The HALT Library

Release version 2.02.07.15

Martha Mercaldi
Michael D. Smith
Glenn Holloway
{mercaldi,smith,holloway}@eecs.harvard.edu
Division of Engineering and Applied Sciences
Harvard University

July 15, 2002

Abstract

HALT, the Harvard Atom-Like Tool, is a library used for studying program behavior and the performance of computer hardware. HALT works by instrumentation, i.e., by mechanically changing a program's code so that it collects information about its own operation. You can use it to profile a program, e.g., to fugure out which parts of the code consume most of the running time. But you can also use HALT to investigate how efficiently a memory caching scheme supports the memory behavior of a program when applied to a range of benchmarks.

In fact, because HALT is extensible, you can design your own experiments for studying the behavior of programs and the performance of machines that execute them. HALT lets you control instrumentation by saying precisely where and how measurement code should be inserted. The class of behavioral events that you can track at run time is open-ended. HALT is structured to make it easy to adapt to new architectures, or to experimental variants of existing architectures. To port the library to a new target platform, you just specify some things that are peculiar to that platform, like how run-time analysis routines are to be called.

While HALT makes the usual applications of instrumentation (such as profiling) easy to set up, it also allows you to selectively override its behavior to do non-standard things.

Contents

1 Introduction

The HALT library is an instrumentation package for Machine SUIF. It allows the user to modify a program as it is compiled to perform run-time analysis of its own behavior. HALT helps record and summarize useful facts about where a program spends its time and how it accesses data. The resulting profiles can be used as inputs to compiler optimizations, and they can shed light on opportunities for new optimizations, either in software or hardware.

For example, HALT is a useful tool in support of static profile-driven optimizations. And many dynamic techniques, such as "walk-time optimization," just-in-time compilation, run-time code generation, instruction set emulation, and binary translation, also require the sort of run-time information that a system such as HALT provides.

The name HALT stands for Harvard Atom-like Tool, after the Atom tool developed by Digital Equipment Corporation (now part of Compaq) [2]. Atom is a customizable system for instrumenting already compiled Alpha executables.

HALT runs as part of compilation. Since parts of HALT have to perform target-specific tasks, HALT must be extended for each target architecture with which you wish to use it. Currently HALT is implemented for both the $x86$ and Alpha targets.

Section 2 contains an example of how you might use HALT to generate a simple edge profile. Section 3 contains all you about HALT to user it for standard tasks like profiling. Section 4 is as an extender's guide, providing implementation specifics to help you adapt HALT to a new platform or to use it in a new way.

2 How HALT is Used

Here, in a nutshell, is how you use HALT for typical instrumetation-based tasks like profiling.

- You run a pass that marks points in the program that you want to have instrumented. A pass called `label`, provided with HALT, handles typical cases for you, but you can also do this in a pass of your own.
- You write run-time *analysis routines*, one for each kind of instrumentation that's been marked in your program. E.g., if you have instrumented procedure entries, then you write a routine called `_record_entry` to be called each time an instrumented procedure is entered.
- You use HALT's instrument pass to do the actual insertion as part of Machine-SUIF compilation.
- You run the resulting program, linked with your run-time analysis routines, on representative input data, producing some analysis, i.e., some data about the program's performance. For example, the analysis data might be an execution count for each instrumented program point.
- Finally, you map the collected data back to the program in a form useful for your intended application. There's no single recipe for doing this. If your goal is to provide a profile for manual optimization, you might identify "hot spots" by source file, procedure name and line number. If you're guiding a profile-driver optimizer, then you must put your analysis information into the form that it expects to read.

2.1 Example: Edge Profiling

HALT is easy to use for edge profiling, allowing you to gather data needed to determine how many times each control flow graph (CFG) edge is traversed during the execution of a program.

Say you had the CFG for your program with numbered nodes. The information that HALT would provide would indicate which direction control went every time it reached a control-flow-related instruction. HALT obtains this information by inserting calls on run-time analysis routines just before each such instruction. The arguments to an analysis routine for a branch indicate which branch it is and which edge from the branch node is being taken.

To mark each branch, HALT assigns it a unique identifier, distinct from that of other branches in the program. To identify an out-edge for a branch node, HALT uses a 0-based index into the sequence of its successor nodes. In the case of a conditional branch that index is 0 when the branch is not taken (the "fall-through" case) and 1 when the branch is taken. For a multiway branch with k out edges, the index ranges between 0 and $k - 1$.

The purpose of analysis routines for edge profiling is to record the number of times each branch edge is taken during runs on sample inputs. To connect the measured frequencies back to the program, you need a map from each unique identifier back to the corresponding point in the control flow graph (CFG). One way to create such a map is with a pass called `summarize`. It produces a representation of the CFG labeled with unique identifiers.

Having mapped each HALT unique identifier to a point in the CFG, you can use the frequencies of branch edges to calculate how many times each edge in the CFG was traversed during the program execution.

The following commands use HALT to gather the run-time data needed to complete the edge profile described above.

```
do_label -entry -exit -cbr -mbr foo.input foo.labeled
do_instrument -target_lib x86_halt foo.labeled foo.instrumented
```

To produce the CFG labeled with unique identifiers:

```
do_summarize foo.labeled foo.summary
```

2.2 Other Applications of HALT

HALT can gather other types of profiles in addition to the simple edge profile described above. Profiles are especially important in guiding profile-driven optimizations. Below are some possibilities for other uses of HALT.

Node profile. An even simpler program profile than an edge profile is a node profile, which tells you the number of times each basic block was executed. For node profiling, you use the `-bb` option of the `label` pass, which marks the start of each basic block as an instrumentation point.

Path profile. Node and edge profiles are called *point* profiles. A *path* profile helps you understand the execution frequencies of important basic block sequences, which is valuable information because it allows you to exploit correlations that aren't exposed in point profiles. HALT has been used for depth-bounded path-profiling in studies of static correlated branch prediction and profile-directed instruction scheduling [4]. In this application, arbitrary paths up to a certain history depth are monitored.

We are currently using HALT to implement an alternative path-profiling approach developed by Thomas Ball and James Larus [3]. The Ball-Larus method assigns a unique number to each acyclic path in a program. The numbering scheme is designed so that very lightweight instrumentation, placed at strategically-chosen program points, is sufficient to identify which path has been taken whenever a path endpoint is reached. Our implementation of the Ball-Larus approach extends the HALT library to accommodate new kinds of instrumentation. We sketch these extensions in Section 4.2.

Branch prediction evaluation. One can evaluate branch prediction hardware by modelling the hardware being tested and comparing its predictions for each branch with the direction actually taken by the program, information that HALT can provide.

Cache simulation. Using instrumentation data on memory references, you can use HALT to simulate and compare different cache models and replacement policies.

3 User's Guide

This section explains how to mark places in code being compiled where instrumentation should be inserted, how to use the `instrument` pass to insert the actual instrumentation code, and how to write the analysis routines that the instrumented code will call at run time.

3.1 Identifying Instrumentation Points

You identify each instrumentation point by adding a HALT annotation to an instruction. The annotation marks a point of interest in the code being compiled and it indicates what kind of instrumentation needs to be done. To attach annotations, you can either use HALT's `label` pass, which performs numbers and labels selected instrumentation points in a SUIF file, or you can place the HALT annotations using a pass of your own. The latter alternative allows you to be more selective, and you may well have an existing pass can that can be adapted to do the labeling as an optional side effect.

A HALT annotation has the key `k_halt`. Its associated value contains two or more constants: a *kind identifier*, a *unique identifier*, and possibly other values that the instrumentation code will use at run time. The kind identifier is an integer that distinguishes one kind of instrumentation point (such as a conditional branch) from another (such as a procedure entry). Different kinds require different handling, both at instrumentation time and at run time. The unique identifier distinguishes the specific instrumentation point. It is often literally unique to one point in the compiled code, but nothing in HALT requires that to be the case. You could, for example, create a separate identifier space for each instrumentation kind, if you prefer.

The additional values attached to the HALT annotation are also up to the user. Suppose that you want to instrument a program to measure the number of instructions that it executes (dynamically) on each run. To do so, you can attach the number of instructions in each basic block to a HALT annotation at the entry of the block. Each time that point is reached at run time, instrumentation code calls an analysis routine (written by you) that adds the static count to a running total.

3.1.1 HALT's `label` pass

The `label` pass looks at each procedure in the program, going through each node in the procedure's control flow graph, and for each node, going through the instructions contained in it. Depending on the options you give on the `label` command line, the pass looks for instructions that satisfy predicates such as `is_cbr` (for conditional branches) or `reads_memory` (for loads). When it finds an instruction satisfying an appropriate condition, the `label` pass attaches a HALT annotation to it, incorporating a freshly generated unique identifier and the kind identifier of instrumentation to be performed.

What `label` labels. Here are the instrumentation kinds that HALT knows about initially.

5 ⟨*kinds list* 5⟩≡ 17a ▷

```
namespace halt {

enum {
    STARTUP = 0,        // whole-program entry
    CBR,                // conditional branch
    MBR,                // multi-way branch
    ENTRY,              // procedure entry
    EXIT,               // procedure exit
    SETJMP,             // setjmp call
```

```
    LONGJMP,            // longjmp call
    LOAD,               // reads memory
    STORE,              // writes memory
    BLOCK,                      // basic block
    CYCLE               // cycle count
};
} // namespace halt
⟨last kind value 17b⟩
```

Note that kind identifiers are wrapped in a C++ `namespace` declaration. When you use a HALT kind identifier, you may either qualify it with the namespace, as in `halt::CBR`, or open the `halt` namespace with the declaration

```
using namespace halt;
```

Here are sketches of the various kinds:

- Program start (`STARTUP`):

  The `STARTUP` point is always at the entry of the main procedure of the compiled program. The `instrument` pass instruments that point without needing an annotation to guide it. The `STARTUP` analysis routine that you define can allocate and initialize storage, install signal handlers, and otherwise prepare for run-time analysis. Typically, it invokes `atexit` to register a termination handler, a function that finalizes analysis when the program exits. For example, the termination handler might write a profile to disk.

- Conditional branch (`CBR`):

  The analysis routine for a conditional branch takes two arguments, the unique identifier of the instrumentation point and the branch condition. The condition is 1 when the branch is about to take, and 0 when it will fall through. The `label` pass annotates conditional branches when given the `-cbr` option.

- Multiway branch (`MBR`):

  The analysis routine for a multiway branch takes the unique identifier and the the zero-based index of the branch target chosen.[1] The `label` pass annotates multiway branches when given the `-mbr` option.

- Procedure entry (`ENTRY`):

  The analysis routine for a procedure-entry point receives just the unique identifier of that point. The `-proc` option to `label` causes every procedure entry point to be annotated. (It also affects return-point annotation. See the next entry.)

- Procedure exit (`EXIT`):

  The analysis routine for a procedure-exit (return) point receives just the unique identifier of that point. If you give `label` the `-ret` option, then it annotates every return instruction giving each one a distinct unique number. If you use both `-ret` and `-proc`, each procedure entry and exit receives its own unique number. However if `-proc` is used without `-ret`, each return point is given the same unique number as the entry of the procedure that contains it. That makes it easy to keep track of which procedure is the "innermost" currently active at run time.

- `setjmp` call (`SETJMP`):

  The analysis routine for a `setjmp` is called just before the original `setjmp` call. It takes a unique identifier and the `jmp_buf` value that is to be passed to `setjmp`. The `label` pass annotates `setjmp` calls when you give it the `-setlongjmp` option.

[1] The number of targets of a multiway branch and the index associated with each isn't always obvious from the source code that gives rise to it (e.g., a C `switch` statement). The `default` label can correspond to more than one index in a multiway dispatch, or it might be omitted completely if the front end has ruled out the need for it.

- `longjmp` call (`LONGJMP`):

  The analysis routine for a `longjmp` takes three parameters: a unique identifier plus the `jmp_buf` and "status" values that are to be passed to `longjmp`. The `label` pass annotates `longjmp` calls when you give it the `-setlongjmp` option.

- Load (`LOAD`):

  This kind of instrumentation covers any instruction that reads from memory. The analysis routine takes three arguments: the unique identifier of the memory-fetch instruction, the address in memory from which data is read, and the number of bytes transferred. To annotate memory-fetch instructions, add the `-load` flag to the `label` command line.

- Store (`STORE`):

  This kind of instrumentation covers any instruction that writes to memory. The instrumentation is placed just after the original store. The analysis routine takes three arguments: the unique identifier of the store instruction, the address in memory to which data is written, and the number of bytes transferred. To annotate store instructions, add the `-store` flag to the `label` command line.

- Basic block (`BB`):

  Basic-block instrumentation is inserted at the entry of each node of a CFG except the entry and exit nodes. The only argument to the analysis routine is the unique identifier. You can cause basic-block entries to be markd for instrumentation by using the `-bb` option of the `label` pass.

- Cycle (`CYCLE`):

  A `CYCLE` event represents the advancement from one processor cycle to the next. These instrumentation points are marked by a scheduler, not by the `label` pass. The only analysis-routine argument for a `CYCLE` point is its unique identifier.

How `label` assigns "unique" identifiers. The `label` pass assigns a fresh unique identifier to each instrumentation point that it annotates. By default, it starts at 0 and adds 1 to form subsequent identifiers. To give it an explicit starting value, add `-unique` *identifier* to the command line. When it finishes, `label` prints the next unused identifier on its standard output. This allows you to assign numbers that are unique over a whole program. You apply `label` to each file of the program in turn, capturing the final identifier for each and making that the starting value for the next.

The effects of the `label` pass are cumulative. If you want the unique numbers for procedure entry, say, to be drawn from a different space than branch numbers, you could run `label` twice, once with the `-proc` option and again with the `-cbr` and `-mbr` options. The resulting identifiers would still be unique within kinds, but not across kinds. Sometimes, that's the scheme that leads to the most efficient run-time organization.

3.1.2 Generating your own HALT annotations

You might prefer to write your own SUIF pass to insert HALT annotations. If your research interests focus on particular parts of a program, then you most likely already have a SUIF pass that knows how to locate these parts, so modifying this pass to add HALT annotations isn't too difficult.

Here is the definition of the HALT annotation class.

7 ⟨*Class* HaltLabelNote 7⟩≡

```
class HaltLabelNote : public Note {
 public:
   HaltLabelNote() : Note(note_list_any()) { }
   HaltLabelNote(long kind, long id) : Note(note_list_any())
       { _replace(0, kind); _replace(1, id); }
   HaltLabelNote(const HaltLabelNote &other) : Note(other) { }
   HaltLabelNote(const Note &note) : Note(note) { }

   long get_kind() const                    { return _get_c_long(0); }
   void set_kind(long kind)                 { _replace(0, kind); }
   long get_unique_id() const               { return _get_c_long(1); }
   void set_unique_id(long id)      { _replace(1, id); }

   int  get_size_static_args(void)          { return _values_size() - 2; }
   long get_static_arg(int pos)             { return _get_c_long(pos+2); }
   void set_static_arg(int pos, long value) { _replace(pos + 2, value); }
};
```

Static versus dynamic arguments. We call analysis routine arguments that are known at compile time *static* arguments. The unique identifier of an instrumentation point is an example of a static argument. A *dynamic* argument however is an argument to the analysis routine that cannot be known until run time. For example, you can't know whether or not a conditional branch is taken until the program is run. So the condition code of a conditional branch is considered a dynamic argument to the analysis routine.

The `HaltLabelNote` allows for more than one static argument. The generic HALT `label` pass uses only the unique identifier of an instrumentation point as a static argument. However, when you write your own pass to insert `HaltLabelNote`s you may wish to include other static arguments. You do so by first creating a `HaltLabelNote` value and then calling the `set_static_arg` and method to append static argument values. Section 4.2 contains an example that uses extra static arguments.

When an analysis routine is called, its dynamic arguments come first, followed by its static arguments. That allows you to add extra static arguments for particular instrumentation points without having to define new instrumentation kinds. You just need to write analysis routines that accept the extra arguments.

3.2 The `Instrument` Pass

Once there are HALT annotations in the program, the `instrument` pass comes along to manipulate and insert the proper instructions to make the calls to analysis routines. The following command line would run the `instrument` pass.

```
do_instrument -target_lib x86_halt foo.labeled foo.instrumented
```

There is only one flag for the command line, the `-target_lib` flag, which indicates the target library to use to generate the machine code. This can be either `x86_halt` or `alpha_halt` or one you write yourself if you have extended Machine SUIF and HALT to another target. If you've added a new target library to Machine SUIF you must also extend HALT to generate instrumentation code in that machine language. Therefore `-target_lib x86_halt` works on the `instrument` pass command line, but `-target_lib x86` does not. There is more information about extending HALT for a new target in Section 4.

3.3 Writing HALT Analysis Routines

The instrument pass, according to the kind of instrumentation, inserts a call to the user-defined analysis routine for that kind. These routines are named using the following formula: `_record_`*kind*. That means that the analysis routine for conditional branches is called `_record_cbr`. Here is a list of the analysis routine declarations for the kinds HALT can currently instrument.

```
void _halt_startup(void);
void _record_cbr(unsigned long cond, unsigned long id);
void _record_mbr(unsigned long index, unsigned long id);
void _record_entry(unsigned long id);
void _record_exit(unsigned long id);
void _record_load(long bytes, unsigned long ea, unsigned long id);
void _record_store(long bytes, unsigned long ea, unsigned long id);
void _record_block(unsigned long id);
void _record_cycle(unsigned long id);
void _record_setjmp(unsigned long buf, unsigned long id);
void _record_longjmp(unsigned long buf, unsigned long status, unsigned long id);
```

In each of the above functions, the last argument, the instrumentation point's unique identifier, is the only static argument. Each argument before that is a dynamic one, whose value must be developed at run time.

These analysis routines are written in C, but can also be written in C++. If you write the functions in C++ you must be sure to use the `extern "C"` specifier in the declaration so that the function names won't be mangled.

4 Extending the HALT System

There are two main reasons for wanting to extend HALT. First, because the action of HALT is partly target-specific, you may want to extend its functionality to a new target platform. Second, you may want to perform a new kind of instrumentation, e.g., to monitor a kind of run-time event that isn't handled by the basic library. We'll give examples of such events in Section 4.2.

To understand how to extend HALT, you need to know something about how the `instrument` pass operates. The examples in this section use the *x*86 target. You can find the complete code adapting HALT to that platform in the `x86_halt` directory of the Machine-SUIF distribution. It may be helpful to look over this code when porting the library to a new target.

What the `instrument` pass does. The `instrument` pass scans the representation of code being compiled, looking for HALT annotations attached to instructions. When it finds such a note, it applies the function `halt_recipe` to its kind identifier to obtain a *recipe* for performing instrumentation.[2] A recipe is a C++ object of class `HaltRecipe`. It represents the series of steps needed to instrument one event. The `instrument` pass "applies" the recipe to the annotated instruction; that is, it invokes the `operator()` method of class `HaltRecipe`, passing the location of the HALT-annotated instruction and some other information (described below) as arguments. The `operator()` method inserts instrumentation code in the neighborhood of the annotated instruction (just before it or just after).

A recipe is customized for the target and for the kind of instrumentation that it handles. It has to know about the target because it involves emitting machine-specific instructions. It has to know about the instrumentation kind because collection of information about instrumented events is different for the different kinds. The recipe for instrumenting a conditional branch on the *x*86, for instance, has to know how to pass arguments to a function called at run time (i.e., by pushing them onto the stack in reverse

[2] `halt_recipe` is the HALT library's addition to the Machine-SUIF OPI. So it can be used in target-independent code (like the `instrument` pass), but its semantics in any given application depend on the prevailing target context [5].

order) and it has to know how to emit code that grabs the the branch condition at run time (which it does by inspecting the code leading up to the branch).

Class `HaltRecipe`. You extend HALT by encoding new recipes in subclasses of the abstract class `HaltRecipe`. There is a subclass for each target family (e.g., `HaltRecipeX86`). Its methods handle things that are needed by most recipes for that family (e.g., emitting code to call a function at run time). And there is a concrete subclass of the target-specific class for each instrumentation kind (e.g., `HaltRecipeX86Cbr` for $x86$ conditional-branch events). This is the class whose `operator()` method knows the exact recipe for that target and kind.

Almost every recipe inserts a function call into the original program. In addition to the actual call, there must be code for saving and restoring registers affected by the call and for developing the dynamic arguments (the ones whose values can't be known until run time).

For a particular target, most of these tasks can be handled in the same way for each kind of instrumentation. There are `HaltRecipe` methods for each task, and most are implemented at the target-specific level (e.g., in class `HaltRecipeX86`). If necessary, however, any of these methods can be overridden by the class for a particular kind (such as `HaltRecipeX86Cbr`).

A recipe collects instrumentation code in the `instr_pot` data field of class `HaltRecipe`, which is an array of instruction lists. As the recipe is followed at a given instrumentation point, these lists are filled with code snippets for the tasks making up the recipe. For example, `instr_pot[SAVE]` receives the instructions for saving registers. After `instr_pot` has been filled, the final step is to pour the instrumentation code snippets from the "pot" into the original instruction stream in the right order.

Here's how class `HaltRecipe` is declared:

10 ⟨*class* `HaltRecipe` 10⟩≡

```
class HaltRecipe {
  public:
    HaltRecipe() { prepare_pot(); }
    virtual ~HaltRecipe() { /* scrub_pot(); */ } // (SUIF dtor-ordering problem)

    // insert target- and kind-specific instrumentation for one point
    virtual void
        operator()(HaltLabelNote note, InstrHandle handle, CfgNode *block,
                   const NatSet *live_before, const NatSet *live_after) = 0;
  protected:
    InstrList *instr_pot[halt::RECIPE_SIZE];    // code snippets for this...
                                                // ...instrumentation point
    Vector<Opnd> args;                          // args to analysis routine

    // target-independent helpers
    void prepare_pot();                         // initialize instr_pot
    void scrub_pot();                           // clean up instr_pot
    void follow_recipe(int kind,                // fill instr_pot by calling...
                       const NatSet *live);     // ... target-specific methods
    void insert_instrs(halt::InsertPoint,       // pour instr_pot into stream...
                       CfgNode*, InstrHandle);  // ...at indicated point

    // target-specific helper methods for use in operator()
    virtual void static_args(HaltLabelNote) = 0;
    virtual void build_save_set(NatSet *save, const NatSet *live) = 0;
    virtual void setup_stack() = 0;
    virtual void save_state(NatSet *save) = 0;
    virtual void insert_args() = 0;
    virtual void insert_call(ProcSym*) = 0;
    virtual void clean_args() = 0;
    virtual void restore_state(NatSet *save) = 0;
    virtual void destroy_stack() = 0;
```

```
};
```

As the declaration above shows, `HaltRecipe` is an abstract class: it defines only four of the methods that it declares. Two of them, `prepare_pot` and `scrub_pot`, are normally only used by the `HaltRecipe` constructor and destructor, respectively. The other two, `follow_recipe` and `insert_instrs`, are useful when you implement `operator()` for particular target machines and instrumentation kinds. `follow_recipe` fills `instr_pot`, and `insert_instrs` empties it into the instruction stream of the program being instrumented. The arguments to `follow_recipe` are a kind identifier (e.g., `CBR` for a conditional branch point) and a set representing the live registers at the instrumentation point (see the discussion of `operator()` below). The arguments to `insert_instrs` give the position at which instrumentation code should be inserted: a flag indicating whether to place it before or after the original instruction, together with the block and the position of that instruction within the block. (An example of how to use `follow_recipe` and `insert_instrs` is coming up.)

`operator()` method. The virtual `operator()` method in class `HaltRecipe` embodies the recipe for a particular target machine and instrumentation kind. It takes the following arguments:

- `note`, the HALT annotation containing the instrumentation kind and static arguments;
- `handle`, a handle on the annotated instruction;
- `block`, the basic block containing the annotated instruction;
- `live_before` and `live_after`, sets of registers live before and after the annotated instruction, respectively.

The sets `live_before` and `live_after` contain abstract register numbers for the target machine. Register liveness is needed during instrumentation to minimize the number of registers that have to be saved and restored around the inserted code. The `live_before` set is used when the insertion point precedes the annotated instruction; `live_after`, when it follows. The `instrument` pass provides both since it doesn't know which is actually needed.

The typical `operator()` method creates code that develops any needed dynamic analysis-routine argument values, it calls `follow_recipe` to fill `instr_pot` with these and other needed instructions, and then it calls `insert_instrs` to insert the contents of `instr_pot` into the instruction stream of the program.

As an example of how to write this method, consider the case of an $x86$ conditional branch that has been annotated for instrumentation. Here is the code at such a branch after `instrument` has finished with it:

```
cmpl    $0,%ecx        # original
pushfl                 # instr_pot[SAVE]
pushl   %eax           # instr_pot[SAVE]
setne   %al            # instr_pot[KIND]
movzbl  %al,%eax       # instr_pot[KIND]
pushl   $42            # instr_pot[ARGS]
pushl   %eax           # instr_pot[ARGS]
call    _record_cbr    # instr_pot[CALL]
addl    $8,%esp        # instr_pot[CLEAN]
popl    %eax           # instr_pot[RESTORE]
popfl                  # instr_pot[RESTORE]
jne     main._tmp123   # original
```

Before instrumentation, the program performed a compare (`cmpl`) followed by a jump-if-not-equal (`jne`). The code that `instrument` has inserted between these two original instructions:

- saves registers that are live before the branch (`EFLAGS` and `EAX`);

- realizes the branch condition as an integer in EAX: zero if the branch will fall through, or one if it will take.
- pushes the analysis-routine arguments in reverse order: first the static argument 42, which is the unique identifier of the instrumentation point, and then the dynamic argument in EAX;
- calls the analysis routine (_record_cbr);
- flushes the arguments from the stack (8 bytes' worth);
- restores the saved registers.

Here is the *x*86 definition of operator() for the conditional-branch kind (CBR):

12 ⟨*x86* CBR operator() 12⟩≡

```
void
HaltRecipeX86Cbr::operator()
    (HaltLabelNote note, InstrHandle handle, CfgNode *block,
     const NatSet *live_before, const NatSet *live_after)
{
    debug(2, "%s:HaltRecipeX86Cbr", __FILE__);

    Instr *mi;
    Opnd opnd_reg_al  = opnd_reg(AL,  type_s8);
    Opnd opnd_reg_eax = opnd_reg(EAX, type_s32);

    // find the setCC opcode matching the branch's jCC opcode
    int setcc = jcc_to_setcc(get_opcode(*handle));

    // build one dynamic arg in EAX: a 1 if branch will take, else 0
    mi = new_instr_alm(opnd_reg_al, setcc, opnd_reg_eflags);
    append(instr_pot[KIND], mi);                        // setCC %al

    mi = new_instr_alm(opnd_reg_eax, MOVZX, opnd_reg_al);
    append(instr_pot[KIND], mi);                        // movzx %al -> %eax

    // put argument operands (dynamic and static) into 'args'
    args.push_back(opnd_reg_eax);
    static_args(note);

    // save/restore regs live before the branch, including EFLAGS
    NatSetDense to_save = *live_before;
    to_save.insert(EFLAGS);

    follow_recipe(CBR, &to_save);
    insert_instrs(BEFORE, block, handle);
}
```

The first half of the method above is about making the instructions that produce the dynamic argument for the analysis routine _record_cbr; in the example, these are the setne and movzbl instructions.[3] The first puts the branch condition into register AL, which is the low byte of EAX. The second zero-extends that value into the full EAX register. These two instructions go into instr_pot[KIND], which is the segment of instr_pot used for purely kind-specific instrumentation code.

The next part of this operator() fills the sequence called args with operands that represent arguments to the analysis routine. The args field of class HaltRecipe is where other methods expect to find these operands. First operator() appends the dynamic-argument operand to args. Since the argument value will be in EAX at run time, it uses a register operand representing EAX. Next, static arguments are inserted

[3]Method jcc_to_setcc is a helper in class HaltRecipeX86. Given a conditional-jump opcode of *x*86, such as JNE, it returns the corresponding opcode (e.g., SETNE) for extracting a Boolean from the EFLAGS register as a byte-sized value.

by the static_args helper, which takes values from the HALT annotation and appends corresponding operands to args. (In the example, there's just one static argument, namely the unique identifier 42.) Finally, like almost all operator() implementations, this one ends with calls on follow_recipe, to fill the other segments of instr_pot, and on insert_instrs, to move all instr_pot segments into the instruction stream. It chooses to insert instrumentation code before the annotated instruction, rather than after. For a conditional branch, this makes sense because the point after the instruction won't always be reached. This choice is reflected in the use of live_before as the set of registers to save and restore (second argument to follow_recipe) and in the use of the BEFORE token when calling insert_instrs.

When you write an operator() method, you have complete control of what happens at the instrumentation points to which it applies. As in the conditional-branch example, however, most of the work is usually handled by helper methods, leaving only kind-specific tasks to be implemented explicitly.

What goes in the pot. In the example above, instr_pot[KIND] is the instruction list in which code for generating a dynamic argument is built up. This segment of instr_pot, intended for kind-specific purposes, is usually the only one that you'll need to mention explicitly in an operator() method. The other segments are filled by methods that, while target-specific, are usually kind-independent.

Here is the full list of instr_pot segments, in the order of their appearance in the instrumented code:

13 ⟨*recipe component names* 13⟩≡

```
enum {
    SETUP = 0,  // code to build stack frame
    SAVE,       // save necessary register state
    KIND,       // create kind-specific arguments
    ARGS,       // argument setup
    CALL,       // call to user-provided instrumentation routine
    CLEAN,      // clean-up argument storage
    RESTORE,    // restore saved register state
    DESTROY,    // code to destroy the stack frame
    RECIPE_SIZE
};
```

Some targets don't need to use every segment. For example, the $x86$ implementation doesn't use the SETUP and DESTROY segments because that ISA has stack push and pop instructions that let you allocate and free stack space while saving and restoring state. The uses of the other segments are illustrated by the conditional-branch example above.

Target-specific methods. Here is a rundown of the target-specific methods that HaltRecipe declares. Although these usually handle the kind-independent aspects of instrumentation-code preparation, you can always override them at the kind level if necessary.

- static_args(HaltLabelNote note): Appends operands for static arguments to the args sequence. The first static argument is always the unique_id attribute of note. The rest, if any, come from note's static_arg sequence.
- build_save_set(NatSet *save, const NatSet *live): Looks at the set of live registers and converts it into a set of maximal registers which need to be saved before the call to the analysis routine. This set, called the *save set*, is unioned into the result parameter save. Elements of save and live are abstract register numbers, as defined by the target library.
- setup_stack(): Builds code in instr_pot[SETUP] to make room on the stack for saving state. The amount of space needed is determined by other methods and communicated through a protected field.
- save_state(NatSet *save_set): Builds code in instr_pot[SAVE] to store all registers given by save_set on the stack.

- `insert_args()`: Builds code in `instr_pot[ARGS]` for passing arguments to a subsequent call instruction. Takes the argument operands from the `args` sequence (a field of class `HaltRecipe`).
- `insert_call(ProcSym *routine)`: Builds a call to the given analysis `routine` in `instr_pot[CALL]`.
- `clean_args()`: Builds code in `instr_pot[CLEAN]` to clean up after the `insert_args` method, if necessary. This method must also clear the `args` sequence by calling `args.resize(0)`.
- `restore_state(NatSet *save_set)`: Builds code in `instr_pot[RESTORE]` to restore the registers in `save_set` from the stack.
- `destroy_stack()`: Builds code in `instr_pot[DESTROY]` to free the stack space allocated by `setup_stack`.

Following the recipe. In the $x86$ conditional-branch example, only one of the target-specific methods described above was mentioned explicitly: `static_args` was called to begin the `args` sequence. The rest are called from `follow_recipe`, which invokes them an order that's intended to allow reasonable communication between the methods. For example, the `save_state` method is called before `setup_stack` because the former might need to record some information for the latter. Here is the definition of `follow_recipe`:

14a ⟨*method* `follow_recipe` 14a⟩≡
```
inline void
HaltRecipe::follow_recipe(int unique_id, const NatSet *live)
{
    NatSetDense save_set;
    build_save_set(&save_set, live);

    insert_args();
    clean_args();

    insert_call(halt_proc_sym(unique_id));

    save_state(&save_set);
    restore_state(&save_set);

    setup_stack();              // save_state may record the size of the state
    destroy_stack();
}
```

Inserting the instrumentation. The function `insert_instrs`, a call to which is the last thing done by `operator()`, performs the actual insertion of the instructions from `instr_pot` into the instruction stream of the program. `Insert_instrs` takes three arguments: the insertion point, the node in which the instructions are to be inserted, and the instruction handle of the annotated instruction. The insertion point argument is simply of type `InsertPoint` (shown below) and indicates on which side of an annotated instruction to place the instrumentation. In some cases this doesn't make a difference, but in other cases it can. For example, the instrumentation for a store instruction is placed after the store instruction itself, so that the analysis routine could, since it has the address in memory and number of bytes of data being stored, look at the data that has been stored. This has to be done after the store is executed because otherwise the data would not yet be at that address.

14b ⟨*insertion-point names* 14b⟩≡
```
enum InsertPoint {
    BEFORE,
    AFTER
};
```

4.1 How to Create a HALT-Ready Target Library

Recall from Section 2.1 that a typical command line for the `instrument` pass looks like this:

```
do_instrument -target_lib x86_halt foo.labeled foo.instrumented
```

The `-target_lib` option must name a library that has all the functionality of a regular Machine-SUIF target library and also supplies target-specific functionality for HALT. In particular, it must provide the $x86$ implementation of the `halt_recipe` function.

Creating a context object. This customization uses the "context object" machinery described in *An Extender's Guide to the Optimization Programming Interface and Target Descriptions* [5]. The HALT library declares a class called `HaltContext` that specifies the interface that a HALT-ready target library must implement. To make a HALT-ready target-specific library such as `x86_halt`, you combine the context class of an existing target library, such as `x86`, with a target-specific subclass of `HaltContext`, using C++ multiple inheritance. An instance of the resulting class is the context object through which a target-independent pass like `instrument` queries the description of the target machine. The same context object that supplies register characteristics and implements predicates about instruction semantics also provides the target-specific results for the `halt_recipe` function.

The `HaltContext` class is particularly simple:

15a ⟨*class* `HaltContext` 15a⟩≡
```
class HaltContext {
  public:
    HaltContext() { }
    virtual ~HaltContext() { }

    virtual void halt_begin_unit(OptUnit*) { }
    virtual void halt_end_unit  (OptUnit*) { }
    virtual HaltRecipe *halt_recipe(int) const = 0;
};
```

As you see, there is only one method to implement: `halt_recipe` takes an integer instrumentation-kind indicator and returns a `HaltRecipe` pointer.

To carry the implementation, you subclass from `HaltRecipe`. For example, in the `x86_halt` library, you will find:

15b ⟨*class* `HaltRecipeX86` 15b⟩≡

And the implementation of `HaltRecipeX86::halt_recipe` just looks up the given kind indicator in a vector of recipes and returns the result.

To create the context object for your library, you derive another context class by multiple inheritance. For example, in `x86_halt`:

15c ⟨*x86 HALT context inheritance* 15c⟩≡
```
class X86HaltContext : public virtual X86Context,
                       public virtual HaltContextX86
{ };
```

As described in the OPI extender's guide [5], you establish this target-context object at runtime by registering a context-creator function, such as the one for `x86_halt`:

15d ⟨*context creator for* `x86_halt` 15d⟩≡

Here "registration" means storing the function in `the_context_creator_registry` under the name of the library. For instance, the function `init_x86_halt` contains the line:

```
the_context_creator_registry[k_x86_halt] = context_creator_x86_halt;
```

Organizing your recipes. As mentioned earlier, recipe classes in the `x86_halt` and `alpha_halt` libraries that come with HALT form a hierarchy rooted at class `HaltRecipe`. The primary subclasses are `HaltRecipeX86` and `HaltRecipeAlpha` (one in each of the HALT-enabled target libraries), and under these are the kind-specific classes like `HaltRecipeX86Cbr`. Typically, you only define `operator()` in the secondary classes, but sometimes you may want to override other virtual methods at this level, and you may also want to add helper methods and variables.

4.2 Extending HALT for Path Profiling

As mentioned in Section 2.2, we have implemented the Ball-Larus path-profiling technique [3] using HALT. While point profiling can identify hot *spots*, path profiling identifies hot *traces*, multi-block sequences that are worth subjecting to concentrated optimization. You can't just extrapolate from point frequencies. You lose information at every join point. For example, consider a piece of code that looks schematically like

```
if (...)
  A;
else
  B;
if (C)
  D;
else
  E;
```

The edge frequencies for `AC`, `BC`, `CD`, and `CE` might all be nearly the same, and yet paths `ACE` and `BCD` could be very hot, while `ACD` and `BCE` are very cold.

Path profiling detects such correlations that point profiling cannot, but it is tricky to extract the extra precision with acceptable efficiency.

Here's how the Ball-Larus method does it. At every point that can be the start of an acyclic path (e.g., the entry of a procedure or the head of a loop), a special register called the *path sum* register is set to a small constant value (which may differ from one path-start point to another). Then, at strategically-chosen points along any path, the path sum is incremented or decremented by other small constants. The Ball-Larus method cleverly picks the places to adjust the path sum and the values of the constant adjustments so that, when the end of a path is reached, the path sum register holds a small positive integer that uniquely identifies the path just taken. Furthermore, the path identifiers for a given procedure are in $[0, k)$, where k is the number of Ball-Larus paths in the procedure. So it is easy to use a path's identifying number to index an array of counters that collect path frequencies.

Instrumentation kind extensions. Our implementation adds two kinds of instrumentation to the ones that come with HALT, and it extends the purpose of an existing kind. For the points along a path where the path sum is incremented, we introduce a "lightweight" instrumentation kind, meaning that the inserted code doesn't save or restore state and it doesn't call an analysis routine. It can be just a single instruction that adds a signed constant to the path sum register. The kind identifier for this first new kind is `INC`.

The second new kind is for the ends of paths, where we want the instrumentation to record the traversal of the path just ended and to get ready to identify the next one. This time we use an analysis routine.

It needs to take the value of the path sum as an argument. At the end of a path, the path sum is the integer identifier of the path just traversed. The analysis routine can use it to find and increment a counter representing the execution frequency of the path. After calling the analysis routine, the path-end instrumentation code must initialize the path-sum register for detecting the next path. The kind identifier for this second new instrumentation kind is `PATHEND`.

Our third extension is a simple addition to the role of the `ENTRY` kind, which normally just records the fact that a procedure has been entered. For our Ball-Larus path-profiling implementation, we extend this kind to initialize the path-sum register as well. This prepares it for identifying any of the paths that begin at the entry of the procedure in question.

Instrumenting Ball-Larus paths. To identify Ball-Larus paths and compute the constant values that are used at runtime to initialize and increment the path sum, we've written a pass called `blpp`. This pass inserts `HaltLabelNote` annotations that mark the points at which the `PATH_SUM_INIT`, `PATH_SUM_INCR`, and `PATH_SUM_READ` instrumentation code should be inserted. These annotations carry the constant values computed by the Ball-Larus algorithm. Since `blpp` adds all the HALT annotations needed for path profiling, we don't have to run the `label` pass at all.

The `blpp` pass also creates a text file describing the Ball-Larus paths that it has identified. This file is read later, during profiling runs, so that analysis routines know how many procedures are being profiled and how many paths each one contains. And it is used still later to correlate profile results with source code.

After running `blpp`, we use the `instrument` pass as in ordinary point profiling, to turn the annotations into instrumentation code. We run `instrument` before register allocation because our path-specific instrumentation kinds introduce a virtual register to hold the path sum. This scheme relies on register allocation to assign it to a real register before the instrumented program is ready for execution.

Adding the extended kinds. Recall from Section 3 that the builtin kind identifiers like `CBR` and `LOAD` are defined as `enum` constants in namespace `halt`.

17a ⟨*kinds list* 5⟩+≡ ◁5

```
namespace halt {

enum {
    STARTUP = 0,        // whole-program entry
    CBR,                // conditional branch
    MBR,                // multi-way branch
    ENTRY,              // procedure entry
    EXIT,               // procedure exit
    SETJMP,             // setjmp call
    LONGJMP,            // longjmp call
    LOAD,               // reads memory
    STORE,              // writes memory
    BLOCK,                      // basic block
    CYCLE               // cycle count
};
} // namespace halt
⟨last kind value 17b⟩
```

The HALT library also uses a preprocessor macro called `LAST_HALT_KIND` to remember the value of the last builtin kind:

17b ⟨*last kind value* 17b⟩≡ (5 17a)

```
#define LAST_HALT_KIND halt::CYCLE
```

The new kinds we're adding must have identifiers whose values don't conflict with the ones built in, so we use the definition of LAST_HALT_KIND to start the new series of kind identifiers.

18a ⟨*blpp kinds list* 18a⟩≡

Then we redefine LAST_HALT_KIND to allow for possible further extensions.

18b ⟨*path-profiling last value kind* 18b⟩≡

Let's look at how the path-profiling instrumentation kinds are implemented for the $x86$ target. For each kind there is a recipe class, that is, a subclass of HaltRecipeX86. As with the builtin kinds discussed earlier, the point of these subclasses is to provide specialized operator() methods that control the insertion of instrumentation code.

One things that these methods have in common is that they need to refer to the path sum. They do this using a special local variable symbol with name __path_sum. A helper function called path_sum produces an operand representing this local variable.

Kind PATH_SUM_INIT. The operator() method for the procedure-entry instrumentation that initializes the path-sum register is short, because it only needs to insert one instruction.

18c ⟨*x86* PATH_SUM_INIT operator() 18c⟩≡

```
void
HaltRecipeX86PathSumInit::operator()
    (HaltLabelNote, InstrHandle handle, CfgNode *block,
     const NatSet*, const NatSet*)
{
    debug(2, "%s:HaltRecipeX86PathSumInit", __FILE__);

    Instr *mi =
        new_instr_alm(opnd_path_sum(), x86::MOV, opnd_immed(0, type_u32));
    append(instr_pot[KIND], mi);

    insert_instrs(AFTER, block, handle);
}
```

Note that the method above doesn't call follow_recipe, since the stages of a typical recipe are unnecessary in this case. It inserts a single instruction to zero the path sum after the instruction holding the HALT annotation with kind PATH_SUM_INIT. The blpp pass places this annotation on the proc_entry instruction of each procedure. Apart from the kind itself, this note carries no values.

Kind PATH_SUM_INCR. The instrumentation for incrementing the path sum is nearly as simple.

18d ⟨*x86* PATH_SUM_INCR operator() 18d⟩≡

```
void
HaltRecipeX86PathSumIncr::operator()
    (HaltLabelNote note, InstrHandle handle, CfgNode *block,
     const NatSet*, const NatSet*)
{
    debug(2, "%s:HaltRecipeX86PathSumIncr", __FILE__);

    Opnd path_sum  = opnd_path_sum();
    Opnd incr = opnd_immed(note.get_static_arg(0), type_s32);

    Instr *mi = new_instr_alm(path_sum, x86::ADD, path_sum, incr);
    append(instr_pot[KIND], mi);

    insert_instrs(AFTER, block, handle);
}
```

Again, a single instruction alters the path sum, this time by the addition of a signed constant. The value of the constant is carried by the HALT annotation that marks the instrumentation point.

Kind `PATH_SUM_READ`**.** This kind of instrumentation consumes the path-sum value at the end of a path and then resets the path-sum register. In the most general case, the inserted code must

- Add one last increment to the path sum.
- Call the analysis routine, passing the unique identifier of the instrumentation point, another numeric identifier for the current procedure, and the value of the path sum.
- Reinitialize the path-sum register to a constant value in preparation for identifying the next path.

The static arguments to the analysis routine, i.e., the numeric identifiers for the current procedure and the current point within it, are given by the HALT note. The dynamic argument is just the path sum. Sometimes the final path-sum increment along a path is folded into the end-of-path handling; that's when the increment instruction is inserted before the analysis routine is called. In all cases, we need to reset the path sum after the call. The value to set it to comes from the HALT annotation as a "static argument", although it is not a value that's passed to the runtime routine.

19 ⟨*x86* `PATH_SUM_READ operator()` 19⟩≡

```
void
HaltRecipeX86PathSumRead::operator()
    (HaltLabelNote note, InstrHandle handle, CfgNode *block,
     const NatSet *before, const NatSet *after)
{
    debug(2, "%s:HaltRecipeX86PathSumRead", __FILE__);

    Opnd path_sum  = opnd_path_sum();

    long incr_value = note.get_static_arg(0);
    if (incr_value != 0)
    {
        Opnd incr = opnd_immed(incr_value, type_s32);
        Instr *mi = new_instr_alm(path_sum, x86::ADD, path_sum, incr);
        append(instr_pot[KIND], mi);
    }

    // dynamic argument
    args.push_back(opnd_path_sum());

    // static arguments
    args.push_back(opnd_immed(note.get_unique_id(), type_s32));
    args.push_back(opnd_immed(note.get_static_arg(2), type_s32)); // proc id

    // Save away the value to which restore_state will cause the
    // path sum to be set.
    next_path_sum = note.get_static_arg(1);

    follow_recipe(PATH_SUM_READ, after);

    insert_instrs(AFTER, block, handle);
}
```

The method above is unusual in a couple of ways. Normally, the `static_args` helper would be used to put all static argument values ontto the `args` list of the analysis routine. In this case, however, the annotation has one `static_arg` field (the new path-sum value) that is not meant to be passed in that way. So the static arguments to the analysis routine are handled manually.

Furthermore, the code to reset the path sum has to be inserted *after* the analysis-routine call. The `operator()` method doesn't have a kind-specific "pot" for instructions that must go after the call, so instead class `HaltRecipeX86PathSumRead` overrides the helper method `restore_state`, which is called by `follow_recipe` after the call has been inserted in the instruction stream. The constant to load into the path sum is transmitted to `restore_state` through the instance variable `next_path_sum`. `restore_state` first invokes the method that it overrides, and then adds one instruction to reset the path-sum register.

20 ⟨*x86* `PATH_SUM_READ restore_state` 20⟩≡
```
void
HaltRecipeX86PathSumRead::restore_state(NatSet *saved_reg_set)
{
    debug(2, "%s:restore_state", __FILE__);

    HaltRecipeX86::restore_state(saved_reg_set);

    Opnd init = opnd_immed(next_path_sum, type_u32);
    Instr* mi = new_instr_alm(opnd_path_sum(), x86::MOV, init);
    append(instr_pot[RESTORE], mi);
}
```

5 Summary

The HALT system is a flexible and extensible way for a user to instrument a program to study its run-time behavior. HALT comes with tools for setting up typical instrumentation-assisted measurements. Using the Machine-SUIF OPI as leverage, HALT gives precise control over placement of instrumentation and transmission of data to run-time analysis routines. The library's design makes it easy to extend to new target machines and to new kinds of experiments based on instrumentation.

6 Acknowledgments

This work has been supported in part by an DARPA/NSF infrastructure grant (NDA904-97-C-0225), a NSF Young Investigator award (CCR-9457779), and a NSF Research Infrastructure award (CDA-9401024). We also gratefully acknowledge the generous support of this research by Advanced Micro Devices, Compaq, Digital Equipment, Hewlett-Packard, International Business Machines, Intel, and Microsoft.

Cliff Young and Mike Smith developed HALT as part of Machine SUIF version 1 [1]. The current library retains much of his original design. Adon Hwang, Vijak Sethaput and Dario Vlah made the initial port to Machine SUIF 2. This was completed by Mike Smith and Martha Mercaldi.

References

[1] C. Young. *The Harvard Atom-like Tool (Halt) Manual.* Harvard University, 1998.

[2] A. Sricastava and A. Eustace *ATOM: A System for Building Customized Program Analysis Tool* Proc. ACM SIGPLAN '94 Conf. on Prog. Lang. Design and Impl. New York: ACM, June 1994.

[3] T. Ball, J. Larus *Efficient Path Profiling* Proc. 29th IEEE/ACM Int. Symp. on Microarchitecture. Paris, France: ACM/IEEE, Dec. 1996: 46-57.

[4] C. Young and M. Smith. *Static Correlated Branch Prediction* ACM Transactions on Programming Languages and Systems, 21.5 (Sept. 1999): 1028-1075.

[5] G. Holloway and M. D. Smith. *An Extender's Guide to the Optimization Programming Interface and Target Descriptions.* The Machine-SUIF documentation set, Harvard University, 2000.