2, l3 \in \textbf{String}$

$mem' = (s', h, se, sc, tr)$,

$s' = (\textbf{tvar} == gettag\ (getvalue\ x\ mem)) \rightarrow$

$setregvalue\ \text{in}\mathbf{R}_s(\text{pc})\ (fetchaddr\ \texttt{label}(l1)\ code)\ s,$

$[]\ ((t == gettag\ (getvalue\ x\ mem)) \rightarrow$

$$setregvalue\ \text{in}\mathbf{R}_s(\texttt{pc})\ (fetchaddr\ \texttt{label}(l2)\ code)\ s,$$
$$[]\ setregvalue\ \text{in}\mathbf{R}_s(\texttt{pc})\ (fetchaddr\ \texttt{label}(l3)\ code)\ s)$$

$$\mathcal{I}[\![\texttt{jump}(t,c,x,y,l)]\!]((code,mem,in,out)) = ((code,mem',in,out))$$
$$\text{where } mem = (s,h,se,sc,tr),$$
$$mem' = (s',h,se,sc,tr),$$
$$s' = (compare\ (getvalue\ x\ mem)\ (getvalue\ y\ mem)\ t\ c) \rightarrow$$
$$setregvalue\ \text{in}\mathbf{R}_s(\texttt{pc})\ (fetchaddr\ \texttt{label}(l)\ code)\ s,$$
$$[]\ incr\ \text{in}\mathbf{R}_s(\texttt{pc})\ 1\ s$$

The comparison operation, c, of values of $x, y \in$ **Adrable** whose tag type is indicated by t as integer, float or untagged, is one of the following: equality, inequality, $<$, $\leq$, $>$, $\geq$. We assume this operator is defined.

$$\mathcal{I}[\![\texttt{cut}(v)]\!]((code,mem,in,out)) = ((code,mem',in,out))$$
$$\text{where } mem = (s,h,se,sc,tr),\ v \in \textbf{Adrable},$$
$$mem' = (s''',h,se,sc',tr),$$
$$s' = setregvalue\ \text{in}\mathbf{R}_s(\texttt{b})\ (getvalue\ v\ mem)\ s,$$
$$sc' = chpsuffix\ (getvalue\ v\ mem)\ sc,$$
$$s'' = setregvalue\ \text{in}\mathbf{R}_s(\texttt{hb})\ ((hd\ sc')\downarrow 2)\downarrow 2\ s',$$
$$s''' = incr\ \text{in}\mathbf{R}_s(\texttt{pc})\ 1\ s''$$

Following are the semantics of the three instances of choicepoint management instruction.

$$\mathcal{I}[\![\texttt{choice}(1/n,r,l)]\!]((code,mem,in,out)) = ((code,mem',in,out))$$
$$\text{where } mem = (s,h,se,sc,tr),\ n \in \mathbf{N},$$
$$mem' = (s''',h,se,sc',tr),$$
$$sc' = cons\ (addr,cp)\ sc,$$
$$addr = (getvalue\ \texttt{b}\ mem) + 1,$$
$$cp = ((regslots\ r\ s\downarrow 2), (getregvalue\ \text{in}\mathbf{R}_s(\texttt{h})\ s),$$
$$(getregvalue\ \text{in}\mathbf{R}_s(\texttt{tr})\ s), (getregvalue\ \text{in}\mathbf{R}_s(\texttt{e})\ s),$$

$$(getregvalue\ \mathbf{inR_s}(\mathtt{cp})\ s),$$
$$(getvalue\ \mathtt{b}\ mem), (fetchaddr\ l\ code)),$$
$$s' = incr\ \mathbf{inR_s}(\mathtt{b})\ ((length\ r) + 6)\ s,$$
$$s'' = setregvalue\ \mathbf{inR_s}(\mathtt{hb})\ (getvalue\ \mathtt{h}\ mem)\ s',$$
$$s''' = incr\ \mathbf{inR_s}(\mathtt{pc})\ 1\ s''$$

The operator *regslots* is defined as follows:

$$regslots : \mathbf{R_a}^* \rightarrow \mathbf{ArgRegVal} \rightarrow \mathbf{DW}^*$$
$$regslots = \lambda r.\lambda s.((r == nil) \rightarrow nil,$$
$$[]\ cons\ (getargreg\ (hd\ r)\ s)\ (regslots\ (tl\ r)\ s))$$

$$\mathcal{I}[\![\mathtt{choice}(i/n, r, l)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
where $mem = (s, h, se, sc, tr),\ i \in \mathbf{N},\ n \in \mathbf{N}, i < n,$
$$mem' = (s'', h, se, sc', tr),$$
$$(hd\ sc)\downarrow 2 = (rl, h, tr, e, cp, b, re),$$
$$sc' = cons\ (rl, h, tr, e, cp, b, (fetchaddr\ l\ code))\ (tl\ sc),$$
$$s' = loadregs\ r\ rl\ s,$$
$$s'' = incr\ \mathbf{inR_s}(\mathtt{pc})\ 1\ s'$$

The operator *loadregs* is defined as follows.

$$loadregs:\ \mathbf{R_a}^*_\perp \rightarrow \mathbf{DW}^* \rightarrow \mathbf{RegState} \rightarrow \mathbf{RegState}$$
$$loadregs : \lambda r.\lambda v.\lambda s.((r = nil) \rightarrow s,$$
$$[]\ ((hd\ r) = \perp \rightarrow true$$
$$[]\ loadregs\ (tl\ r)\ (tl\ v)\ (setregvalue\ (hd\ r)\ (hd\ v)\ s)))$$

The list of registers may also contain $\perp$ to signify an argument register that need not be restored. The domain of such register lists is denoted as $\{\mathbf{R_{a\perp}}\}^*$ in the above operator definition.

$$\mathcal{I}[\![\mathtt{choice}(n/n, r, l)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
where $mem = (s, h, se, sc', tr),\ n \in \mathbf{N},$
$$mem' = (s', h, se, sc, tr),$$

$$(hd\ sc) = (rl, h, tr, e, cp, b, re),$$
$$sc' = (tl\ sc),$$
$$s_1 = loadregs\ r\ rl\ s,$$
$$s_2 = setregvalue\ \mathbf{inR_s}(\mathtt{b})\ b\ s_1,$$
$$s_3 = setregvalue\ \mathbf{inR_s}(\mathtt{hb})\ h\ s_2,$$
$$s' = incr\ \mathbf{inR_s}(\mathtt{pc})\ 1\ s_3$$

$$\mathcal{I}[\![\mathtt{fail}]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
where $mem = (s, h, se, sc, tr)$,
$$mem' = (s', h, se, sc, tr')$$
$$s_1 = restoreregs\ ((hd\ sc)\downarrow 2)\downarrow 3\ tr\ s,$$
$$tr' = trsuffix\ ((hd\ sc)\downarrow 2)\downarrow 3\ tr,$$
$$s_2 = setregvalue\ \mathbf{inR_s}(\mathtt{e})\ ((hd\ sc)\downarrow 2)\downarrow 4\ s_1,$$
$$s_3 = setregvalue\ \mathbf{inR_s}(\mathtt{cp})\ ((hd\ sc)\downarrow 2)\downarrow 5\ s_2,$$
$$s_4 = setregvalue\ \mathbf{inR_s}(\mathtt{h})\ (getregvalue\ \mathbf{inR_s}(\mathtt{hb})\ s_3)\ s_3,$$
$$s' = setregvalue\ \mathbf{inR_s}(\mathtt{pc})\ ((hd\ sc)\downarrow 2)\downarrow 7\ s_4$$

The operator *restoreregs* is defined as follows.

restoreregs: $\mathbf{DW} \rightarrow \mathbf{Trail} \rightarrow \mathbf{RegState}$

$$restoreregs = \lambda a.\lambda t.\lambda s.((a == ((hd\ t)\downarrow 1)) \rightarrow s$$
$$[]\ restoreregs\ a\ (tl\ t)\ (setregvalue\ ((hd\ tr)\downarrow 2)\downarrow 1\ ((hd\ tr)\downarrow 2)\downarrow 2\ s))$$

$$\mathcal{I}[\![\mathtt{test}(c, t, x, l)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
where $mem = (s, h, se, sc, tr)$,
$$mem' = (s', h, se, sc, tr),$$
$$s' = (((gettag\ (getvalue\ x\ mem)) == t) \rightarrow$$
$$((c == \mathbf{eq}) \rightarrow$$
$$setregvalue\ \mathbf{inR_s}(\mathtt{pc})\ (fetchaddr\ \mathtt{label}(l)\ code)\ s$$
$$[]\ incr\ \mathbf{inR_s}(\mathtt{pc})\ 1\ s)$$
$$[]\ ((c == \mathbf{ne}) \rightarrow$$

$$setregvalue\ \text{in}\mathbf{R_s}(\mathbf{pc})\ (fetchaddr\ \mathtt{label}(l)\ code)\ s$$
$$[]\ incr\ \text{in}\mathbf{R_s}(\mathbf{pc})\ 1\ s))$$

3.5.3 Unification Instructions

$$\mathcal{I}[\![\mathtt{deref}(x,y)]\!]((code,mem,in,out)) = ((code,mem',in,out))$$
$$\text{where } (s,h,se,sc,tr) = setvalue\ y\ (deref\ (getvalue\ x\ mem)\ mem)\ mem,$$
$$mem' = (s',h,se,sc,tr),$$
$$s' = incr\ \text{in}\mathbf{R_s}(\mathbf{pc})\ 1\ s$$

The operator $deref$ is defined as

$$deref : \mathbf{Adrable} \rightarrow \mathbf{Memory} \rightarrow \mathbf{DW}$$
$$deref = \lambda d.\lambda m.((gettag\ (getvalue\ d\ m) == \mathbf{tvar}) \rightarrow$$
$$((getheapvalue\ (getdataval\ (getvalue\ d\ m))\ h\ ==\ d) \rightarrow\ d$$
$$[]\ deref\ (getheapvalue\ (getdataval\ (getvalue\ d\ m))\ h)\ m)$$
$$[]\ d)$$
$$\text{where } m = (s,h,se,sc,tr).$$
$$\mathcal{I}[\![\mathtt{equal}(x,y,l)]\!]((code,mem,in,out)) = ((code,mem',in,out))$$
$$\text{where } mem = (s,h,se,sc,tr),$$
$$mem' = (s',h,se,sc,tr),$$
$$s' = (getvalue\ x\ mem == getvalue\ y\ mem) \rightarrow$$
$$incr\ \text{in}\mathbf{R_s}(\mathbf{pc})\ 1\ s,$$
$$[]\ setregvalue\ \text{in}\mathbf{R_s}(\mathbf{pc})\ (fetchaddr\ \mathtt{label}(l)\ code)\ s$$

$$\mathcal{I}[\![\mathtt{unify}(x,y,t1,t2,l)]\!]((code,mem,in,out)) = ((code,mem'',in,out))$$
$$\text{where } mem' = unify\ x\ y\ mem,$$
$$mem'' = ((mem' == \perp) \rightarrow$$
$$setregvalue\ \mathbf{pc}\ (fetchaddr\ \mathtt{label}(l)\ code)\ s,$$
$$[]\ ((incr\ \mathbf{pc}\ 1\ mem'\downarrow 1), mem'\downarrow 2, mem'\downarrow 3, mem'\downarrow 4, mem'\downarrow 5)$$

The values of $t1, t2$ are either **any**, **var** or **nonvar** indicating the tag values of

$x, y \in$ **Adrable** respectively, if known. Their value is any if no information is known. These values are used to optimize the unification operation. Thus they are ignored in the semantics of the **unify** instruction. The unification operator $unify$: **Adrable** $\rightarrow$ **Adrable** $\rightarrow$ **Memory** $\rightarrow$ **Memory**$_\perp$ maps unification of two addressable entities given a memory state either to another memory state if unification succeeds or to $\perp$ if it fails. The *unify* operator is defined as follows.

$$
\begin{array}{l}
unify = \lambda x.\lambda y.\lambda m.(\\
\quad \text{let } tx = gettag\ (getvalue\ x\ m), \\
\quad\quad ty = gettag\ (getvalue\ y\ m), \\
\quad\quad vx = getdataval\ (getvalue\ x\ m), \\
\quad\quad vy = getdataval\ (getvalue\ y\ m), \\
\quad\quad m = (s, h, se, sc, tr)) \\
\quad \text{in } ((tx == \mathtt{tvar}) \bigwedge ((ty \neq \mathtt{tvar}) \bigvee (vx > vy))) \rightarrow \\
\quad\quad ((s', h, se, sc, tr') \\
\quad\quad\quad \text{where } s' = incr\ \mathbf{in R_s}(\mathtt{tr})\ 1\ (setvalue\ x\ (getvalue\ y\ m)\ m)\downarrow 1 \\
\quad\quad\quad\quad tr' = cons\ ((getregvalue\ \mathbf{in R_s}(\mathtt{tr})\ s) + 1, \\
\quad\quad\quad\quad\quad\quad (x, (getvalue\ x\ m)))\ tr \\
\quad\quad [\!]\ ((ty == \mathtt{tvar}) \rightarrow ((s', h, se, sc, tr'), \\
\quad\quad\quad \text{where } s' = incr\ \mathbf{in R_s}(\mathtt{tr})\ 1\ (setvalue\ y\ (getvalue\ y\ m)\ m)\downarrow 1, \\
\quad\quad\quad tr' = cons\ ((getregvalue\ \mathbf{in R_s}(\mathtt{tr})\ s)+1, (x, (getvalue\ y\ m)))\ tr \\
\quad\quad [\!]\ ((tx == \mathtt{tint}) \bigvee (tx == \mathtt{tflt}) \bigvee (tx == \mathtt{tatm})) \rightarrow \\
\quad\quad\quad ((vx \neq vy) \rightarrow \perp \\
\quad\quad\quad\quad [\!]\ m) \\
\quad\quad [\!]\ ((tx == \mathtt{tlst}) \rightarrow \\
\quad\quad\quad ((ty \neq \mathtt{tlst}) \rightarrow \perp \\
\quad\quad\quad\quad [\!]\ (m'' \text{ where } m' = unify\ (getheapvalue\ (getdataval\ x)\ h) \\
\quad\quad\quad\quad\quad\quad\quad\quad (getheapvalue\ (getdataval\ y)\ h)\ m, \\
\quad\quad\quad\quad\quad m'' = unify\ (getheapvalue\ (getdataval\ x) + 1\ h) \\
\quad\quad\quad\quad\quad\quad\quad (getheapvalue\ (getdataval\ y) + 1\ h)\ m), \\
\quad\quad [\!]\ ((tx == \mathtt{tstr}) \rightarrow
\end{array}
$$

$$((ty == \mathtt{tstr}) \rightarrow \perp$$
$$[] (unify\_str\ 1\ (arity\ vx)\ vx\ vy\ m)))))))$$

The operator that iteratively unifies two heap addresses is defined as follows.

$unify\_str : \mathbf{N} \rightarrow \mathbf{N} \rightarrow \mathbf{Address} \rightarrow \mathbf{Address} \rightarrow \mathbf{Memory} \rightarrow \mathbf{Memory}_{\perp}$

$$unify\_str = \lambda i.\lambda j.\lambda x.\lambda y.\lambda m.((i > j) \rightarrow m,$$
$$[]\ unify\_str\ (i+1)\ j\ x\ y$$
$$(unify\ (getheapvalue\ (x+i)\ m{\downarrow}2)$$
$$(getheapvalue\ (y+i)\ m{\downarrow}2)\ m)$$

$$\mathcal{I}[\![\mathtt{trail}(x)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
where $mem' = (s', mem{\downarrow}2, mem{\downarrow}3, mem{\downarrow}4, tr')$,
$tr' = cons\ ((getregvalue\ \mathbf{inR_s}(\mathtt{tr})\ mem{\downarrow}1) + 1,$
$(x,\ (getvalue\ x\ mem)))\ mem{\downarrow}5,$
$s' = incr\ \mathbf{inR_s}(\mathtt{pc})\ 1\ (incr\ \mathbf{inR_s}(\mathtt{tr})\ 1\ mem{\downarrow}1)$

$$\mathcal{I}[\![\mathtt{move}(x, y)]\!]((code, mem, in, out)) = ((code, mem'', in, out))$$
where $mem'' = ((incr\ \mathbf{inR_s}(\mathtt{pc})\ 1\ mem'{\downarrow}1), mem'{\downarrow}2, mem'{\downarrow}3,$
$mem'{\downarrow}4, mem'{\downarrow}5)$,
$mem' = setvalue\ y\ (getvalue\ x\ mem)\ mem$

$$\mathcal{I}[\![\mathtt{push}(d, r, n)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
where $mem = (s, h, se, sc, tr)$,
$mem' = (incr\ \mathbf{inR_s}(\mathtt{pc})\ 1\ (incr\ (getvalue\ r\ mem)\ n)\ s)$,
$(cons\ (((getvalue\ r\ mem)+1),\ (getvalue\ d\ mem))\ h)$
$se, sc, tr)$

$$\mathcal{I}[\![\mathtt{adda}(d, n, r)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
where $mem = (s, h, se, sc, tr)$,
$mem' = (incr\ \mathbf{inR_s}(\mathtt{pc})\ 1\ setvalue\ r$

(incr (getvalue d mem) n) mem)

The pragma instructions contain information that allows better native code translation. The pad instruction is provided to facilitate double word load and store on some architectures. These instructions do not contribute to the functionality of the BAM in any way. The semantics of hash table instructions hash and pair are also not provided. Their functionality of table look-up and entry specification in the hash table is assumed. Further, the semantics of arithmetic operation instructions are not provided as they are evident.

The state, argument and permanent registers in the BAM code generated by the Aquarius compiler are represented as `r(e)`, `r(b)`, ..., `r(0)`, `r(1)`,... and `p0`, `p1`, ... respectively. These are however represented as `e`, `b`,..., `r0`,`r1`,... and `p0`, `p1`, ..., respectively in the above semantics specification to avoid any possible confusion with the usage of parentheses for unambiguous representation of function application. Further, in the BAM code generated, a tagged dataword is denoted as `T^V` where `T` is the tag and `V` is the data value as opposed to its representation as `(T, V)` in our semantics.

3.6 Summary

The theme of this dissertation is partial evaluation of the abstract machine code generated by an optimizing Prolog compiler. Issues involved in partial evaluation of a low-level language have not been studied yet, as far as we know. This work attempts to study partial evaluation of abstract machine code generated by an optimizing Prolog compiler with a GFA phase. The primary goal of performing partial evaluation is to expose opportunities of code optimizations. Consequently, an overview of Prolog — the language and its execution model – was presented in Section 2.1.1. Partial evaluation is done of the abstract machine code generated by Aquarius Prolog compiler that compiles Prolog to native code via Berkeley Abstract Machine (BAM). A detailed description of the Berkeley Abstract Machine (BAM) architecture is pre-

sented. Section 2.2 provides the basic process of Prolog compilation to BAM using a simple example.

This chapter defines an implementation-independent specification of BAM to provide the foundation for showing the correctness of these specializations detailed in Chapter 4. It further provides the specification for the implementation of the BAM code specializer. The implementation of the specializer provides a basis for exposing various issues involved in partial evaluation of BAM code.

Chapter 4

Program Specialization

This chapter presents a concise introduction to a program transformation technique known as *program specialization*. The goal of this transformation is to improve program performance. Specialization of high-level language programs such as Lisp, Prolog, and C has been studied for several years. This dissertation studies program specialization of a low-level language viz., BAM code.

Program specialization of low-level languages is conceptually similar to that of high-level languages. However, the difference in the data abstraction and the context of its usage, viz., as a compiler optimization phase, lead to an entirely different specialization algorithm and a different set of issues related to the machine model of the language. As a simple example, high-level language programs have well-defined program modules such as functions and/or procedures. Such program modules are typically specialized for certain values of their parameters. Low-level language code lacks such modularization and needs to be analyzed to identify "modules" along with their "parameters". The specialization algorithm must then respect any such "modularization" in the context of the machine model to discover opportunities for specialization and affect them. The transformations performed need to be correct in the context of the machine execution model. These considerations entail a BAM code specialization algorithm and proof of correctness of the transformations affected.

This chapter presents an algorithm to perform BAM code specialization followed by the various possible BAM code transformations and a proof of their correctness using the denotational semantics presented in Chapter 3.

Section 4.1 gives a brief introduction to program specialization and some terminology. The reasons for perceived opportunities to optimize BAM code using program specialization (Section 4.1.1) and the structural partitioning of the BAM code to facilitate specialization (Section 4.2) are presented. Given the partitioning and denotational semantics of BAM, transformations that result in optimizations are shown to be correct (Section 4.3). Program specialization is illustrated and pertinent issues such as choicepoint optimization in the context of BAM code specialization are discussed with the help of two examples (Section 4.4). This provides the necessary background to the various issues of BAM code specialization discussed in the subsequent chapters.

4.1 Introduction

Specializing programs by using the portion of (any possibly known) program input that remains constant during repeated runs is termed as *partial evaluation*. This technique may be used at compile-time to improve program performance. Such specialization results in a (possibly) new program, called a *residue*. Stipulating the program input is termed as *input specification*. Stipulating the program input that remains constant across several runs of the program is termed as *constant input specification*. Given a program and its constant input specification, performance of the residue is no worse than that of the original program for any input whose constant portion is the same as specified by the constant input specification. A formal characterization of a *partial evaluator* follows.

Let $\mathcal{L}$ denote a language and $\mathbf{P}_{\mathcal{L}}$ the set of programs written in $\mathcal{L}$. Let $\mathbf{V}$ be the domain of values that expressions of $\mathcal{L}$ may be assigned to. $\mathbf{P}(\mathbf{V})$ denotes the power-domain of $\mathbf{V}$. Let $E_{\mathcal{L}} : \mathbf{P}_{\mathcal{L}} \rightarrow \mathbf{V} \rightarrow \mathbf{V}$ be the evaluation function corresponding to the language $\mathcal{L}$. Let $\mathbf{S}$ be the set of possible specifications of values in $\mathbf{V}$ and $f : \mathbf{S} \rightarrow \mathbf{P}(\mathbf{V})$, be the "concretization" function that maps a specification to a set of values it denotes. For any input specification $s \in \mathbf{S}$ of program $p \in \mathbf{P}_{\mathcal{L}}$, $(f\ s) = v_s \cup v_d$, where $v_s \in \mathbf{V}$ is the set of values that are constant during repeated

runs of p, termed as "known" at specialization time or *static* values, and $v_d \in \mathbf{V}$ is the set of values "unknown" at specialization time, i.e., non-static values or *dynamic* values.

For any program $p \in \mathbf{P}_{\mathcal{L}}$ and any input specification $s_i \in \mathbf{S}$,

$$E\ p\ (f\ s_i) = (f\ s_o)$$

where $s_o \in \mathbf{T}$ is an output specification and $(f\ s_o) \in \mathbb{P}(\mathbf{V})_\perp$ denotes the set of output values. If a program fails to terminate, the output is undefined and is represented by $\perp$.

A program specializer $\xi : \mathbf{P}_{\mathcal{L}} \rightarrow \mathbf{S} \rightarrow \mathbf{P}_{\mathcal{L}}$ is a function such that

$\forall v \in (f\ s_i)\ ((E_{\mathcal{L}}\ p\ a) \neq \perp \Rightarrow (E_{\mathcal{L}}\ p\ a) = (E_{\mathcal{L}}\ (\xi\ p\ s_i)\ a))$, where $a = v_s \cup v_d$.

The program $(\xi\ p\ s_i)$ is the residue. The above succinctly captures the definition of program specialization given by Ershov [26], Jones [42] and Ruf [57] but does not indicate the specialization process that results in the residue. Further, the behavior of the residue is undefined when the program p fails to terminate for an input specification thus allowing any value to be output by the residue in that case. A description of the specialization process in general and for BAM code in particular are discussed later in this chapter.

Note from the above definition of the program specializer that p and its residue $(\xi\ p\ s_i)$ take different inputs, viz., $v_s \cup v_d$ and v_d respectively. It is so defined to emphasize that the output of the residue solely depends on the dynamic part of the input and that the static values are "hard-wired" into the program to form the residue. Equivalently, the inputs to the program and the residue may be the same, viz., $v_s \cup v_d$ with an understanding that the residue does not consume the static input values, v_s, during its execution.

A program may contain constructs that evaluate to constant values either depending on static input or independently. Thus, for a program p and an input specification s_i, the set of static values, $v_s = v_{ss} \cup v_{sc}$, where v_{ss} is the set of static values that depend on input static values specified by s_i and v_{sc} is the set of static values independent of those specified in s_i.

Program specialization involves the following two tasks:

- computation of constructs that are completely dependent on static values, v_s. This is referred to as *reducing* the constructs to "simpler" versions.

- retaining those that depend on the dynamic values, v_d. This is referred to as *residualizing* the construct to be computed at run-time.

These are accomplished by symbolically executing the program in the context of static values during which the specializer needs to decide whether the construct can be *reduced* or *residualized*.

Specialization may be performed even if it is only known that an input value is static but not necessarily the actual input value. The program constructs are annotated as static or dynamic by a pre-specialization analysis done according to *congruence* principle and is known as *Binding Time Analysis* (BTA) [42]. The congruence principle states that a program construct is classified as static only if all its constituents are static. Otherwise, it is classified as dynamic. Program constructs are reduced or residualized by the specialization phase according to these annotations. Such a specialization process, during which the reduce/residualize decision is made based on analysis performed prior to specialization step, is known as *off-line* specialization. If the reduce/residualize decision is made based on analysis done at specialization time when the static values are also available, then it is called *online* specialization. Program variable binding information is computed and used "on the fly". Online specialization does not involve a pre-specialization analysis phase.

The specialization may also take advantage of user annotations to help the reduce/residualize decisions. Such annotations may be used to provide hints to the analysis regarding the static/dynamic properties of a variable. If a specializer does not use user annotations to make these decisions, then it is referred to as an *automatic* specializer.

If the specializer can specialize itself then it is termed to be *self-applicable*. Self-applicability has been an important topic in program specialization and automatic program generation. Traditionally, Futamura's three projections [26, 42] for gener-

ating program specializers, compilers and compiler-compilers leverage on the self-applicability of the program specializer.

This thesis addresses the issue of generating optimized BAM code using specialization during Prolog compilation. In an effort to verify usability of specialization as a compilation phase in a real-world compiler, every attempt is made to minimize the time added to compilation time by designing efficient data structures using the C language. Consequently, the implementation language of the specializer is different from the language being specialized (BAM code). Hence, self-applicability is not an issue in this thesis and will not be discussed further.

The mathematical foundations of program specialization have been traced to Kleene's *s-m-n* theorem [45]. This theorem states that for any function

$f(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m)$, there is a computable function $s_{\{x_1,x_2,\ldots,x_n\}}$ such that

$$s_{\{x_1,x_2,\ldots,x_n\}}(y_1, y_2, \ldots, y_m) = f(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m)$$

for all $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m$. The function $s_{\{x_1,x_2,\ldots,x_n\}}$ is referred to as *specialized* version of f for the arguments $x_1, x_2, \ldots, x_n$. Jones [42] provides a chronological history of the development of the field of program specialization. Program specialization has been applied to functional programming languages such as a subset of Lisp [44] and Scheme [9,19,57], logic programming languages such as Prolog [47,58], constraint logic programming languages [37] and imperative languages like C [4]. Program specialization has in general been used for high-level languages where both the data and the control abstraction are much higher than that of the machine language.

The present work studies partial evaluation of BAM, a low-level abstract machine with data and control abstraction very close to that of a RISC architecture. Partial evaluation of BAM described in the following chapters has two fundamental differences with partial evaluation of a high-level language. Firstly, no source language or source program knowledge is assumed or used during partial evaluation of BAM. Often such information is both available and is used during partial evaluation of high-level languages. As an example, consider Mixtus [58] – an automatic

partial evaluator of Prolog. It uses predicate parameter information, Prolog execution semantics that builds a goal stack (**GStack** in Algorithm 1 in Chapter 2) and cut placement and existence information along with other information during various phases of partial evaluation. The BAM code partial evaluator described herein assumes no knowledge of Prolog nor has any access to the Prolog sources used to generate the BAM code being partial evaluated. It does not depend on the GFA phase which generates the mode/type information regarding the predicates of the Prolog source. It depends solely on the BAM machine model, a multi-stack machine, and its instruction semantics. Thus this work demonstrates the compile-time optimizations achievable by partial evaluation of low-level code.

BAM stacks are tailored to facilitate execution of Prolog-specific features such as backtracking. This leads to the second difference between partial evaluation of high-level language and low-level code. The explicit memory and stack manipulations possible using instructions of a low-level machine like BAM, present a different set of issues to be discussed later. The BAM instruction set facilitates memory accesses in a manner similar to that of a RISC architecture. Such opportunities in a high-level language (or even C) are restricted by the type system. We present the required background for specialization of BAM, outline the BAM specialization process and show the correctness of the transformations employed during the specialization process.

4.1.1 Opportunities to specialize BAM Code

Prolog is a dynamic-typed language. BAM has a finite set of data types. Hence each Prolog variable of a predicate in its corresponding BAM translation can potentially assume any of the BAM data types at run-time. Consequently, the compiled BAM code consists of a code stream for each basic BAM data type a Prolog variable can assume at run-time. Run-time type checks dispatch execution flow to corresponding code stream depending on the type of the variable.

In other words, the abstract machine code is generic enough to facilitate execu-

tion of code corresponding to data-types that are known only at run-time. Global Flow Analysis of Prolog programs have been traditionally [2] used to restrict the generic code to those code streams corresponding only to data types of the values a variable may be assigned at run-time and not all of the possible ones.

Abstract interpretation (AI) based GFA [20,21] of Prolog programs was shown to provide a means for inference of predicate variable run-time data type information that may be used to generate optimized code [65,67,71] or less generic code. The basic methodology employed in AI-based GFA is to map the program value domain to an abstract domain and to analyze/execute the program over the abstract domain instead of the value domain. Several different abstract domains along with corresponding abstract execution and analysis algorithms for pure logic programs and Prolog programs have been proposed for mode, type and data dependence analyses [10,24,53,66]. Getzinger [29] presents a taxonomy of several domains and analyses algorithms. Abstract machine code streams generated for each predicate are then restricted to those predicate variable data types inferred by the GFA algorithm. Similarly, run-time checks are also reduced with this information. Thus, code that handles run-time data types which a variable is known not to have are optimized away. AI-based GFA has been used in this manner to improve code quality and speed-up the resulting executable.

We propose that performing partial evaluation or program specialization at compile-time exposes opportunities for further optimizations. Figure 4.1 shows two options for performing PE during the compilation phases of the Aquarius compiler. If performed on Prolog source as a pre-GFA phase, PE can result in inference of more specific types by the GFA whenever possible. PE may also be performed as a post-GFA phase. This work focuses on further optimizations that can be achieved by a post-GFA partial evaluation.

As pointed out in Section 4.1 program invariants that depend on the input static values, v_{ss} and that are independent of these, v_{sc} are used during program specialization. BAM code specialization done here is intended to be used as a compilation phase. It does not require any explicit input specification. Thus v_{ss}

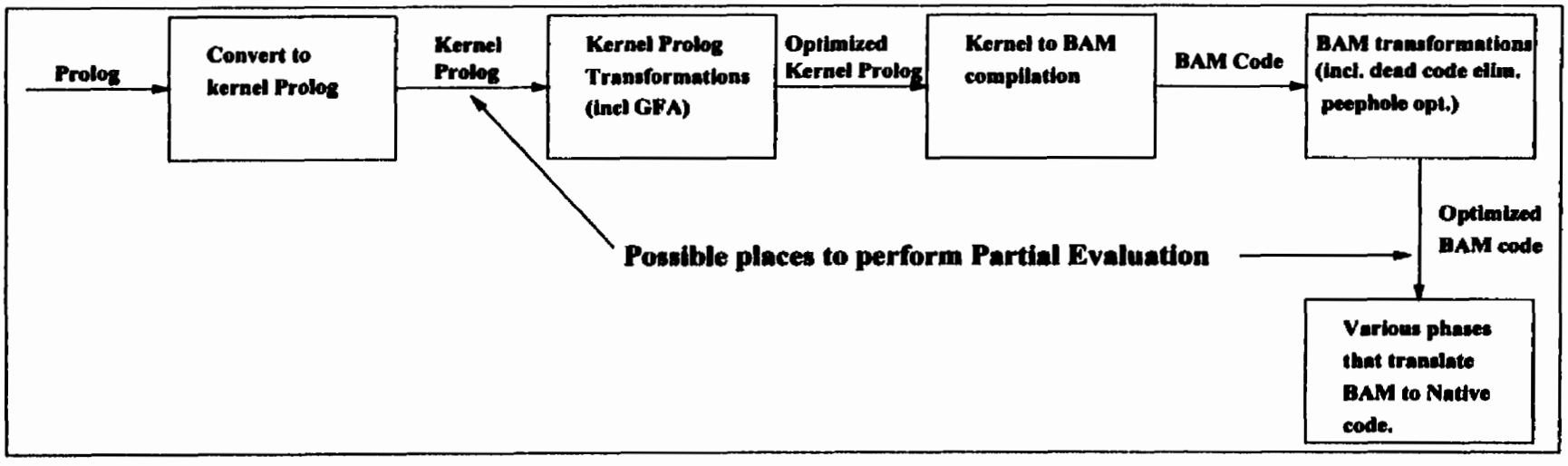

Figure 4.1: Aquarius Prolog compilation phases

$= \phi$. Only the static values uncovered during partial evaluation are used by the process. Thus the partial evaluation of BAM code done here is independent of the input static values and of any user annotations. It is *automatic*.

An alternate view may be taken of the partial evaluation described in this thesis. As explained in Section 4.1, a pre-specialization analysis step typically annotates the program to facilitate the specialization process. Traditionally, Binding Time Analysis (BTA) is used to annotate the program constructs. For the PE process described here, we may view the GFA of the Prolog source as a pre-specialization analysis. The process is then guided by the results of the analysis. The Aquarius compiler supports several user annotations that are may be used by the GFA algorithm to increase the granularity of the deduced data types [35]. The resulting BAM code is thus restricted to code streams for possibly lesser number of data types of the predicate variables of a given program. Partial execution specializes BAM code for these data types. As the language on which GFA is performed is different from that on which PE is performed, the usage of the analysis information in the PE is not directly evident. Thus it may be argued the specialization process is not purely online. However, since the reduce/residualize decision is made at PE-time (as explained in Chapter 5), we consider it to be online partial execution.

WAM, as mentioned briefly in Chapter 2 is the precursor to BAM. It is also the fundamental abstract machine for several popular Prolog compilers and interpreters such as SICStus Prolog [32] and Quintus Prolog [1]. However, the instruction set of WAM consists of complex or coarse-grained instructions that provide little opportunity for specialization. Several efforts were made to create opportunities for specialization to improve the performance of compiled Prolog by "extending" or "specializing" WAM instructions. SEPIA [51,52], SICStus Prolog [12] and the special purpose instruction set of Quintus Prolog are some of the many realizations of such extensions. However, the finer-grained RISC-like instruction set of BAM offers greater specialization opportunities as will be discussed in Section 4.4.

Partial evaluation of BAM code at compile-time has the following potential benefits:

- traditional optimizations such as dead-code and dead-check elimination, expression evaluation and constant propagation are done automatically. Ruf's work [57] on online partial evaluation of a substantial subset of Scheme arrives at a similar conclusion as well in the context of functional programming languages.

- these optimizations/transformations, in turn, enable further back-end optimizations.

It is generally recognized [7] that the above mentioned traditional optimizations are based on partial evaluation. However, there seems little online partial evaluation effort during abstract machine based compilation – particularly to generate optimized code. More specifically, it has not been used to compile high-level, dynamically typed logic programming languages like Prolog. Bulyonkov [11] proposes an algorithm for performing polyvariant partial evaluation for programs written in a low level language much simpler than BAM. Thus the focus of the present work is to investigate the various issues involved in optimizing abstract machine code using partial evaluation.

The definition of program specialization given in Section 4.1 does not specify the specialization process. Such a process for BAM code specialization is discussed in the following section.

4.2 Overview of BAM Code Specialization

Most specializers symbolically execute the program with the available static values and transform the program constructs to simpler equivalents. The constructs of a program are *evaluated* in the context of *partial* knowledge. As the constructs of BAM code are instructions and instruction evaluation is usually known as instruction execution, we refer to the process of specialization of BAM code as *partial execution*(PE). Note that such a specialization process consists of two phases; first symbolic execution in which the program is executed with available static values;

second code transformation in which program constructs are transformed to less expensive equivalents.

Program execution involves a series of transitions from one computation (or execution) state to another. Syntactic constructs in program source corresponding to these computation states are referred to as *program points*. Further, the state of program execution can be meaningfully comprehended at these points. The computation state at a program point is referred to as a *program state*. Syntactic structures such as functions, procedures and predicates in a program are designated as program points during execution of high-level language programs. Unlike high-level languages, BAM code has no pre-defined syntactic structure. Any sequence of BAM instructions is a syntactically legal BAM program. A structure is provided to a BAM instruction sequence by partitioning it into a control flow graph (CFG) [2] of basic blocks. Section 4.2.1 discusses the semantics of CFG representation. Such a partitioning facilitates characterization of basic block entry points as program points.

4.2.1 Partitioning BAM Code into CFG

A basic block is conventionally defined as "a sequence of (zero or more) instructions with no branch instructions, except perhaps the last instruction, and no branch targets or labels, except perhaps at the first instruction" [72] (known as the *leader* of the block). Thus a basic block has a single control flow entry point and a single control flow exit point. A block entry point represented by the unique block label is a program point. A CFG representation of a given BAM code is a graph with basic blocks as nodes and with edges between these nodes representing the program's control flow. Each basic block has a unique number associated with it.

In a conventional CFG, an edge between two basic blocks denotes a transfer of control from the predecessor to its successor node. Two or more out-edges of a node denote transfer of control from the predecessor to one of the successors. These edge semantics capture the control flow due to conventional branch instructions

of three-address code [2]. The BAM instruction set has several of such branch instructions. However, `choice/3` and `fail/0` instructions do not have conventional branch instruction semantics. Further, the CFG does not explicitly represent the control flow transfer due to `call/1` and `return/0`. Thus, an edge in the CFG of a BAM code may either represent a conventional or a BAM-specific branching instruction.

The branching or flow change instructions of BAM are classified into the following categories based on the way they affect the control flow and the information they create and access.

1. **Regular flow change instructions:**

   The instructions `equal/3`, `jump/1`, `jump/5`, `jump_ind/1`, `switch/3`, `switch/5` and `test/4` either change the control flow to an address label that is an explicit operand or to the next instruction. None of these instructions creates or saves information to facilitate return of the control flow to a following block at a later program execution point. These are similar to the conventional branch instructions.

2. **Procedural flow change instructions:**

   The instructions `call/1` and `return/0` create and access data not explicitly present as an instruction operand. The control flow changes are same as the conventional stack-based procedure calls [2]. The flow change target is an explicit operand of `call/1` instruction. The `return/0` instruction returns the control flow to the following block in accordance with the information stored on the environment stack by the immediately preceding `call/1` instruction.

3. **Choicepoint flow change instructions:**

   The `choice/3` instructions create and access data that are both explicitly present as instruction operands and are on the choicepoint stack. The control falls through to the basic block containing the next instruction but can return to the current block to go through the alternate path as indicated by the data

created. Thus a block with a `choice/3` instruction has two edges – a fall through edge and a retry edge.

4. **Backtracking instruction:**
   The instruction `fail/0` accesses choicepoint stack data and has no explicit operands specified. Execution of this instruction sets the control flow to a basic block determined at run-time. Thus a basic block with `fail/0` as last instruction has no successors.

The out-edge semantics of basic blocks with procedural flow change instructions can be illustrated by an example BAM code. Instead of presenting an arbitrary BAM code sequence we use a simple Prolog example and present BAM code generated during its compilation. This will also provide an opportunity to relate program points in the BAM code with those of the Prolog program for the purposes of comprehension. Consider the program `sample.pl` in Figure 4.2 whose BAM code is shown as a CFG in Figure 4.3.

```
main :- p(X),q(X).

p(X) :- r(X).  q(3).
p(X) :- s(X).  r(3).

s(4).
```

Figure 4.2: Program `sample.pl`

Edges out of nodes 6, 10 and 14 are examples of regular flow control change. Control flow transfer occurs exactly along one of the edges. The control transfer occurring due to `call(p/1)` instruction in block 0 to block 2 is not represented by an edge. Similarly the control transfer back to block 1 due to `return` instruction either in in block 13 or 17 is also not explicitly shown as an edge. Such transfer of control due to `call/1` and `return/0` might occur at various program points. The location

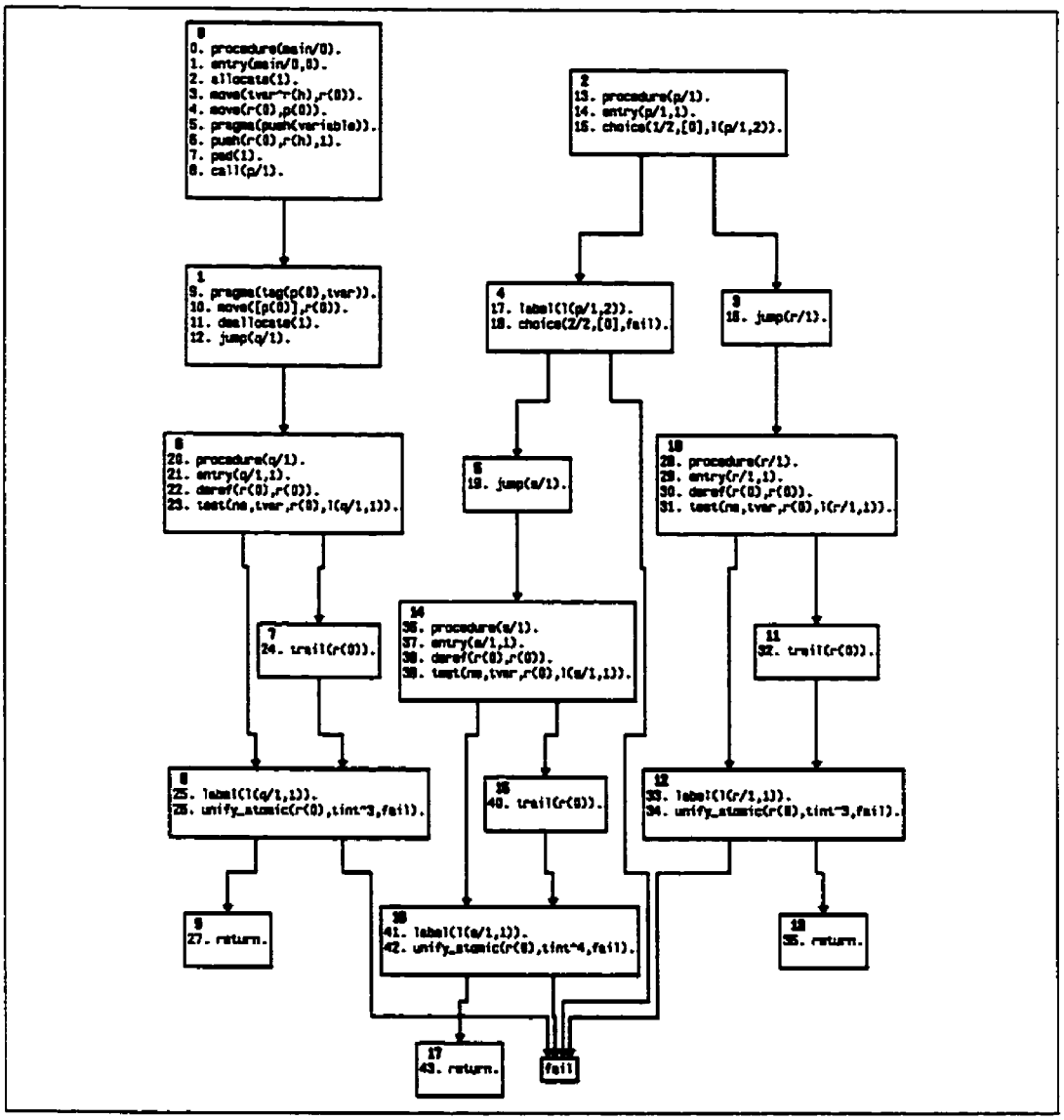

Figure 4.3: CFG of BAM code of sample.pl

of control transfer due to `procedure/1` can be uniquely determined by its operand. Similarly, the return address is available in the continuation pointer `cp`. Hence, successors of blocks with `call/1` and `return/0` instructions are not represented in the CFG explicitly. Consequently, a basic block with a `call/1` instruction has only one successor – the block executed upon returning from the called procedure. The basic block with `return/0` instruction has no successors.

The edges out of blocks 2 and 4 in Figure 4.3 are examples of control flow change due to `choice/3` instruction. Exactly two edges represent control flow. The first one denotes flow control to the block beginning with the instruction following the `choice/3` instruction in the code and the second to the block whose leader has the label specified as alternate choice in the `choice/3` instruction. Thus the control flow along the edges out of a block with `choice/3` instruction are not necessarily mutually exclusive for a given set of register values unlike the case of the out-edges of blocks with regular flow control instructions. The control might return to flow through the alternate edge.

The instruction `fail/0` that triggers control flow to backtrack is represented in a basic block by itself as shown in Figure 4.3. This block has no successors.

4.2.2 Polyvariant Specialization of BAM Code

With the description of a partitioning of given BAM code, we proceed to describe the process of its partial execution. This is done by traversing the basic blocks of the CFG starting at the program entry point. Recall that a compiled Prolog program has only one designated entry point. Instructions in each basic block are executed in the order they occur by building datawords in the registers and on the BAM stacks. However, at PE-time, the datawords built might be incomplete as only their data tags are known. The corresponding data values are usually known only at run-time. Thus the BAM code execution is performed with partial datawords only. During partial execution, if the direction of control flow due to a regular flow change instruction is not decidable, the CFG is traversed depth-first along each

of the mutually exclusive edges. BAM memory is constructed along each of the ensuing paths. Such traversal is termed as *speculative* traversal since the result of the condition in the flow change instruction is assumed to hold along the respective paths. Opportunities to transform instructions are exposed and utilized during partial execution to result in basic block residues.

A basic block may lie on more than one of the traversal paths of partial execution. Partial execution of such a block might have to be performed with respect to different partial information along each of these traversal paths. This might result in different residues for the same block. Such a specialization process, where multiple specialized versions of code at program points is generated, is known as *polyvariant* partial evaluation [11, 42].

Performing the above partial execution process naïvely will often lead to non-termination of the process and possible code explosion [39, 41]. Non-termination occurs due to repeated traversal of the same basic blocks. Such repeated traversal also results in code explosion as the process yields the same residue for each of the block partial executions. Criteria to recognize such attempts to partial execution of blocks previously traversed are required to correctly terminate the PE process as well as to limit the generated residue.

Partial execution, as mentioned earlier, involves transformations of instructions whenever possible. A very common transformation involves replacing an instruction that transfers control flow to a program point (such as a procedure/function/predicate call) with an instance of the code corresponding to the program point. Such a transformation is called *unfolding* [50] at program points. Non-termination of the partial execution can result from repeated unfolding of directly or indirectly recursive procedures. Even if the procedures are not recursive, naïve unfolding might result in code explosion. Criteria to recognize such attempts to perform repeated unfolding are required to let the PE process terminate and limit the generated residue.

Consider a loop in the BAM code. Partial execution of a loop whose upper bound is known at compiler-time could also lead to code explosion. This is similar to problems encountered in loop unrolling [72] performed during conventional com-

piler optimizations. Criteria to prune the depth to which a loop may profitably be unrolled are required to minimize the code explosion.

In essence, criteria to decide whether partial execution may proceed further at a program point or not is crucial both to the partial execution termination and to avoid needless code explosion. Such a criteria allow specialization to occur only a finite number of times at any given program point. Consequently, at each program point, several of its versions may be created each corresponding to the repeated but finite partial executions that specialize the program point to the set of static values.

Such criteria make use of the results of partial execution of program units (basic blocks) that are recorded. This technique of recording the results is usually referred to as *caching* or *tabling*. The criteria help in deciding whether to proceed with PE of a basic block or a previously computed specialization use a previous result. Chapter 6 discusses the issues relating to the criteria used in this work.

Traditionally, the following three steps are used to perform polyvariant program specialization irrespective of the language [39, 42].

1. Obtain a description of all computation states reachable during program execution with the available variable values. The variable values may be available as invariant inputs known at specialization time or run-time invariants exposed at specialization time.

2. Restrict these computation states to those reachable from the entry point of the program being specialized. Also incorporate the known data into these states to yield possibly several specialized versions of the program's control points.

3. Optimize the residue further using traditional optimizations to yield a residue.

4.2.2.1 Residue Generation During Partial Execution

Partial execution of BAM code outlined in the previous section may be viewed as involving a series of transitions from one computation state to another. A state

transition occurs due to partial execution of one instruction. The BAM registers and memory built with (possibly) partial datawords, constitutes the computation state. PE of an instruction transitions a given computation state to a new one. The two states, viz., one transitioned from and the one transitioned to are abstract representations of the corresponding states resulting from execution of the instruction. The run-time computation state data values are abstracted to their data tags in the corresponding PE-time computation state. The cumulative state transitions due to partial execution of instructions in a basic block are referred to as *block state* transitions. Correspondingly the cumulative state transitions due to partial execution of instructions immediately after a `call/1` and a corresponding `return/0` instruction are referred to as *procedure state* transitions. Instructions to which execution control transfers are usually referred to as *targets*. Leaders of all basic blocks are considered as program points during partial execution of BAM code. The computation state at the program point is the memory state of BAM. Thus, PE of the basic block at a given program point is performed in the context of its memory state. This decision to perform the PE of a block or not, is made at PE-time based on the criteria detailed in Chapter 6.

Partial execution of basic block instructions in the context of the current memory state and their transformation to simpler instructions whenever possible results in a residue. Residues of all basic blocks of the CFG are thus generated for all the program states possible at all run-time entries of the blocks. PE of a basic block is performed if its residue for the current memory state was not generated earlier. Additionally, as a procedure entry may also be a program point, PE of the procedure entry basic block is performed if the residue of the procedure for the current memory state was not generated earlier.

As mentioned earlier, the results of PE of each basic block are recorded. To check whether PE of a block was performed earlier for a given memory state, the memory state and the resulting residue need to be recorded after each PE of a block. This means that a block is "parameterized" by a memory state. It may be noted that the granularity of data accesses by most of the BAM instructions is a dataword.

All such accesses are done either via argument registers or permanent registers. Further the architecture specification assumes that all instruction operands to be dereferenced, except those of the instruction **deref/2** [67]. Thus execution of the instructions in a basic block are characterized by contents and accesses made by the registers in the block. In other words, the memory state at the end of the PE of a basic block reflects changes due PE of the block instructions only while the rest of the memory areas remain the same. Consequently, a block need not be parameterized by a memory state it executes in. Instead, it is sufficient to parameterize the block with the argument and permanent registers live at that program point. Such registers are referred to as *reference registers*. Similarly, to answer the question "Is a residue generated for the block at current program point given the context of current memory state?", it is sufficient to check if PE of the block was performed for the current reference register contents instead of the whole memory state. Additional analysis of instructions with larger data access granularity allows augmenting reference registers with other instruction operands, if needed, to parameterize the block. This is illustrated in Section 4.4.2.

The above discussion provides a background for various issues that need be addressed while performing partial execution of BAM code. We now outline an algorithm to perform partial execution of the CFG by iterative depth-first traversal in Algorithm 2. The goal is to provide details of the process of partial execution of BAM code using the algorithm.

Algorithm 2 Empirical Partial Execution Algorithm

1: Perform depth-first traversal of the CFG.
2: **for all** basic blocks *bb* of the CFG **do**
3: **if** no residue for *bb* with respect to current memory state exists **then**
4: Partial Execute *bb* to get a residue, bb_{res}.
5: **else**
6: Let bb_{res} be the residue of *bb*.
7: **end if**
8: Record the current control flow path from parent of *bb* to bb_{res} for code generation.
9: **end for**

The test for a previous partial execution of a basic block and a resulting residue is done at Step 3 of the algorithm. This is referred to as *version check*. As explained earlier, since a basic block may be parameterized with reference registers, the version check tests if a residue for the block was previously generated for static reference registers. This test ensures that code explosion and non-termination do not occur. The version check is described in Chapter 6.

The amount of residue generated during the execution of this partial execution algorithm depends on the size of static value domain of the reference registers at PE-time. We show that this domain is indeed finitely small within the PE framework described above.

There are a finite number of registers in a given basic block. At PE-time, each of these registers may contain dataword whose datatag is one of the finite set {`tvar`, `tint`, `tpos`, `tflt`, `tstr`, `tneg`, `tatm`, `tlst`}. Thus even if one residue for each of the registers with each of these datatags were generated, only a finite number of residues for a block are generated. Consequently, the number of residues generated during the whole process of partial execution is finite and the partial execution process is will terminate. An algorithm to optimize the number of reference registers used for version check is discussed in Chapter 5. Further the reasons for termination given here are in concurrence with *bounded static variable* conditions laid out by Glenstrup and Jones [30] and Holst [38].

4.2.2.2 Consolidation of Residue

The BAM code partial execution outlined above generates residues of the basic blocks along all possible run-time CFG paths. All such run-time paths traversed during partial execution are recorded. The residues generated are consolidated to yield an optimized version of the BAM code on which PE was performed. This is done after the completion of partial execution. It involves adjustment of control flow targets to residues instead of the original basic blocks. Further, trivial transition eliminations such as removal of unconditional jumps to a following instruction are also affected. Issues relating to the formation of a run-time path during partial

execution, introducing a residue into the path, code generation corresponding to a given run-time path and related optimizations done are discussed in detail in Chapter 7.

4.3 Specialization of Instructions

Transforming instructions of a basic block during its partial execution results in a residue. If two different instructions I_1 and I_2 transform a given memory state, M, to the same memory state M', then the instructions are transformable to one another in the code in the context of the memory state **M**. If the transformed instructions execute in lesser number of cycles than the original instructions, then the residue may be expected to execute more efficiently. The most important criterion for any transformation is to ensure that the program output remains the same as that of the original. Hence any possible instruction transformation should be ensured to be correct. The memory state transformations done by all BAM instructions are specified by the denotational semantics of BAM (Chapter 3). Using the denotational semantics specification, the correctness of all instruction transformations done during partial execution is shown in Section 4.3.

As a first step towards showing correctness of instruction transformations, the following classification of registers that occur in a basic block is performed.

If a register content has known datatags or a known dataword (i.e., both datatag and datavalue) at partial execution time, then the register is termed as *static* register. A static register is called either *tag-static* or *data-static* to signify the knowledge of tag or complete datavalues (i.e., datatag and datavalue). Registers whose tag values or dataword values are known only at execution time are called as *dynamic* registers.

As specified in the previous section, an instruction or a sequence of instructions maps (or transforms) a memory state to another. Let $p \in$ **ProgState** denote a memory state during the execution of a sequence of BAM instructions. Let `instr` be a sequence of one or more instructions to be executed next. Let `instr` map p to a memory state $p' \in$ **ProgState**. If a sequence of one or more instructions `instr'` also

maps p to p' then `instr` and `instr'` are termed *equivalent instructions*. Thus the sequence `instr` may be replaced with `instr'` in a code fragment containing `instr` provided the program state is p prior to the execution of `instr'`.

We now present instructions and their equivalents specialized for partially known memory/operand contents represented by the current program state p. We show the equivalence of the instructions and their specialized versions within the denotational semantics framework presented above. Thus a foundation is laid out for showing that the partial execution methodology presented is correct.

We introduce a no-operation instruction `nop` similar to that found in several processor architectures. The execution of this instruction has no effect on the memory state except incrementing the program counter `pc`. Its semantics are

$$\mathcal{I}[\![\mathbf{nop}]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
$$\text{where } mem = (s, h, se, sc, tr),$$
$$mem' = (s', h, se, sc, tr),$$
$$s' = (incr\ \text{in}\mathbf{R}_s(\mathbf{pc})\ 1\ s)$$

Correctness is shown for only those instructions that present an opportunity to be transformed.

4.3.1 Specialization of Conditional Control Flow Instructions

As shown below, it is often possible to use the tag information of a register value to eliminate redundant tests and reduce the conditional control flow instructions to unconditional jumps or eliminate them altogether.

Consider the valuation function of `switch/5` instruction given in Section 3.5.2. This three-way branch instruction depends on the tag information of the addressable entity x available from program state $p = (code, mem, in, out)$. The value of x may either be static or dynamic. In case of a static x, the equivalent instructions for the three cases are as follows:

Case 1. tag of x known to be `tvar`.

This implies that the condition (**tvar** == *gettag* (*getvalue* x *mem*)) is true and the semantics reduce to

$$\mathcal{I}[\![\texttt{switch}(t, x, l1, l2, l3)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$

where $mem = (s, h, se, sc, tr)$,

$t \in \mathbf{T}$, $x \in$ **Adrable** and $l1, l2, l3 \in$ **String**

$mem' = (s', h, se, sc, tr)$,

$s' = (setregvalue\ \text{in}\mathbf{R}_s(\text{pc})\ (fetchaddr\ \texttt{label}(l1)\ code)\ s)$

The semantics of the instruction `jump`($l1$) are exactly the same as above. Thus the `switch/5` instruction is equivalent to `jump/1` instruction. This unconditional jump is a simpler instruction involving no tag comparison unlike the original `switch/5` instruction.

Case 2. tag of x is as specified by t.

This implies that the condition (t = *gettag* (*getvalue* x *mem*)) is true and the semantics reduce to

$$\mathcal{I}[\![\texttt{switch}(t, x, l1, l2, l3)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$

where $mem = (s, h, se, sc, tr)$,

$t \in \mathbf{T}$, $x \in$ **Adrable** and $l1, l2, l3 \in$ **String**

$mem' = (s', h, se, sc, tr)$,

$s' = (setregvalue\ \text{in}\mathbf{R}_s(\text{pc})\ (fetchaddr\ \texttt{label}(l2)\ code)\ s)$.

This is equivalent to the semantics of the instruction `jump`($l2$).

Case 3. tag of x is neither **tvar** nor as specified by t. The semantics reduce to

$$\mathcal{I}[\![\texttt{switch}(t, x, l1, l2, l3)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$

where $mem = (s, h, se, sc, tr)$,

$t \in \mathbf{T}$, $x \in$ **Adrable** and $l1, l2, l3 \in$ **String**

$mem' = (s', h, se, sc, tr)$,

$s' = (setregvalue\ \text{in}\mathbf{R}_s(\text{pc})\ (fetchaddr\ \texttt{label}(l3)\ code)\ s)$

This is equivalent to the semantics of the instruction `jump`($l3$).

In each case, the evaluation function reduces to a `jump/1` to the appropriate label. Thus with a tag-static value `switch/5` instruction is equivalent to `jump/1` instruction. In case of a dynamic addressable entity, the `switch/5` instruction remains unchanged.

Next consider the valuation function for `test/4` instruction given in Section 3.5.2. This is a two-way branch instruction – depending on equality or inequality as specified by `eq` or `ne` – of the tag of x and a given tag t. If the addressable entity x is static, the `test/4` instruction may be simplified as follows.

Case 1. If the tag of x is the same as t and c is specified as `eq` the semantics of `test/4` reduce to

$$\mathcal{I}[\![\texttt{test}(c,t,x,l)]\!]((code,mem,in,out)) = ((code,mem',in,out))$$
$$\text{where } mem = (s,h,se,sc,tr),$$
$$mem' = (s',h,se,sc,tr),$$
$$s' = setregvalue\ \text{in}\mathbf{R}_s(\texttt{pc})\ (fetchaddr\ \texttt{label}(l)\ code)\ s$$

This is equivalent to the semantics of the instruction `jump`(l).

Case 2. If the tag of x is the same as t and c is specified as `ne` the semantics of `test/4` reduce to

$$\mathcal{I}[\![\texttt{test}(c,t,x,l)]\!]((code,mem,in,out)) = ((code,mem',in,out))$$
$$\text{where } mem = (s,h,se,sc,tr),$$
$$mem' = (s',h,se,sc,tr),$$
$$s' = incr\ \text{in}\mathbf{R}_s(\texttt{pc})\ 1\ s))$$

This is exactly the same semantics as those of the `nop` instruction indicating that the execution of `text/4` in the current program state is redundant and thus may be replaced with a `nop`.

Case 3. If the tag of x is different from t and c is specified as `eq` the semantics of `test/4` reduce to that of `nop`.

```
procedure(pred/2).
        ...Argument dereferencing instructions...
        choice(1/4,ArgRegLst1,l(pred/2,2)).
        ...Head unifications and body instructions...
        return.
label(l(pred/2,2)).
        choice(2/4,ArgRegLst2,l(pred/2,3)).
        ...Head unifications and body instructions...
        return.
label(l(pred/2,3)).
        choice(3/4,ArgRegLst3,l(pred/2,4)).
        ...Head unifications and body instructions...
        return.
label(l(pred/2,4)).
        choice(4/4,ArgRegLst4,fail).
        ...Head unifications and body instructions...
        jump/1.
```

Figure 4.4: Schematic of choicepoint creation in BAM

Case 4. If the tag of x is different from t and c is specified as `ne` the semantics of `test/4` reduce to that of `jump`(l).

Similarly, it may be shown that the semantics of `jump/5` reduce to those of `jump/1` or to `nop` with data-static values of the operands x and y.

4.3.2 Specialization of Choicepoint Instructions

Specializing choicepoint instructions by partial execution may reduce (or even eliminate) the number of choicepoints created at execution time. Here we discuss the methodology for specializing choicepoint instructions that will serve as a background for the choicepoint optimization detailed later.

First, a brief review of how choicepoint creation code is generated by the Aquarius Prolog compiler is in order. Suppose a predicate `pred/2` consists of four clauses and each of these clauses is compiled to BAM code with labels, say, `l(pred/2,2)`, `l(pred/2,3)`, and `l(pred/2,4)` respectively, as shown in Figure 4.4. A schematic

of this BAM code partitioned as CFG is shown in Figure 4.5.

The `choice/3` instruction in basic block **A** creates a choicepoint which contains the address of next alternative to be tried, i.e., address of the instruction `label(1(pred/2,2))`, along with the argument register values given in **ArgRegLst1**. First, the control flow proceeds along path $\mathbf{P_A}$. If the execution fails, the control flow enters block **B** and the `choice/3` instruction in **B** restores the argument registers listed in **ArgRegLst2** from the choicepoint. It also updates the next alternative to

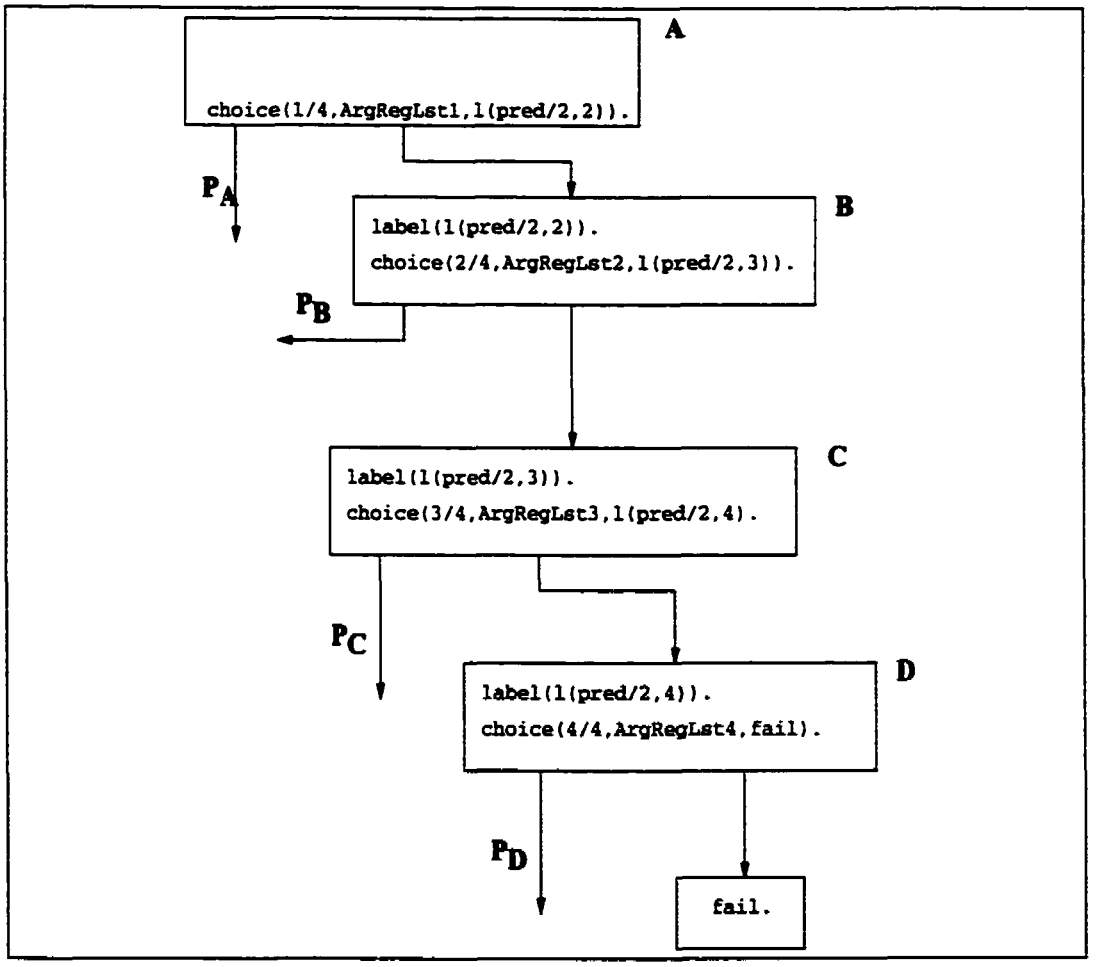

Figure 4.5: Schematic CFG of a predicate **pred/2** with four alternate choices

try with the address of the instruction `label(1(pred/2,3))`. Execution proceeds along the edge $\mathbf{P_B}$. The rest of the choices of **pred/2** are attempted similarly if the current choice is a failure. The `choice/3` instruction in the last alternative **D** restores the argument registers to those listed in **ArgRegLst4** from the choicepoint

and pops the choicepoint off the choicepoint stack. If the last alternative fails, the call pred/2 fails.

Given the static register information at the entry of procedure(pred/2), suppose partial execution reveals that execution along the path $\mathbf{P_C}$ fails. Then the choicepoint instruction in the code block with label 1(pred/2,2) may be transformed to bypass block with label 1(pred/2,3). In other words, choice(2/4, ArgRegLst2, 1(pred/2,3)) in block **B**, can be transformed to choice(2/4, ArgRegLst3, 1(pred/2,4)). The correctness of this transformation may be shown using the denotational semantics of choice/3 and fail/0 instructions as follows.

Let mem be the memory state at the time of entry into block **A** and $\mathtt{b_a}$ and $\mathtt{h_a}$ be the values of b and h. The memory state at the exit of block **A** results from application of valuation function of choice($1/n, r, l$) specified in section 3.5.2 to the memory state mem. The register hb contains $\mathtt{h_a}$ at the exit of **A**. Let $\mathtt{b_b}$ be the value of b after the choicepoint is created by the instruction choice(1/4,ArgLst1,1(pred/2,2)). Let $\mathtt{h_b}$, $\mathtt{cp_b}$, $\mathtt{e_b}$ and $\mathtt{tr_b}$ be the values of h, cp, e and tr respectively at the exit of block **A**. Hence (*hd sc*), which represents the top of choicepoint stack, is ($\mathtt{b_b}$, (ArgRegLst1, $\mathtt{h_b}$, $\mathtt{tr_b}$, $\mathtt{e_b}$, $\mathtt{cp_b}$, $\mathtt{b_b}$, addr_pred_2_2) where addr_pred_2_2 = *fetchaddr* 1(pred/2,2)) *code* at the entry of block **B**.

The top of choicepoint stack at the exit of **B** i.e., entry of block **C** is

($\mathtt{b_b}$, ((ArgRegLst1), $\mathtt{h_b}$, $\mathtt{tr_b}$, $\mathtt{e_b}$, $\mathtt{cp_b}$, $\mathtt{b_a}$, addr_pred_2_3),

where addr_pred_2_3) = *fetchaddr* 1(pred/2,3) *code* and (ArgRegLst1) represents the contents of the argument registers in ArgRegLst1. Further, the registers listed in ArgRegLst2 have the values restored from ArgRegLst1 that were stored in the choicepoint. The register pc has the address of the instruction following the choice/3 instruction in block **B**. Now, suppose it was revealed that partial execution executes fail/0 along the each of the paths on $\mathbf{P_C}$. This implies that execution along the path $\mathbf{P_C}$ fails and partial execution of fail instruction restores the memory state to the correct state. Let mem_b be the memory state before the execution of this fail instruction. State transformation of memory due to execution of fail may be computed by applying its valuation function ($code, mem_b, in, out$), where

code $\in$ **ProcCode** is the BAM code on which partial execution is being performed, *in* and *out* are its input and output.

$$\mathcal{I}[\![\texttt{fail}]\!]((code, mem_b, in, out)) = ((code, mem_c, in, out))$$

where $mem_b = (s, h, se, sc, tr)$,

$mem_c = (s', h, se, sc, tr')$

$s_1 = restoreregs\ \mathtt{tr_b}\ tr\ s$,

$tr' = trsuffix\ \mathtt{tr_b}\ tr$,

$s_2 = setregvalue\ \mathrm{in}\mathbf{R}_s(\mathtt{e})\ \mathtt{e_b}\ s_1$,

$s_3 = setregvalue\ \mathrm{in}\mathbf{R}_s(\mathtt{cp})\ \mathtt{cp_b}\ s_2$,

$s_4 = setregvalue\ \mathrm{in}\mathbf{R}_s(\mathtt{h})\ \mathtt{h_a}\ s_3$,

$s' = setregvalue\ \mathrm{in}\mathbf{R}_s(\mathtt{pc})\ \mathtt{addr\_pred\_2\_3}\ s_4$

Evidently, the valuation function of `fail` maps mem_b to mem_c wherein the state registers e, tr, cp are restored from the top of the choicepoint stack.

Now partial execution proceeds to block **C** wherein PE of instruction `choice(3/4, ArgRegLst3, 1(pred/2,4))` transforms $\mathtt{mem_c}$ as specified by the following valuation function:

$$\mathcal{I}[\![\texttt{choice(3/4, ArgRegLst3, 1(pred/2, 4))}]\!]((code, mem_c, in, out)) = ((code, mem_d, in, out))$$

where $mem_c = (s, h, se, sc, tr)$,

$mem_d = (s'', h, se, sc', tr)$,

$sc' = cons\ ((\mathtt{ArgRegLst1}), \mathtt{h_b}, \mathtt{tr_b}, \mathtt{e_b}, \mathtt{cp_b}, \mathtt{b_a}, \mathtt{addr\_pred\_2\_4})\ (tl\ sc)$,

$s' = loadregs\ r\ (\mathtt{ArgRegLst1})\ s$,

$s'' = incr\ \mathrm{in}\mathbf{R}_s(\mathtt{pc})\ 1\ s'$

Knowing that partial execution fails along $\mathbf{P_C}$ fails, given mem_b, the instruction `choice(2/4, ArgRegLst2,1(pred/2,3))` may be transformed to `choice(2/4, ArgRegLst3,1(pred/2,4))` as explained above. The memory state transformation of mem_b done by this transformed instruction is as follows:

$$\mathcal{I}[\![\texttt{choice(2/3, ArgRegLst3, 1(pred/2, 4))}]\!]((code, mem_b, in, out)) =$$

$$((code, mem', in, out))$$

where $mem_b = (s, h, se, sc, tr)$,

$mem' = (s'', h, se, sc', tr)$,

$sc' = cons\ ((\texttt{ArgregLst1}), \mathtt{h_b}, \mathtt{tr_b}, \mathtt{e_b}, \mathtt{cp_b}, \mathtt{b_a}, \texttt{addr\_pred\_2\_4})\ (tl\ sc)$,

$s' = loadregs\ r\ rl\ s$,

$s'' = incr\ \mathrm{in}\mathbf{R}_a(\mathtt{pc})\ 1\ s'$

mem' is exactly the same as mem_d. Hence the transformation is correct. This transformation renders block **C** to be dead-code. This type of choicepoint specialization may be performed if the fall-through path following a choicepoint update instruction may be shown to fail during partial execution of the BAM code. The implementation of the associated analysis is detailed in Section 6.4.

Two special cases of this optimization occur. The first is when all alternatives to the first choice can be shown to fail at PE time but the first choice can not be shown to fail. In Figure 4.5, it may be shown that partial execution fails along $\mathbf{P_B}$, $\mathbf{P_C}$ and $\mathbf{P_D}$, but not along $\mathbf{P_A}$. In such a case the choicepoint creation itself may be inhibited as shown in Section 4.4.2. The second case, when it may be shown that partial execution fails along $\mathbf{P_A}$, an entirely different kind of choicepoint transformation is done (Section 6.4). The correctness of these transformations follows from the proof of the general case given above.

The basic idea behind choicepoint optimization is to avoid creation and/or manipulation of a choicepoint with retry addresses that fail. Choicepoint optimization can save the time of manipulation as well as attempting to execute code on paths known to fail. Also in programs that can generate exponential number of choicepoints, this optimization can result in reducing potential swapping problems.

4.3.3 Specialization of Unification Instructions

Using static registers, some unification instructions may be specialized as follows. The semantics of `deref/2` instruction are specified in Section 3.5.3. The follow-

ing possibilities exit for specializing this instruction depending on the compile-time information available.

Case 1. x is tag-static, x and y refer to the same register and x has a tag other than `tvar`.

The semantics of `deref/2` reduce to

$$\mathcal{I}[\![\texttt{deref}(x,x)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
$$\text{where } (s, h, se, sc, tr) = setvalue\ x\ ((getvalue\ x\ mem))\ mem,$$
$$mem' = (s', h, se, sc, tr),$$
$$s' = incr\ \text{in}\mathbf{R}_s(\text{pc})\ 1\ s$$

since the condition (*gettag (getvalue x m)* == `tvar`) in the *deref* operator evaluates to false thus evaluating *deref (getvalue x mem) mem* to *(getvalue x mem)* which in turn, evaluates to the contents of x which are re-mapped to x by the expression *setvalue x ((getvalue x mem)) mem*. In other words, the content of x is re-mapped to itself – a vacuous operation. Thus the semantics reduce to

$$\mathcal{I}[\![\texttt{deref}(x,x)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$
$$\text{where } mem = (s, h, se, sc, tr)$$
$$mem' = (s', h, se, sc, tr),$$
$$s' = incr\ \text{in}\mathbf{R}_s(\text{pc})\ 1\ s$$

These semantics are equivalent to those of **nop** instruction. In other words, `deref/2` instruction can be replaced by the **nop** instruction in this case.

Case 2. x is tag-static, x and y are different registers and x has a tag other than `tvar`.

As above, (*deref* (*getvalue x mem*) *mem*) in the specification of `deref/2` reduces to (*getvalue x mem*). The expression

setvalue y (*deref* (*getvalue x mem*) *mem*) *mem*

reduces to *setvalue y* (*getvalue x mem*) *mem*. Thus the semantics of `deref/2` reduce to

$$\mathcal{I}[\![\texttt{deref}(x,y)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$

where $(s, h, se, sc, tr) = setvalue\ y\ ((getvalue\ x\ mem))\ mem$,

$mem' = (s', h, se, sc, tr)$,

$s' = incr\ \text{in}\mathbf{R}_s(\texttt{pc})\ 1\ s$

which may be rewritten as

$$\mathcal{I}[\![\texttt{deref}(x,y)]\!]((code, mem, in, out)) = ((code, mem'', in, out))$$

where $mem'' = ((incr\ \text{in}\mathbf{R}_s(\texttt{pc})\ 1\ mem'\downarrow 1), mem'\downarrow 2, mem'\downarrow 3,$

$mem'\downarrow 4, mem'\downarrow 5)$,

$mem' = setvalue\ y\ (getvalue\ x\ mem)\ mem.$

This is exactly the same as the semantics of the **move/2** instruction. Thus the **deref/2** instruction is equivalent to **move/2** given the above static memory information.

Case 3. In all other cases, the instruction **deref/2** may not be further specialized at compile time.

■

Next consider the semantics of the **unify/5** instruction as given in Section 3.5.3. The following specialization options exist based on the static memory information available.

Case 1. x and y are tag-static and have tag values other than **tvar**.

If the tag values of x and y are different, the **unify/5** instruction is reduced to **jump**(l) as the condition (mem ==$\perp$) in the semantics specification of **unify/5** holds in this situation.

Case 2. x and y are tag-static, and x has a tag other than **tvar** and y a tag **tvar**, the **unify/5** instruction is equivalent to the sequence of instructions

trail(y).

move(x, y).

The equivalence is shown as follows.

Given the above static information, a `unify/5` instruction maps a program state $(code, mem, in, out) \in$ **ProgState** to a new program state $(code, mem'', in, out)$

where $mem = (s, h, se, sc, tr)$,

$mem' = (s', h, se, sc, tr')$

$s' = incr\ \mathbf{inR_s(tr)}\ 1\ (setvalue\ y\ (getvalue\ x\ mem)\ mem)\downarrow 1$,

$tr' = cons\ ((getregvalue\ tr\ s) + 1, (y, (getvalue\ y\ mem)))\ tr$,

$mem'' = ((incr\ \mathbf{pc}\ 1\ s'), h, se, sc, tr')$

by applying appropriate simplifications to the semantic specification of `unify/5`.

Given the same static information, a sequence of `trail/1` and `move/2` instructions map the program state $(code, mem, in, out) \in$ **ProgState** to a new program state $(code, mem'', in, out)$ as follows:

The `trail`(y) instruction maps to $(code, mem^t, in, out)$

where $mem_t = (s_t, h, se, sc, tr_t)$

$s_t = incr\ \mathbf{inR_s(pc)}\ 1\ (incr\ \mathbf{inR_s(tr)}\ 1\ s)$,

$tr_t = cons\ ((getregval\ \mathbf{inR_s(tr)}\ s){+}1,\ (y,\ getvalue\ y\ mem))\ tr$.

The `move`(x, y) instruction then maps $(code, mem_t, in, out)$ to $(code, mem_m, in, out)$

where $mem_m = (s_m, h, se, sc, tr_t)$, and

$s_m = incr\ \mathbf{inR_s(pc)}\ 1\ (setvalue\ y\ (getvalue\ x\ mem_t)\ mem_t)\downarrow 1$

Neither the heap(h), the environment stack(se), nor the choicepoint stack(sc) are changed during the partial execution of `unify/5` or the sequence `trail/1`, `move/2`. Further in both cases $y \in$ **Adrable** and the trail stack are updated in the same manner. Since the instruction sequence consists of two instructions, the pc register is updated twice. The instruction `unify`$(x, y, t1, t1, l)$ and the sequence

```
trail(y).
move(x, y).
```

perform exactly same memory updates except for the value of pc. The value of pc in both the cases is equivalent as it points to the instruction following original `unify/5` instruction. Thus the transformation may be shown to be correct for the given static information.

A similar transformation may be shown to hold if x has a static `tvar` tag and y either has a static `tvar` tag with the value part that is lesser than that of x or has a static tag other than `tvar`. The `unify/5` in either of these cases is semantically equivalent to

`trail(`x`).`

`move(`y, x`).`

Case 3. x and y are data-static with non-pointer datatags.

If x and y are equal, the instruction may be reduced to `nop/0`. Otherwise, it may be reduced to `jump(`l`)`.

The condition

$$((tx == \texttt{tint}) \bigvee (tx == \texttt{tflt}) \bigvee (tx == \texttt{tatm}))$$

in the *unify* operator is true given the static information, viz., x and y are non-pointer datatags.

(a) If the data value is not the same:

The semantics of `unify/5` reduces to $\perp$. This means that the condition ($mem == \perp$) holds in the semantics of `unify/5` and (*fetchaddr* `label(fail)` *code*) evaluates to the execution of `fail/0` instruction as explained earlier. Thus in this case the `unify/5` instruction may be replaced with `fail/0` without any change in execution semantics.

(b) If the data value is the same:

control falls through to the next instruction and hence the `unify/5` instruction may be replaced by `nop/0`.

■

The equal/3 instruction whose semantics are given in Section 3.5.3 may be specialized using static register information as follows.

Case 1. Let x and y be tag-static. If their tag values are not the same, the semantics of equal/3 reduce to

$$\mathcal{I}[\![\texttt{equal}(x, y, l)]\!]((code, mem, in, out)) = ((code, mem', in, out))$$

where $mem = (s, h, se, sc, tr)$,

$mem' = (s', h, se, sc, tr)$,

$s' = setregvalue\ \mathbf{in}\mathbf{R}_s(\texttt{pc})\ (fetchaddr\ \texttt{label}(l)\ code)\ s$

since the comparison *(getvalue x mem == getvalue y mem)* fails as the datatags of x and y are unequal. The above semantics are the same as the semantics of jump(l). Thus equal/3 is equivalent to jump/1 given the above static memory information.

Case 2. Let x and y be data-static registers with same tag values. equal/3 reduces to to jump(l) if the data values of x and y are equal. Else it reduces to nop.

With this background, the partial execution process and instruction transformation are illustrated in the following section.

4.4 Illustration of Partial Execution

4.4.1 Example 1

The following example illustrates partial execution outlined in Algorithm 2. Figure 4.6 shows a simple Prolog program and the BAM code generated during Aquarius compilation using global flow analysis. CFG representation of the BAM code is shown in Figure 4.7.

Partial execution of the BAM code may be viewed as application of the valuation function $\mathcal{B}$ to the BAM code stream in Figure 4.6. Associated instruction transformations are also illustrated in the example below. For easy reference, BAM instructions are annotated in the code with numbers.

```
:- option(analyze).

main :- num(X),p(X,Y), write(Y).

p(2,Y) :- Y = 10.
p(3,Y) :- Y = 20.

num(3).
```

```
% Aquarius Prolog compiler
% Copyright (C) 1989-92 Peter Van Roy
% All rights reserved.
% Creation date Wed Aug 12 19:31:38 PDT 1992

% Modes generated:
% mode(main,true,true,true,n)
% mode(num(A),uninit_reg(A),true,(ground(A),rderef(A)),n)
% mode(p(A,B),uninit_reg(B),(ground(A),rderef(A)),(ground(B),
%                            ground(A),rderef(B),rderef(A)),n)
% mode('$ init_main/0',true,true,true,n)


0: procedure(main/0).
1:      entry(main/0,0).
2:      allocate(0).
3:      call(num/1).
4:      call(p/2).
5:      move(r(1),r(0)).
6:      deallocate(0).
7:      jump(write/1).

8: procedure(num/1).
9:      entry(num/1,1).
10:     move(tint^3,r(0)).
11:     return.

12: procedure(p/2).
13:     entry(p/2,2).
14:     test(ne,tint,r(0),fail).
15:     equal(r(0),tint^2,l(p/2,2)).
16:     move(tint^10,r(1)).
17:     return.
18: label(l(p/2,2)).
19:     equal(r(0),tint^3,fail).
20:     move(tint^20,r(1)).
21:     return.

22: procedure('$ init_main/0'/0).
23:     entry('$ init_main/0'/0,0).
24:     return.
```

Figure 4.6: Program example1.pl along with its BAM code

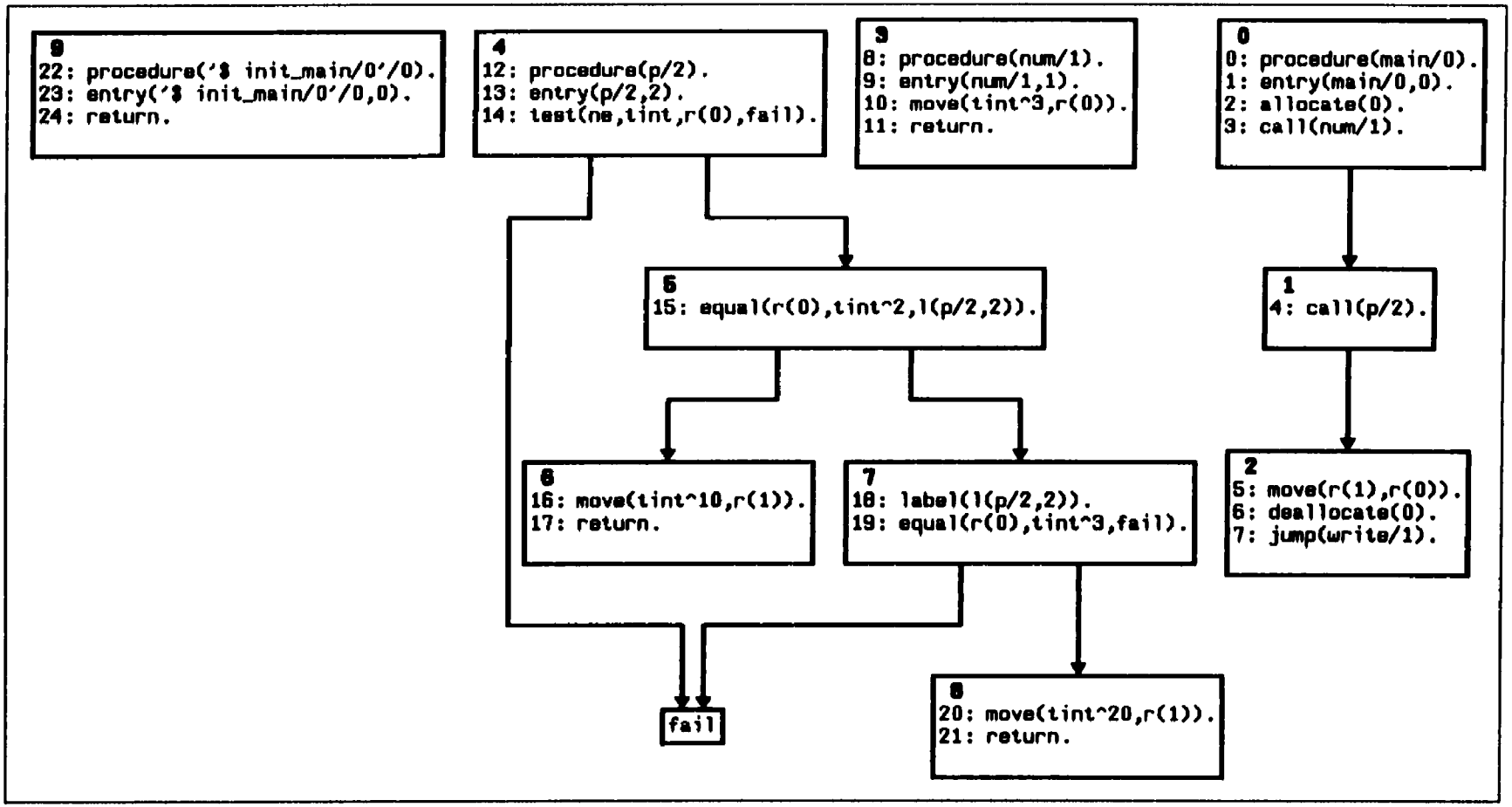

Figure 4.7: CFG of BAM code of `example1.pl` program

Block 9 consists of code that identifies the entry point for the assembler and linker. It is not used during the PE process but is retained. In subsequent examples, it is assumed to exist and is not explicitly shown in the CFG of BAM code.

Block 0 is the entry point of the CFG in Figure 4.7. Execution of instruction 3 and consequently of basic block 3 loads register `r(0)` with the dataword `tint^3`. Execution continues to block 1. Execution of the call instruction 4 transfers control to block 4. At the entry of block 4, `r(0)` is a static register. In the context of current program state, the flow change instruction 14 may be replaced with a `nop` since tag of the static register `r(0)` is known to be `tint` as shown in Section 4.3.1. This results in a specialized basic block 4′ with the following instructions.

```
12: procedure(p/2).
13: entry(p/2,2).
14: nop.
```

Control flow then falls through to block 5.

In block 5, since the value of `r(0)` in current memory state is `tint^3`, the equality test of instruction 15 fails. Thus the `equal/3` instruction is equivalent to `jump(1(p/2,2))` as shown in Section 4.3.1. This results in a specialized basic block 5′ with the following instruction.

```
15:       jump(1(p/2,2)).
```

The control flows to block 7 in which instruction 19 is equivalent to a `nop` as the data-static register `r(0)` is equal to `tint^3`. This results in the following specialized basic block 7′.

```
18: label(1(p/2,2)).
19: nop.
```

The symbolic execution continues to block 8 where `tint^20` is loaded into `r(1)`. The control then flows to block 2 where instruction 5 with data-static `r(1)` may be transformed to `move(tint^20,r(0))` resulting in the following specialized block 2′.

```
0: procedure(main/0).
1:       entry(main/0,0).
2:       allocate(0).
3:       call(num/1).
4:       call(p/2).
5:       move(tint^20,r(0)).
6:       deallocate(0).
7:       jump(write/1).

8: procedure(num/1).
9:       entry(num/1,1).
10:      move(tint^3,r(0)).
11:      return.

12: procedure(p/2).
13:      entry(p/2,2).
14:
15:
16:
17:
18: label(l(p/2,2)).
19:
20:      move(tint^20,r(1)).
21:      return.

22: procedure('$ init_main/0'/0).
23:      entry('$ init_main/0'/0,0).
24:      return.
```

Figure 4.8: Residue of the BAM code of `example1.pl`

```
5:       move(tint^20,r(0)).
6:       deallocate(0).
7:       jump(write/1).
```

Thus the run-time path of code shown in Figure 4.7 is **0 – 3 – 1 – 4 – 5 – 7 – 8 – 2.** As shown above, the blocks **2**, **4**, **5** and **2** may be specialized and replaced with their corresponding residues.

Figure 4.8 shows the residue resulting after the **nop** instructions and trivial jumps to following instruction are eliminated. Such a step is referred to as *code consolidation*. Instruction numbers of dead blocks and **nop** instructions are left behind to highlight the redundancies.

The `jump(l(p/2,2))` instruction at **15** is eliminated since it is a trivial jump to an immediately following location. This simple example illustrates elimination of three redundant comparisons otherwise performed at run-time, thus contributing to an improvement in its execution time. This program may also be optimized by performing partial evaluation of the Prolog source [58] to achieve a result similar to

that obtained by partial execution of the BAM representation. However the finer granularity of the data structures at the BAM level offer greater opportunities for optimizing BAM code by partial execution than optimizing Prolog code by partial evaluation.

4.4.2 Example 2

```
main :- read(List),
        last(List, Last),
        write(Last).

% last(List, LastElement).
last([Element],Element).
last([_|Rest],Last):- last(Rest,Last).
```

Figure 4.9: Program `last.pl`

A second example illustrates choicepoint optimization by partial execution as explained in Section 4.3.2.

Consider the standard definition of the predicate `last/2` shown in Figure 4.9. Figure 4.10 shows the CFG representation of its BAM code generated by the Aquarius compiler with global flow analysis. It is well known that most WAM-based compilers compile `last/2` to abstract machine code that creates a choicepoint left on the stack upon successful exit of a call to `last/2`. This choicepoint is removed only when control backtracks to `last/2` for alternative solutions and fails. It is the case with Aquarius Prolog compiler as well. The choicepoint created in block **11** is left on the stack upon successful exit via block **12**. By performing partial execution, the choicepoint creation may be inhibited when the predicate call `last(List,LElem)` succeeds with `LElem` instantiated to some non-variable value which is an element of `List`.

Symbolic execution of the instructions in the entry block **0** results in allocation

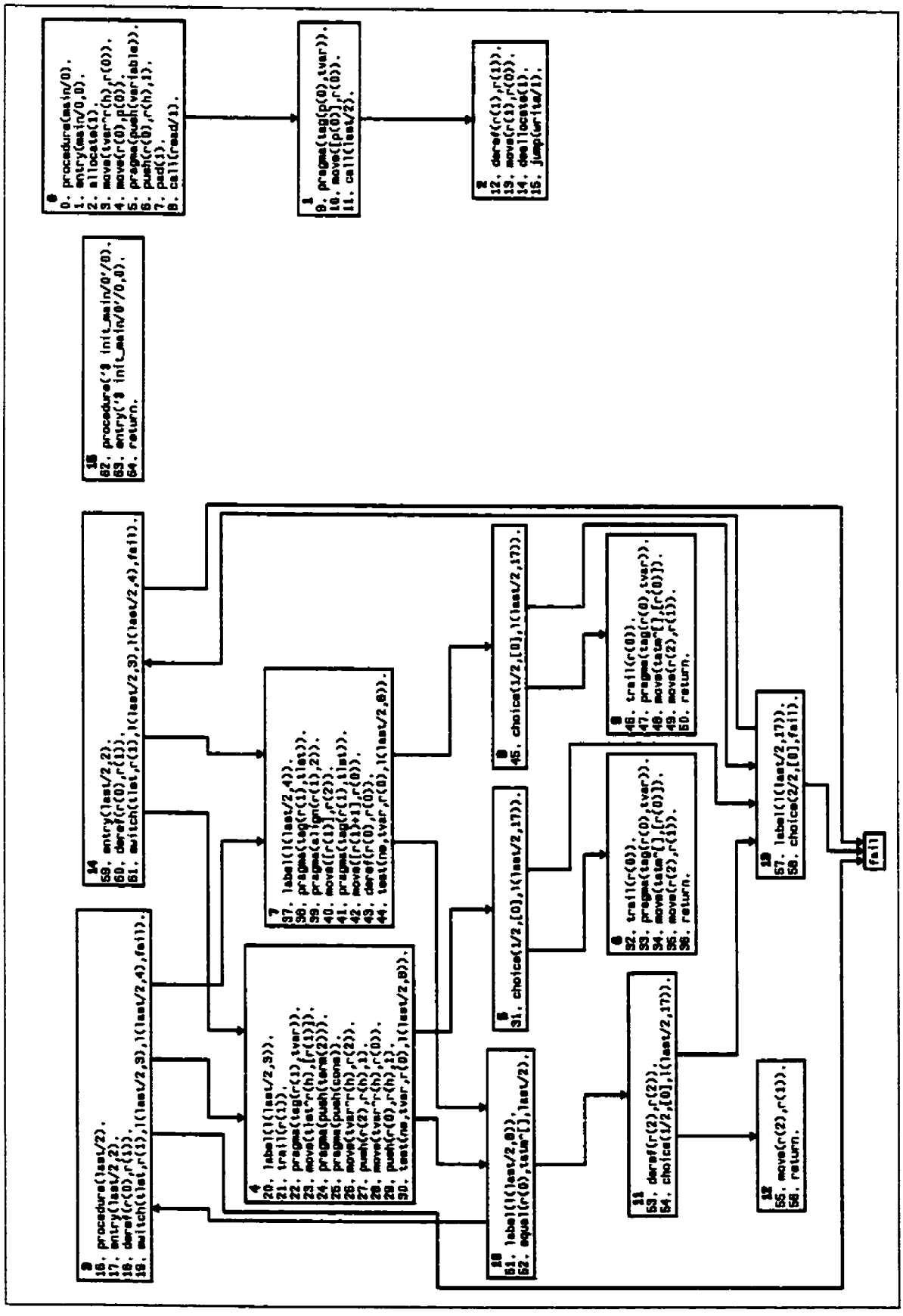

Figure 4.10: CFG for last/2

of an environment on the environment stack, and creation of a variable on the heap. The memory updates resulting from the call to built-in `read/1` are not determinable at PE time. Thus the only information known about `r(0)` and `p(0)` is that their tag values are `tvar`. The value read is built on the heap and `r(0)` holds its address.

Execution of block **1** loads the contents of the address stored in `p(0)` to `r(0)`. Thus `r(0)` and `p(0)` are the reference registers of block **1**. Of these `r(0)` is the dynamic register. A call to `last/2` results in the transfer of control to block **3**.

The reference register set of block **3** is {`r(0)`, r(1)}. Of these `r(0)` is the only register whose values are used in block **3**. At entry time it is not static. Register `r(1)` is dynamic since instruction 18 dereferences `r(0)` to `r(1)`. As the flow change instruction 19 depends on dynamic register `r(1)`, partial execution is to be performed along all possible control flow paths. Choicepoint elimination is illustrated on the path going through block **7**.

Control proceeds on this path when the tag of `r(1)` is `tlst`. Symbolic execution of the block instructions load `r(2)` with the head of the list pointed to by `r(1)`. As the head value is not known at PE time, `r(2)` is dynamic. Similarly, register `r(0)` that has a pointer to the tail of the list, is dynamic. Thus partial execution must be done on both the possible flow directions upon execution of instruction **44**. We focus on the flow that executes the block **10** where tag of `r(0)` is not `tvar`. Since r(0) is dynamic, two possible flow directions exist upon the execution of **52**. Consider the path leading to block **11** when `r(0)` contains `tatm^[]`. Execution of instruction 53 in block **11** dereferences `r(2)` — a dynamic register — into itself and creates a choicepoint which contains the current value of `r(0)` — `tatm^[]`. Execution the flows to block **12** where instruction 55 copies the contents of register `r(1)` to `r(2)`. Control transfers to block **2** where `r(1)` is dereferenced and loaded into `r(0)`. Thus the current execution path is **0-1-3-7-10-11-12-2**. As there is a choicepoint created in this path, it is evident that this is only one of the several possible speculative paths. All the other speculative paths due to the choicepoint have to be traversed for PE to discover the run-time execution path as illustrated in the following.

The alternate path from block **11** goes to block **13**. The `choice/3` instruc-

tion at 58 restores `tatm^[]` to `r(0)` and removes the choicepoint and control flows to block **14**. Of the reference register set {`r(0)`, `r(1)`}, `r(0)` is static. PE of instruction **61** results in failure. As there are no more alternate execution paths in the choicepoint, the program exits with failure. Thus all alternate paths from the choicepoint created in block **11** fail leading to the safe removal of the choicepoint instruction **54** and correspondingly at **58** to yield a complete specialized path $\mathbf{0-1-3-7-10-11'-12-2}$ where block **11'** is

```
53. deref(r(2),r(2)).
54. nop.
```

Partial execution detailed above follows the final loop of the iterative traversal of the call `last(List,LElem)` that succeeds with an instantiation for `LElem` given a list `List` with a non-variable last element and an uninitialized `LElem`.

The memory usage will be reduced by avoiding redundant choicepoints. Less time is spent in manipulation of such choicepoints. Savings corresponding to above optimization in `last.pl` are measured as follows. Since the optimizations affect the execution speed upon first backtracking into the call, a failure driven loop that calls `last([1],L)` 100000 times is used to test the gains. On a lightly loaded SPARCstation 1+ with 16MB main memory the unoptimized executable takes 2.0s of user time whereas the optimized version takes 1.4s of user time – a speedup of 1.4 using Amdahl's law [36]. The user time corresponds to *user* time — time spent in the program itself — as reported by System V version of the command `time`. The memory usage improvement is too small to measure in the present case as one choicepoint occupies only a few bytes. The predicate call with only one element is chosen so that the speed-up resulting from avoiding the creation of the redundant choicepoint is not over shadowed by the rest of the computations in the predicate. The memory usage improvement can potentially be significant in larger programs where more choicepoint optimizations are possible.

It is well known that unnecessary choicepoint creation may be avoided by rewriting `last/2` as shown in Figure 4.11 wherein the first arguments of the two clauses

```
% last(List, LastElement).
last([X|Xs], Last) :- last_2(Xs, Last, X).

last_2([], Last, Last).
last_2([X|Xs], Last, _) :- last_2(Xs, Last, X).
```

Figure 4.11: Program `last-tweaked.pl`

of the predicate definition `last_2/3` (Figure 4.9) represent two mutually exclusive choices. This is evident in its BAM code shown in Figure 4.12. It may be observed that a redundant choicepoint is created in the BAM code of `last/2` when the first two arguments are not mutually exclusive. There is no necessity to transform the "pure" version of `last/2` predicate (Figure 4.9) to avoid redundant choicepoint creation. Such redundant choicepoint creation may be optimized away by partial execution of BAM code of the pure version itself as shown above. The resultant optimized BAM code has the same quality as that of the BAM code of Figure 4.11 with regards to choicepoint creation.

The instruction 30 viz., `test/4` in BAM code shown in Figure 4.10 may be transformed to `nop` since the tag of `r(0)` is known to be `tvar` owing to instruction 28. Although the redundant choicepoint creation was optimized away by re-writing the predicate `last/2`, the `test/4` instruction at 41 in Figure 4.12 is not. Both these optimizations may be achieved by partial execution of the BAM code of the pure version itself. This implies that the programmers need not spend extra time to design predicates with mutually exclusive arguments.

A similar choicepoint optimization may be performed on the code of another standard predicate `min_list/2` that finds the minimum of a given list of numbers.

```
min_list([X],X).
min_list([X|L],M) :-
        min_list(L,Y),
```

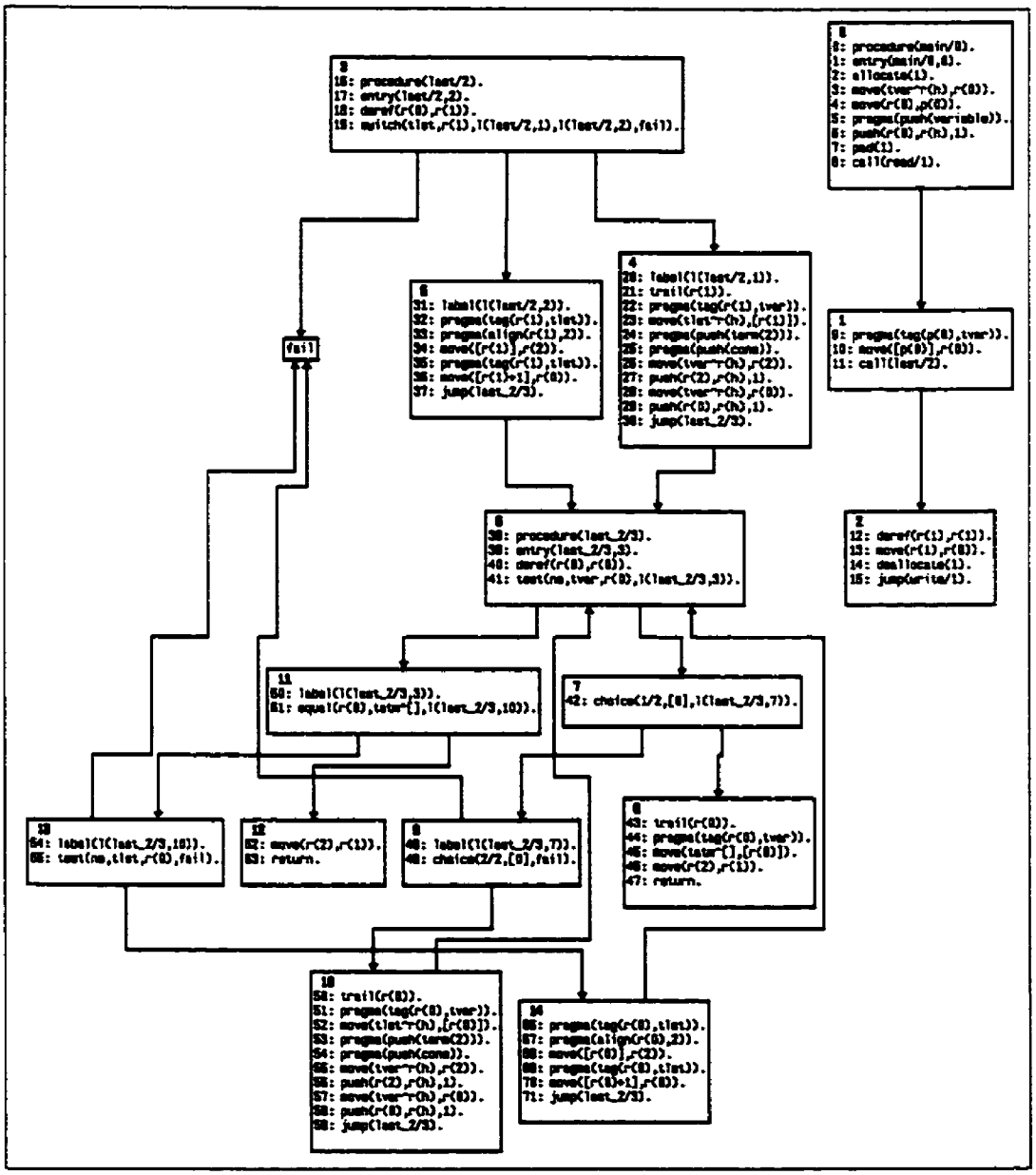

Figure 4.12: CFG for hand-optimized `last/2`

```
        minimum(X,Y,M).

minimum(A,B,A) :- A<B,!.
minimum(A,B,B) :- A>=B.
```

The choicepoint optimization created in the predicate `min_list/2` results in a speedup by a factor of 1.2 on a lightly loaded SPARCstation 1+ with 16MB main memory.

4.5 Summary

In this chapter, a brief introduction to program specialization and associated terminology is presented. Then opportunities for performing BAM code specialization, called partial execution, are discussed. The partial execution frame work is then sketched out by providing a structure to the BAM code. A high-level algorithm to perform PE is presented to provide the starting point of the detailed partial execution process that follows in later chapters. All possible individual transformations are shown to retain semantic correctness using the denotational semantics described in Chapter 3. With this background and introduction to the partial execution process, two complete examples are presented to illustrate the various optimization opportunities.

Any code transformation has to be based on different types of analyses to preserve correctness of the results of the code. Transformations performed during BAM code partial execution also rely on several analyses as well as the results of partial execution itself. Chapter 5 presents BAM code analyses employed in conjunction with PE to affect optimizations.

Chapter 5

Structure of a BAM Partial Executor

5.1 Introduction

The PE-based optimization process designed and implemented in this thesis constantly ensures the correctness of two abstractions – viz., program semantics and the BAM memory abstraction of the program. The program semantics are maintained with the help of an internal representation (IR) of the candidate BAM code. This IR is built with its CFG at the core. As explained in earlier chapters, the process consists of two interleaved phases – viz., symbolic execution of the code and code transformation. The BAM memory model is used to ensure correct symbolic execution. The correctness of code transformation is ensured by maintaining additional information. This additional information pertains both to the candidate BAM code structure and to its run-time behavior. The PE-based optimization process consists of three components: a front end, a partial execution driver (PE driver) and a code consolidator. In the front end, BAM code is partitioned and syntactically analyzed. Code partitioning was described in Section 4.2.1. Since BAM code parsing and partitioning code to basic blocks uses standard techniques, a detailed description of the front end is not given. However, syntactic analysis carried out in the front end is described in various sections of this chapter. The PE driver controls the partial execution that is comprised of symbolic execution and code transformation. It ensures that the BAM memory model is consistent with the symbolic execution and updates the internal representation of the BAM code. The PE driver maintains a

control stack to control the entire partial execution process.

This chapter begins with a description of extensions of BAM data structures. These provide place holders for run-time characteristics of the BAM code. This is followed by the PE driver algorithm. We take a top-down approach to describe the the data structures that are built around the CFG. In other words, we illustrate the need of the data structures when explaining the appropriate algorithm step rather than lay them out before presenting the algorithm.

5.2 Augmenting BAM Memory Areas To Support PE

The BAM memory areas described in Chapter 2 are designed to hold run-time information. To analyze and characterize the run-time behavior of BAM code, additional place holders are necessary and these are described in this section.

5.2.1 Partial Execution Registers

The following pieces of information are maintained to ensure correct partial execution. Each of the pieces is stored in a register or other data structure as indicated in its description. A non-BAM register defined to hold any such additional information is called a partial execution register or *PE register*.

- Basic block information – The current block number being partially executed is stored in a PE register `bb`. The parent block number of `bb`, i.e., the block from which control passed to the current block, is stored in a PE register `pb`.

- Continuation block information – The number of the basic block to which the control flow returns upon successful completion of current procedure is stored in the PE register `cbb`. The number of the parent basic block, i.e., the block with a `call/1` instruction to the current procedure, is stored in PE register `cp_par`.

- The PE register, `proc_blk`, stores the number of the entry block of the current procedure being partially executed.
- The index into a table of input-output value pairs associated with the current procedure is stored.
- A new stack called the *allocate stack* whose items hold environment allocation information and information about the block containing the call to the current procedure. This data structure is discussed in Section 5.2.2.

5.2.2 Augmenting Environment Stack for PE

BAM code resulting from compiling a Prolog predicate is referred to as a *procedure*. A procedure begins with the instruction `procedure/1`. As described in Section 2.2.4, two different BAM code streams for a predicate are generated depending on the number of predicates in the clause body. If the body has more than one predicate, code to allocate an environment is generated. It is followed by code to set up call arguments and the call instruction `call/1`. An unconditional jump, `jump/1`, is generated instead of a call to the last predicate in the body. Code to deallocate the environment is generated before that unconditional jump. If the predicate body has one predicate call it is translated to an unconditional jump, `jump/1`. No environment allocation or deallocation instructions are generated in this case.

Thus a procedure is entered via a `call/1` or a `jump/1` instruction. Further, an environment is not created at every procedure entry. These run-time BAM execution characteristics necessitate additional mechanisms to keep track of procedure entries and environment creation during partial execution. A separate stack, referred to as an *allocate stack*, is used for this purpose. Partial execution of a call or a jump to a procedure creates an allocate stack item irrespective of the allocation of an environment by the procedure. A jump to an instruction `procedure/1` is treated as a procedure call except that the values of `cp` (the continuation register), `cbb` (the block continuation PE register) and `cp_par` (the continuation block parent PE register), are not updated. Partial execution of `allocate/1` sets a flag *alloc_flag*

in the top allocate stack item to indicate environment creation in the procedure. Correspondingly, during partial execution of `deallocate/1`, the *alloc_dirty* flag is set to indicate that the current procedure has deallocated an environment from the top of the environment stack. An allocate stack item consists of the following:

- The entry block number of the current procedure.
- The *alloc_flag* flag to record whether an environment is allocated. The same flag is used to record whether the current procedure was a last call optimized to a jump. This may be done without inconsistency since an environment is deallocated before the last call and no access to the environment stack is done during the call.
- A pointer into a table holding the current input value set of the calling procedure. This table also holds output value sets of the calling procedure. This table is referred to as *in-out* table.
- The *alloc_dirty* flag.

As discussed in Section 2.2.1.1, a BAM environment stores the permanent register values that occur in the predicate body clauses, a pointer to the previous environment and the return instruction address of the current predicate, namely, the current value of continuation pointer `cp`. For the purposes of partial execution, the environment is augmented with the following additional information:

- The return block address of the current predicate, i.e., the current value of `cbb`. Note that `cp` stored in the environment points to the first instruction of this block.
- The predecessor of the return block address stored above. This is stored since the CFG has no edge representing the control flow due to a procedure call or a return from a procedure call, as described in Section 4.2.1.

Chapter 6 describes how this additional information is used.

5.2.3 Augmenting the Choicepoint Stack for PE

Section 2.2.1.2 described the information stored in a BAM choicepoint. The following values are additionally stored in a choicepoint to aid analysis performed during partial execution.

- Basic block number, `cbb`;
- Basic block number of the parent of the continuation block, i.e., the value of PE register `cp_par`;
- Basic block number of the parent of the current block, i.e., the value of PE register `pb`;
- Control stack top (described in Section 5.4.2);
- Allocate stack;
- *Choice success information*, described below.

Speculative symbolic execution involves symbolic execution that assumes a dynamic register to have a certain static value. We refer to partial execution that involves speculative symbolic execution as symbolic partial execution. Speculative symbolic execution is done while an alternative retry address is attempted or while continuing PE along several possible successors of a basic block. All alternate choices available via a choicepoint are speculatively symbolic executed during partial execution. PE thus proceeds on several corresponding execution paths. Success or failure of the alternate choices is recorded in a data structure, called the *choice success information*. A choice is recorded as failure only if it can be shown to fail at PE time. Otherwise, it is recorded as a success. This may be viewed as a conservative approximation of run-time behavior of the program.

The heap and trail stacks are not augmented with any additional information for the purposes of PE.

5.3 BAM Code Partial Execution Driver

Consider a program **P** and its CFG $\mathbf{P_G}$. The sub-graph of $\mathbf{P_G}$ traversed from the beginning to the termination of one invocation of **P** is referred to as an *execution thread.* Any given program has several possible execution threads each of which is characterized by a sub-graph of the CFG rooted at the block containing the program entry point `procedure(main/0)`. One and only sub-graph corresponds to any single program invocation. Every subsequent invocation of the program might be characterized by a different sub-graph. This variation of execution threads between different program invocations depends on run-time inputs, if any. Since run-time inputs are not known at PE time, it is not possible to find the precise execution thread of a program invocation and its sub-graph. Consequently, all sub-graphs that represent possible run-time execution paths are discovered by performing partial execution. Instructions in blocks along each of these paths are optimized whenever possible by transforming them to simpler but equivalent ones. The PE driver traverses the CFG built by the front end to schedule basic block partial execution. Partial execution results in basic blocks that either are specialized, if enough information is available in the current program state, or belong to the original CFG otherwise. The PE driver schedules speculative symbolic execution when more than one PE-time control flow option exist. The code generation phase uses the resultant sub-graphs recorded in the *PE-flow graph* to generate optimized code while eliminating trivial transitions.

Algorithm 3 presents a PE driver algorithm based on the empirical partial execution algorithm outlined in Chapter 4. The driver traverses the CFG in a depth-first manner using a control stack to schedule partial execution along all execution threads of the given BAM code. Algorithm 3 hinges of three fundamental phases:

1. Checking and updating BAM memory state to ensure correct partial execution,
2. Checking if a residue(version) of the current basic block exists for the current memory state, and

Algorithm 3 Partial Execution Driver Algorithm

1: Let **G** be the control flow graph of the BAM code being partially executed and $\mathbf{G_{root}}$ be its root node.

2: Let **E** be a graph called *PE-flow graph*.
3: Let *cur_mem_state* be the current BAM memory state.
4: Let *total_instructions* be the total number of instructions in the BAM code.
5: For any basic block, *blk*, let $cont_{blk}$ hold *resumptions* resulting from partial execution of *blk*.
6: Set *node* = 0.
7: Set **CStack** = *nil*.
8: Set initial BAM memory state.
9: push ($\mathbf{G_{root}}$,*cur_mem_state*) onto **CStack**.

10: **while CStack** $\neq$ *nil* **do**
11: *(node,mem)* = pop(**CStack**)
12: **if** (pc = *success* or pc = *failure*) **then**
13: Record the edge between the parent of *node* and *node* in **E**
14: **else if** pc > *total_instructions* **then**
15: Flag error indicating out of code space access.
16: **else** /* `Continue partial execution` */
17: **if** *node* has no residue for the reference registers in *mem* **then**
18: Set current memory state to *mem*.
19: ($node_{res}$, $cont_{node}$) := *partial_execute(node)*
20: **if** $node_{res}$ is same as *node* **then**
21: $node_{new}$:= *node*.
22: **else**
23: $node_{new}$:= $node_{res}$.
24: Record $node_{new}$ as the residue of *node*.
25: Record $node_{new}$ in **E**.
26: **end if**
27: *update_cstack*($cont_{node}$)
28: **end if**
29: **end if**
30: **end while**

3. Transforming instructions to their simpler equivalents whenever possible.

The rest of the chapter is devoted to explaining various aspects of the Algorithm 3. The algorithm uses several characteristics of a basic block and a procedure. A description of these characteristics is follows.

5.3.1 Characteristics of a Basic Block

Several syntactic and run-time characteristics of a basic block are used by the PE driver. The following information is associated with a given basic block.

- Pointers to its successors and ancestors.
- A list of focus registers that parameterize the block (Section 5.3.2).
- Pointers to its versions generated during partial execution.
- A flag indicating whether the block contains a `choice/3` instruction.
- Procedure analysis information viz., register and in-out table information of the procedure containing this basic block (Section 5.3.3).
- Strongly Connected Component (SCC) related information used to identify loops in CFG (Section 5.5.2).

We discuss focus registers, procedure and SCC related information in the following sections. The rest are self-explanatory.

5.3.2 Parameterizing a Basic Block With Optimal Reference Registers

Each basic block may be parameterized with the reference registers introduced in Chapter 4. When a basic block is partially executed with respect to a certain set of static reference register values producing a residue, we say a version of the basic block is generated. The PE driver in Algorithm 3 checks whether a version of the

current basic block corresponding to current values of all the reference registers exists in the current memory. However, only some of the reference registers (i.e., permanent and argument registers accessed by the instructions in a basic block) are affected during the partial execution of the basic block. These registers are called *active registers*. Thus, associated with every basic block is a set of active registers. Consequently, it is sufficient to partially execute a basic block only when it has not been specialized for the current values of the active registers.

Specializing a basic block with respect to all dynamic registers leads to code explosion. Thus the specialization must be restricted to static active registers. This technique is similar to that of specializing a function with respect to its static arguments as done in partial evaluation of functional programming languages [42,57]. Specialization of programs with respect to static/invariant entities has been studied in various contexts. Haraldsson's online partial evaluator *Redfun* [34] is considered to be the first attempt at this. Consel and Khoo [18] define *facets* that a provide means for user-specification of static properties in the context of both online and offline partial evaluation of a first-order language and provide a formal framework. The current work follows the conventional methodology of specialization with respect to static properties. However, no user-specification, either of static properties or of input values are expected during the online specialization. Further, we specialize a low-level language and do not attempt a self-applicable specializer. Additionally, the partial execution algorithm has no information about the Prolog predicates whose BAM code translation is being partially executed nor of their arguments. In summary, this work differs from others by not relying on user specifications or on any syntactic knowledge of the Prolog sources. Further, the low-level of abstraction of its source allows the technique to be used as a compiler phase.

BAM registers may be accessed for two purposes — to read or to update their contents. The terms *read* and *define*, respectively, are used to distinguish these accesses. The first access of some of the active registers in a basic block may be to define before reading, irrespective of their value at block entry. Thus, instead of specializing the basic block for all static active register values, it is sufficient to

specialize it for the current values of only those active registers that are read from in the basic block and ignore the static active registers that are defined. Four distinct cases of active register accesses occur within a basic block. A register may be

1. read before being defined within the block
2. read after being defined within the block
3. only read within the block
4. only defined within the block

Let $\mathbf{R_{read}}$ and $\mathbf{R_{def}}$ denote the set of read and defined registers in a basic block, respectively. Static active registers that are either read before being defined (type 1), or only read (type 3), contain relevant static values. At any program point, the contents of these registers provide the invariants for specialization of the basic block. Thus, a basic block is specialized for static active registers of this set. The set containing the union of registers of types 1 and 3 is called the *focus* register set, denoted $\mathbf{R_{foc}}$. Consequently, a basic block is specialized if it has not been specialized for the current static values of its focus register set. The focus registers in a given basic block may be found using Algorithm 4.

A basic block is thus parameterized with its focus register set. The read/defined classification of an active register is based on its operand position in an instruction and so can be performed in the front end. Static-dynamic classification of focus registers is a PE-time property and is done while symbolically executing the BAM instructions.

A CFG may contain basic blocks with no active registers. However, PE of the block still needs to be done to correctly set the BAM memory state for partial execution of any of its successor basic blocks.

5.3.3 Characteristics of a Procedure

Recall that a basic block is a collection of instructions with only one entry and one exit. A collection of basic blocks with only one entry and many possible exits

Algorithm 4 Find focus registers

```
1: We assume that instructions in the basic block are arranged in a linked list
   whose head is ll_hd. ll_nd→instr is the instruction pointed to by ll_nd.
2: R_read = Φ
3: R_def = Φ
4: R_foc = Φ

5: while ll_nd ≠ nil do
6:   for each operand, r, of ll_nd→instr do
7:     if r is a read register then
8:       R_read = R_read ⋃ {r}
9:       if r ∉ R_def then
10:        R_foc = R_foc ⋃ {r}
11:      end if
12:      if r is a defined register then
13:        R_def = R_def ⋃ {r}
14:      end if
15:    end if
16:  end for
17:  ll_nd = ll_nd→next
18: end while
```

is a procedure. These two levels of partitioning of BAM code facilitate analyses of instruction sequences in BAM code. These analyses result in recognizing various basic block and procedure properties that guide the PE driver. The previous section described one such property of a basic block, reference registers of the block that parameterize it.

A procedure has two kinds of properties that are used during partial execution process - syntactic and run-time. For example, the number of argument registers that appear in the procedure is a syntactic property. Syntactic analysis of a procedure begins with building of the call graph [2] and a list of argument registers that occur in the procedure code. The number of permanent registers used in the procedure is not recorded separately. This information is readily available at PE-time from the current environment. This initial internal representation is augmented with several other syntactic analyses that are described in this chapter.

Prolog predicates may be written so that the same positional argument is used to pass a value into its body (i.e., used as an input argument) at one call site and

to pass a value out of the body (i.e., used as an output argument) at another site. Further, the type of the argument can vary from one call site to another. Consider the following definition of add/3

```
add(X,Y,Sum) :- integer(X), integer(Y), Sum is X + Y.
add(X,Y,Sum) :- list(X), list(Y), append(X,Y,Sum).

append([],L,L).
append([H|T], L, [H|R]) :- append(T,L,R).
```

where, `integer(X)` and `list(X)` are built-in type checking predicates that succeed if `X` is of integer type and list type, respectively. The addition operation is *overloaded* or extended to list arguments. Thus the arguments `X` and `Y` may be of integer type at one call site and of list type at another. Further, in the case that the arguments of add/3 are of list type, any two of the three arguments may be used as input arguments to compute the third.

BAM code for such predicate calls may be specialized according to the type of arguments at a given call site. Thus a predicate call may be made with more than one set of input instantiations. Each of these calls may result in corresponding output instantiations. A table that records the mapping of input-output static values is maintained per procedure. This table is referred to as the *in-out table* and is part of information associated with the basic block, as detailed earlier. It is updated at every procedure entry and exit.

Any given procedure may have more than one exit. Thus more than one set of output values may be associated with a given input value set. Additionally, by PE we may discover that some procedure exits lead to failure. Consequently, an in-out table entry is a pair, $\langle \mathbf{V}_{in}, \mathbf{V}_{out} \rangle$ such that

$$\mathbf{V}_{in} = \{\langle r, v \rangle \mid r \text{ is an argument register and } v \text{ is its static value at procedure entry}\}$$

and

$$\mathbf{V}_{out} = \{O \mid O = \{\langle r, v \rangle \mid r \text{ is an argument register and}$$

v is its static value at procedure exit}

5.4 PE Driver Execution

The PE driver, given in Algorithm 3, traverses the CFG using a stack-based version of the traditional depth-first traversal algorithm [2] beginning at the procedure with first instruction `procedure(main/0)`[1] It builds successor basic block information on a stack referred to as the *control stack* – **CStack**. A **CStack** item contains a pointer to the successor block and a pointer to the program state with which the block's partial execution is to proceed. **CStack** is referred to as control stack since its contents, detailed in Section 5.4.2, control and drive the partial execution.

The PE driver initializes the BAM memory areas to their respective start states before the CFG traversal begins. Thus heap, choicepoint stack, environment stack and trail stack are set to empty and the corresponding register values `h`, `b`, `e`, and `tr` are set to uninitialized values. Program counter `pc` is set to the first instruction to be executed viz., `procedure(main/0)`. The continuation pointer `cp` is uninitialized. Either of the two special values of `pc`, viz., *success* and *failure* (as defined in Section 3.4.9), are used to designate successful and unsuccessful completion of an execution thread, respectively.

The state of augmented BAM memory areas (Section 5.2) along with the additional information maintained to support partial execution is referred to as the *memory state* and forms the **CStack** item. The memory state is ensured to be correct at all program points to guarantee the correctness of the partial execution process. Memory state is used in various stages of partial execution such as loop-checking, updating PE-flow graph, restoring BAM memory state correctly upon loop detection and performing choicepoint optimizations.

[1] Aquarius Prolog considers the first predicate in the first program file being compiled as the program entry point. This predicate should be of arity zero [35]. For ease of notation we always use `main/0` to denote entry point.

5.4.1 Semantics of Dereferencing during Partial Execution

Following is a discussion of issues that resulted in the ultimate choice to represent a dynamic BAM register in the implementation of the partial executor. A dynamic register denotes a Prolog variable that is unbound at PE-time. In BAM, it is represented as a self-referential heap location, as described in Chapter 2. Since any value of a register whose tag is known at PE-time is considered static, we need a distinguish a self-referential, pointer-tagged value from its dereferenced dynamic value. We begin by designating a BAM register with a special unique datum δ as a dynamic register. Thus dereferencing a self-referential heap location yields the value δ. Although this representation is inadequate, it is used as a preliminary step to reveal some subtle design considerations and subsequently arrive at a correct representation. This is done with the help of the CFG in Figure 5.2 for the predicates `main/0` and `num/1` of the program in Figure 5.1. The BAM code in Figure 5.2 is generated by the Aquarius compiler with GFA phase turned on.

```
:- option(analyze).

main :- num(X),p(X,Y), write(Y).

p(2,Y) :- Y = 10.
p(3,Y) :- Y = 20.

num(3).
```

Figure 5.1: Program `simple.pl`

Consider partial execution of the call to `num/1` in block 0. At the entry of block 3 the register `r(0)` contains `tvar^0`. Further, heap location 0 contains `tvar^0`. Symbolic execution of `deref(r(0),r(0))` in block 3 sets `r(0)` to δ thus revealing it to be a dynamic register due to the content of heap location 0 which is a self-referential dataword with the `tvar` tag. Since the branch instruction `test(ne,tvar,r(0),1(num/1,1))` involves a dynamic register, a speculative symbolic execution is performed along

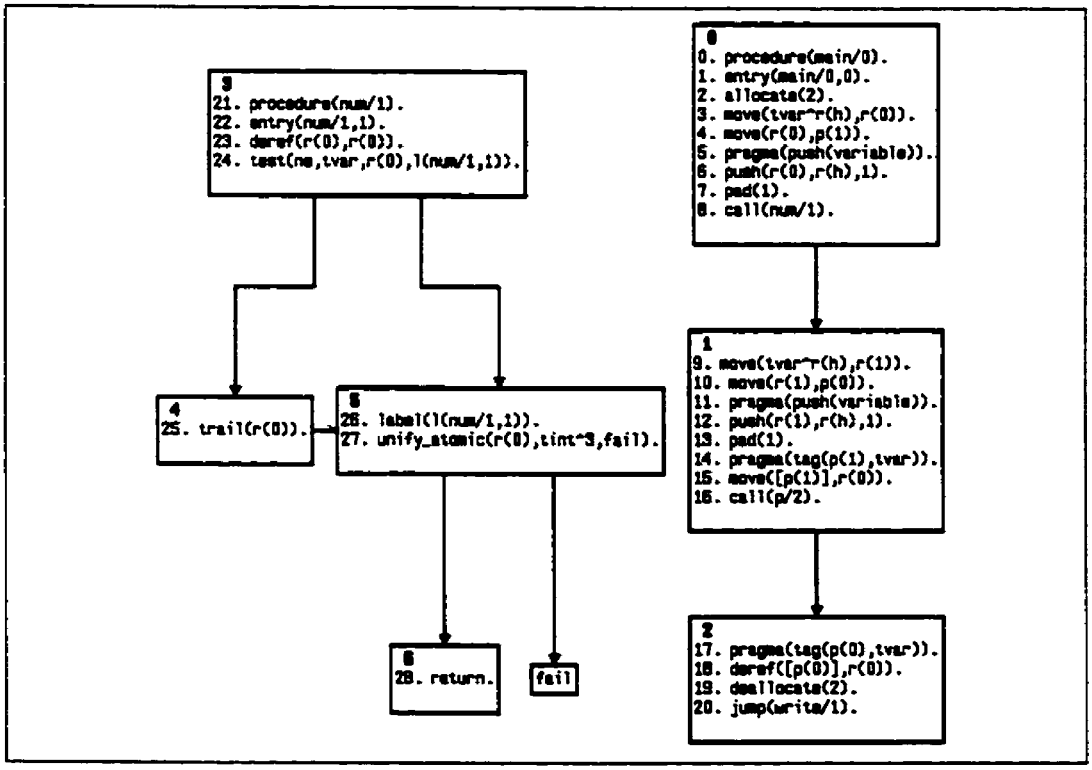

Figure 5.2: Example to demonstrate deferred dereferencing of a dynamic register with tvar-tagged value

both successors of 3, viz., 4 and 5, by setting `r(0)` to have the tag `tvar` and to have a non-`tvar` tag at entry time respectively. Partial execution of blocks 4 and 5 are otherwise done with exactly the same memory state.

Consider the PE along the execution thread 3-5-6 of Figure 5.2. At the entry of block 5, `r(0)` is known to have a non-`tvar` tag. Hence, the speculative successful PE of `unify_atomic(r(0),tint^3,fail)` sets the value of `r(0)` to atomic value `tint^3`. By examining what happens at run-time, this is revealed to be only partially correct. At run-time, the BAM memory at this program point (i.e., at the entry of block 6) not only contains the dataword `tint^3` in register `r(0)` but also in the heap address x. The heap address 0 contains the dataword `tint^`x. In essence, PE of `unify_atomic/3` can not set the heap to the correct state since `r(0)` contains only data-tag information, that indicates it as having a non-`tvar` tag. `r(0)` has no datavalue which at run-time is the heap address. The datavalue information lost due to dereferencing `tvar^0` to δ is the heap address 0 pointing to the heap location of the atomic value unifying with `tint^3`.

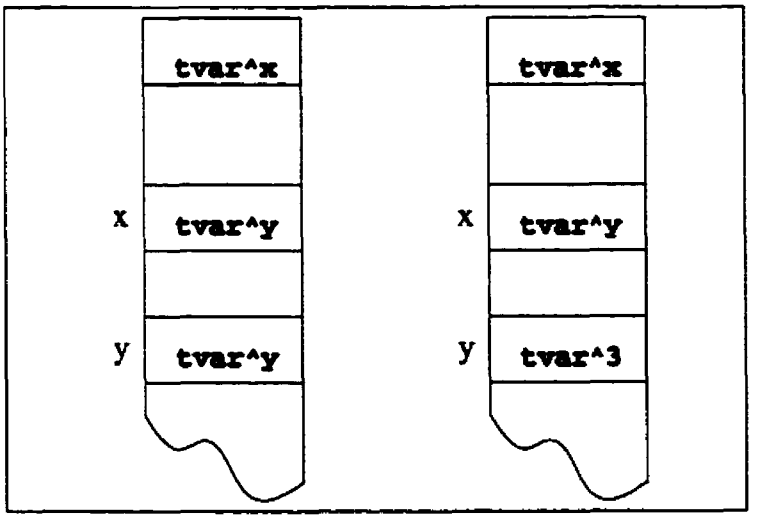

Figure 5.3: Example dereferencing chains

The loss of heap address described above also results in incorrect partial execution on the path 3-4-5-6 as follows. PE on this execution thread simulates run-time execution of procedure `num/1` with a `r(0)` containing a dataword `tvar^0` at the entry of block 3. In general, the heap address indicated by the datavalue of `r(0)` might

be the beginning of a dereference chain with either a self-referential dataword with tag `tvar` or a non-`tvar` tagged dataword as shown in Figure 5.3. It will be resolved by the `deref/2` instruction to the dataword at the end of the chain. In the current example, dereferencing `r(0)` (instruction 23) sets `r(0)` to δ thus losing the last heap pointer in the chain of heap pointers. Hence, PE of `trail(r(0))` in block 4, trails nothing. The run-time execution of `unify_atomic(r(0),tint^3,fail)` not only sets `r(0)` to `tvar^0` but also sets the heap location whose address is the datavalue of `r(0)` before it is updated. However, as described earlier, during PE of block 5 the heap is not updated. This leads to the conclusion that the representation of a dynamic value needs to carry the datavalue that indicates the heap location to ensure correct trailing and heap update during term unification.

A new technique is designed to avoid this loss of information. A self-referential dataword is dereferenced to a dataword with a special datatag `dvar`. The new tag `dvar` is only known to and used by the PE driver to perform and maintain run-time information of the program. Hence neither BAM execution semantics nor the memory model need be changed. The datavalue of the `dvar` tagged dataword is the heap address of the self-referential dataword at the end of a possibly long dereferencing chain. This scheme allows us to indicate that the current register is dynamic while retaining the heap address of the atomic value generated during unification. This preserves correctness of partial execution and loses no information. This new technique is referred to as *deferred dereferencing*.

Now we examine similar issues involved in dereferencing dynamic register that dereferences to a `tstr`-tagged dataword at partial execution time. As noted in Section 2.2.2, a `tstr`-tagged dataword also holds a pointer to the sequence of heap memory addresses that contain the functor and arguments of the structure. The number of contiguous heap locations holding the structure information is embedded in the atomic representation of the functor. Such a representation necessitates another special tag to identify a dynamic register that dereferences to a `tstr`-tagged dataword at partial execution time. Similar to the tag `dvar`, this new tag is only known to and used by the PE driver to perform and maintain run-time information

of the program. The need for this new tab is discussed with the help the following example. Figure 5.5 shows the CFG of the BAM code of the predicate `str/1` defined in program given in Figure 5.4.

```
main :- str(X),p(X,Y), write(Y).

p(f(a,b),Y) :- Y = 10.
p(g(c,d),Y) :- Y = 20.

str(f(a,b)).
```

Figure 5.4: Program to illustrate the need for `dstr`

Consider what happens at run-time entry into procedure `str/1` with `X` instantiated to a structure, say `f(a,b)`. Register `r(0)` contains a dataword `tvar^`x where x is the heap address containing the contiguous structure information at the time of entry into block 33 as shown in Figure 5.5. Dereferencing `r(0)` in block 33 sets `r(0)` to `tstr^`y where is the heap address of the beginning of the contiguous heap locations that hold the structure.

Now, consider partial execution of an arbitrary call to `str/1` with a dynamic `r(0)`. At the entry of block 33, the register `r(0)` contains the dataword `tvar^`x, where x is the heap address holding the value of `X`. For simplicity, suppose the heap address contains the self-referential dataword `tvar^`x. (That is, assume the dereference chain is of length zero). Partial execution of `deref/2` in block 33 sets `r(0)` to `dvar^`x. Speculative partial execution along edge 33-35 needs to proceed as if the variable `X` is a structure. Thus the tag value of `r(0)` is set to `tstr`. To avoid loss of the last heap address at the end of the dereferenced chain, we retain x as the datavalue in `r(0)`. The speculative partial execution needs to build a structure on the heap. PE proceeds along the path 36-38-39-41-42 by speculating `r(0)` to be instantiated to this structure.

Thus during partial execution, the value of `r(0)` needs to record that the datavalue

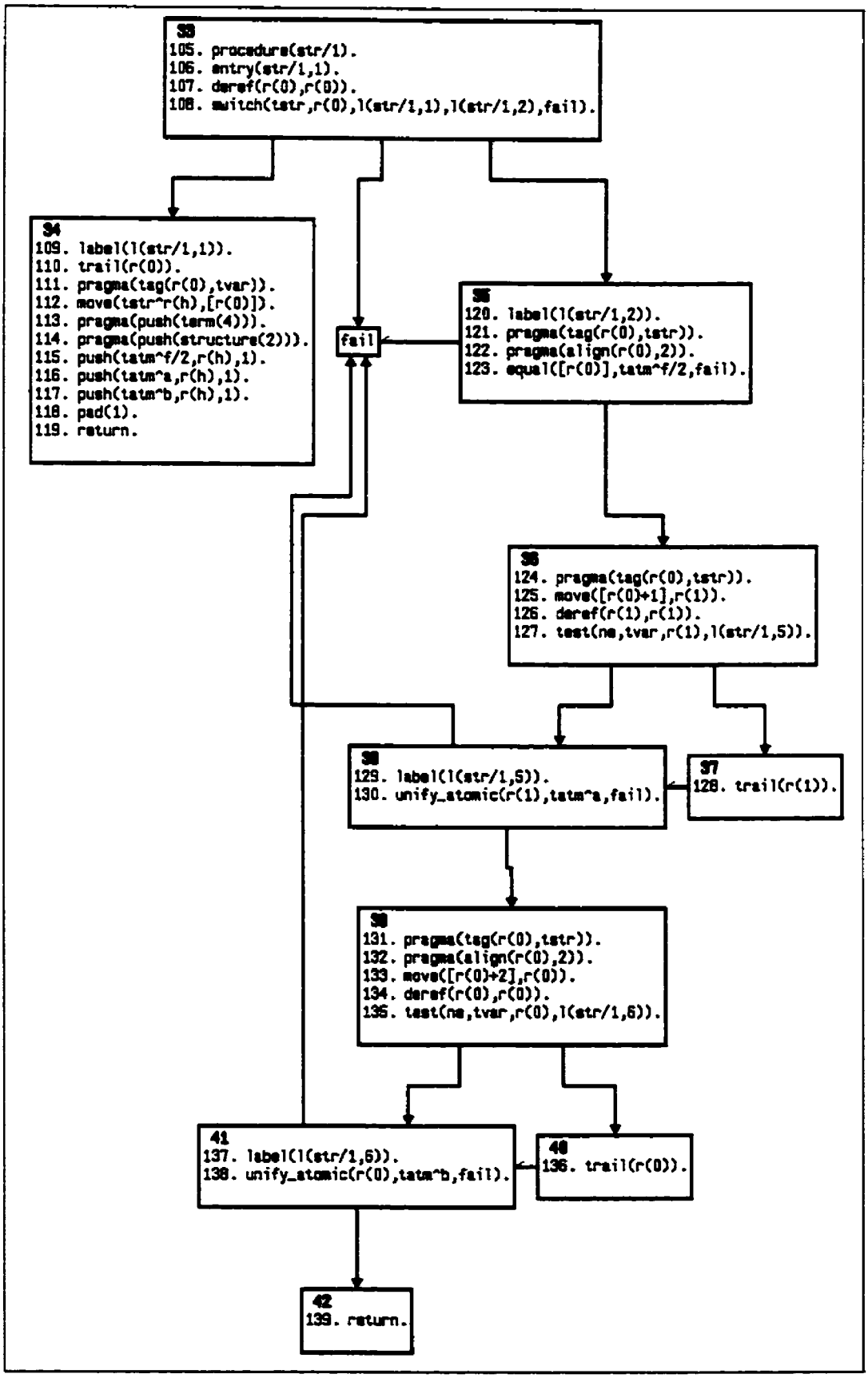

Figure 5.5: CFG of code with deferred dereferencing of dynamic register with `tstr`-tag

is in fact a self-referential tvar-tagged dataword that dereferences to tstr^x where x is only known later. Such a dataword is represented using a special tag, dstr. This new tag also maintains the semantics of a tstr-tagged dataword since a self-referential tstr-tagged dataword has no well-defined meaning.

At run-time, a compound term built on the heap is accessed by indirectly addressing its tstr-tagged dataword. If the structure is unknown during partial execution and indicated by a dstr-tagged value, the location of the structure's functor/arity and its arguments can not be determined. The dstr tag provides the required hint to update the heap along with the register involved.

In summary, a structure, whose form and heap location are not known, is represented on the heap by a self-referencing dataword with a tag dstr. The tag dstr, distinct from BAM tag tstr, is used to signify a dynamic structure. Its heap location is retained to facilitate subsequent access of this dataword when the structure's form and heap space become known and is created.

Thus, a dataword with either of the tags – dvar or dstr – is dynamic. A dataword dvar^V stands for a heap value obtainable by dereferencing the heap location V. A dataword dstr^V denotes an unknown structure whose preamble dataword on the heap is at location V. A dvar-tagged dataword will never be formed on the heap but a dstr-tagged dataword may. Datawords with tags dvar or dstr are created by the dereferencing operation. However, since they are only known to the PE-driver, the semantics of dereferencing operation described in earlier chapters need not be changed. These tags are part of extensions made by the PE-driver to maintain run-time information.

Deferred dereferencing is not required of datawords with tlst tag, the third dataword of "pointer" type. This is because the heap space needed to store its constituents viz., the head and that tail, is known and may be created accordingly at PE-time even if the contents of list itself are unknown.

5.4.2 Control Stack

The PE driver uses the control stack to control the partial execution process. Each control stack item holds a pointer to the block along with the memory state in which the block is to be partially executed. A **CStack** item is comprised of the following:

- current values of BAM registers and PE registers.
- a pointer into the in-out table entry that corresponds to the current input-output values of the procedure being partially executed;
- current heap, choicepoint, environment and trail stack values;
- current allocate stack value;
- any additional register values that need setting upon restoration of BAM memory done at step 18 of Algorithm 3.

As shown in Algorithm 3, partial execution of a block, *blk*, results in a set of data items, $cont_{blk}$ and a residue *res*. $cont_{blk}$ is called a resumption set of *blk*. Each of the data items in the set corresponds to one block to be partial executed after *blk* and is called the *resumption* of *blk*. A resumption holds three items – memory state, *resumption target* and *resumption register values*. A resumption target is a pair $(\mathtt{pc}_r, \mathtt{bb}_r)$ where $\mathtt{bb}_r$ is the block at which PE resumes and $\mathtt{pc}_r$ is its first instruction. *resumption register values* holds one or more register-value pairs. These registers are set to the corresponding values when PE resumes at block $\mathtt{bb}_r$. *update_cstack* uses this resumption set to form control stack items. The resumption set is formed as follows, depending on the control flow change instruction of the block *res*.

1. The block *res* has only one successor whose pointer is explicitly available in *blk*. In this case, the resumption set of *blk* contains only one resumption that holds the current memory state and the successor block as resumption target.
2. The block *res* has several successors whose pointers are explicitly available in *blk*. In this case, the resumption set of *blk* contains one resumption for

each of the successors. Each resumption holds the current memory state, the corresponding resumption target and resumption register values, if any.

3. The last instruction of *res* is `fail/0`. The run-time behavior of BAM in this situation is to resume execution from the next choice stored in the current choicepoint. In this case, the resumption set of *blk* contains only one resumption that holds the memory state saved in the current choicepoint, the alternative choice as resumption target and resumption register values, if any. *update_cstack* also records a failure of the current choice in the choicepoint apart from forming a control stack item. Note that the values of `pc` and `bb` will never be **FAILURE** except when the partial execution is complete and when there are no more choicepoints available in the choicepoint stack.

4. The last instruction of *res* is either a `return/0`, a `jump/1` or a `call/0` to a non-local target that is assumed to succeed and thus is treated as equivalent to `return/0`. Partial execution of `return/0` sets the current memory such that partial execution may continue along the block number `cbb`. In this case, the resumption set of *blk* contains a single resumption that holds the current memory state, resumption target pair (*cbb*, first instruction number of *cbb*) and resumption register values, if any. *update_cstack* also records success of the current choice in the current choicepoint.

Thus, the control stack keeps track of all the run-time execution threads to be traversed after completing partial execution of the current block. The additional block specific speculative information mentioned above constitutes all the register datavalues set on a speculative PE path.

5.5 Loop Detection and Termination of Partial Execution

In presenting the details of Algorithm 3, we have already described the properties of a basic block and a procedure, their parameterization with reference registers, and the control stack that controls PE. We now describe the loop-handling mechanism used in the algorithm.

The depth-first traversal of the CFG schedules blocks to be partially executed. However, partial execution of the block is actually performed only if it was not performed earlier with respect to the reference registers (more precisely, static focus registers). A *PE-loop* is detected if a residue exists for the block scheduled for partial execution for its reference registers. A *PE-loop* may or may not correspond to a loop in the BAM code. This section discusses detecting and handling of PE-loops (Step 15 in Algorithm 3).

Recall that the top of **CStack** contains both the block, *blk*, and the memory state, *mem*, in which *blk* must be partial executed. The PE driver restores the memory state to *mem*, but before partial executing *blk* it considers the following options.

1. *blk* may be the entry block of a procedure. Let $\mathbf{P}_{in}$ be the set of input argument-static value pairs for the current memory. The PE driver performs partial execution of the procedure block based on the in-out table entry for $\mathbf{P}_{in}$ as follows.

   (a) If the in-out table has a record of output values $\mathbf{P}_{out}$ corresponding to $\mathbf{P}_{in}$, then the procedure need not be partially executed. The existence of output value information corresponding to the current static input argument registers implies that the procedure has already been partially executed. The current memory is updated with the set of output values in $\mathbf{P}_{out}$ to reflect the execution of the procedure and partial execution continues as if the procedure has been partially executed.

If $\mathbf{P}_{out}$ has more than one set of output values that correspond to $\mathbf{P}_{in}$, speculative partial execution is set up by setting the current memory state to each of these output values in turn and continuing partial execution along the execution threads that correspond to those output values.

Figure 5.6 illustrates this case. Partial execution of `call(p/n)` in block **A** results in a resumption that holds the current memory state and block **C** as resumption target. Hence PE of **C** is scheduled. Let $\mathbf{A}_{out}$ be the static output values at the exit of **A**. Then the input static values $\mathbf{C}_{in}$ of block **C** are a proper subset of $\mathbf{A}_{out}$. Let **D**, **E**, **F** and **G** be the exit (or leaf) blocks of **p/n**. Assume **p/n** was previously partially executed for the static input values $\mathbf{C}_{in}$. The in-out table corresponding to procedure **p/n** will have an entry $\langle \mathbf{C}_{in}, \langle \mathbf{D}_{out}, \mathbf{E}_{out}, \mathbf{F}_{out}, \mathbf{G}_{out} \rangle \rangle$ where $\mathbf{D}_{out}, \mathbf{E}_{out}, \mathbf{F}_{out}$ and $\mathbf{G}_{out}$ are the output static value sets corresponding to the procedure exit points.

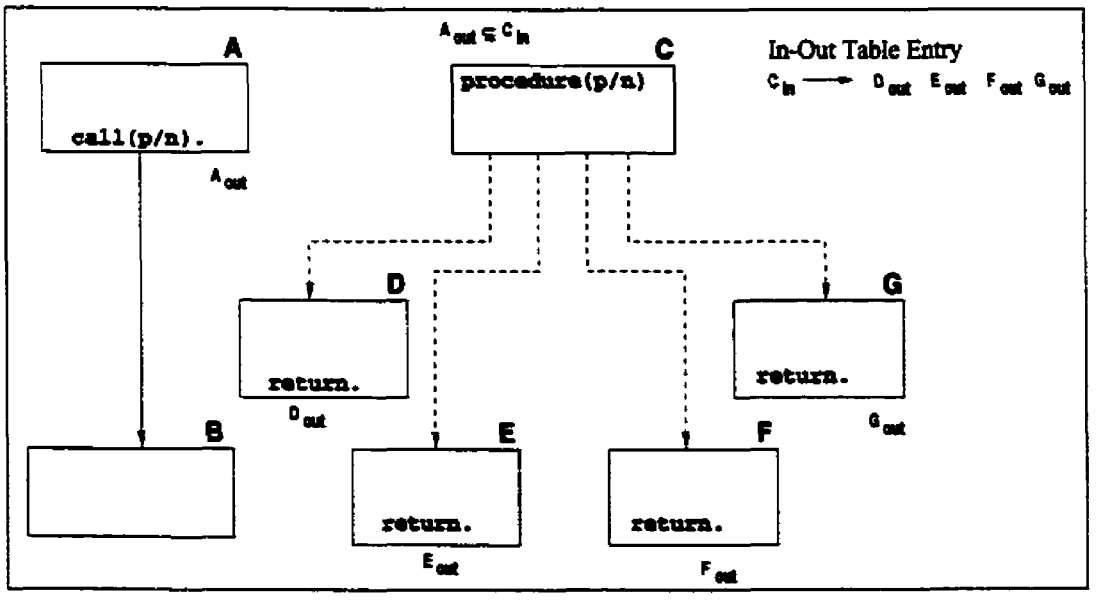

Figure 5.6: Schematic illustration of procedure *in-out* value usage

Since the **p/n** has been previously partially executed for $\mathbf{C}_{in}$, the PE driver schedules four speculative partial execution threads through block **B** with memory set to the output values $\mathbf{D}_{out}, \mathbf{E}_{out}, \mathbf{F}_{out}, \mathbf{G}_{out}$ respec-

tively.

Thus this technique results in specialization of procedure calls for specific call sites while ensuring termination of partial execution.

(b) If in-out table has no record of output values $\mathbf{P}_{out}$ that correspond to $\mathbf{P}_{in}$, then the procedure is partially executed.

2. *blk* is a non-procedure block. Then, a check is performed similar to one used to determine whether a procedure was previously partially executed. This check determines if the current block has a residue corresponding to the current static focus registers of the block. The version check is made possible by maintaining the following run-time basic block residue information.

   - The static registers for which a residue was generated along with a pointer to the residue;
   - A unique identification of the residue block;
   - The result of partial execution of the block.

   Thus the version check simply verifies whether the current basic block has been partially executed given the current static focus registers. If so, a loop is said to be detected and the residue is recorded in the flow graph that records the partial execution flow. Otherwise, the block is partially executed.

5.5.1 Handling a Basic Block Execution Loop

Assume that *blk'* is the residue of block *blk* for static focus registers s. Let the static focus registers at the current stage of partial execution also be s and hence a loop is detected. Once a loop is detected prior to partial execution of *blk*, three alternate situations exist.

- Previous partial execution of *blk* resulted in a local failure. This would have set the partial execution along the retry address, say `r`, in the choicepoint. Presently, since the loop is detected, it is incorrect to let partial execution

continue along r since the retry address in the current choicepoint could be different. Instead, partial execution is set to continue along the retry address in the current choicepoint and the residue is recorded in the execution flow graph.

- Previous partial execution of *blk* resulted in global failure due to PE of an instruction like `jump('$ flt_error/2')`, where `'$ flt_error/2'` is the label to the built in floating-point error condition handler. Then, the current block is not partially executed. Instead, global failure is recorded in the PE-flow graph and depth-first traversal of the CFG continues.

- Partial execution of *blk* was successful and generated a residue *blk'*. This is the most common case. We assume partial execution will succeed along the current path and return out of the current procedure by restoring the continuation pointer (cp). The memory is set to the static output values recorded in the in-out table of the procedure being returned from.

Here, we take advantage of the fact that control flow does not enter a block from outside the procedure to which the block belongs. In other words, all target labels of `jump`, `switch` and `test` instructions are either within the procedure or are built-ins. Any previous partial execution thread through *blk* with static values *s* would have traversed the same blocks as the current partial execution through *blk* will traverse. Thus it is sufficient and correct to return out of the current procedure thereby setting memory state with the static output recorded during previous PE of the current procedure. This is accomplished by *simulating a return* using Algorithm 5.

5.5.2 PE Loops and Code Loops

Strongly connected components (SCCs) are commonly used to identify loops in a CFG. SCCs in the BAM code CFG are computed in the front end. Syntactic loops within each closed procedure of the CFG are found with the well-known algorithm of Tarjan [64], detailed by Wolfe [72] and sketched in Algorithm 6.

Algorithm 5 Simulation of a return out of a procedure when a loop is detected

1: **if** an environment was allocated by the current procedure **then**
2: Restore the values of cp, cpp, cp_par and e from the environment top.
3: Pop the environment stack top.
4: **end if**
5: **if** *blk* is a choicepoint block **then**
6: **if** **CStack** has no blocks that are scheduled to be partially executed after creation of current choicepoint **then**
7: Set retry address in the current choicepoint to next retry address.
8: **end if**
9: **end if**
10: Partial execute the instruction `return/0`

Following are the data structures used in the algorithm.

- n is the global counter for assigning pre-order numbers, initialized to zero. V is the set of graph nodes.
- *CountSCC* is the total number of strongly connected components found, initially zero.
- *Stack* is a stack of nodes, initially empty.
- $NPre(x)$ is the pre-order number assigned to each node, initially zero for each node.
- $Lowlink(x)$ keeps track of whether each node has a path to a spanning forest ancestor.
- $SCC(x)$ is the SCC number assigned to each node; two nodes with the same SCC number are in the same strongly connected component.
- $InStack(x)$ is a flag indicating whether the node is on the stack; initially set $FALSE$ for every node.

A conventional loop in the given code is called a *syntactic loop* to distinguish it from the *PE-loop* described above. Each SCC denotes a syntactic loop in the BAM code. Each syntactic loop entry need not correspond to a PE-loop. Consider

Algorithm 6 Algorithm for finding strongly connected components

```
1: for x ∈ V do
2:    NPre(x) = 0
3:    InStack(x) = FALSE
4: end for
5: n = 0
6: CountSCC = 0
7: Stack = Φ
8: for x ∈ V do
9:    if NPre(x) == 0 then
10:      SCCRecurse(x)
11:   end if
12: end for

13: Procedure SCCRecurse(x)
14: Lowlink(x) = NPre(x) = n = n = n + 1
15: Push x onto Stack
16: InStack(x) = TRUE
17: for y ∈ succ(x) do
18:   if NPre(y) == 0 then
19:      SCCRecurse(y)
20:      Lowlink(x) = min(Lowlink(x), Lowlink(y))
21:   else if NPre(y) < NPre(x) ⋀ InStack(y) then
22:      Lowlink(x) = min(Lowlink(x), NPre(y))
23:   end if
24: end for
25: if NPre(x) == Lowlink(x) then
26:   CountSCC = CountSCC + 1
27:   repeat
28:      pop w off Stack
29:      InStack(w) = FALSE
30:      SCC(w) = CountSCC
31:   until w == x
32: end if
```

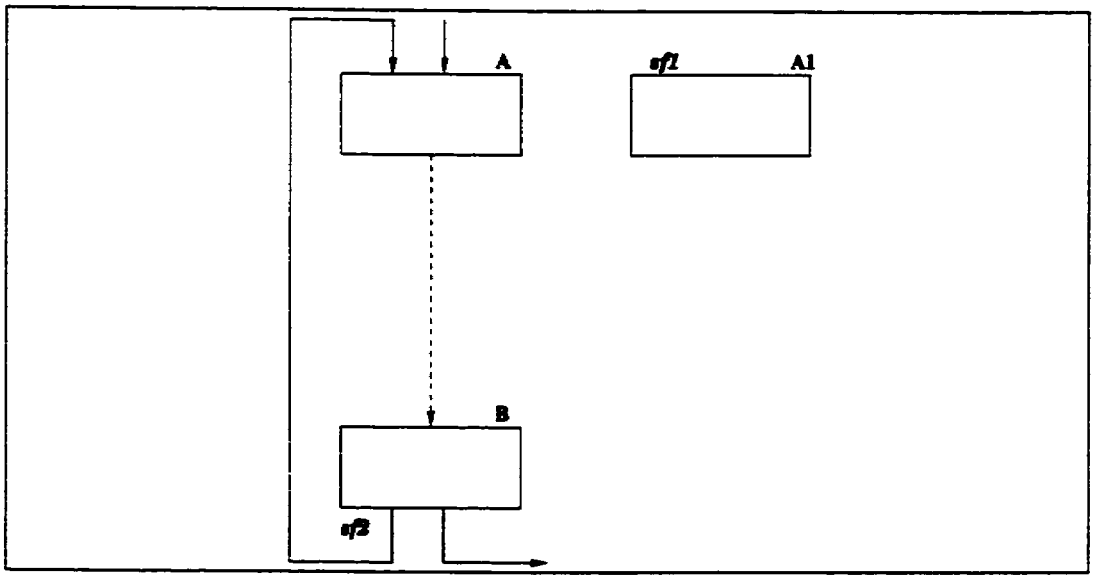

Figure 5.7: Schematic to Illustrate PE-loop and Syntactic Loop

the schematic illustrating a syntactic loop **A-...-B-A** in Figure 5.7. Assume that when the block **A** is partial executed for the first time with static focus registers *sf1* it generates a residue **A1**. If the static focus register values at the exit of partial execution of **B**, say *sf2*, are a proper subset of *sf1*, a PE-loop is detected. In such a case the syntactic loop **A-...-B-A** is the same as the PE-loop. However, if *sf2* is not a proper subset of *sf1* the PE driver schedules the partial execution of **A** in the context of *sf2* to generate a possibly different residue. In such a case, the syntactic loop is different from the PE-loop. Such characterization represents a conservative form of classical loop unrolling [54, 72]. The conservative nature is due to the loop checking criterion (as implemented) that tests for the equality of only the datatags of static register values. However, the effect of aggressive loop unrolling can be achieved within the partial execution framework presented here by extending the equality test to the entire dataword contents.

5.5.3 Termination of Partial Execution

Partial execution of given BAM code terminates upon traversing all the blocks of its CFG. With the loop detection and handling techniques described in earlier sections and the finite number of residues generated as described in Section 4.2.2,

termination of partial execution is straightforward. Partial execution of the complete CFG traverses all the possible execution threads. Thus any path in the graph not traversed during partial execution exposes dead code. Thus dead-code elimination is a by-product of partial execution.

5.6 Partial Execution of a Basic Block

Partial execution of a given basic block in the context of current memory can be performed by a simple loop that performs symbolic instruction execution and instruction transformation together using Algorithm 7.

Algorithm 7 Basic block partial execution

```
1: for all basic block instructions do
2:    if static operand information available then
3:       transform the instruction according to the transformations detailed in Section 4.3.
4:    end if
5:    Symbolically execute the instruction.
6: end for
```

An instruction is transformed to a simpler equivalent if sufficient static information is available as detailed in Section 4.3. Symbolic execution of an instruction is done after instruction transformation to ensure correctness of the transformation in the context of correct BAM memory state.

Depending on the result of basic block partial execution, the **CStack** is updated so that partial execution continues correctly. Further, the result of block PE is recorded in the residue information.

The following information, pertaining to the run-time behavior of the block and results of the partial execution, is collected during partial execution of a block:

- Residue generation: If at least one of the instructions in the block is transformed, it is recorded that a residue is generated.
- Local failure: If PE of the block resulted in failure, a local failure is recorded.

- Global failure: If the PE of the block resulted in failure, a global failure is recorded.

5.7 Implementation of BAM Partial Executor

It is highly desirable that the partial execution phase does not significantly add to the compilation time of Aquarius Prolog compiler. The front end first parses the candidate BAM code, builds the CFG and syntactically analyses it. The PE driver then performs the partial execution process by building and maintaining the various data structures described earlier. Except for the implementation of parsing functionality, the partial executor is implemented using C. The Aquarius compiler can generate a BAM code listing during compilation of a Prolog program, BAM instructions emitted as Prolog terms. Hence, Sicstus Prolog is called by the partial executor to parse them. The implementation consists of over 15,000 lines of C code along with over 200 lines of Prolog code.

5.8 Summary

The main contribution of this chapter is a description of the partial execution driver. A description is provided of additional memory areas and data structures. These augment BAM memory areas such as PE registers, allocate stack, control stack to facilitate partial execution. A description is given of partitioning the BAM code at two levels – viz., procedure and basic block level – and parameterizing these partitions to facilitate block-level and procedure-level analyses. Such partitioning is crucial since BAM code is RISC-like and attributing a form to such code facilitates definition of program points and program states. Techniques to identify dynamic registers and dereferencing are described.

The PE driver traverses the partitioned BAM code while collecting its run-time behavior in the augmented data structures. A combination of syntactic analyses and run-time analyses guides the partial execution with the help of a control stack and

a loop-handling mechanism. These techniques are also discussed in this chapter. Code loops are syntactically identified while parsing the BAM code. The loop recognition and handling discussed in this chapter ensures termination of partial execution process as well as limits code explosion.

Partial execution of instructions uses syntactic and run-time information extracted as discussed in this chapter. Section 5.6 described the algorithm to partially execute a basic block. We describe in the next chapter the symbolic execution and transformation of individual BAM instructions within the framework layed out in this chapter.

Chapter 6

Instruction Level Partial Execution and Analyses

While partially executing BAM code, the PE driver utilizes information relating to one or more of the following aspects of the program:

- syntactic representation of the BAM instructions;
- run-time behavior of the BAM triggered by execution of the given code;
- run-time behavior of the BAM independent of the current code sequence.

Program-related information is collected during several analyses phases and is used to control partial execution as well as to perform transformation of instruction sequences.

Symbolic execution of each of the BAM instructions is done according to the denotational semantics presented in Chapter 3. This phase of the BAM partial executor also relies on program information obtained by the program analysis phases. This chapter describes the partial execution of all BAM instructions whose denotational semantics were presented earlier. After an instruction is partially executed, the PE driver employs various techniques to ensure BAM memory correctness and to continue partial execution. These techniques use several analyses results and are described in this chapter.

6.1 PE of BAM Instructions

In general, PE of an instruction involves three steps: symbolic execution, transformation and analysis. When an instruction is not transformed to a simpler one, an *identity transformation* is said to have applied. When an instruction may be eliminated, a *nop transformation* is said to have applied. A resumption set is generated using the successor information of a block during partial execution of the last instruction in a block. The analysis associated with each instruction refines the resumption set, as necessary. The PE driver updates the control stack according to the resumption set as explained in Section 5.4.2. When an instruction refines a resumption, it is explicitly specified in the following description of instruction PE. Otherwise, it is omitted. Further, PE of instructions not described below involves only symbolic execution. The implementation of symbolic execution of such instructions ensures that the memory state is maintained correctly.

6.1.1 PE of Procedural Control Flow Instructions

6.1.1.1 PE of `procedure(P)`

- Symbolic execution: Increment `pc` by 1.
- Analysis: Perform the following steps:
  - Record current static argument register values in the in-out table associated with the current procedure.
  - Set the current value of the PE register `proc_blk` to the current block, i.e., entry block of the current procedure.
  - Save the current index into the in-out table of the calling procedure in the corresponding allocate stack item. This value is restored upon return from the current procedure so that partial execution continues in the calling procedure.
- Transformation: An identity transformation is applied.

6.1.1.2 PE of `allocate(N)`

- Symbolic Execution: Create an environment on the top of the environment stack with the current values of `e`, `cp`, `cp_par` and `cbb`, along with space for `N` permanent registers.
- Analysis: Set the `env_alloc` flag of the topmost non-dirty environment on the environment stack to indicate allocation of the environment.
- Transformation: An identity transformation is applied.

6.1.1.3 PE of `deallocate(N)`

- Symbolic Execution: Pop the environment top off the environment stack.
- Analysis: The allocate stack top corresponds to the environment just popped off. Set the allocate stack top to "dirty" indicating that the corresponding environment was popped off.
- Transformation: An identity transformation is applied.

6.1.1.4 PE of `call(N)`

- Symbolic Execution: The called procedure may be a translation of a Prolog predicate accessible during compilation, a Prolog built-in (e.g., =/2, +/2) or a Prolog predicate whose definition is unknown at compile-time but will be available at link-time. In the first case, the current value of pc is saved in cp and pc is set to the address of the call site with label `N`. In the latter two cases, partial execution assumes that the called procedure will return successfully and simply increment the pc to the next instruction.
- Analysis: The value of `cbb` is appropriately set to the successor of the current block. A new allocate stack item is created and pushed on to the allocate stack. This new item records the current block. The `env_alloc` and `alloc_dirty`

flags are set to indicate no environment allocation and no environment deallocation. A pointer to the current input-output values is also saved in the allocate stack item to enable their restoration upon successful return to the current procedure's partial execution.

- Transformation: An identity transformation is applied.

Partial execution of the `call/1` instruction distinguishes between a procedure call known at PE-time and one unknown at PE-time. Code is available for further analysis in the former case. It is conservatively assumed that the call succeeds in the later case unless the call is to global failure. This allows the PE driver to perform basic block and procedure analyses as described in Section 5.3.

6.1.1.5 PE of `return/0`

- Symbolic Execution: Restore the value of cp to pc resulting in return from a procedure call.

- Analysis: The values of `proc_blk` – current procedure to which partial execution returns – and the pointer to the current in-out values are restored from the allocate stack item. The top of allocate stack is popped. Recall that the in-out table records the argument register values of the current procedure. The input values corresponding to the current procedure will not change due to the return. Current argument register values, that represent the procedure being returned from, are registered in the in-out table. When a block is recognized as having a residue resulting from a previous partial execution (as discussed in Section 5.5.1), a call return is simulated using algorithm 5.

- Transformation: An identity transformation is applied.

6.1.1.6 PE of `jump(L)`

- Symbolic Execution: If the jump target, `L`, is a user-defined procedure, the values of pc and `bb` are set to those of the target block's first instruction and

the target block number respectively. The value of cbb is adjusted accordingly. If the jump target is a known failure label, partial execution of the instruction `fail/0` is performed. The resumption target in the resumption is indicated with a special value. This triggers the PE driver to set the control along an alternate choice and records the failure in the current choicepoint as explained in Section 5.4.2.

- Analysis: The jump target is tested for a `procedure/1` label. If it is a procedure, the jump is in fact a last call that was optimized to a jump. In such a case, an allocate block with appropriate initialization is pushed onto the allocate stack.

- Transformation: An identity transformation is applied.

6.1.2 PE of Conditional Control Flow Instructions

6.1.2.1 PE of `switch(T,R,L1,L2,L3)`

- Symbolic execution: If the tag of R is static, its symbolic execution updates pc and cbb to the jump target. Only one resumption is created. If R is dynamic, speculative PE is set up along the paths leading to blocks labelled L1, L2 and L3 respectively. This is done by creating a resumption set with three resumptions: each with an encapsulation of current BAM memory state, L1, L2 and L3 as branch targets, respectively, and (R,tvar), (R, T) and (R, none) respectively as resumption resister values. The PE driver pushes three control stack items using these resumptions.

- Analysis: No additional analysis necessary.

- Transformation: If the tag of R is static at PE-time, the instruction is transformed to `jump(L)`, where L is L1 if the datatag of R is tvar, L2 if T or L3 if any other. If R is a dynamic register, the identity transformation is applied.

6.1.2.2 PE of `test(E,T,R,L)` instruction

- Symbolic execution: If `R` is static, the `pc` is set to the address of the instruction at label `L` if either of the following is true:

  1. If `E` is `eq` and if the tag of `R` is equal to `T` or
  2. If `E` is `ne` and if the tag of `R` is not equal to `T`.

  If `R` is static, the `pc` is set to the address of the following instruction in all other cases.

  If `R` is dynamic, two speculative PE is set up – one along the path leading to block labelled `L` and the other leading to the fall-through block. This is done by creating a resumption set with two resumptions. Both of them contain an encapsulation of current BAM memory state. One resumption has the block with label `L` as resumption target and the other has fall-through block as its resumption target.

- Analysis: No additional analysis necessary.

- Transformation: If `R` is static, the instruction is transformed to `jump(L)` if the above tag tests succeed; else the instruction is transformed to `nop/0`. No transformation is done if `R` is dynamic.

6.1.2.3 PE of `jump(T,C,A,B,L)`

PE of `jump/5` occurs in one of the following three scenarios:

1. Values of the registers `A` and `B` are static and the condition specified in `C` evaluates to true.
2. Values of the registers `A` and `B` are static and the condition specified in `C` evaluates to false.
3. Values of `A` and `B` are dynamic.

- Symbolic execution: In case 1, symbolic execution results in creation of a resumption containing the block with label `L` as resumption target. In case 2, symbolic execution results in creation of a resumption containing (`ni`, `nb`) as resumption target, where `ni` is the next instruction and `nb` is the fall-through block. In case 3, speculative execution is set up by creating a resumption set containing two resumptions each with the current BAM memory state encapsulated. One resumption has the block with label `L` as its resumption target and the other has the fall-through block as its resumption target.

- Analysis: No additional analysis necessary.

- Transformation: In case 1, the instruction is transformed to `jump(L)`. In case 2, the instruction is transformed to `nop/0`. In case 3, the identity transformation is applied.

6.1.2.4 PE of `choice(I/N, Rs, L)`

- If `I = 1`,

  - Symbolic execution: a new choicepoint is created on the choicepoint stack. Apart from the current BAM memory state, the choice success information of the current choicepoint, if one exists, is saved in the choicepoint.

  - Analysis: A resumption containing the current memory state and address of next instruction as resumption target is created.

- If $1 < I \leq N$,

  - Symbolic execution: The BAM memory state is restored from the current choicepoint. The next choice is set to the label `L` in the current choicepoint.

  - Analysis: A resumption containing the restored memory state and the address of next choice in the current choicepoint as resumption target is created.

- Further Analysis and transformation: In all cases where $1 < I \leq N$, success or failure of the choice whose PE was just completed is registered in its choice success information. However, during the symbolic execution of a `choice/3` instruction with `I = N`, the choicepoint is not popped off the choicepoint stack. The retry address in the choicepoint stack top is set to a special value `FAILURE` to indicate that the choicepoint is merely left on the stack for the purposes of analysis done for choicepoint optimization. This choicepoint is popped off the stack during choicepoint optimization described in Section 6.4. Any possible choice instruction transformation is also done during this phase.

6.1.2.5 PE of `cut(R)`

The compiler generates a `move(r(b),R)` instruction that stores the value of the choicepoint stack top `b` in argument register `R` at the entry of the predicate with a cut in its body. The built-in predicate `!` is compiled to the instruction `cut(R)` which restores `b` and `hb` to their values at predicate entry thus rolling back the choicepoint state to that at the entry of predicate. This results in ignoring all choicepoints created in the body goals of the current clause, thus committing the choices made by the body goals.

- Symbolic execution: Sets the values of `b` to that stored in `R` and restores `hb` from the current choicepoint.
- Analysis: If the choicepoint stack has not grown since entry into the procedure, it follows that the value of `b` (and hence `hb`) has not changed.
- Transformation: If `b` is unchanged, the `cut/1` instruction is transformed to a `nop/0` instruction. If the value of `b` (and hence that of `hb`) has changed, no transformation is done.

The transformation of a `cut/1` instruction is illustrated using the "steadfast" version of `max/3` [55] shown in Figure 6.1. Figure 6.2 shows CFG of the BAM code generated with the GFA-based optimization turned on. The `cut/1` instructions in

```
:-option(analyze).

main :-
        read(X),
        read(Y),
        max(X,Y,Z),
        write(Z),
        nl.

max(X,Y,Z) :- X >= Y, !, Z=X.
max(X,Y,Y).
```

Figure 6.1: Definition of `max/3` Predicate

block **19** and block **15** may be transformed to `nop/0` as it can be shown at PE-time that the value of `r(b)` remains unchanged between its storage in block **5** and its restoration in blocks **19** and **15**.

6.1.2.6 PE of `trail(V)`

- Symbolic execution: Only `tvar`-tagged datawords are trailed. Thus `V` contains a PE-time dataword with `tvar` tag. Its datavalue is pushed on the trail stack.
- Transformation: None needed.
- Analysis: None needed.

6.1.2.7 PE of `fail/0`

- Symbolic execution: Untrails all variable bindings from the trail stack, restores the BAM memory state from the current choicepoint and forms a resumption containing current memory state and current retry address in the choicepoint as resumption target.
- Analysis: Several BAM memory areas are updated to assist analysis performed during partial execution of the `fail/0` instruction as described below. First, the choice success information corresponding to the current choice is updated

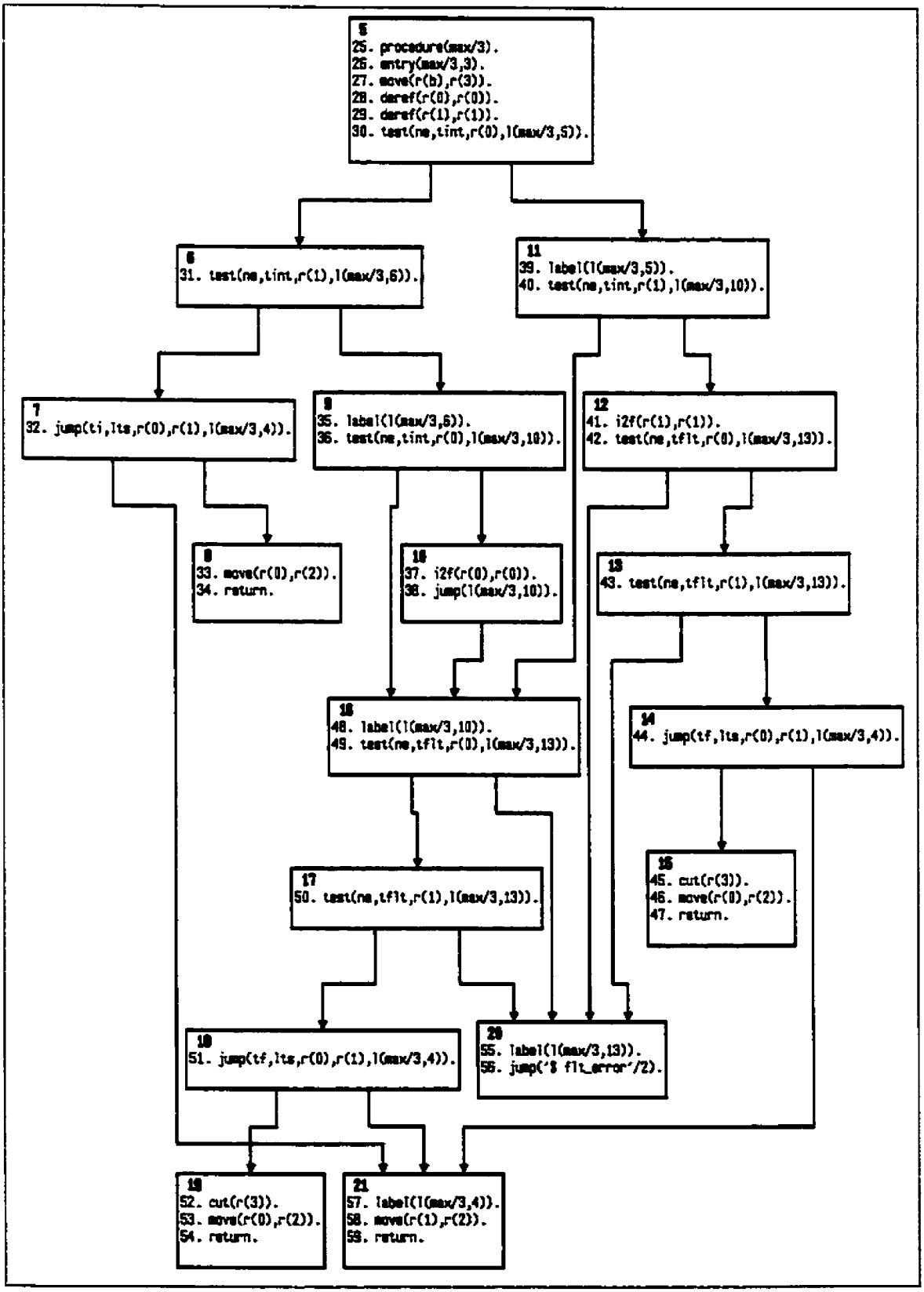

Figure 6.2: CFG of BAM Code for predicate max/3

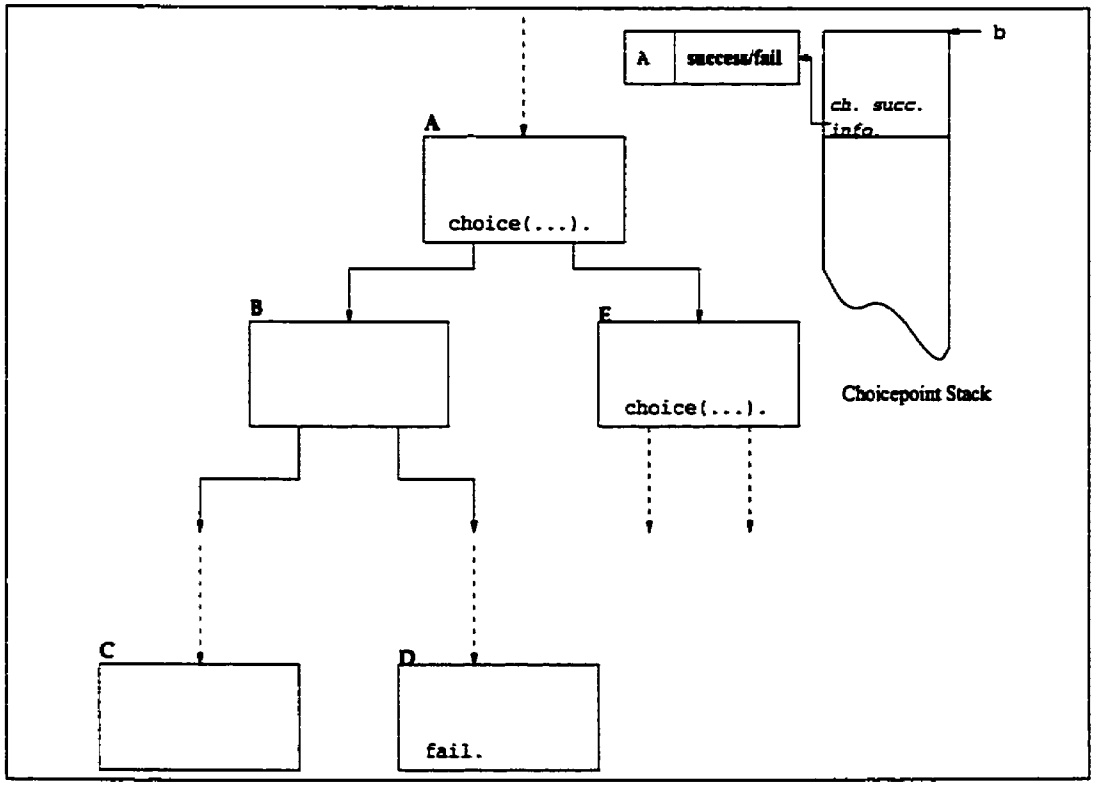

Figure 6.3: Schematic CFG to illustrate choice success update

in the current choicepoint. The current choice is marked as *failure* in the choice success information only if all previous speculative partial execution paths traversed via the current choice are also false. For example, consider the CFG structure in Figure 6.3. Let the `choice/3` instruction in block **A** be the choicepoint creating instruction (i.e., with first argument `1/N` where `N` > 1). Let the path **A-E** be the path of next alternate choice (retry address). Let **B** ... **C** and **B** ... **D** be two speculative execution paths along the first choice attempted. Partial execution along each path can eventually lead to success or to failure. The choice success information stored in the current choicepoint records a *failure* of the current choice only if PE along **all** such speculative paths results in failure. recorded. Thus *failure* is recorded in the current choicepoint during partial execution of `fail/0` instruction in block **D** only if PE along the path **B** ... **C** does not fail.

Partial execution of the `choice/3` instruction with first argument `N/N` (de-

scribed earlier) does not simulate the run-time BAM behavior of popping the top of the choicepoint stack. It is left on the choicepoint stack with the next retry address set to FAILURE to indicate completion of all speculative PE paths to be traversed via the current choicepoint. Hence, if the retry address in the choicepoint is FAILURE, the choicepoint optimization is performed.

- Transformation: This phase either transforms some of the choice/3 instructions to nop/0 or to jump/1 instructions.

6.1.3 PE of Unification Instructions

6.1.3.1 PE of deref(X,Y)

Since several subcases exist, the following discussion is not presented under the headings of symbolic execution, analysis and transformation, as done till now.

Depending on whether the content of X is a non-pointer tag or a pointer tag, there exist two possibilities for partial execution of deref(X,Y).

Consider the case when the content of X has a non-pointer tag. There exist two possibilities:

- the second operand Y is the same as the first, i.e, the instruction is deref(X,X), then the instruction is transformed to a nop/0 since at run-time the first operand dereferences to itself in the same register.

- the second operand Y is different from the first; the deref(X,Y) instruction is transformed to a move(X,Y), since no further dereferencing of the non-pointer tagged value may be done.

Consider the case where X has a pointer-tagged content that dereferences (Section 2.2.3) to a non-pointer tagged dataword V. Then the instruction can be transformed to a move(V,Y) instruction. This is similar to constant propagation as done in conventional compilers. However, the static values propagated in the current work are not restricted to known constant values – viz., fully known non-pointer

tagged datawords. The values propagated are datawords whose tags are known at PE-time. Hence partial execution may be considered as a more general technique that encompasses "conventional" constant propagation.

If the pointer-tagged dataword content of X does not dereference to a non-pointer PE-time dataword, symbolic execution and analysis steps for each of the possible pointer tags is done as described below. No transformation of instructions is possible in these cases.

- Let X dereference to a dataword with a tag tvar or dvar and data-value dv. At run-time, dereferencing tvar-tagged dataword results in either a self-referential heap location or a non-pointer tagged dataword at heap location dv. At PE-time, this is indicated by setting the second operand to dvar-tagged dataword. This dataword indicates that its run-time value is actually obtained by dereferencing of this dataword as described in Section 5.4.1.

- Let X dereference to a dataword with a tag tstr or tlst. The second operand value is set to this dataword.

6.1.3.2 PE of equal(V1, V2, L)

- Symbolic execution: If datavalues of V1 and V2 are fully known at PE time, set pc to L or to the following instruction according to the equality test. Otherwise, schedule speculative PE is performed by forming a resumption set with two resumptions, each containing the current memory state. One of them has the next instruction as resumption target and the other has the instruction label(L) as resumption target.

- Transformation: If the datavalues of V1 and V2 are fully known at PE time, the instruction is transformed to either a nop/0 or to jump(L) depending on whether the datavalues are equal. However, if the datavalues are not fully known, but have static data-tags (i.e.,data-tags other than dvar or dstr), the instruction may be transformed to nop/0 or to jump(L) depending on whether the data-tags are equal.

If the instruction is transformed to a **nop**, a resumption with current memory and the next instruction as resumption target is formed. If it is transformed to `jump/1`, a resumption with current memory and the instruction `label(L)` as resumption target is formed.

- Transformation: If the datavalues of `V1` and `V2` are fully known at PE-time and are equal, the instruction is transformed to a `nop/0`. If they are fully known but not equal, the instruction is transformed to `jump(L)`. If the datavalues of `V1` and `V2` are not fully known at PE-time, but have same static datatags (i.e.,datatags other than **dvar** or **dstr**), the instruction is transformed to `nop/0`. If they have different static datatags, it is transformed to `jump(L)`.

  If the instruction is transformed to a **nop**, a resumption with current memory and the next instruction as resumption target is formed. If it is transformed to `jump/1`, a resumption with the current memory and the instruction `label(L)` as resumption target is formed.

- Analysis: No analysis needed.

6.1.3.3 PE of `unify(R1,R2,F1,F2,fail)`

Partial execution of `unify/5` depends on the static or dynamic nature of `R1` and `R2` as described below.

Let contents of `R1` and `R2` be `T1^V1` and `T2^V2` respectively.

- Case 1:
  - Symbolic execution: If both `T1` and `T2` are **dvar**, then symbolic execution of `unify/5` trails `V`, the greater of the heap addresses `V1` and `V2`. In other words, the most recently created heap location is trailed. Then the value of the operand containing datavalue `V` is set to that of the other operand.
  - Transformation: None.

- Analysis: None.

- Case 2:
  - Symbolic execution: If only one of T1 and T2 is dvar, then symbolic execution of unify/5 trails the corresponding datavalue and sets the operand with dvar-tagged dataword to the value of the other.
  - Transformation: None.
  - Analysis: None

- Case 3:
  - Symbolic execution: If T1 is tvar and T2 is a non-tvar tag or if V1 is more recently created heap address than V2 (i.e., V1 > V2), the heap value V1 is pushed onto the trail stack and R1 is set to the value of R2.
  - Transformation: The instruction unify(R1,R2,F1,F2,fail) is transformed to the sequence:

    ```
    trail(R1).  move(R2,R1).
    ```

  - Analysis: None.

- Case 4:
  - Symbolic execution: If T2 is tvar and T1 is a non-tvar tag or if V2 is a more recently created heap address than V1 (i.e., V2 > V1), the heap value V2 is pushed onto trail stack and R2 is set to the value of R1.
  - Transformation: The instruction unify(R1,R2,F1,F2,fail) is transformed to the sequence:

    ```
    trail(R2).  move(R1,R2).
    ```

  - Analysis: None.

- Case 5:

- Symbolic execution: If `T1` is different from `T2` and neither is a `dvar` nor `tvar` tag, symbolic execution of the instruction `fail` is performed such that PE continues along alternate execution threads.
- Transformation: The instruction is transformed to `fail`.
- Analysis: None.

- Case 6:
  - Symbolic execution: If `T1` and `T2` have the same tag other than `dvar`, `dstr`, `tvar`, `tstr` or `tlst`, either `pc` is set to the next instruction or symbolic execution of instruction `fail` is done depending on whether `V1` and `V2` being equal or not, respectively.
  - Transformation: The instruction is transformed either to `nop` or to `fail` depending on whether `V1` and `V2` being equal or not, respectively. Correspondingly, the partial execution continues to the next instruction or to `fail/0`. PE will continue along alternate execution threads, if they exist, in the latter case as explained in Section 6.1.2.7.
  - Analysis: None.

- Case 7:
  - Symbolic execution: If `T1` and `T2` are either `tstr` or `tlst`, the heap locations `V1` and `V2` are unified using Algorithm 8. It adapts the classical unification algorithm [3] that facilitates unification of heap addresses containing only partial information. It unifies two heap locations h_1 and h_2 and builds any necessary heap data as much as possible to maintain correctness of partial execution.
  - Transformation: None.
  - Analysis: None.

Algorithm 8 $unify(h_1, h_2)$: *boolean*

1: Let the contents of h_1 and h_2 be `T1^V1` and `T2^V2`.
2: If `T1` and `T2` are both `dvar`, there is insufficient information to do anything further. Return success.
3: If only `T1` (`T2`) is `dvar`, trail the heap location `V1` (`V2`) and set it to `T2^V2` (`T1^V1`). Return success.
4: If both `T1` and `T2` are `tvar`, trail the most recently created heap address among `V1` and `V2` and set it to `tvar^V` where V is the greater of `V1` and `V2`. Return success.
5: If `T1` and `T2` are different non-pointer tags, or if they are same non-pointer datawords with different datavalues, unification is not possible; return failure.
6: If `T1` and `T2` are both `tstr` or `tlst`, dereference `V1` and `V2` to `W1` and `W2`. Return $unify$(`W1, W2`).

6.1.3.4 PE of `unify_atomic(V,A,fail)`

1. If `V` contains a static non-`tvar` or non-`dvar` tagged value and the contents of `V` are the same as `A`:
   - Symbolic execution: The pc is incremented. A resumption with current memory state and the next instruction as resumption target is formed.
   - Transformation: The instruction is transformed to a `nop/0`.
2. If either the (non-`tvar` and non-`dvar`) datatag or the datavalue of the content of `V` is different from that of `A`:
   - Symbolic execution: PE of the instruction `fail/0` is carried out.
   - Transformation: The instruction is transformed to `fail/0`.
3. If `V` contains a PE-time datatag:
   - Symbolic execution: The pc is incremented. The datavalue is trailed and `V` is set to `A`.
   - Transformation: Not done.
4. Analysis: No additional analysis is needed.

6.1.3.5 PE of `move(S,D)`

- Symbolic execution: The PE-time contents of `S` are moved into `D`. The `pc` is incremented.
- Transformation: None needed.
- Analysis: None needed.

6.1.3.6 PE of `push(S,R,N)`

- Symbolic execution: The `pc` is incremented. The PE-time contents of `S` onto the stack with stack pointer `R`.
- Transformation: None needed.
- Analysis: None needed.

6.1.3.7 PE of `adda(S,O,D)`

- Symbolic execution: The `pc` is incremented. If `S` is static, `D` is set to a value whose tag is that of `S` and data value is (datavalue of `S` + `O`). If `S` is not static, only the tag of `D` is set to that of `S`.
- Transformation: None needed.
- Analysis: None needed.

6.1.4 PE of Arithmetic Instructions

Partial execution of arithmetic instructions mainly involves symbolic execution. The instruction is transformed only if the block containing it is not part of a program loop. Program loops may be unrolled to achieve the effect of classical loop unrolling [54] as discussed in Section 5.5. Since the present work does not perform limited loop unrolling, arithmetic instructions in a program loop are only symbolically executed and not transformed.

However, if an arithmetic instruction is not in a block that is part of a program loop and its operands have static datavalues then it is transformed into `move(S,D)` instructions where `S` is the result of the arithmetic computation to be done by the instruction and `D` is the destination of the operation. Additionally, the following cases are handled accordingly during partial execution of arithmetic instructions with one static operand that is the identity value for that operation:

- Either of the source operands of an `add/4` or `sub/4` instruction has a static non-pointer tag and a datavalue of 0.
- Either of the source operands of `mul/4` has a static non-pointer tag and a datavalue of 1 or 0.
- The numerator operand of `div/4` has a static non-pointer tag and a datavalue of 0.
- The denominator operand of `div/4` has a static non-pointer tag and a datavalue of 0. This is transformed to a jump to arithmetic failure.
- Either of the source operands of an `and/4` instruction has a static non-pointer tag and a datavalue of 0.

6.2 Maintaining BAM Memory Correctness

The BAM memory state at any given program point during partial execution is an abstraction of its corresponding run-time state at that program point. By abstraction, we mean the following. The register contents either have the same data tags or a dynamic tag; heap locations differ only by the size of the run-time data structure. This is illustrated using the following code whose BAM code CFG is shown in Figure 6.4.

```
main :- read(X), p(X,Y), write(Y).


p(a,1).
```

If the PE-time BAM memory state abstracts the corresponding run-time BAM memory state at a given program point, then we say that BAM memory correctness is maintained at that program point. The BAM memory correctness is said to have been maintained for a given program if is correctness is maintained at every program point in the program. We further illustrate in this example the use of SCC information to ensure the correctness of an instruction transformation involving a static operand.

Consider the content of the register `r(0)` at the entry of block **1**. Its contents have the same PE-time and run-time data tag, viz., `tvar`. Now let us consider the heap. Instruction 6 in block **0** pushes a dataword onto the heap. The call to `read/1` builds datawords on the heap that are unknown at PE time. Instruction 12 in block **1** pushes another dataword. At PE-time there are no datawords between the datawords pushed by instructions 6 and 12. However, at run-time they are separated by datawords pushed by the call to `read/1`. Top of the heap, `r(h)`, at entry into block **3** at PE-time is different from that at run-time for the same reason. The partial execution algorithm does not assume the size of data that might be written on the heap by calls whose code is not available. Instead PE continues with the present value of `r(h)`. In this example, `r(h)` contains 0 both before and after the call to `read/1`, which is assumed to succeed. The heap is adjusted later to approximate its run-time state once more information about the heap location becomes available (Section 6.2.1).

In the present example, since `tvar^r(h)` is a static term, it is possible to transform instruction 3 to `move(tvar^0,r(0))` and consequently consider `r(0)` to be static. However, `r(h)` points to a BAM memory area whose PE-time and run-time values differ. So, the above transformation is not performed because it does not preserve the correctness of BAM memory. Transformations involving only static datavalues are performed and those involving pointers to BAM memory areas are not performed.

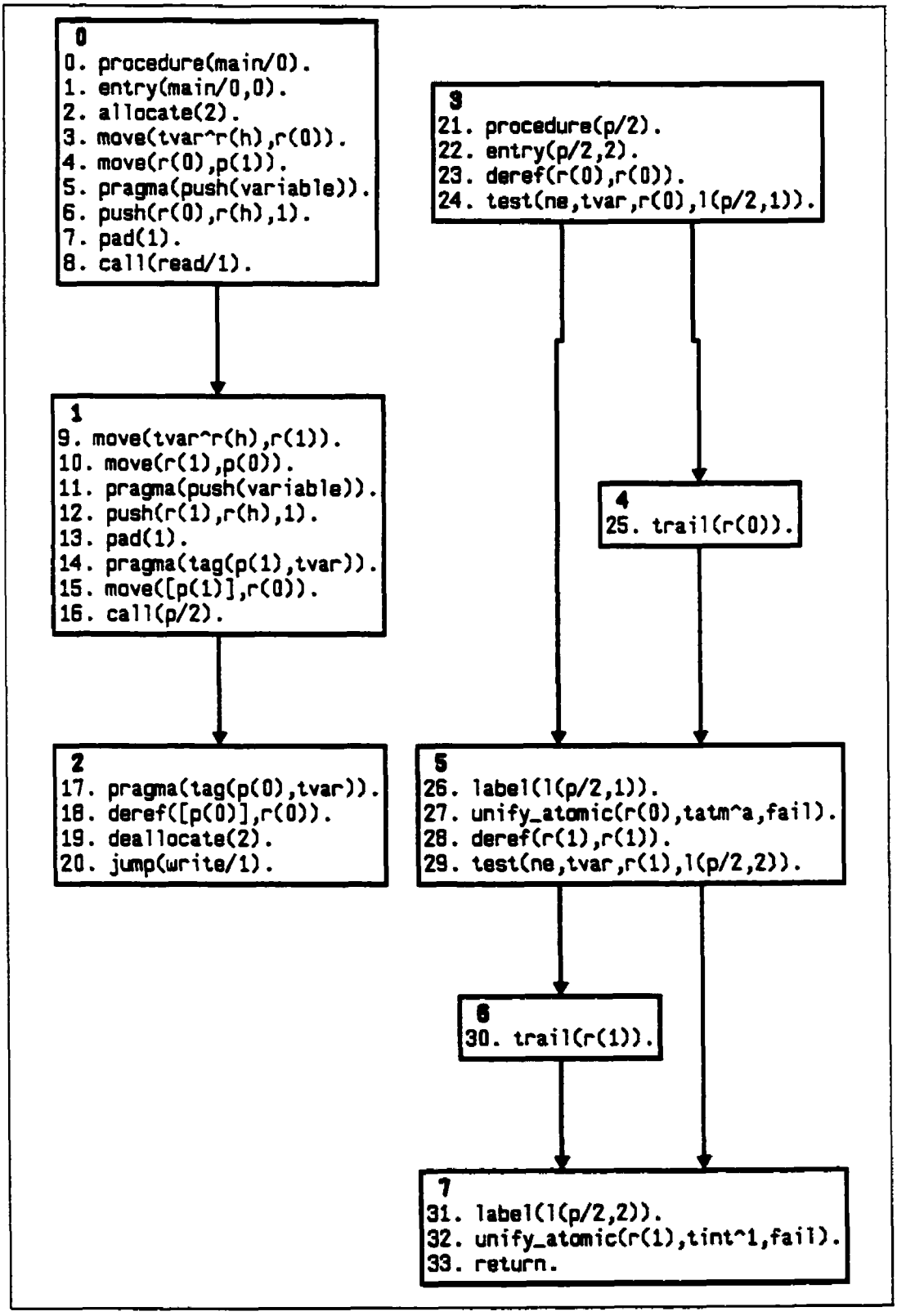

Figure 6.4: Example to illustrate transformation of an instruction with static pointer-tagged operand

6.2.1 Speculative Partial Execution

BAM memory correctness has to be maintained not only at every program point but also before and after partial execution of each instruction. Symbolic execution of instructions as detailed earlier in the chapter maintains correct BAM memory state. Resumption plays a crucial role in ensuring correctness of BAM memory state at the entry and exit of basic blocks. This section discusses techniques employed to ensure BAM memory correctness when a block is partially executed speculatively. On the other hand let **S** be a block not being speculatively partially executed speculatively and let **P** be its parent from which PE control reached **S**. Partial execution of **S** is simply started with the memory state encapsulated in the resumption at the end of PE of **P**. We refer to such PE as deterministic partial execution.

Speculative partial execution is set up when a retry address is being attempted or when a conditional control instruction involves a dynamic register. With speculative PE, more than one block successor is partially executed. Hence, additional argument register values must be set along each execution path. For example, assume the flow change instruction in the current block being partially executed is `switch(T,R,L1,L2,L3)` with a dynamic operand `R`. As explained in Section 6.1.2.1, a resumption set with three resumptions each containing a copy of the current memory state results from its PE. Further, two of the three resumptions correspond to flow control along blocks labeled `L1` and `L2`. These resumptions indicate that `R` has a tag `tvar` and `T`. The third resumption indicates failure.

The resumption set is used to form a **CStack** item. The resumption registers are stored in the **CStack** frame as a set of argument register-dataword pairs. The **CStack** top indicates the block to be partially executed and the memory state in which it needs to be partially executed. The memory state is set to that indicated in the **CStack** top. Resumption register information is used to adjust the memory state set using Algorithm 9. This phase ensures that the heap at PE-time is correctly approximated to that at run-time. The value of each register `r` being set depends on the addressing mode of `r` as well as its current content. Since speculative PE

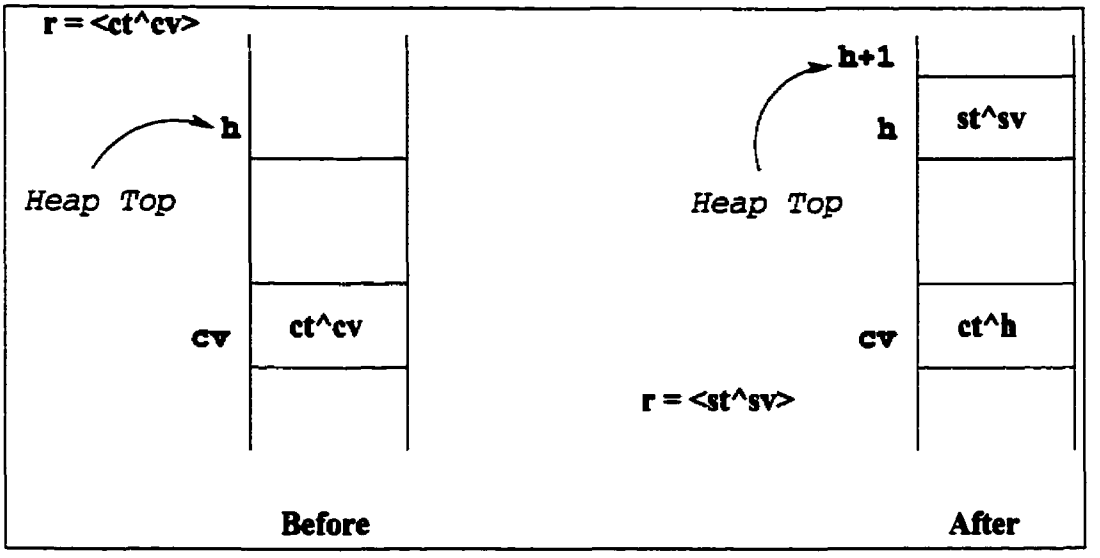

Figure 6.5: Setting a non-state register r to a tvar-tagged dataword

is scheduled, each r is a dynamic register. If r is an immediate operand then it is considered to be a heap address and the heap location r is set to st^sv; if r is r(h) or r(b) then the register r(h) or r(b) is set to st^sv accordingly. If r is neither a state register nor an immediate operand then the heap is built to reflect its run-time state depending on the current value of r. Algorithm 10 is used to build the heap. The case constructs used in this algorithm do not fall through to the next case. Figures 6.5 and 6.6 depict the heap before and after setting r to a tvar-tagged dataword and tatm-tagged functor dataword, respectively.

Algorithm 9 Algorithm to set memory correctly for speculative partial execution

1: **for all** (r, st^sv) in resumption register list **do**
2: Let r be the argument register.
3: Let the current content of r be ct^cv.
4: Let h represent the current heap top i.e., content of the register r(h).
5: **if** r is an immediate operand **then**
6: Set heap location h to st^sv; increment content of r(h) by 1.
7: **else if** r is r(h) or r(b) **then**
8: Set the value of the register r(h) or r(b) to st^sv.
9: **else**
10: Adjust heap and set the non-state register r using Algorithm 10.
11: **end if**
12: **end for**

Algorithm 10 Algorithm to adjust the heap while speculatively setting a dynamic non-state register

```
1: switch (st)
2:   case TPOS or TNEG or TINT or TINT :
3:     Set heap location h to contain st^sv.
4:     Point heap location cv to h i.e., set the contents of heap location cv to
       ct^h; increment content of r(h) by 1.
5:     Set r to contain st^sv.
6:   case TVAR :
7:     Set the value of sv to h.
8:     Set heap location h to contain st^sv.
9:     Point heap location cv to h i.e., set the contents of heap location cv to
       ct^h;
10:    Ensure the tag of the contents of heap location ct is TVAR; increment content
       of r(h) by 1.
11:    Set r to contain st^sv.
12:  case TATM :
13:    if sv is known and is of the form f/n then
14:      Set heap location h to contain st^sv.
15:      Point heap location cv to h i.e., set the contents of heap location cv to
         ct^h;
16:      Ensure the tag of the contents of heap location ct is TSTR;
17:      Create n self-referential TVAR-tagged datawords on the heap starting at
         heap location h; increment content of r(h) by n.
18:      Set r to contain st^sv.
19:    else
20:      Set heap location h to contain st^sv.
21:      Point heap location cv to h i.e., set the contents of heap location cv to
         ct^h; increment content of r(h) by 1.
22:      Set r to contain st^sv.
23:    end if
24:  case TLST :
25:    Set heap location h to contain TLST^h+1; increment content of r(h) by 1.
26:    Point heap location cv to h i.e., set the contents of heap location cv to
       ct^h;
27:    Create two self-referential TVAR-tagged datawords on the heap starting at
       heap location h; increment content of r(h) by 2.
28:    Set r to contain st^sv.
29:  case TSTR :
30:    Set heap location h to contain DSTR^h; increment content of r(h) by 1.
31:    Point heap location cv to h i.e., set the contents of heap location cv to
       ct^h.
32:    Set r to contain DSTR^h; increment content of r(h) by 1.
33: end switch
```

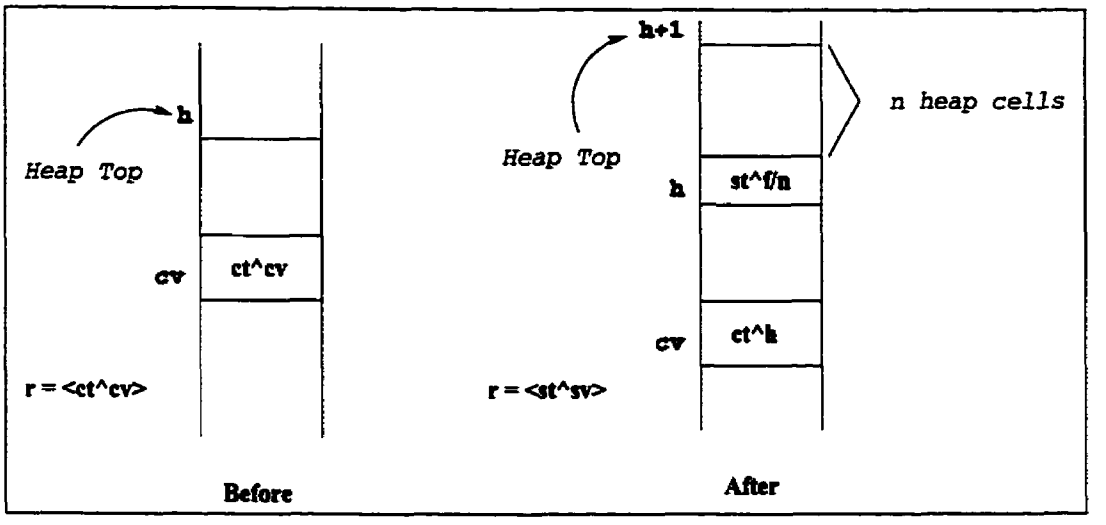

Figure 6.6: Setting a non-state register r to a tatm-tagged dataword

6.3 Memory Correctness When a PE-Loop Exists

BAM memory state needs to be maintained correctly as part of the loop handling mechanism explained in Section 5.5. This is done with the help of the allocate stack as explained in this section.

A loop detected during partial execution is deemed to be a return out of the current procedure. In other words, it is assumed that there is a run-time path that eventually succeeds and returns from the current procedure. The memory state is set to reflect this assumption. Although this is true for most general cases, exceptions arise and the memory state is set accordingly as explained in the following.

To illustrate such a situation, let *blk* be the block which is to be partially executed, and blk_{res} the residue of *blk* resulting from its previous PE with respect to the current state of static focus registers. This implies that a loop has been detected and no further PE of *blk* is necessary. The previous PE could have indicated that the run-time execution of the block *blk* would result in a success or failure.

- PE of previous block indicates run-time success, the most general case. Then, PE proceeds by returning from the current procedure. The simulated return from the procedure performs the following:

  – Sets the environment stack in preparation for the partial execution to

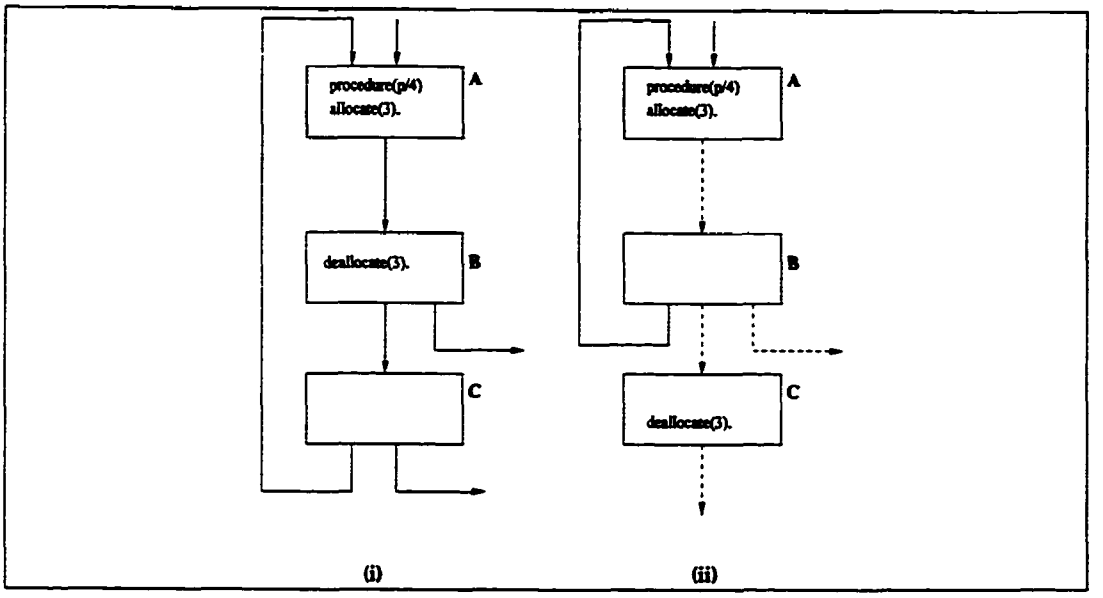

Figure 6.7: Schematic illustration of PE-loops detected

continue at instruction cp, the first instruction the block cbb.

- Sets the correct values of PE registers.

Partial execution of `return/0` simply sets the value of pc to that of cp (see Section 6.1.1.5).

The current procedure might or might not have created an environment. In the first case when one is created, the PE-loop could occur either before or after deallocation of the current procedure's environment, as shown in Figure 6.7. Partial execution of `deallocate/1` sets the *alloc_dirty* flag on the top allocate stack item to indicate a deallocation of the environment. The values of cp and cbb are restored off the top of environment stack if the top allocate stack item indicates an environment was created by the current procedure and that environment was not deallocated prior to the loop. This ensures the correctness of the register values for the correct PE of `return/0`.

In the second case, an environment is not created by the current procedure. Since the allocate stack top indicates no environment creation for the current procedure, the current values of cp and cbb are correct and need not be restored from the environment. Thus the partial execution of `return/0` pro-

ceeds correctly. The environment adjustment and continuation register update is shown in Algorithm 11.

- Previous block PE indicates possible run-time failure as illustrated with the help of the following basic block.

```
move([r(0)],r(1)).
move([r(0)+1],r(0)).
equal(r(1),tint^7,fail).
```

  Suppose that the tag of `[r(0)]` is `tlst`. Then `equal(r(1), tint^7, fail)` will be transformed to `fail` and partial execution of the block reveals run-time execution of the block will fail.

  In this situation, the failure is recorded in the current choicepoint as detailed in Section 6.1.2.4 and **CStack** is not updated allowing PE to continue in the depth-first order. Thus PE proceeds along either an alternate speculative execution path available in the current choice or an alternate choice if one is available. In case no more alternate choices exist in the current choicepoint, PE continues by popping the top of the **CStack**. The allocate stack is adjusted either by a `return/0` or a simulated return out of the current procedure.

6.4 Choicepoint Optimization

The basic idea behind the choicepoint optimization is to transform `choice/3` instructions such that run-time execution does not attempt choices along which execution is known to fail at PE-time. The PE driver uses Algorithm 12 to perform this optimization as described below.

Let **ChPt** be the current choicepoint. Let **B** be the set of blocks containing `choice/3` instructions that access **ChPt**. Let $\mathbf{ChS} = \{(B_i, R_i) \mid B_i \in \mathbf{B}\}$, where R_i is either *success* or *failure* indicating success or failure of PE along all paths

Algorithm 11 Setting BAM memory upon loop detection

```
1: if blk_res indicates a local failure then
2:    Simulate local failure by partial execution of fail/0 (Section 6.1.2.7).
3: else if olk_res indicates a global failure then
4:    Set partial execution path along the next alternative in the depth first order.
5: else
6:    if alloc_flag of itm indicates environment allocated in current procedure
      then
7:       if alloc_dirty of itm is not set then
8:          Restore cp, cbb and the PE registers from top environment stack item.
9:          pop environment stack.
10:      end if
11:   end if
12: end if
13: if blk has a choice/3 instruction then
14:   Adjust next choice in current choicepoint.
15: end if
16: Partially execute return/0
```

accessible via the fall-through edge of block $\mathbf{B_i}$. Assume the set **ChS** is sorted in the order the partial executor attempts to execute blocks **B**. For any two blocks $\mathbf{B_j}$, $\mathbf{B_k} \in \mathbf{B}$ such that the label of $\mathbf{B_j}$ is a retry address that is partially executed before that of $\mathbf{B_k}$, there exists an ordering between $\mathbf{B_j}$ and $\mathbf{B_k}$. The notation $\mathbf{B_j} < \mathbf{B_k}$ means that zero or more blocks in **B** might have been partially executed after $\mathbf{B_j}$ and before $\mathbf{B_k}$. $\mathbf{B_{i+j}}$ and $\mathbf{B_{i-j}}$ indicate the block tried j blocks after and before trying $\mathbf{B_i}$, respectively, during partial execution. $\mathbf{B_1}$ indicates the block that contains the `choice/3` instruction that creates the choicepoint on the stack.

Figure 6.8 shows a sequence of `choice/3` instructions. The sequence of blocks containing the choice instructions correspond to the choicepoint created by the `choice/3` instruction in block **A**. The sequence `L1, L2, ..., Ln-1` represents retry addresses attempted during PE. If, for example, PE along the edge marked $\mathbf{P_A}$ fails and that along the edge marked $\mathbf{P_B}$ succeeds, the `choice/3` instruction in block **A** may be transformed to `jump(L1)`. A complete choicepoint optimization was illustrated for the code shown in Figure 4.10 in Section 4.4.2.

Algorithm 12 Choicepoint optimization

1: Find the block $\mathbf{B_f}$ such that $(\mathbf{B_f}, \mathit{failure}) \in \mathbf{ChS}$ and $\forall i$ where $\mathbf{B_i} < \mathbf{B_f}$, $(\mathbf{B_i}, \mathit{success}) \in \mathbf{ChS}$.
2: Find the block $\mathbf{B_s}$ such that $(\mathbf{B_s}, \mathit{success}) \in \mathbf{ChS}$ and $\forall i$ where $\mathbf{B_f} < \mathbf{B_i}$ and $\mathbf{B_i} < \mathbf{B_s}$, $(\mathbf{B_i}, \mathit{failure}) \in \mathbf{ChS}$.
3: **while** ($\mathbf{B_s}$ exists) **do**
4: **if** ($\mathbf{B_f}$ is $\mathbf{B_1}$) **then**
5: Let choice(1/N, $\mathtt{R_f}$, $\mathtt{L_f}$) be the choicepoint instruction in $\mathbf{B_f}$.
6: Let choice(I/N, $\mathtt{R_s}$, $\mathtt{L_t}$) be the choicepoint instruction in $\mathbf{B_s}$ and $\mathtt{L_s}$ be the label of $\mathbf{B_s}$.
7: Transform choice(1/N, $\mathtt{R_f}$, $\mathtt{L_f}$) in $\mathbf{B_f}$ to jump($\mathtt{L_s}$).
8: Transform choice(I/N, $\mathtt{R_s}$, $\mathtt{L_t}$) to choice(1/N, $\mathtt{R_f}$, $\mathtt{L_t}$).
9: Let M be an empty list of instructions.
10: **for** $i = 1$ to *length*($\mathtt{R_f}$) **do**
11: **if** ($\mathtt{R_f}$[i] $\neq$ $\mathtt{R_s}$[i] $\bigwedge$ $\mathtt{R_s}$[i] $\neq$ no) **then**
12: Append the instruction move($\mathtt{R_f}$[i], $\mathtt{R_s}$[i]) to M.
13: **end if**
14: **end for**
15: Insert M after the choice/3 instruction in block $\mathbf{R_s}$, if M is not empty.
16: **else**
17: Transform choice(I/N, $\mathtt{R_i}$, $\mathtt{L_i}$) in $\mathbf{B_{f-1}}$ to choice(I/N, $\mathtt{R_i}$, $\mathtt{L_s}$), where $\mathtt{L_h}$ is the label of $\mathbf{B_s}$.
18: **end if**
19: Find the block $\mathbf{B_k}$ such that $(\mathbf{B_k}, \mathit{failure}) \in \mathbf{ChS}$ and $\forall i$ where $\mathbf{B_f} < \mathbf{B_i} < \mathbf{B_k}$, $(\mathbf{B_i}, \mathit{success}) \in \mathbf{ChS}$. Set $\mathbf{B_k}$ to be $\mathbf{B_f}$.
20: Find the block $\mathbf{B_n}$ such that $(\mathbf{B_n}, \mathit{success}) \in \mathbf{ChS}$ and $\forall i$ where $\mathbf{B_f} < \mathbf{B_i}$ and $\mathbf{B_i} < \mathbf{B_n}$, $(\mathbf{B_i}, \mathit{failure}) \in \mathbf{ChS}$. Set $\mathbf{B_n}$ to be $\mathbf{B_s}$.
21: **end while**

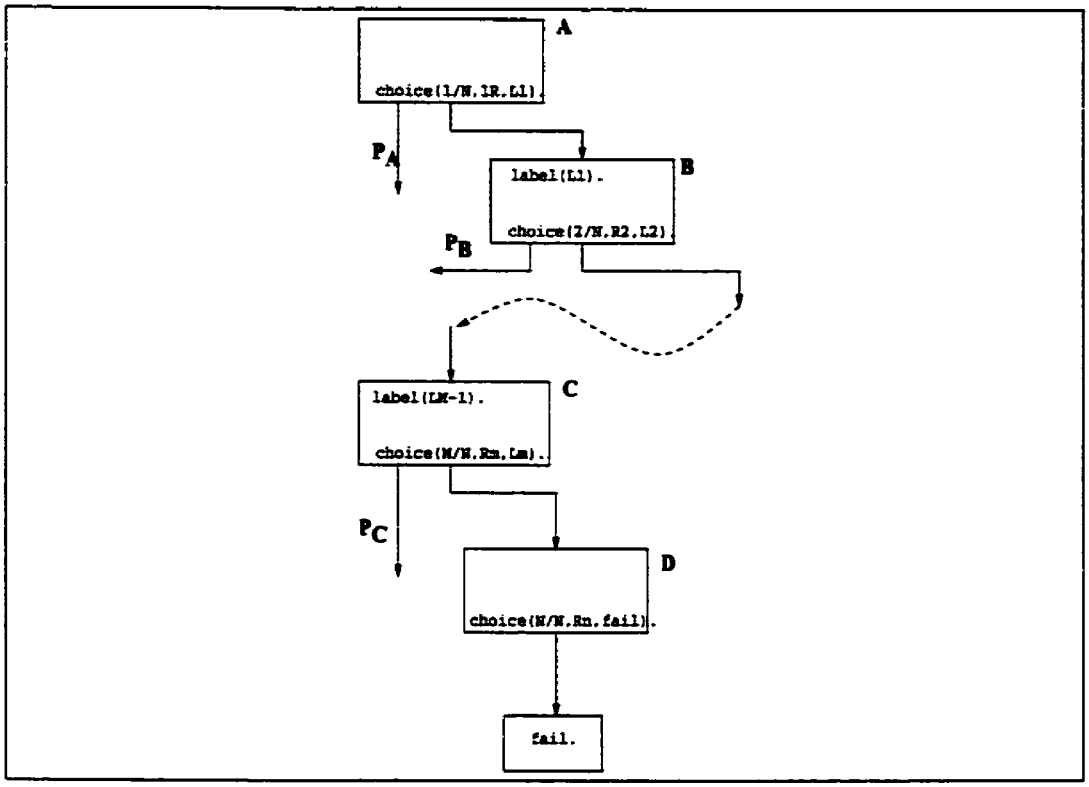

Figure 6.8: Schematic CFG to illustrate choicepoint optimization

6.5 Summary

This chapter provides the details of partial execution of BAM instructions. The run-time semantics of BAM instructions (Chapter 2) are extended to accomodate run-time analyses whose results are used by the PE driver to affect transformation/optimization appropriately. Further, the crucial aspect of maintaining the PE-time memory state to be an abstraction of the corresponding run-time state is discussed. The choicepoint optimization algorithm used is presented. The next chapter discusses consolidation of the residues generated during partial execution along with benchmarking of the resulting residues.

Chapter 7

BAM Code Regeneration

Partial execution described in the earlier chapters results in residues for each basic block. This chapter describes the last step of the PE process which includes regeneration of BAM code of all the block residues. The regenerated BAM code of a given program is referred to as *program residue*. First a description is given of simple mechanism that is used to lay out BAM code in the residue blocks into a file. The code lay out is done according to the PE-flow graph.

7.1 Code Consolidation

As described in Chapter 5, the PE driver records the control flow between all residue blocks in the PE-flow graph. The partial execution of a block may or may not result in a block that is different from the original. Yet, the resulting block is referred to as the residue. The PE-flow graph may hence contain basic blocks in addition to those in the CFG of the original BAM code. New control flow edges from and to any new blocks are also recorded in the PE-flow graph. If the PE of a given program does not result in a residue for even one basic block, then the PE-flow graph is identical to the original CFG.

Algorithms 13, 14 and 15 is used to regenerate BAM code for the whole program after the completion of partial execution. It uses the information stored in both PE-flow graph and the original CFG. The resulting BAM code is input to the Aquarius Prolog compiler which assembles and links the object file to generate an executable.

Algorithm 13 Top-level loop for BAM code regeneration

```
1: Let BStack be a global stack of basic block numbers; ProcList be the global
   list of procedures that is initially empty; R be the root of PCFG, the PE-flow
   graph.
2: Add R to end of ProcList.
3: for each (procedure block B ∈ ProcList) do
4:    Push residue of B onto BStack.
5:    while ( BStack is not empty) do
6:       Pop the top of BStack into B
7:       regenerate_code(B)
8:    end while
9: end for
```

The top-level loop shown in Algorithm 13 performs a depth-first traversal of all procedures in the PE-flow graph, **PCFG** and maintains two data-structures viz., **BStack** and **ProcList**. **BStack** is the stack of basic blocks used for depth-first traversal of a given procedure. **ProcList** is a list of procedures still to be traversed for code regeneration.

Following are the important issues related to the BAM code regeneration algorithm. Firstly, at the time of partial execution of a flow change instruction containing a label operand, say `Lbl`, it is not known if partial execution of the block with the label `Lbl` will result in a residue different from the original. This necessitates a post-residue generation patch up phase wherein the control flow is correctly set. The only exception where the target labels of an instruction are correctly set during partial execution is for `choice/3` instructions. This is done during choicepoint optimization phase. Consequently, in Algorithm 14 that the label `choice/3` instruction is not changed during code regeneration. Algorithm 15 replaces the input instruction I with the transformed one, if any.

Next, the flow change instructions `return/0` and `fail/0` neither have successors nor label operands. Thus they are simply emitted without any change during code regeneration of blocks containing these instructions. The CFG does not record the called procedure block as a successor to the block with `call/1`. Such a calling procedure block, however, has one and only one successor block whose leader is the instruction executed after the call returns. Code regeneration for block with a

Algorithm 14 Algorithm to regenerate BAM code of a basic block

*regenerate_code(**B**: block)*:**SB**: block

```
1: if (code for B is not regenerated) then
2:   Emit all instructions except the last instruction, li, in B.
3:   switch (li)
4:     case jump(L) :
5:       if (L is not fail) then
6:         SB := transform_instruction(B, li, L).
7:         if (SB is a procedure block) then
8:           Add SB to the end of ProcList.
9:         else
10:          Push SB onto BStack.
11:        end if
12:      end if
13:    case call(L) :
14:      SB := transform_instruction(B, li, L).
15:      Add SB to the end of ProcList.
16:      Push the successor of B onto BStack.
17:    case choice(I/N, R, L) :
18:      if (L is not fail) then
19:        Push block with label L onto BStack.
20:      end if
21:      Push the fall-through successor of B onto BStack.
22:    case equal(X, Y, L) or test(E, T, X, L) or jump(T, C, X, Y, L) :
23:      if L is not fail then
24:        SB := transform_instruction(B, li, L).
25:        Push SB onto BStack.
26:      end if
27:      Push the fall-through successor of B onto BStack.
28:    case switch(T, R, L1, L3, L3) :
29:      for all L ∈ {L1, L2, L3} do
30:        if (L is not fail) then
31:          SB := transform_instruction(B, li, L).
32:          Push SB onto BStack.
33:        end if
34:      end for
35:      Push the fall-through successor onto BStack.
36:    case unify(X, Y, T, U, L) or unify_atomic(X, Y, L) :
37:      if (L is not fail) then
38:        Push block with label L onto BStack.
39:      end if
40:      Push the fall-through successor onto BStack.
41:  end switch
42:  Emit li and mark that code for B is regenerated.
43: end if
```

Algorithm 15 Algorithm to regenerate a BAM instruction

transform_instruction(**B**: *block*, **I**: *instruction*, **L**: *label*): *block*

1: Let **T** be the block whose label is **L** in the original CFG.
2: Let **T'** be the block whose parent is **B** and is a residue of **T** as recorded in **PCFG**.
3: **if** (**T'** is different from **T**) **then**
4: Form a new label unique $\mathbf{L}_{new}$.
5: Change the label of the block **T'** to $\mathbf{L}_{new}$.
6: Replace label operand **L** with $\mathbf{L}_{new}$ in instruction **I**.
7: **end if**
8: Return **T'**.

`call/1` instruction is handled accordingly. Due to last call optimization(LCO), the target of a `jump/1` instruction might be a procedure. Code regeneration for block with `jump/1` takes this into consideration.

The root CFG block which is the program entry block may have at most one version that is different from itself since the block will be entered only once during PE resulting in the only time a residue is generated for the program entry block. Finally, a separate pass is made over the code generated to remove any unconditional jumps to the next instruction. The resulting BAM code is used to generate an executable.

7.2 Summary

The present work uses the SPARC port of Aquarius Prolog compiler. Its back-end compiles BAM code to SPARC assembly instructions which in turn are assembled by the SPARC assembler [35] to generate a native executable. The resultant of code regeneration phase, described in this chapter, is a stream of BAM instructions. BAM instructions are Prolog terms and can be compiled by Aquarius compiler which directly invokes the back-end and the assembler to produce a native executable. An evaluation of the partial execution process is described in the next chapter by compiling some benchmark programs to native executables.

Chapter 8

Evaluation of BAM Code Partial Execution

In this chapter, we present an evaluation of the partial execution described in the earlier chapters. First we present an discussion to provide the context for the benchmarking done to evaluate the work. Then we present the benchmarking methodology and the results.

8.1 Evaluation Context

In an increasingly complex world of programming languages and processor architectures, high performance of applications developed that use these languages is achieved by a combination of several architecture-independent and architecture-independent compiler optimizations. It is well-known that a combination of various analysis and optimizations is needed to expose further optimizations in the later phases of a compiler [15]. A specific set of optimizations by themselves rarely produce a highly optimized executable.

The principal focus of the present work is to study the issues involved in design and implementing a low-level language partial evaluator that enables several common compiler optimizations. Ideally, the benefits of the set of optimizations performed by partial evaluation would be fully evident along with low-level optimizations. Conventionally, a high-level language compiler builds more than one internal representation of the code in various phases. The low-level optimizations are performed before the executable is written out. To implement assembly-level

optimizations we either need to interface with the existing SPARC back-end and/or assembler, or implement an assembler optimization phase that performs low-level optimizations. Neither of these is feasible given that this work is a single-person project. Hence there is no phase that performs any machine-specific optimizations, such as code motion, inter-procedural code scheduling, software pipelining, to name a few.

The lack of a machine-specific optimizer can prove to be a handicap in generating a fully optimized executable thereby hiding the real performance improvement due to partial execution. The need for such a phase is even more evident when the program residue is larger than the original, as is often the case when PE is applied. The code size can affect the load time [27] of the executable. The additional code will also affect the code layout which in turn can degrade the performance due to instruction and/or data cache access patterns, despite any speed-up achieved due to partial execution. A machine-specific code motion optimization can alleviate this problem whenever possible.

Table 8.1 gives a list of the programs used for benchmarking the partial executor implementation described in this thesis. These programs are taken from the benchmarks used in presenting the performance of the Aquarius Prolog compiler [67]. The Aquarius benchmark suite consists of "examples of realistic programs during computations representative of Prolog" [67]. We have chosen some small and medium-sized programs that facilitate manual verification of correctness the entire PE process and the generated BAM and SPARC assembly code. Keeping the lack of machine-specific optimizer in context, the performance of these programs was measured to get an indication of the potential speedup PE optimizations can produce.

8.1.1 Benchmarking Methodology

Here we describe the process used to perform partial execution on benchmark programs. This process is illustrated using one of the benchmarks, `qsort.pl` that implements quick sort. This program contains the definition of `qsort/3` and of

the program entry predicate [1] that has no arguments and has a call to `qsort/3` in its body with the appropriate arguments instantiated. We refer to the predicate `qsort/3` as the *top-level predicate*. The program entry predicate and the top-level predicate are separated into two Prolog files. The file containing top-level predicate is compiled using global flow analysis to BAM code, say `orig.b`. The partial executor is run with `orig.b` as input resulting in a BAM code residue, `residue.b`. Then, `residue.b` is compiled along with the file containing the program entry predicate call resulting an executable corresponding to residue. The performance of this executable is compared with that resulting from compiling the file with entry predicate and the top-level predicate. The following alternate compilation may also be employed. The entire original Prolog source containing both the top-level and entry predicates may be compiled to BAM code using global flow analysis. This is followed by partial execution of the BAM code to yield a residue. The residue is then compiled using the Aquarius Prolog compiler to result in an executable. However, this does not provide the correct measure of effectiveness of partial execution since both GFA and PE have the additional information about the modes with which of the top-level predicate are called. This information is not necessarily available in general programs. Thus this manner of compiling benchmarks was not employed.

The benchmarks were each run 10 times on a lightly-loaded Sun SPARCstation 10/30 with 64MB of memory. The best and worst of these 10 times are discarded and the rest were averaged to eliminate any extraneous machine states that are not in a typical run of the program. The benchmarking results are presented in Table 8.2. The correctness of all the transformations done by the partial executor and the output of all the programs was manually verified.

By performing PE on `allperms` and `fibo`, redundant conditional instructions and consequently dead-code were discovered. This resulted in a final executable with 90 and 140 SPARC instructions lesser, respectively. The deadcode corresponds to BAM code in a never taken subgraph of an execution thread.

[1] The Aquarius compiler considers `main/0` as the program entry predicate by default

Table 8.1: Benchmarks	
Program Name	Description
allperms	Computing permutations of integers in three ways - insertion; reverse and append; findall(41 lines)
fibo	Computing Fibonacci number with starting value of 0 (21 lines)
tak	Recursive integer arithmetic (12 lines)
ops8	Symbolic differentiation (25 lines)
queens_8	Solve 8 queens puzzle (all solutions)
zebra	A logical puzzle based on constraints (37 lines)
qsort	Quick sort of a list of numbers (11 lines)

Table 8.2: Execution times		
Program	With PE phase	Without PE phase
allperms		
with insertion	0.073s	0.074s
with reverse and append	4.01s	4.42s
with findall	71.54s	78.6s
fibo	36893ms	38188ms
tak	120ms	120ms
ops8	0.042ms	0.032ms
queen_8	26.1ms	28.5ms
zebra	130ms	100ms
qsort	0.74ms	0.75ms

No PE-based optimization was possible on **tak**. This benchmark was chosen to demonstrate that not all programs benefit from partial execution. It has been observed that the program has very few alternate execution threads. Consequently, PE-based optimizations are not possible in its BAM representation. As with all compiler optimizations, all programs may not benefit from PE-based optimizations. We observe that programs with several alternate execution threads can potentially benefit from such optimizations.

A performance degradation was observed due to PE of **ops8** and **zebra**. A residue that increased the code size by approximately 50% and 20%, respectively. A visual inspection of SPARC assembly reveals that code layout and the call-graph of the residue's executable are different from those of the original. Given the manual verification done of the correctness of PE transformations, we believe that the difference in code layout and call-graph contribute to the degradation in performance. It was not possible to verify this conjuncture due to lack of machine-level optimizer or sophisticated disassembler. Either of these would have facilitated rearrangement of SPARC code generated by the post-PE back-end of Aquarius compiler.

The changes in code layout and call graph were observed using the binary dumping tool, `objdump`, from the GNU tool set. Both redundant branch elimination and specialization resulted due to PE of **queens_8**. Although PE of `qsort` optimizes away the redundant choicepoint creation which reduces the memory footprint, no speedup is observed.

8.2 Summary

In summary, the above described evaluation process which is a combination of visual inspection for transformation correctness, result verification for execution correctness and CPU time utilization for performance measurement indicate the following: PE-based optimizations do result in speedup in programs with several execution threads. By inspecting the residue's executable, we believe that a post-PE machine-specific optimization phase can enhance the benefits of PE on the low-level code,

viz., BAM code. For example, machine-specific optimizations, such as code layout or inter-block instruction scheduling and software pipelining that involve the new code generated, could enhance the quality of the executable. In their absence, as is the case now, the resultant change in performance for the better or worse is due to PE-based optimization alone. Machine-specific optimizations are even more necessary due to potential procedure inlining done by the partial executor in several cases.

Chapter 9

Conclusions and Future Work

9.1 Research Contributions

The primary intent of this thesis was to investigate the application of partial evaluation on a low-level language during compilation. This investigation was geared as an important step towards answering the question "Can PE yield efficient low-level machine code?" posed by Jones [40]. To achieve the above stated goal, we proposed a new compiler back-end optimization technique based on partial evaluation of low-level RISC-like code.

9.1.1 New Compiler Back-End Optimization Technique

We studied various issues that are involved in design and implementation of such a partial evaluator as a back-end phase in a real-world Prolog compiler. We also presented solutions to problems that seem unique to partial execution of low-level code such as deciding correct program units for partial execution (Section 4.2.1), correctly keeping track of changed return address (Section 5.2.2), deciding candidate static registers (Section 5.3.2) etc. Based on inspection of resulting executable code and conclusions of other researchers [7,15,17] we believe that the full impact of such a PE-based optimization phase would be visible in conjunction with other aggressive machine-specific optimizations that take advantage of opportunities exposed by transformations done in the PE phase.

9.1.2 Optimization Framework

Another important contribution of the thesis is the demonstration that PE provides a framework of several conventional optimizations such as constant propagation, dead-code elimination, common sub-expression evaluation and, – to a lesser extent – loop unrolling. The first three optimizations are illustrated in examples shown in Figure 4.4. Loop-unrolling is described in Section 5.5. Further a Prolog-specific optimization called choicepoint elimination is also demonstrated within the framework of PE (Section 4.3.2). In conventional compilers, optimizations such as constant propagation and deadcode elimination are performed as separate phases [54]. We demonstrated that the effects of these optimizations may be obtained using a PE phase.

9.1.3 Semantics Specification of Low-Level Code

Further, we present a technique of implementation-independent specification of the Berkeley Abstract Machine using denotational semantics. Such a specification facilitates verification of the correctness of any transformation and provides a precise definition of instructions for an implementation of the partial evaluator.

9.2 Related work and applicability

9.2.1 Partial Evaluation of Prolog

Related work was discussed at several places in the thesis while discussing various issues such as partial evaluation in general, general language compilation techniques, Prolog compilation techniques and program transformation issues. A brief summary of the same along with aspects of the present work that can be prove beneficial in relation to Prolog follows. Partial evaluation typically is applied to programs written in a high-level language – either functional or Object-oriented [22,39,42]. Sahlin [58] implemented an automatic partial evaluator for full Prolog called Mixtus. Similar efforts were made by Prestwich [56] and Lakhotia [47]. These implementations of

partial evaluation, however, were not aimed to be used as compiler phase. None of them are geared towards program performance improvement and hence no performance evaluation is done. The present work aims at presenting PE as a compiler back-end phase that generates optimized code. Further, issues such as code partitioning and memory models that need to be considered when performing partial evaluation of high-level language programs differ from those while performing partial evaluation of low-level language programs. However, in this thesis we present a novel PE termination methodology (Section 5.5) that involves a loop termination technique. This can be applied during PE of any program with procedure-like constructs.

9.2.1.1 Prolog Programming Environment

Several data structures were designed augmenting BAM memory model (Section 5.2) to facilitate speculative partial execution. Similar data structures and associated algorithms can be used during implementation of a Prolog debugger or a Prolog tracer. We also present a new technique referred to as deferred dereferencing (Section 5.4.1) and it can find application in Prolog program analyses such as GFA [29, 67].

9.2.2 Partial Evaluation of Low-Level Code

Little work has been done in the area of applying partial evaluation to low-level code except that done by Bulyonkov [11]. Program performance evaluation was, however, not presented by Bulyonkov. The present work is one of the few that studies the related design and implementation issues. It finds a utility for partial evaluation as a compiler backend phase and provides a framework for several conventional compiler optimizations that are performed often in an unconnected manner.

9.2.3 Conventional Compiler Technology

9.2.3.1 Binary Translation

Partial execution techniques detailed in this thesis may be applied in several areas where conventional compiler back-end optimization techniques are widely being applied. One such area is *binary translation* [61] whose value has been recognized in recent years. Binary translation is a technique that translates an arbitrary executable binary of one architecture to executable binary for another architecture. It usually involves two phases – code translation and optimization. This technique makes application programs available on platforms on not otherwise supported by the vendor and when the user has no sources for recompilation. Although, the source and target languages differ for a binary translator, its primary functionality is to interpret the source language and emit equivalent target language instructions. Typically the translation process involves mapping a source instruction to one or more target machine instructions. Instead of emitting a generic "canned" sequence of target machine instructions, specialized code for the program memory state can be generated by performing partial execution during the translation phase. Several of the analysis techniques described in this work may be applied during the binary translation phase to result in an optimized code. The optimization phase that follows translation phase will receive a more optimized version.

9.2.3.2 Dynamic Optimization

Another potential application area is *dynamic optimization* or *dynamic compilation* [8, 23, 31]. This emerging area of research refers to techniques that facilitate optimizing a program in memory at run-time. For example, it is common for compilers to generate executables using a common instruction set architecture (ISA) such as the 80486 to run on an Intel-based system or the 21064(EV4) to run on an Alpha-based system. If, however, the program is being executed on a later architecture implementation that would most likely support advanced instructions having higher performance than those in the base architecture, it is possible to recognize

this fact and replace such instructions with their better performing counterparts. In essence the executing program is *specialized* for the architecture implementation it is running on. For instance, the sequence of instructions used to load a byte in an executable generated for a base 21064 architecture may be replaced with a single `ldb` instruction if it is running on a more recent version of Alpha processor. Using PE techniques, the program can be specialized for the host architecture implementation in several ways: its instruction schedules may be modified on the fly for better performance; its memory access pattern may be tuned with the knowledge of the cache sizes. For example, a 21164 has a wider instruction pipeline and has more number of function units in comparison to a 21064 processor. Hence the instruction scheduling needs to be different to exploit the performance advantage a 21164 offers. These system parameters provide the static information for the specialization of the program being executed. Partial execution techniques described in this thesis may be directly applied in the context of dynamic optimization.

9.2.3.3 Link-Time and Post-Link Optimization

Another potential area of application for the PE techniques described in this thesis is in tools [17, 62] that perform link-time/post-link-time processing. Information such as relocation, memory aliasing can be extracted from the disassembly of the executable using PE techniques described in this thesis. Such tools usually [17] depend on the existence of this information as part of the executable. However, using partial evaluation this information can be correctly reconstructed to facilitate further analysis and code optimizations.

9.3 Future Work

The following issues should next be addressed to firmly establish PE as a viable technique in the mainstream machine-dependent back-end compiler optimizations. Firstly, a post-PE phase that performs machine specific optimizations such as inter-procedure instruction scheduling and software pipelining needs to be implemented

either in the compiler backend or in the assembler. This will complete the environment in which performance of PE-based optimizations may completely be evaluated. Further, tools that non-intrusively profile a program at execution time need be developed to validate the observed performance. These tools will allow us to study issues such as effects of increase in code size, changes in code layout that seem to be a very common result of PE.

Next, it will be interesting to study the impact of partial execution based optimizations for the increasingly complex processor architecture implementations with advanced mechanisms such as pipelined out-of-order instruction issue, simultaneous multi-threading (SMT), predicated execution etc., that challenge conventional back-end optimization techniques.

Two of the main stream processor architectures are taking different approaches to evolve higher performing implementations. Correspondingly, compilers have to evolve to incorporate the new processor functionalities and generate efficient code that exploits processor advances. For example, it was announced that on-chip SMT will be implemented on the next generation of Alpha processors to allow instructions from various processes to be in flight at any given time [25]. A partial executor that maintains a model of the processor execution state during code generation might be one approach to assist compilation for an SMT processor. The IA-64 architecture implementation uses predicated execution and relies on sophisticated compilers [46]. Optimized code generation may be done by eliminating all code streams that can be identified by partial execution. These are two of the examples where partial evaluation can prove to be a valuable tool in compilers for future processors.

References

[1] *Quintus Prolog - Reference Manual*, 1990.

[2] Alfred V. Aho, Ravi Sethi, and Ullman Jeffrey D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[3] Hassan Aït-Kaci. *Warren's Abstract Machine - A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19 no. D-203).

[5] L.O Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

[6] L.O Andersen and C. K Gomard. Speedup Analysis in Partial Evaluation (Preliminary Results). In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–7. Yale University, 1992.

[7] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. Technical Report UCB//CSD-93-781, University of California, Berkeley, 1993.

[8] V Bala, E Duesterwald, and S Banerjia. Transparent Dynamic Optimization: The Design and implementation of Dynamo. Technical report, Hewlett Packard Laboratories, 1999.

[9] Anders Bondorf. Automatic Autoprojection of Higher Order Recursive Equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Selected papers of ESOP '90, the 3rd European Symposium on Programming.

[10] Maurice Bruynooghe, Gerda Janssens, Alain Callebaut, and Bart Demoen. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204. IEEE, August 1987.

[11] M. A Bulyonkov. Polyvariant Mixed Computation for Analyzer Programs. *Acta Informatica*, 21:473–484, 1984.

[12] Mats Carlsson. On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog. In *Logic Programming: Proceedings of the Sixth International Conference*, pages 3–16. MIT Press, 1989.

[13] Mats Carlsson. The SICStus Emulator. Technical Report SICS Technical Report no. T91:15, Swedish Institute of Computer Science, September 1991.

[14] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A Profile-Directed Binary Translator. *IEEE Micro*, 18(2), March/April 1998.

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

[16] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, 1987.

[17] Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin. Spike: An Optimizer for Alpha/NT Executables. In *Proceedings of The USENIX Windows NT Workshop*, August 1997.

[18] Charles Consel and S. D. Khoo. Parameterized Partial Evaluation. In *Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 91–106. ACM Press, June 1991.

[19] Charles Consel, C. Pu, and J. Walpole. Incremental Partial Evaluation: The Key to High Performance, Modularity and Portability in Operating Systems. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46. New York: ACM, June 1993.

[20] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. Also Research Report 92–12, June 1992 at LIENS.

[21] P. Cousot and Cousot R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[22] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization in Object-Oriented Languages. In *Proceedings of the 1995 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)*, pages 93–102, June 1995.

[23] Dean Deaver, Rick Gorton, and Norm Rubin. Wiggins/Redstone: An On-line Program Specializer. In *To appear in Proceedings of Hot Chips 11*, August 1999.

[24] Saumya K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):419–450, July 1989.

[25] Joel Emer. Simultaneous Multithreading: Multiplying Alpha's Performance. Microprocessor Forum 1999, October 1999.

[26] Andrei Ershov and Neil D. Jones. Two Characterizations of Partial evaluation and Mixed Computation. In Andrei Ershov and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages xv–xxi. North-Holland, 1988.

[27] Christopher Fraser. Automatic Inference of Models for Statistical Code Compression. In *Proceedings of ACM SIGPLAN '99 Conference on PLDI*, pages 242–246, May 1999.

[28] John Gabriel, Tim Lindholm, E. L. Lusk, and R. A. Overbeek. A Tutorial on the Warren Abstract Machine for Computational Logic. Technical Report ANL-84-84, Argonne National Laboratory, Argonne, Illinois, June 1985.

[29] Thomas Getzinger. *Abstract Interpretation for the Compile-time Optimization of Logic Programs.* PhD thesis, University of Southern California, December 1993.

[30] Arne J. Glenstrup and Neil D. Jones. BTA Algorithms to Ensure Termination of Off-line Partial Evaluation. In *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, LNCS, June 25–28 1996.

[31] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan Eggers. An Evaluation of Staged Run-Time Optimizations in DyC. In *Proceedings of ACM SIGPLAN '99 Conference on PLDI*, pages 293–304, May 1999.

[32] Programming Systems Group. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, PO Box 1263, S-164 28 Kista, Sweden, release 3 #2 edition, June 1995.

[33] M. A Guzowski. Towards Developing a Reflexive Partial Evaluator for an Interesting Subset of LISP. Master's thesis, Department of Computer Science, Case Western Reserve University, January 1988.

[34] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation.* PhD thesis, Linköping University, 1977. Published as Linköping Studies in Science and Technology Dissertation No. 14.

[35] Ralph C. Haygood. *Aquarius Prolog - User Manual*, March 1993.

[36] John Hennessey and David Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[37] T. H. Hickey and D. H. Smith. Toward the Partial Evaluation of CLP Languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 43–51. New York: ACM, 1991.

[38] Carsten K Holst. Finiteness Analysis. In John Hughes, editor, *Proceedings of FPCA, 5th ACM Conference, Cambridge, MA*, number 523 in Lecture Notes in Computer Science, pages 473–495. Springer-Verlag, August 1991.

[39] Neil D. Jones. Automatic Program Specialization: A Re-Examination From Basic Principles. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, Gammel Avrnæs, Denmark, 18-24 October, 1987, pages 225–282. North-Holland, 1987.

[40] Neil D. Jones. Challenging Problems in Partial Evaluation and Mixed Computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, Gammel Avrnæs, Denmark, 18-24 October, 1987, pages 1–14. North-Holland, 1987.

[41] Neil D. Jones. What Not to Do When Writing an Interpreter for Specialization. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237. Springer-Verlag, 1996.

[42] Neil D. Jones, Carsten K. Gomard, and Sestoft Peter. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science, 1993.

[43] Neil D. Jones, P Sestoft, and H Sondergaard. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In J.-P Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[44] Neil D. Jones, Peter Sestoft, and Harold Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[45] S. C. Kleene. *Introduction to Metamathemetics.* Van Nostrand, New York, 1952.

[46] Allan Knies and Jesse Fang. IA64 Architecture and Compilers. Tutorial at Hot Chips 11, A Symposium on High-Performance Chips, August 1999.

[47] Arun Lakhotia and Leon Sterling. ProMiX: A Prolog Partial Evaluation System. In Leon Sterling, editor, *The Practice of Prolog*, chapter 5, pages 137–179. MIT Press, 1990.

[48] John W Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[49] L.A Lombardi and B. Raphael. LISP as the Language for an Incremental Computer. In E.C. Berkeley and D.G. Bobrow, editors, *The Programming Language Lisp: Its Operation and Applications*, pages 204–219. MIT Press, Cambridge, MA, 1964.

[50] Burstall R. M and Darlington John. A Transformation System for Developing Recursive Programs. *Journal of the Association of Computing Machinery*, 24(1):44–67, January 1977.

[51] Micha Meier. Compilation of Compound Terms in Prolog. In *Proceedings of North American Conference on logic Programming*, pages 63–79, October 1990.

[52] Micha Meier et. al. SEPIA - An Extendible Prolog System. In *Proceedings of the 11th World Computer Congress IFIP'89*, pages 1127–1132, August 1989.

[53] C.S Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming*, 2:43–66, 1985.

[54] Steven Muchnick. *Advanced Compiler Design.* Morgan Kaufman, 1997.

[55] Richard O'Keefe. *The Craft of Prolog.* MIT Press, Cambridge, Massachusetts, 1990.

[56] Steven Prestwich. The PADDY Partial Deduction System. Technical Report ECRC-92-6, European Computer-Industry Research Centre, April 1993.

[57] Erik Ruf. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, Computer Systems Laboratory, Departments of Electrical Engineering & Computer Science, March 1993.

[58] Dan Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, March 1991. SICS Dissertation Series 04.

[59] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[60] David A. Schmidt. *The Structure of Typed Programming Languages.* MIT Press, 1994.

[61] Richard Sites, Anton Chernoff, Mathew Kirk, Maurice Marks, and Scott Robinson. Binary Translation. *Communications of the ACM*, 36(3):69–81, February 1993.

[62] Amitabh Srivastava and David Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Programming Language Design and Implementation*, June 1994.

[63] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, second edition edition, 1994.

[64] Robert E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[65] Andrew Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, June 1991.

[66] Pascal Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog Using Type Graphs. Technical Report CS-93-52, Brown University, November 1993.

[67] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

[68] Peter Van Roy and Alvin Despain. High-performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer*, 25(1):54–68, January 1992.

[69] Raf Venken. A Partial Evaluation System for Prolog: Some Practical Cosiderations. *New generation Computing*, 6:279–290, 1988.

[70] David H.D. Warren. An Abstract Prolog Instruction Set. Technical Report Technical Note 309, SRI International, Artificial Intelligence Center, Menlo Park, CA, August 1983.

[71] R Warren, Manual Hermenegildo, and Saumya K. Debray. On the Practicality of Global Flow Analysis. In Robert Kowalski and Kenneth Bowen, editors, *International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

[72] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.