# DYNAMIC ADVANCE LOAD REMOVAL

Chad Rosier and Saurabh Sharma
{mcrosier,ssharma2}@ncsu.edu
May 8th, 2005

## 1 Introduction

Dynamic optimization refers to any optimization that is performed after static compilation. Typically, the target application executes under the control of a dynamic optimization system which monitors the execution of the program and applies dynamic optimizations in hopes of reducing overall execution time. Observing the dynamic behavior of an application opens up many opportunities for optimization not available at static compile time. Thus, dynamic optimization has the potential to play an especially important role for statically scheduled machines (*i.e.*, EPIC/VLIW architectures).

For example, consider data dependencies that limit the amount of instruction level parallelism (ILP) found in many application. These data dependencies are often caused by an ambiguous memory dependency between a load and a store where it cannot be determined at static compile time if the instructions involved access overlapping memory locations. To eliminate these dependencies an *advanced load* allows a load to be moved above a store even if it is not known whether the load and the store may reference overlapping memory locations. Unfortunately, in the event of misspeculation expensive recovery code must be executed to restore the state of the program. Due to the high cost of executing recovery code and due to the static compiler's inability to determining when an aliasing issue exists most compilers are hesitant to use advanced loads.

In this work, we implement a dynamic optimization system that can be used to complement a static compiler by allowing the static compiler to aggressively insert advance loads without paying the penalty of misspeculation. We have implement a dynamic optimization, referred to as *dynamic advance load removal*, that dynamically identifies misspeculative advance loads and removes such advance loads while the program executes. Our implementation is transparent to the user and does not require the target application to be recompiled.

## 2 Dynamic Binary Rewriting System

We begin this section by introducing data speculation and our algorithm for removing such speculation. We then introduce features of our target architecture, the Itanium 2 processor, that impact the design of our dynamic optimization system and then briefly discuss the use of Intel's PIN tool for profiling. Finally, we conclude by showing the rewriting process in action.

### 2.1 Data Speculation

In the event of an ambiguous memory dependency the order in which a load and store instruction execute cannot be changed. To overcome this scheduling limitation, the Itanium 2 ISA includes the advance load and check instructions. An advance load, and those instructions dependent upon the load, can be speculatively scheduled above a potentially aliasing store. After the store executes a check instruction is used to determine if the speculation was successful or not. The code below is an example of how to use the advance load and check instructions for data speculation:

**Original Code:**

```
 st [r4] = r12      // ambiguous store
 ld r6 = [r8] ;;    // load to advance
 add r5 = r6, r7
 st [r18] = r5
```

**Speculative Code:**

```
 ld.a r6 = [r8] ;;  // advance load
 add r5 = r6, r7
 st [r4] = r12
 chk.a r6, recover  // check advance load
 back:
 st [r18] = r5
  . . . . .
  . . . . .
recover:
 ld r6 = [r8] ;;
 add r5 = r6, r7
 br back
```

In the event that the speculation was found to be incorrect the check instruction branches to recovery code to restore the state of the program. The decision to perform such a transformation is highly dependent upon the likelihood and cost of recovering from an unsuccessful data speculation.

## 2.2 Advance Load Removal

In this work, we seek to identify and remove data speculation in the form of advance loads that are predominantly incorrect. Our approach does not reschedule instructions, but rather changes speculative instructions into nop instructions. Speculative instructions include the advance load and all instructions before the check instruction that are dependent upon the advance load. After the speculative instructions are rewritten the check instruction is transformed into an unconditional branch to the recovery code. The example below shows how our algorithm is applied to a speculative region of code:

**Speculative Code (after applying dynamic advance load removal):**

```
 nop.m ;;           // ld.a -> nop.m
 nop.i              // add -> nop.i
 st [r4] = r12
 bra recover        // unconditional branch
 back:
 st [r18] = r5
  . . . . .
  . . . . .
recover:
 ld r6 = [r8] ;;
 add r5 = r6, r7
 br back
```

Removing speculative code and rescheduling the non-speculative code would most likely improve performance beyond that of our algorithm. This is attributed to the fact that changing the advance load and dependent instructions to nops does not reduce the length of the static schedule. For this work our primary focus was to investigate the challenges of dynamically rewriting a binary and to discuss the complexities surrounding such a non-trivial task. In the next subsections we discuss the issues encounter during the development of our system as well as the difficulty of dynamically rescheduling code.

### 2.3 Itanium 2 Architecture

The Itanium 2 Processor uses the Explicitly Parallel Instruction Computing (EPIC) paradigm where the compiler, rather than hardware, explicitly exposes instruction-level parallelism. Like VLIW processors, instructions are grouped into bundles that can be executed in parallel. Each bundle contains three instructions and a template field that indicates the type of instructions in the bundle and well as which instructions can execute in parallel. The template also indicates the mapping of instruction slots to execution unit types. Not all possible mappings[1] of instructions to units are available. This limited number of mappings often prohibits our ability to apply our dynamic advance load removal optimization.

Transforming speculative instructions into their nop equivalent is trivial. However, when transforming a check instruction into an unconditional branch we are in fact changing the type of instructions in the bundle (*i.e.*, a memory instruction to a branch instruction). Therefore, we must also change the bundle template which may or may not be possible due to the restricted number of mapping. For example, assume we have the following bundle:

```
[MII] chk.a.nc r32, recover
      cmp4.lt.unc p18,p19=r9,r36
      nop.i 0x0
```

We have one memory operation and one integer ALU operation. After transforming the check instruction to an unconditional branch we have one integer ALU operation and one branch operation. The transformed bundle is shown below:

```
[MIB] nop.m 0x0
      cmp4.lt.unc p18,p19=r9,r36
      br.cond.sptk.few recover
```

This transformation works because we have a valid template mapping from MII to MIB and an empty 3rd issue slot. However,if the 3rd slot were not a nop.i then we would not have a valid mapping from MII to IIB (or any of the other 5 combinations of I,I, and B). It is also important to note that the reordering of instruction in the bundle is valid because all instructions in the bundle execute in the same cycle.

To overcome these limitation we could either dynamically reschedule the code or insert an empty bundle after the check instruction during static compilation. Unfortunately, the former solution was not feasible given the amount of time alloted for the project and the later would potential take away from our dynamic optimization system as it would require the binary to be recompiled.

### 2.4 PIN

The primary challenge of dynamic optimization rests in the performance requirements for runtime optimization. At a minimum the dynamic optimization system must not slow the performance of a program. For

[1] Only 24 of the possible 216 mappings ($6\ instructions^{\ 3\ slots}$ = 216) are available.

this reason we chose Intel's PIN tool for dynamic instrumentation because it has many features (*e.g.*, code caching) designed specifically for reducing the overhead of runtime profiling. Further, PIN provides an API that is well suited for this work. Specifically, PIN provides functionality for instrumenting both instructions and basic blocks, categorizing instructions, and inspecting and modifying register values.

Despite PIN's rich API, no mechanism existed for determining which instructions were grouped together within a bundle. To solve this problem a set of functions were developed to read bundles directly from the ELF image. With an IP provided by PIN we can index into the ELF image to determine the template for any bundle. Section 6.1 contains the source for reading the text region of an ELF image. Section 6.2 shows the PIN instrumentation code as well as the use of the `ELF_text` buffer which contains the text region of the target application.

To determine if the removal of an advance load is appropriate, each advance load instruction and the first instruction in the recovery code are instrumented with a counter. These counters are used to determine the misspeculation rate: $\frac{references\_of\_recovery\_code}{references\_of\_advance\_load}$. The instrumentation code can be found in section 6.2. The user defines the threshold for when to apply dynamic advance load removal. By default out system currently tries to apply our optimization to all advance loads (*i.e.*, the threshold is set to 100%).

## 2.5 Mutator Process

We now discuss the steps for deciding when and how to modify the target application. In our system we use a user defined threshold to decide when to rewrite the target application. The threshold is based on the number of basic blocks executed thus far and is found by instrumenting all basic blocks with a single counter that is incremented upon entry into the block. Once the threshold is reached PIN is detached and the *mutator processes* is forked. This code is found in section 6.3.

To share information between PIN and the mutator process standard IPC system calls are used to create a shared memory segment. This code can be found in section 6.4. Entries within the *Advance Load Table* include the IPs of the advance load and associated check instruction, the number of times the advance load and recovery regions were referenced, the register associated with the advance load, the slot within the bundle that contains the advance load, and the slot within the bundle that contains the check instruction. From this table we know the location all advance loads as well as how often they are speculatively correct or incorrect. The data structures are provided below:

```
/* Adv Load / Check Inst Table Entry */
struct entry_st {
  unsigned long lda_addr;    // Address of bundle that contains lda
  unsigned int lda_slot;     // lda slot number
  unsigned long chk_addr;    // Address of bundle that contains chk
  unsigned int chk_slot;     // chk slot number
  unsigned int lda_refs;     // Num of times the adv load referenced
  unsigned int recover_refs; // Num of times the recovery code referenced
  unsigned int reg;          // Register for advload
};

/* The advance load / check table */
struct entry_st *TablePtr;
```

The first step in the mutation process is to attach to the target application using the `ptrace()` system call. The arguments to ptrace include the action to be taken (i.e., `PTRACE_ATTACH`) and the process id (pid) of the target application. The pid is passed to the mutator processes as a command line argument

when PIN calls the `exec()` system call. After successfully attaching to the target application each entry in the Advance Load Table is checked to determine if the actual accuracy, in terms of speculative correctness, is below a user defined accuracy threshold. If below the threshold the advance load removal optimization is applied. Otherwise no action is taken. As previously mentioned, the process of advance load removal includes changing the check instruction into an unconditional branch to the recovery code and converting all speculative instructions into their associated nop instructions. The main mutation function can be found in section 6.5.

When converting the check instruction to an unconditional branch we limit our rescheduling of the unconditional branch to within the original bundle. As discussed in section 2.3 this limits our opportunity to apply our dynamic advance load removal optimization. Therefore, if we are unable to convert the check instruction then we do not apply our optimization. The code for converting check instructions can be found in section 6.6. The code for converting the speculative instruction to nops can be found section 6.7. To convert the speculative instructions we compare the instructions in the recovery region to those in the speculative region, removing any redundant instructions from the speculative region.

# 3 Results

In this section we explain how to use our dynamic optimization system as well as apply our dynamic advance load removal optimization to a simple program.

## 3.1 Setup

For the example we assume that PIN (version 0.179) has been successfully installed on an Itanium 2 processor in the `$PIN_HOME` directory and our files have been unpackaged under the `$PIN_HOME/PinTools/advload/` directory. In addition to this report the following files were submitted: `Makefile`, `advload.cc`, `advload.h`, `mutator.cc`, `utils.cc`, `utils.h`, `pix_sat`, and `tom-16bit.ppm`. To use the IPC system calls the following commands must first be executed in the `advload/` directory: `'touch e'`, `'touch r'`, and `'touch t'`. Finally, execute the `'make'` command to build the system.

## 3.2 Usage

The usage information for the dynamic optimization system is provided below. They can also be seen by running the `./advload` command with no arguments.

```
Usage: ./advload -f <infile> [optional args] -- <infile>

Optional Arguments:
  -h              Help
  -a <acc>        Mutate if accuracy is below threshold
                   Default: 100 (e.g.,100.0 = mutate everything)
  -d              Dump modified binary with ".mut" extension
  -s <seq>        Begin mutating after seq sequences
                   Default: 10000000
  -S              Dump statistical info for each Adv Load
```

The required `-f` flag is used to specify the target binary. The `-a` flag is used to set the accuracy threshold. For example, to set the accuracy threshold to 50% the flag would be `-a 50`. Any advance load that is speculatively correct 50% of the time or less will be removed. The `-d` flag dumps a mutated copy of the

binary with `.mut` appended to the original binary name. The `-s` flag sets the number of sequences to be seen prior to mutating the target application. Finally, the `-S` flag when set dumps statistical information for each advance load to the `lda_chk.stats` file.

### 3.3 Example

In this subsection we apply our dynamic optimization to a simple test case. We are using a small benchmark (`pix_sat`) which includes only a single advance load. Using `objdump -d pix_sat` we show the speculative and recovery code below:

Speculative Code (Before applying optimization):

```
4000000000000960: [MMI] (p16) ld2.a r32=[r3],2
4000000000000966:       (p19) st1 [r35]=r2
400000000000096c:       (p20) cmp4.lt.unc p21,p22=r37,r0;;
4000000000000970: [MMI] (p21) st1 [r35]=r0
4000000000000976:       (p22) st1 [r35]=r37
400000000000097c:       (p16) sxt2 r36=r32;;
4000000000000980: [MII] (p16) chk.a.nc r32,40000000000009c0
4000000000000986:       (p16) cmp4.lt.unc p18,p19=r9,r36
400000000000098c:             nop.i 0x0
```

Recovery Code (Before applying optimization):

```
40000000000009c0: [MMI] (p16) adds r40=-2,r3;;
40000000000009c6:       (p16) ld2 r32=[r40],2
40000000000009cc:             nop.i 0x0;;
40000000000009d0: [MII]       nop.m 0x0
40000000000009d6:       (p16) sxt2 r36=r32
40000000000009dc:             nop.i 0x0;;
40000000000009e0: [MIB] (p16) cmp4.lt.unc p18,p19=r9,r36
40000000000009e6:             nop.i 0x0
40000000000009ec:             br.few 4000000000000990;;
```

To execute we run the following command: `./advload -f pix_sat -d -- pix_sat`. From this we generate the mutated binary (`pix_sat.mut`). Below we show the speculative code after applying our dynamic optimization. The recovery code is not shown because this code is not modified.

Speculative Code (After applying optimization):

```
4000000000000960: [MMI]       nop.m 0x0
4000000000000966:       (p19) st1 [r35]=r2
400000000000096c:       (p20) cmp4.lt.unc p21,p22=r37,r0;;
4000000000000970: [MMI] (p21) st1 [r35]=r0
4000000000000976:       (p22) st1 [r35]=r37
400000000000097c:             nop.i 0x0;;
4000000000000980: [MIB]       nop.m 0x0
4000000000000986:             nop.i 0x0
400000000000098c:       (p16) br.cond.sptk.few 40000000000009c0
```

After applying our optimization a total of three speculative instructions (`ld2.a`, `sxt2`, and `cmp4.lt.unc`) are removed. First the advance load is removed. Then the `sxt2` instruction in removed due to the true dependence on the original advance load. Next the `cmp4.lt.unc` instruction is removed due to the true dependence on the `sxt2` instruction. Finally, we see that the check instruction has been converted into an unconditional branch. This results in the bundle template changing from `[MII]` to `[MIB]`. Also notice that the predicate register of the check instruction is carried to the unconditional branch.

# 4 Conclusions and Future Work

We now discuss possible future work for improving the performance of our dynamic optimization system:

- PIN has many limitations as used in our dynamic optimization system. First, we were unable to modify the target application until PIN was detached. Otherwise, if we began the mutation process prior to detaching PIN we could disturb the sanctity of PIN or the target application. Also, after detaching PIN we cannot reattach to a running process. Therefore, our system cannot adapt to any phase changes after PIN has been is detached. The primary goal would be to use a dynamic optimization system such as Strata that does not need to be detached to apply the dynamic optimization. This would also allow us to apply the optimization as soon as we find a misspeculative advance load rather than delaying the mutation process until we detach.

- Another necessary component of a dynamic optimization system is the ability to dynamically reschedule. In addition to removing the previously mentioned limitation on converting a check instruction to an unconditional branch this would produce a much tighter schedule.

- `ptrace()` is also very limited as a means to rewrite the target application. Specifically, to rewrite a single bundle two calls to `ptrace()` are required. An alternative method, not explored in this work, would be to open the /proc/PID/mem proc entry to write an arbitrary contiguous chunk using one pwrite syscall for the each bundle. In this manner we can rewrite a bundle with a single system call. Further, what is probably worth doing is making sure you never make more than one pwrite syscall for a given page. So, when several discontiguous changes are needed in the same page, do a single write from the beginning of the first changed bundle to the end of the last changed bundle. Copying the unchanged parts of the page in between probably has less overhead than making multiple calls per page.

- The final component of this work would be to modify a compiler (*e.g.*, ORC) to speculate whenever possible. The target application would benefit greatly from the increased amount of speculation. Ideally, only a small amount of performance would be lost while our dynamic optimization is applied to poorly behaving advance loads.

# 5 Per Author Contributions

## 5.1 Saurabh Sharma

- Primary contribution was to address high-level technical aspects of the project.

- Provided code for reading the text region of an ELF.

- Implemented code for instrumenting advance loads and check instructions.

### 5.2 Chad Rosier

- Sole author of all three project reports and project web page.
- Implemented command line parsing.
- Implemented the code to dump the mutated version of the target binary.
- Implemented the code to dump per advance load statistics.
- Designed and developed all code relating to IPC between PIN and Mutator Process.
- Implemented the code to determine if a check instruction can be converted to an unconditional branch.
- Implemented the code to identify and mutate instructions.
- Implemented code to rewrite bundles in target application.
- Implemented code to rewrite ELF buffer (use to dump modified binary).
- Provided test cases for dynamic optimization system.

## 6 Code Reference

### 6.1 Source for reading bundles directly from an ELF image

```
/*******************************************************
 Function: filereadblock()
 Purpose: Read the a block of data from a file stream
*******************************************************/
void filereadblock(FILE *stream, int offset, unsigned int size, void *addr){

  fseek(stream, offset, SEEK_SET);

  if (size <= 0)
    return;

  if (fread(addr, 1, size,  stream) != size) {
    fprintf(stderr, "Couldn't read %x bytes!\n", size);
    exit(-1);
  }
}

/*******************************************************
 Function: ReadElfText()
 Purpose: Read the .text region of the <filename> binary
*******************************************************/
void ReadElfText(char *filename) {
  FILE *fd;
  Elf64_Ehdr ei;
  Elf64_Shdr *sections, *section_ptr;
```

```
  char *str_table, *name;
  int i;

  /* Open the binary */
  if((fd = fopen(filename, "r")) == NULL) {
    fprintf(stderr, "Cannot open executable %s\n", filename);
    exit(-1);
  }

  /* Get the file header */
  filereadblock(fd, 0, sizeof(Elf64_Ehdr), &ei);

  /* Read the section tables */
  sections = (Elf64_Shdr *) malloc(sizeof(Elf64_Shdr) * ei.e_shnum);
  filereadblock(fd, ei.e_shoff, ei.e_shentsize * ei.e_shnum, sections);

  /* Read the string table */
  if(ei.e_shstrndx){
    section_ptr = sections + ei.e_shstrndx;
    str_table = (char *) malloc(section_ptr->sh_size);
    filereadblock(fd, section_ptr->sh_offset, section_ptr->sh_size, str_table);
  }
  else
    str_table = NULL;

  /* Find the .text section */
  for(i = 0; i < ei.e_shnum; i++, sections++) {
    name = str_table ? str_table + sections->sh_name : NULL;

    if(name == NULL)
      continue;

    /* Found so set the start and end address of the text region. */
    /* Also read the .text section into the ELF_text buffer */
    if(strcmp(name, ".text") == 0) {
      start = sections->sh_addr;
      end = start + sections->sh_size;
      ELF_text = (unsigned char *)allocShm('e', (int)(end-start), IPC_CREAT);
      filereadblock(fd, start, (end - start), ELF_text);
      break;
    }
  }
  free(str_table);
}
```

### 6.2 PIN Instrumentation Function

```
/*****************************************************
 Function: Sequence()
 Purpose: Interpret a sequence (basic block) of
          instructions for instrumentation.
*****************************************************/
void Sequence(INS head, VOID *v)
{
  struct bundle_t bundle;
  unsigned char tplate = 0;   // Bundle template bits
  UINT64 addr = INS_Address(head);
  INT32 dest, src;

  /* Only want to look at the binarys .text region */
  if(addr < start || addr >= end)
    return;

  /* Update sequence start/end table */
  SeqAddrTable[nseq_entries].start_addr = addr;

  PIN_InsertCall(IPOINT_BEFORE, head, (AFUNPTR) inc_sequence, IARG_END);

  for (INS ins = head; ins != INS_INVALID(); ins = INS_Next(ins)){

    /* Get the address of the first instruction in the bundle */
    bundle.addr = INS_Address(ins);

    /* Update the end address of the sequence in the seq addr table */
    SeqAddrTable[nseq_entries].end_addr = bundle.addr;

    /* Read the bundle from the ELF Image */
    readBundle(&ELF_text[bundle.addr-start], &bundle.high, &bundle.low);

    /* Check to see if first instruction is an advance load */
    if(INS_MemType(ins) == TYPE_MEM_A){

      dest = INS_Regw2(ins); /* Get the dest reg for the load */

      /* Insert a call that makes entry in the buffer */
      PIN_InsertCall(IPOINT_BEFORE, ins, (AFUNPTR) update_table_ld,
                     IARG_IP, IARG_QP_VALUE, IARG_INT32, dest,
                     IARG_INT32, 0, IARG_END);
    }

    /* Check to see if first instruction is an check instruction */
    else if(INS_ChkType(ins) == TYPE_CHK_ALAT){
```

```
      src = INS_Regr2(ins);  /* Get the src reg for the chk.a */

      /* Insert a call that makes entry in the buffer */
      PIN_InsertCall(IPOINT_TAKEN_BRANCH, ins, (AFUNPTR)update_table_chk,
                     IARG_IP, IARG_INT32, src, IARG_INT32, 0, IARG_END);
    }

    /* Get the bundle template so we know the number of instructions */
    tplate = ELF_text[bundle.addr - start] & 0x1F;

    /* Check to see if instructions is an advance load */
    for(unsigned int i = 1; i < InstPerBundle[tplate]; i++){
      ins = INS_Next(ins);  // Get the next instruction

      if(INS_MemType(ins) == TYPE_MEM_A){

        /* Get the dest reg for the adv load */
        dest = INS_Regw2(ins);

        /* Insert a call that makes entry in the buffer */
        PIN_InsertCall(IPOINT_BEFORE, ins, (AFUNPTR) update_table_ld,
                       IARG_IP, IARG_QP_VALUE, IARG_INT32, dest,
                       IARG_INT32, i, IARG_END);
      }
      /* Check to see if instructions is a check instruction */
      else if(INS_ChkType(ins) == TYPE_CHK_ALAT){

        src = INS_Regr2(ins); /* Get the src reg for the chk.a */

        /* Insert a call that makes entry in the buffer */
        PIN_InsertCall(IPOINT_TAKEN_BRANCH, ins, (AFUNPTR)update_table_chk,
                       IARG_IP, IARG_INT32, src, IARG_INT32, i, IARG_END);
      }
    }
  }

  /* Update the number of sequences entries */
  if(++nseq_entries >= MAX_ENTRIES){
    fprintf(stderr,"Error: Reached max entries in SeqAddrTable\n");
    exit(-1);
  }
}
```

### 6.3 Code to determine when to fork mutator process

```
/*****************************************************
 Function: Fini()
```

```
 Purpose: Called when we have made the decision to mutate
**********************************************************/
void Fini(){

  /* Step 1: Spawn the mutator process */
  SpawnMutator();

  /* Step 2: Detach PIN from binary to be mutated */
  PIN_Detach();
}

/*********************************************************
 Function: inc_sequence()
**********************************************************/
void inc_sequence(){
  if(++nseq > max_seq)
    Fini();
}
```

### 6.4 Allocating a shared memory segment

```
/*********************************************************
 Function: allocShm()
 Purpose: Allocate (or attach to) a shared memory segment
**********************************************************/
void *allocShm(char key, int size, int shmflag){
  void *ShmPtr;
  int ShmId;
  key_t ShmKey;

  /* Generate IPC key */
  ShmKey = ftok(".", key);
  if(ShmKey == -1){
    fprintf(stderr,"Error: Unable to generate IPC key\n");
    exit(-1);
  }

  /* Allocate shared memory segment */
  ShmId = shmget(ShmKey, size, shmflag | 0666);
  if (ShmId < 0) {
    fprintf(stderr,"Error: Unable to allocate shared memory segment\n");
    perror("perror:");
    exit(1);
  }

  /* Attach to shared memory segment */
  ShmPtr = (void *)shmat(ShmId, NULL, 0);
```

```
  if ((long)ShmPtr == -1) {
    printf("Error: Unable to attach to shared memory space\n");
    exit(1);
  }

  return ShmPtr;
}
```

### 6.5 The main Mutate() function

```
/*******************************************************
 Function: Mutate()
 Purpose: Time to mutate
*******************************************************/
void Mutate(){
  struct bundle_t bundle;
  unsigned int i, j;
  unsigned long slot0, slot1, slot2;
  unsigned long rec_start_addr, rec_end_addr;

  /* Attach to process to mutate */
  ptrace(PTRACE_ATTACH, pid, NULL, NULL);

  /* Wait till target process stalls on system call before mutating */
  wait(NULL);

  for(i = 0; i < entries; i++){

    /* Only mutate if the accuracy is below threshold */
    if(accuracy/100.0 <
       (1.0-(float)TablePtr[i].recover_refs/(float)TablePtr[i].lda_refs))
       continue;

    /* Modify the bundle containing the check instruction */
    if(chkMutate(TablePtr[i].chk_addr, TablePtr[i].chk_slot)){
      fprintf(stderr,"Mutator: Unable mutate check bundle at %lx.\n",
              TablePtr[i].chk_addr);
      continue;
    }

    /* Modify the bundle containing the advance load instruction */
    ldaMutate(TablePtr[i].lda_addr, TablePtr[i].lda_slot);

    /* Determine the starting address of the recovery region */
    bundle.addr = TablePtr[i].chk_addr;
    readBundle(&ELF_text[TablePtr[i].chk_addr-start], &bundle.high, &bundle.low);
    readSlots(bundle, &slot0, &slot1, &slot2);
```

```
    /* Use the unconditional branch we created in slot 2
       to determine starting address of recovery region */
    if(slot2 & 0x01000000000)
      rec_start_addr = TablePtr[i].chk_addr -
                       ((slot2 & 0x001FFFFE000) >> 13) * 0x10;
    else
      rec_start_addr = TablePtr[i].chk_addr +
                       ((slot2 & 0x001FFFFE000) >> 13) * 0x10;

    /* Determine the ending address of the recovery region */
    rec_end_addr = 0x0;
    for(j = 0; j < MAX_ENTRIES; j++){
      if(SeqAddrTable[j].start_addr == rec_start_addr){
        rec_end_addr = SeqAddrTable[j].end_addr;
        break;
      }
    }

    if(rec_end_addr == 0x0){
      fprintf(stderr,"Unable to find the end of the recovery region\n");
      ptrace(PTRACE_KILL, pid, NULL, NULL);
      exit(-1);
    }

    /* Convert redundant speculative instructions to nops */
    redundantElim(TablePtr[i].lda_addr, TablePtr[i].chk_addr,
  rec_start_addr, rec_end_addr);
  }

  /* Detach from process to mutate */
  if(ptrace(PTRACE_DETACH, pid, NULL, NULL) == -1){
    perror("Ptrace Detach Error:");
    exit(-1);
  }

  return;
}
```

### 6.6 Code to convert check instruction to unconditional branch

```
/*******************************************************
 Function: chkMutate
 Purpose: Check to make sure we can modify the bundle
          containing the check instruction.
*******************************************************/
int chkMutate(unsigned long chk_addr, unsigned int slot){
```

```
struct bundle_t bundle;
unsigned long slot0 = 0, slot1 = 0, slot2 = 0;
unsigned char tplate = ELF_text[chk_addr-start] & 0x1F;

switch(tplate){
  /*****************************
   .mii -> .mib and .mii; -> .mib;
   One i unit must be nop.i
  *****************************/
case 0:
case 1:
  bundle.addr = chk_addr;
  readBundle(&ELF_text[chk_addr-start], &bundle.high, &bundle.low);
  readSlots(bundle, &slot0, &slot1, &slot2);

  /* Generate a statically predicted taken branch */
  if(slot2 == NOP_I){
    slot2 = (0x08000000000 | (slot0 & 0x011FFFFE03F));
  }else if(slot1 == NOP_I){
    slot1 = slot2;
    slot2 = (0x08000000000 | (slot0 & 0x011FFFFE03F));
  }else{
    fprintf(stderr,"Unable to convert chk to br in .mii\n");
    fprintf(stderr,"No integer unit is not a NOP\n");
    return -1;
  }

  slot0 = NOP_M; /* Make check instruction a NOP */

  rebuildBundle(&bundle, tplate+16, slot0, slot1, slot2);
  writeBundle(pid, bundle.addr, bundle.high, bundle.low);
  writeElf(bundle.addr-start,&bundle.low, &bundle.high);
  return 0;
case 2: fprintf(stderr,"Unable to convert chk to br in .mi;i\n"); break;
case 3: fprintf(stderr,"Unable to convert chk to br in .mi;i;\n"); break;
case 4: fprintf(stderr,"Unable to convert chk to br in .mlx\n"); break;
case 5: fprintf(stderr,"Unable to convert chk to br in .mlx;\n"); break;
case 6: fprintf(stderr,"Unused Template\n"); break;
case 7: fprintf(stderr,"Unused Template\n"); break;
  /*****************************
   .mmi -> .mmb and .mmi; -> .mmb;
   one i unit must be nop.i
  *****************************/
case 8:
case 9:
  bundle.addr = chk_addr;
  readBundle(&ELF_text[chk_addr-start], &bundle.high, &bundle.low);
```

```
  readSlots(bundle, &slot0, &slot1, &slot2);

  /* Generate a statically predicted taken branch */
  if(slot2 == NOP_I){
    slot2 = (0x08000000000 | (slot0 & 0x011FFFFE03F));
  }else{
    fprintf(stderr,"Unable to convert chk to br in .mmi\n");
    fprintf(stderr,"Integer unit is not a NOP\n");
    return -1;
  }

  /* Make check instruction a NOP */
  if(slot == 0)
    slot0 = NOP_M;
  else
    slot1 = NOP_M;

  rebuildBundle(&bundle, tplate+16, slot0, slot1, slot2);
  writeBundle(pid, bundle.addr, bundle.high, bundle.low);
  writeElf(bundle.addr-start,&bundle.low, &bundle.high);
  return 0;
  break;
case 10: fprintf(stderr,"Unable to convert chk to br in .m;mi;\n"); break;
case 11: fprintf(stderr,"Unable to convert chk to br in .m;mi\n"); break;

  /*****************************
  .mfi -> .mfb and .mfi; -> .mfb;
  One i unit must be nop.i
  *****************************/
case 12:
case 13:
  bundle.addr = chk_addr;
  readBundle(&ELF_text[chk_addr-start], &bundle.high, &bundle.low);
  readSlots(bundle, &slot0, &slot1, &slot2);

  /* Generate a statically predicted taken branch */
  if(slot2 == NOP_I){
    slot2 = (0x08000000000 | (slot0 & 0x011FFFFE03F));
  }else{
    fprintf(stderr,"Unable to convert chk to br in .mfi\n");
    fprintf(stderr,"No integer unit is not a NOP\n");
    return -1;
  }

  slot0 = NOP_M; /* Make check instruction a NOP */

  rebuildBundle(&bundle, tplate+16, slot0, slot1, slot2);
```

```
    writeBundle(pid, bundle.addr, bundle.high, bundle.low);
    writeElf(bundle.addr-start,&bundle.low, &bundle.high);
    return 0;
  case 14: fprintf(stderr,"Unable to convert chk to br in .mmf\n"); break;
  case 15: fprintf(stderr,"Unable to convert chk to br in .mmf;\n"); break;
  case 16: fprintf(stderr,"Unable to convert chk to br in .mib\n"); break;
  case 17: fprintf(stderr,"Unable to convert chk to br in .mib;\n"); break;
  case 18: fprintf(stderr,"Unable to convert chk to br in .mbb\n"); break;
  case 19: fprintf(stderr,"Unable to convert chk to br in .mbb;\n"); break;
  case 20: fprintf(stderr,"Unused Template\n"); break;
  case 21: fprintf(stderr,"Unused Template\n"); break;
  case 22: fprintf(stderr,"Unable to convert chk to br in .bbb\n"); break;
  case 23: fprintf(stderr,"Unable to convert chk to br in .bbb;\n"); break;
  case 24: fprintf(stderr,"Unable to convert chk to br in .mmb\n"); break;
  case 25: fprintf(stderr,"Unable to convert chk to br in .mmb;\n"); break;
  case 26: fprintf(stderr,"Unused Template\n"); break;
  case 27: fprintf(stderr,"Unused Template\n"); break;
  case 28: fprintf(stderr,"Unable to convert chk to br in .mfb\n"); break;
  case 29: fprintf(stderr,"Unable to convert chk to br in .mfb;\n"); break;
  case 30: fprintf(stderr,"Unused Template\n"); break;
  case 31: fprintf(stderr,"Unused Template\n"); break;
  default: fprintf(stderr,"Unknown Template\n"); break;
  }
  return -1;
}
```

### 6.7 Code to convert speculative code to nops

```
/*****************************************************
 Function: ldaMutate
 Purpose: Mutate the advance load to a nop.m
*****************************************************/
void ldaMutate(unsigned long lda_addr, unsigned int slot){
  struct bundle_t bundle;
  unsigned long slot0 = 0, slot1 = 0, slot2 = 0;
  unsigned char tplate = ELF_text[lda_addr-start] & 0x1F;

  bundle.addr = lda_addr;
  readBundle(&ELF_text[bundle.addr-start], &bundle.high, &bundle.low);
  readSlots(bundle, &slot0, &slot1, &slot2);

  switch(slot){
  case 0: slot0 = NOP_M; break;
  case 1: slot0 = NOP_M; break;
  case 2: slot0 = NOP_M; break;
  default:
    fprintf(stderr,"Removing lda from unknown slot\n");
```

```
  }

  rebuildBundle(&bundle, tplate, slot0, slot1, slot2);
  writeBundle(pid, bundle.addr, bundle.high, bundle.low);
  writeElf(bundle.addr-start,&bundle.low, &bundle.high);
}

unsigned long rec[MAX_BUNDLES][3];
unsigned char rec_template[MAX_BUNDLES];

/*******************************************************
 Function: redundantElim();
 Purpose: Remove redundant operations from speculative
          code region.
*******************************************************/
void redundantElim(unsigned long lda_addr, unsigned long chk_addr,
   unsigned long rec_start_addr, unsigned long rec_end_addr){

  bool dirty;
  unsigned char spec_template;
  struct bundle_t bundle;
  unsigned long i, slot[3];
  int j, k, l, m, rec_bundles = 0;

  /* Decode the recovery code */
  for(i = 0; i <= ((rec_end_addr - rec_start_addr)/0x10); i++){

    /* Read the template */
    rec_template[i] = ELF_text[rec_start_addr+(i*0x10)-start] & 0x1F;

    bundle.addr = rec_start_addr + (i*0x10);
    readBundle(&ELF_text[bundle.addr-start], &bundle.high, &bundle.low);
    readSlots(bundle, &rec[i][0],&rec[i][1],&rec[i][2]);

    if(i >= MAX_BUNDLES){
      fprintf(stderr,"Error: redundantElim reached MAX_BUNDLES\n");
      exit(-1);
    }
  }
  rec_bundles = i;

  /* Iterate across speculative bundles and remove redundant operations */
  for(j = 0; j <= ((chk_addr - lda_addr)/0x10); j++){
    dirty = false;

    /* Read the template */
    spec_template = ELF_text[lda_addr+(j*0x10)-start] & 0x1F;
```

```
    bundle.addr = lda_addr+(j*0x10);
    readBundle(&ELF_text[bundle.addr-start], &bundle.high, &bundle.low);
    readSlots(bundle, &slot[0],&slot[1],&slot[2]);

    /* Compare each inst slot with all inst in recovery code */
    for(k = 0; k < 3; k++){

      /* No reason to change a nop to a nop */
      if(slot[k]== NOP_M || slot[k] == NOP_I || slot[k] == NOP_B)
continue;

      /* For each bundle */
      for(l = 0; l < rec_bundles; l++){
/* For each slot */
for(m = 0; m < 3; m++){
  if(slot[k] == rec[l][m]){
    dirty = true;
    slot[k] = getNop[spec_template/2][m];
  }
}
      }
    }

    /* Write modified code to binary */
    if(dirty == true){
      bundle.addr = lda_addr + j*0x10;
      rebuildBundle(&bundle, spec_template, slot[0],slot[1],slot[2]);
      writeBundle(pid, bundle.addr, bundle.high, bundle.low);
      writeElf(lda_addr+j*0x10-start, &bundle.low, &bundle.high);
    }
  }
}
```