Comparing Two Implementations of a Memory Reference Analysis Tool

A Design Project Report

Presented to the Engineering Division of Graduate School

Of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering (Electrical and Computer)

by
I-CHUN LI
Project Advisor: Sally McKee
Degree Date: August 2006

1. Introduction

1.1 Cache Conflicts

The cache is a small, fast storage area where frequently accessed data can be stored, taking advantage of temporal and spatial locality of the accesses. Temporal locality implies that if a memory location is referenced, it will tend to be referenced again in the near future. Spatial locality implies that if a memory location is referenced, memory locations near it will tend to be referenced soon [1].

Generally, a cache is divided into many blocks with fixed-size collection of data containing the requested words retrieved from the main memory. Because the cache is smaller than main

memory, it is not possible to have all objects of interest in the cache at once. When data is loaded and it is not found in the cache, this is called a cache-miss. There are three types of cache misses: compulsory, capacity, and conflict. A compulsory miss happens the first time a data object is referenced and has not had a chance to be loaded into the cache. A capacity miss happens when a cache simply is not big enough to hold all of the data being referenced. A conflict miss is caused when more than one object of interest map into the same block in the cache [2]. These memory conflicts cause frequent swaps between different level memories and increase miss rates. Consequently, the performance will be extremely degraded as well as causing greater power consumption.

The memory conflicts can potentially be eliminated by reorganizing the code or adjusting the memory allocation as high miss-rate parts are known. It requires a profiler to monitor the client program's memory reference and an analysis tool to keep statistics about which memory structures cause cache conflicts. A cache simulator, Cache Stats [3], has been developed to gather and report statistics about these conflicts.

1.2 Cache Utilization Analysis Tool: Cache Stats

Cache Stats is a cache simulator and analyzer which reads in data from an instrumented file and runs this data through a cache simulator. The simulator keeps statistics on all variables

(text, data and bss) and also tracks variables allocated in the heap via malloc, calloc and realloc.

Cache Stats requires four input files: a trace file, a configuration file, a symbol file, and an executable binary. The client code should be instrumented by a program analyzer (FIT and Valgrind in this project) which will generate a memory reference trace file in the format Catch Stats requires. The configuration file defines the behavior of Cache Stats, indicating what kind of cache to simulate, what output to generate, and various other data structure parameters. The total size, block size, and associativities of the L1 and L2 caches can be specified in detail. The symbol file and the executable file are optional. The symbol file, which is generated by the command “nm” in Linux, contains a list of all the variable names and addresses. The executable file is compiled from the non-instrumented code. Cache Stats uses the executable file in conjunction with the "addr2line" tool to determine where in the code memory allocations happen.

When Cache Stats runs, many 64-bits counters are allocated for statistics. Next, the addresses of static variables are loaded from the symbol file. Information on the various memory areas are stored in variables of the “struct memory_area” type which are found via a hash table. The stack is treated as one large unified memory area and given its own memory_area. Dynamic memory allocations are treated specially, and there are additional statistics and tables kept for

them. In an attempt to approximate knowledge of data-types without parsing the source code, Cache Stats groups allocations of same sizes to be of the same data-types. After allocating the main infrastructure, the L1-icache, L1-dcache, and L2-unified cache are initialized and the trace file is opened. The program then loops, using the incoming data from the trace file to simulate the cache and build up the cache conflict information. Conflicts are recorded by taking the memory_area corresponding to the address causing a miss, and registering a conflict with the memory_area of the data structure being replaced. After the trace files ends, the results are reported to an HTML file

1.3 Static Binary Instrumentor vs. Dynamic Binary Instrumentor

Cache Stats requires some method of generating memory traces from the benchmarks of interest. The trace must include all memory accesses and also information on all dynamic memory allocations. There are two common types of analysis tools; one instruments the benchmarks source code, the other instruments only the compiled executable [4]. A source analyzer operates on the source code, and is independent from the machine's architecture or operation system. In contrast, the binary analysis analyzes a program at the level of machine code, either as pre-linked object code or post-linked executable code. It instruments the analysis code to the client binary directly and without any access to the source code.

Two main binary instrumentation methods are discussed here: the static binary instrumentation and the dynamic binary instrumentation (DBI). The static binary instrumentation occurs prior to run-time. It takes time to instrument the analysis code first and then execute the program for analysis. Unlike static instrumentation, dynamic instrumentation injects analysis code into the client program at run-time. DBI has at least two main advantages. First, the client program does not have to be prepared in any way in advance, which makes the analysis process a bit simpler, especially when client programs are frequently modified. Second, it naturally covers all client code. If client code and libraries are mixed, different modules are used, or client uses dynamically generated code, it would be difficult to instrument all codes statically. This guarantees the correctness for general usage.

This project compares two kind of binary instrumentor: FIT, the Flexible Instrumentation Toolkit, a statistic instrumentor [5] and Valgrind, a dynamic instrumentor [6]. FIT's implementation for Cache Stats had been previously developed and was known to generate reasonable results. Nevertheless, as a static instrumentor, FIT requires a slow and unwieldy instrumentation process, something which Valgrind does not need. The two methods of instrumentation will be compared using accuracy and slowdown as metrics, in order to decide whether it is beneficial to replaced FIT with Valgrind as Cache Stats' profiler.

1.4 FIT: The Flexible Instrumentation Toolkit

FIT, a Flexible open-source binary code Instrumentation Toolkit, is designed to be an ATOM compatible binary instrumentor (ATOM [7] was a classic, widely used static binary instrumentation tool, which could insert calls to arbitrary C code before and after functions, basic blocks, and individual instructions. It worked on Alpha only, and thus is unfortunately defunct now.). FIT's instrumentation is static; it requires all object files for the binary being linked, and the object files must be linked with a slightly modified GCC tool-chain. FIT consists of three parts (figure 1.1): the FIT front-end, the FIT instrumentation libraries, and the FIT support library. Like ATOM, FIT, requires an instrumentation file that indicates what points of the program should be instrumented, and an analysis file which defines what analysis code should be executed at those program points. FIT's front-end creates the instrumentor and compiled analysis code. The instrumentation file is linked to the instrumentation library to produce the instrumentor, and the analysis code is linked with FIT support libraries that provide the standard C-functionality. The instrumentor is then run on a binary executable program: it links the analysis code into the binary, and rewrites the binary to call the desired parameters of the analysis code. The detail of the internal organization of FIT's instrumentation is in [8] which is beyond the scope of this report.

FIT uses its own support library to avoid the standard C-library because using the latter will

disturb the run-time data structures of the analysis code in the program. FIT also has mechanisms that attempt to prevent the original data addresses from being changed. FIT was originally chosen to trace the program's memory reference for Cache Stats because of these attempts to preserve as closely as possible the memory access pattern of the original program.

Despite these good features, there are a few reasons that FIT is not suitable for Cache Stats. First, FIT has large overhead during instrumentation time. The memory used when instrumenting SPEC benchmarks can take gigabytes of RAM which will cause thrashing or even out-of-memory situations. Although FIT is a binary instrumentation tool, it requires the original object files from the compilation, and also requires the binary to be linked with a modified gcc tool chain, so effectively you will need the source code available to make full use of FIT. Another issue is that FIT currently only works on C programs, and some client programs of interest are programmed in C++ and FORTRAN. Finally, the static instrumentation requires the binary to be re-instrumented whenever the client code is modified. To go through the whole process whenever a part is changed is time consuming and inefficient.

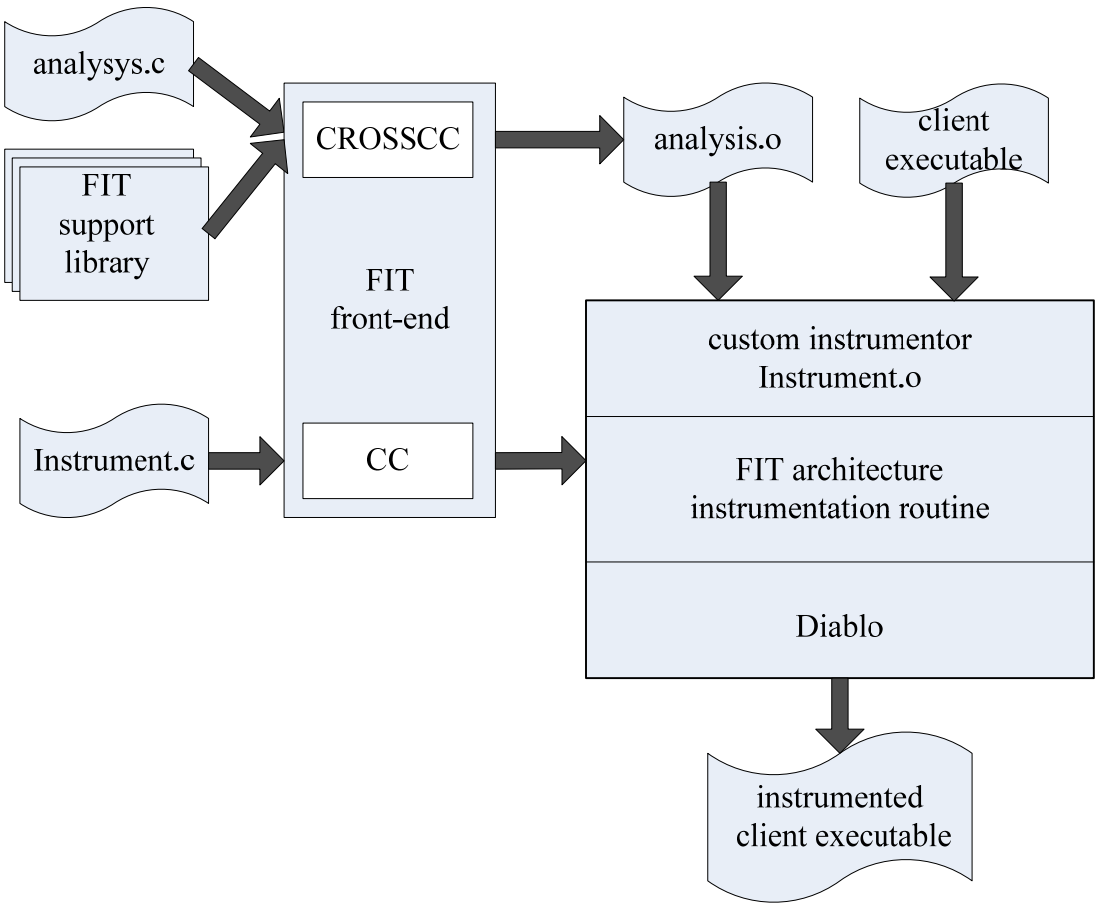

Figure 1.1 The design of FIT

1.5 Valgrind

Valgrind is an open-sourced DBI framework which provides low-level infrastructure to build up supervision tools, also called dynamic binary analysis (DBA) tools, such as profilers and bug detectors [9][10]. The Valgrind core emulates a synthetic software CPU, and Valgrind tools[1], which are plugged into the core, instrument and analyze the running program. Anyone can easily write and add arbitrary instrumentation to programs under Valgrind. This makes Valgrind ideal for experimenting with new kinds of debuggers, profilers, and similar tools.

Because Valgrind is execution-driven and uses binary translation, it covers all the codes of a client program which includes normal executable code, dynamically linked libraries, and dynamically generated code even if the source code is not available. Neither a skin nor its libraries need to be recompiled, re-linked with client programs before being run. Just prefix the client program's command line with Valgrind and everything works. These characteristics allow Valgrind to supervise programs written in any programming language, and it requires no compiler support, no code recompilation, no source code, and no special treatment for libraries. Figure 1.2 (a) gives a conceptual view of normal program execution, from the point of view of the client. The client can directly access the user-level parts of the machine (e.g. general-purpose registers), but can only access the system-level parts of the machine through the operating system (OS), using system calls. Figure 1.2 (b) shows how this changes when a program is run under the control of Valgrind. The client and the Valgrind tool are part of the same process, but the latter mediates everything the client does, giving it complete control over the client.

1. Contrast to Valgrind core, Valgrind tools are plug-in DBA tools of Valgrind. Valgrind's creators call them "skins." The terms "plug-in", "skin" and "Valgrind tool" are used as synonyms in this report.

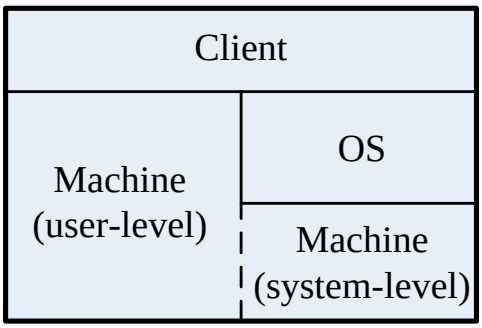

Figure 1.2 (a)

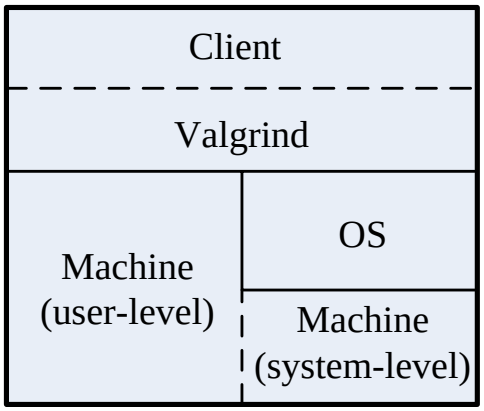

Figure 1.2 (b)

The following components are used at Valgrind start-up:

- Valgrind's loader (a statically-linked ELF executable)
- Valgrind's core (a dynamically-linked ELF executable)
- The plug-in, the skin (a shared object)
- The client program (an ELF executable, or a script)

Figure 1.3 demonstrates their relationship. When Valgrind runs, the loader does the first step to get the other three parts loaded into a single process sharing the same address space; the loader is not present in the final layout. The next stage is the basic block (BB) translation. Valgrind uses dynamic binary compilation and caching that grafts itself onto the client process at start up, and then recompiles the client code, one BB at a time, in a just-in-time (JIT) execution-driven fashion. To avoid the complexity of x86 instruction set, Valgrind translates the block of x86 instructions into its own intermediate representation (IR), a RISC-like instruction set, called UCode. This translation process involves disassembling and optimizing

the client program's x86 code into UCode, which is then instrumented by the skin, and then converted back into x86 code. (The design of UCode makes Valgrind easily be transfer to other platforms without redesigning the instrumentation methodology in the future.) The process utilizes the x86-to-x86 JIT compiler, a basic C library replacement, a low-level memory manager, the support for signals handling, and a scheduler. The result basic blocks are connected and stored in a translation table, a linear-probe hash table, to be rerun as necessary. Basic blocks are translated one-by-one, and once a translation is made, it can be executed (refer to [4] for more detail). The Valgrind core spends most of its execution time making, finding, and running translations. Finally, Valgrind generates the client program's original executing result and reports its own instrumentation's conclusion. The file opened by instrumentation will also be created.

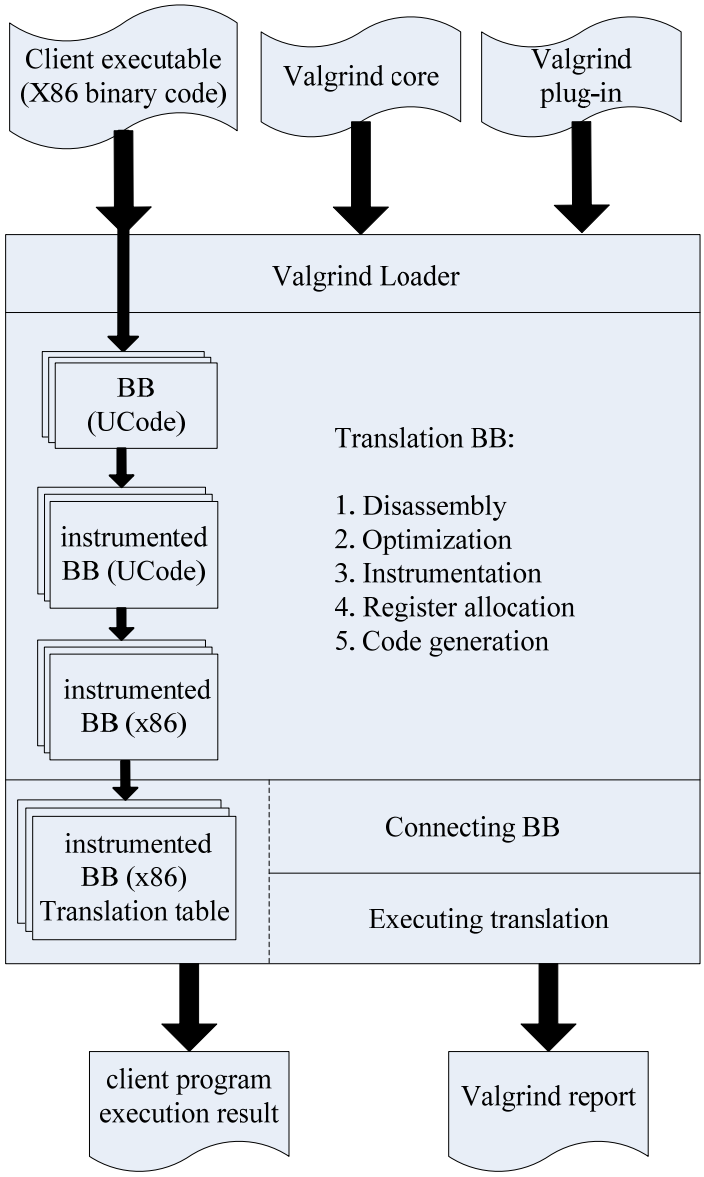
Client executable
(X86 binary code)
Valgrind core
Valgrind
plug-in
Valgrind Loader
BB
(UCode)
instrumented
BB (UCode)
instrumented
BB (x86)
Translation BB:
1. Disassembly
2. Optimization
3. Instrumentation
4. Register allocation
5. Code generation
instrumented
BB (x86)
Translation table
Connecting BB
Executing translation
client program
execution result
Valgrind report

Figure 1.3

2. Implementation of the Memory Reference Tracing Tool under Valgrind

This section presents a Valgrind tool, Cache Tool (CT, tentative name), that generates a trace file from a client program for the Cache Stats tool.

2.1 Writing a Valgrind Tool

Valgrind tools define various functions called by Valgrind's core for instrumenting programs. They are then linked against the *coregrind* library (*libcoregrind.a*, the Valgrind core library) that Valgrind provides as a C library replacement as well as the VEX library (*libvex.a*, the library for dynamic binary instrumentation and translation.) that provides the JIT engine.

Valgrind source code has already provided many tools for debugging, profiling, etc. On of

these skins, Nulgrind, does no instrumentation and can be used for Valgrind's developers to create a new tool [11]. Four basic functions have been set up in it:

- *pre_clo_init()*
- *post_clo_init()*
- *instrument()*
- *fini()*

The first two functions are used for initialization ("*clo*" stands for "command line options"). The "*pre_clo_init()*" contains most of the initialization such as the tool's name, version, and all the functionalities it needs. The "*post_clo_init()*" function is needed only if the tool provides command line options and must do some initialization after option processing takes place. The "*instrument()*" function allows developers to insert code into just-translated basic blocks of UCode. The "*fini()*" function is called when the translation and execution are finished. This is where final results, such as a summary of information collected, are printed. Any opened log files opened in the initialization functions can also be written and closed here.

Standard C library functions are avoided in Valgrind tools. Valgrind provides replacements for most functions in the C standard library to prevent interference and to ensure client programs are totally under Valgrind's control. Conventionally, functions and variables in the Valgrind core and replacement C library use the prefix "*VG_*" for identification. For example,

VG_printf() is used to replaces *printf()*. For Valgrind tools, the abbreviated name is prefixed. Therefore, "ct_" is prefixed in this case, such as: *ct_pre_clo_init(), ct_post_clo_init(), ct_instrument*, and *ct_fini()*.

2.2 Overview of the Trace Implementation

CT requires three parts to trace the client's memory reference:

- Tracing heap allocation: CT uses functions, *ct_malloc(), ct_calloc()* and *ct_realloc()*, to replace the client programs' heap allocation routines *malloc(), calloc(), realloc()*. It also replaces *free()* by *ct_free()* to free the above heap allocations.

- Tracing memory accesses, load and store: CT inserts instrumentation code to client programs' basic blocks to trace the store and the load instructions in *ct_instrument()*.

- Writing a trace file: A trace file is opened by CT, and the information gathered above is written to it according to the trace file format of Cache Stats (see Appendix A).

2.3 Heap Allocation

Heap allocations are dynamic allocation of memory. Currently, Cache Stats and CT handle only C heap allocations, but adding support for C++ and Fortran should be trivial:

- Malloc (size_t *size*): The *malloc()* function allocates a memory block of at least *size*

bytes. The block may be larger than *size* bytes because of space required for alignment and maintenance information.

- Calloc (size_t *num*, size_t *size*): The *calloc()* function allocates storage space for an array of *num* elements, each of length *size* bytes. Each element is initialized to 0.

- Realloc (void **memblock*, size_t *size*): The *realloc()* function changes the size of an allocated memory block. The *memblock* argument points to the beginning of the memory block. If *memblock* is NULL, *realloc()* behaves the same way as *malloc()* and allocates a new block of *size* bytes. If *memblock* is not NULL, it should be a pointer returned by a previous call to calloc, malloc, or realloc. The *size* argument gives the new size of the block in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block can be in a different location. Because the new block can be in a new memory location, the pointer returned by *realloc()* is not guaranteed to be the pointer passed through the *memblock* argument.

- Free (void **memblock*): The *free()* function de-allocates a memory block *memblock* that was previously allocated by a call to *calloc(), malloc(), or realloc()*. The number of freed bytes is equivalent to the number of bytes requested when the block was allocated (or reallocated, in the case of *realloc()*).

The function replacement is an important feature that Valgrind provides and is not directly related to the instrumentation. CT's replacement functions of these standard C memory management functions provide the necessary hooks for the heap memory event callbacks. These replacement functions can control details of allocation information and have code to write the allocation parameters into a trace file. In order to track the heap information, client executables should be dynamically linked. This is because Valgrind uses the *LD_PRELOAD* mechanism to intercept the *malloc()* calls. In the beginning, an allocation list, *ct_malloc_list*, initialized in *ct_pre_clo_init()* is created for re-allocation and accessed as a hash table. Whenever an allocation happens, a data structure *ct_Chunk*[2] records the allocation address, size, kind and PC is added, resized or deleted in the list by the function *add_ct_Chunk()*:

```
typedef
    struct _ct_Chunk {
        struct _ct_Chunk* next;
        Addr    data;                       // ptr to actual block;address
        SizeT   size : (sizeof(UWord)*8)-2; // size requested; 30 or 62 bits
        ct_AllocKind allockind : 2;         // which wrapper did the allocation
        ExeContext*     where;              // where it was allocated
    }
    ct_Chunk;
```

2. We use "chunk" in the following text to indicate the information data structure of allocations stored in the *ct_malloc_list.*

The parameter “kind” in *ct_Chunk* is defined as an enumeration (enum) of four kinds:

```
typedef
    enum {
        ct_AllocMalloc = 0,
        ct_AllocNew = 1,
        ct_AllocNewVec = 2,
        ct_AllocCustom = 3
    }   ct_AllocKind;
```

Only C’s allocations are implemented now and categorized as *ct_AllocMalloc*, other allocation categorization such as *ct_AllocNew* and *ct_AllocNewVec* for C++’s new/new[]/delete/delete[], and *ct_allocCustom* for other types are reserved for future use. A free list is also optional for future use.

2.3.1 Malloc() and Calloc()

Three functions are explained here:

```
void *ct_malloc ( ThreadId tid, SizeT n );
void *ct_calloc ( ThreadId tid, SizeT nmemb, SizeT size1 );
void *ct_new_block ( ThreadId tid, Addr p, SizeT size, SizeT align, \
                     Bool is_zeroed, ct_AllocKind kind, VgHashTable table);
```

The ct_malloc() and ct_calloc() functions replace the equivalent functions in the client program, and return allocation addresses. Actually, Valgrind implements these two allocations in much the same way as the original. The only difference is their size definition. The size of ct_malloc() is defined directly by parameter n, but the size of ct_realloc is the multiple of the element type argument nmemb and the size argument size1. The function ct_new_block() is

called to allocate memory and add chunks to the ct_malloc_list.

In ct_new_block(), the parameter *tid* is the thread id of the allocation. The *ct_new_block()* function assigns to address *p* a memory block having parameters *size* and *align* (*align= VG_clo_alignment= VG_MIN_MALLOC_SZB*, defined in the Valgrind core to ensure all block payloads are *VG_MIN_MALLOC_SZB*-aligned). The *is_zeroed* parameter determines if the element is to be initialized to zero or not (this parameter is true in *ct_malloc()* and false in *ct_malloc()*). The allocation *kind* is set to be 0 as mentioned before. When the allocation is a success, its chunk is stored in *table* (equal to *ct_malloc_list* in CT).

2.3.2 Free()

Three functions are referred to here:

```
void ct_free ( ThreadId tid, void *p );
void ct_handle_free ( ThreadId tid, Addr p, ct_AllocKind kind );
void die_and_free_mem ( ThreadId tid, ct_Chunk* mc, ct_Chunk**, \
                        prev_chunks_next_ptr);
```

The *ct_free()* function replaces the *free()* of client programs. It passes the allocation address *p* to *ct_handle_free()* to retrieve the chunk in the *ct_malloc_list*. Then the chunk is passed to *die_and_free_mem()* to be deleted, and its previous chunk's next pointer points to its next chunk. Finally, the allocation is freed in the memory by *VG_free()*.

2.3.3 Realloc()

The *ct_realloc()* function replaces the realloc() of the client program:

*void *ct_realloc (ThreadId tid, void* p, SizeT new_size);*

It allocates memory almost the same as *ct_new_block()*, with there cases. First, when *new_size* is equal to the original size of the memory block **p*, everything remains the same. Second, when *new_size* is smaller, the allocation size will be changed, and the size parameter in chunk will be updated. Third, when the new size is larger, a new space will be allocated as *ct_new_block()* does, and then the data is copied from the original allocation to a new one. Finally, the original allocation is freed as *ct_free()* does.

The *ct_malloc_list* is created only because re-allocation requires the original allocation's information. If a client program does not use re-allocation, the *ct_malloc_list* and the chunk related code can be removed, and the heap allocation replacement speed can be highly improved.

2. 4 Program Counter (PC)

In the heap allocation replacement functions, the allocation PC will be stored as a member of the ct_Chunk structure. The type is declared as:

```
struct _ExeContext {
        struct _ExeContext * next;
        /* Variable-length array.    The size is VG_(clo_backtrace_size); at
          *             least 1, at most VG_DEEPEST_BACKTRACE.    [0] is the current IP,
```

```
            *           [1] is its caller, [2] is the caller of [1], etc. */
          Addr ips[0];
    };
```

When the allocation happens, the functions' return addresses are stored in the *STACK*, and then the callee address is stored next to it. The callee PC can be found in the variable-length array *ips* here and also in a list which is dealt by *VG_(record_ExeContext)* [13]. Therefore, its caller PC, where the allocation instruction happens, is near the callee's in the *STACK*, in the *ips[1]* location, and it is written into the trace file.

2.5 Basic Block Instrumentation: Load and Store

The Valgrind JIT compiler translates the client program's x86 instructions to its IR (UCode) once per basic block (BB). When a BB is under instrumentation, Valgrind will create a new BB first and then put all the original instructions plus CT's instrumentation into it. The IR is defined in LibVEX [14], the library for dynamic binary instrumentation and translation of Valgrind. Here, only load/store instructions are considered. The loop in *ct_instrument()* will go through each statement in the BB and check whether the statement is a load/store or not. When a load/store statement is found, its address will be stored in *access_address*. Another variable *is_Load* is set to "True" when it is a load; otherwise, it is set to "False." Finally, *access_address* and *is_Load* determines what kind of trace data should be written to the trace file and adds appropriate instructions to do this to the instrumented BB.

2.6 Executing Valgrind Tool and Writing a Trace File

If a client program is normally run like this:

prog arg1 arg2

Then the command line of Valgrind with CT to instrument and execute it is:

./valgrind --tracefile= trace_file --tool=cachetool prog arg1 arg2

The "*--trace_file*" is optional to set the trace file name. Without this argument, CT will use a default file name. Usually, the trace file will be a UNIX fifo (named pipe) because the memory reference data may consume large amounts of disk space (normally about 2 GB in less than ten minutes).

When the execution starts, Valgrind will read the *ct_pre_cloinit()* first. This function defines the aforementioned heap allocation replacement functions and load/store instrumentation. It also calls the *ct_post_clo_init(),* which gets the file name from command, open the trace file and then writes a 16 byte long header. The header is mainly for future compatibility purposes; it reports the machine's size of long in bytes (i.e. 4 on a 32 bit machine, 8 on 64), whether the machine is big- or little-endian, and reports the version of the trace file format being used. When executing, the tool generates trace data for every replacement allocation function and memory reference (address, size, PC). Once instrumented once during the BB instrumentation, the JIT execution will execute the modified version of the BB. When the

execution finishes, the *ct_fini()* closes the file and terminates the whole process.

An example message is like this (use “a.out” as the client program and “tracefile.txt” as the trace file name):

> ./valgrind --tool=cachetool --tracefile="tracefile.txt" a.out
==3207== Cachetool, generates traces for the cachteool program.
==3207== Copyright (C) 2006 - Vince Weaver/yohowo
==3207== Using LibVEX rev 1367, a library for dynamic binary translation.
==3207== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==3207== Using valgrind-3.0.1, a dynamic binary instrumentation framework.
==3207== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
trace file tracefile.txt opened
header written
==3207== For more details, rerun with: -v
==3207==

(the message of the client program shows here)

==3207==

Cachetool:Exiting!

The number 3207 is the process ID and usually unimportant. Next it shows the declaration and information about Valgrind and its tool (CT here). The opened file name is shown in the next part, and then compilation and execution started and displayed the client program’s message. Finally, it showed the exit message when the process finished, and the trace file was generated (if not a pipe).

3. Simulation and Comparison of Implementations under Valgrind and FIT's Instrumentation

This section demonstrates the simulation process of client programs under the Valgrind or FIT profiler and the memory conflict analyzer, Cache Stats. The two tools' accuracy and speed are compared. The FIT instrumentation is presumed correct because it had been used extensively beforehand. The goal of this project is to prove that Valgrind has the same accuracy as FIT, and that Valgrind's implementation process is more convenient and to use as an input for Cache Stat.

3.1 Simulation Process

3.1.1 Machine Specifications

The simulation was done on the Sampaka Cluster belonging to the Computer System Lab at Cornell University. The cluster's specifications are:

- 40 1-U nodes
- Each node has two Pentium 4 2.8GHz Xeon Processors

- Each node has 2GB of RAM
- Connected by 1000Mb/s Ethernet
- Runs SuSE 9.3 Linux booted via PXE
- 90 Gigaflops with 64 CPUS
- Donated by Intel

The Network Batch System (NBS) [15] has been installed in the cluster. NBS is a suite of executable images and command scripts that implements a distributed load-balanced batch execution system. NBS is useful when many processes are required to execute at the same time, and it will count the final CPU time of processes.

3.1.2 Trace File

To run Cache Stats and the instrumented binary at the same time, the trace file should be a fifo named pipe. The trace file that is read is named */tmp/trace.PID* (in a temporary directory */tmp*) where PID is the process ID of the program being traced and assigned by the operation system. The PID is used because if more than one client is tracing, the two process would overwrite each other's traces and ruin results. The trace file can be consume a lot of disk space, so using a pipe can avoid running out of disk, which could potentially interrupting a simulation that may take hundreds of hours.

FIT's instrumentation code has been written to create the pipe file itself. (When Valgrind is used, the pipe is created in a script). To determine the proper trace file name generated by FIT, the FIT run is run in the background and the PID determined with the "$!" shell substitution.

3.1.3 Preparing the Simulation

Before the simulation, some files are prepared or generated first: the script, the symbol name file, and the config file. The script bundles the instrumented program execution and Cache Stats command line to run them at the same time through the pipe, and it simplifies the process of doing multiple runs with long pathnames. Symbol files are create by the command “nm” in Linux, and these files use .nm as extension names. The config file was described in section 1.2.

3.1.4 Benchmarks

Five SPEC CPU 2000[16] benchmark programs were simulated in this project (name, remarks):

- 164.gzip Data compression utility
- 175.vpr FPGA circuit placement and routing
- 186.crafty Chess program
- 177.mesa 3D Graphics library
- 183.equake Finite element simulation; earthquake modeling

The remaining benchmark, SMG2000 [16] (a parallel semi-coarsening multigrid solver) is a part of the ASCI Purple benchmark suite.

3.1.5 FIT implementation Simulation Process

Running the simulation of Cache Stats using FIT requires four steps. First, the client source

code should be compiled by FIT's patched gcc with its tool-chain. Since FIT needs statically linked executables, its gcc compiler will statically link against its own version of standard C library. This process generates an executable called a "nofit executable", which means the file has been compiled by the proper tool-chain but has not been instrumented yet. Second, the nofit executable is used to create the symbol file and also as Cache Stats's input executable because it contains the non-instrumented original client program code. Third, the nofit executable is instrumented by FIT along with the instrumentation and the analysis files. This process generates the final FIT executable that uses ".fit" as its extension. Finally, a script executes the ".fit" file to generate the trace and inputs the trace to Cache Stats along with the config file, symbol file, and nofit executable. Figure 3.1 depict the flows of FIT's implementation and simulation with Cache Stats. The part enclosed by the dashed line can be done together by a script. In the final simulation, the script was submitted via NBS and execution time was measured.

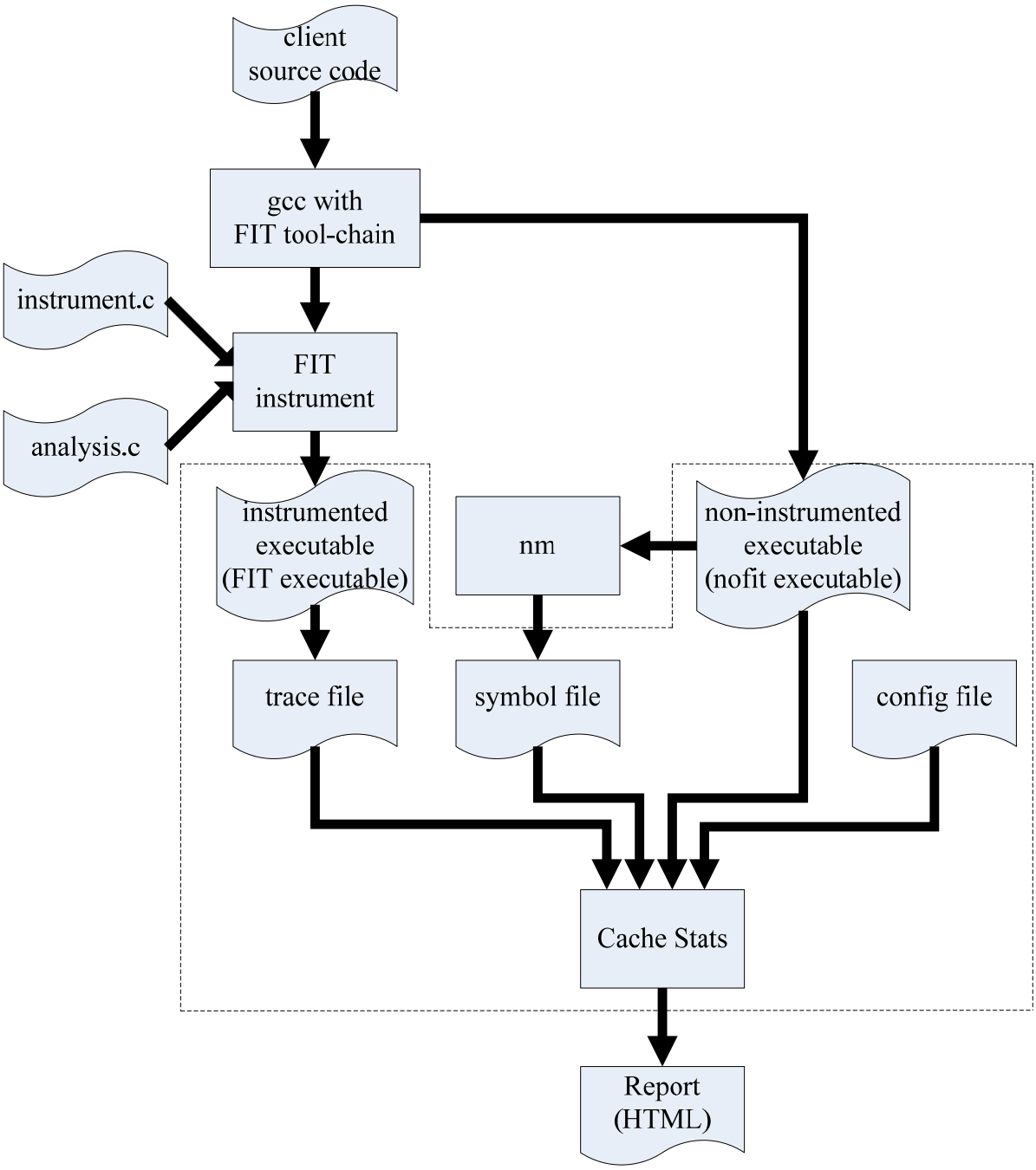

Figure 3.1 FIT & Cache Stats simulation flow chart

3.1.6 Valgrind Implementation Simulation Process

Running the Cache Stats simulation using Valgrind requires three steps. The process is similar

to FIT's (in section 3.1.5), but the third step of instrumenting analysis code and instrument code is not needed, because the client binary only has to be compiled once by gcc. Since Valgrind uses dynamic binary instrumentation, it is not like FIT which has to compile the client code first and then instrument it. Valgrind's JIT compiler instrument the client's binary executable on the fly and the library can be dynamically linked. Figure 3.2 depicts the flow of Valgrind's implementation and simulation with Cache Stats. The part enclosed by the dashed line can be done together by a script. In the final simulation, the script was submitted via NBS and the execution time was measured.

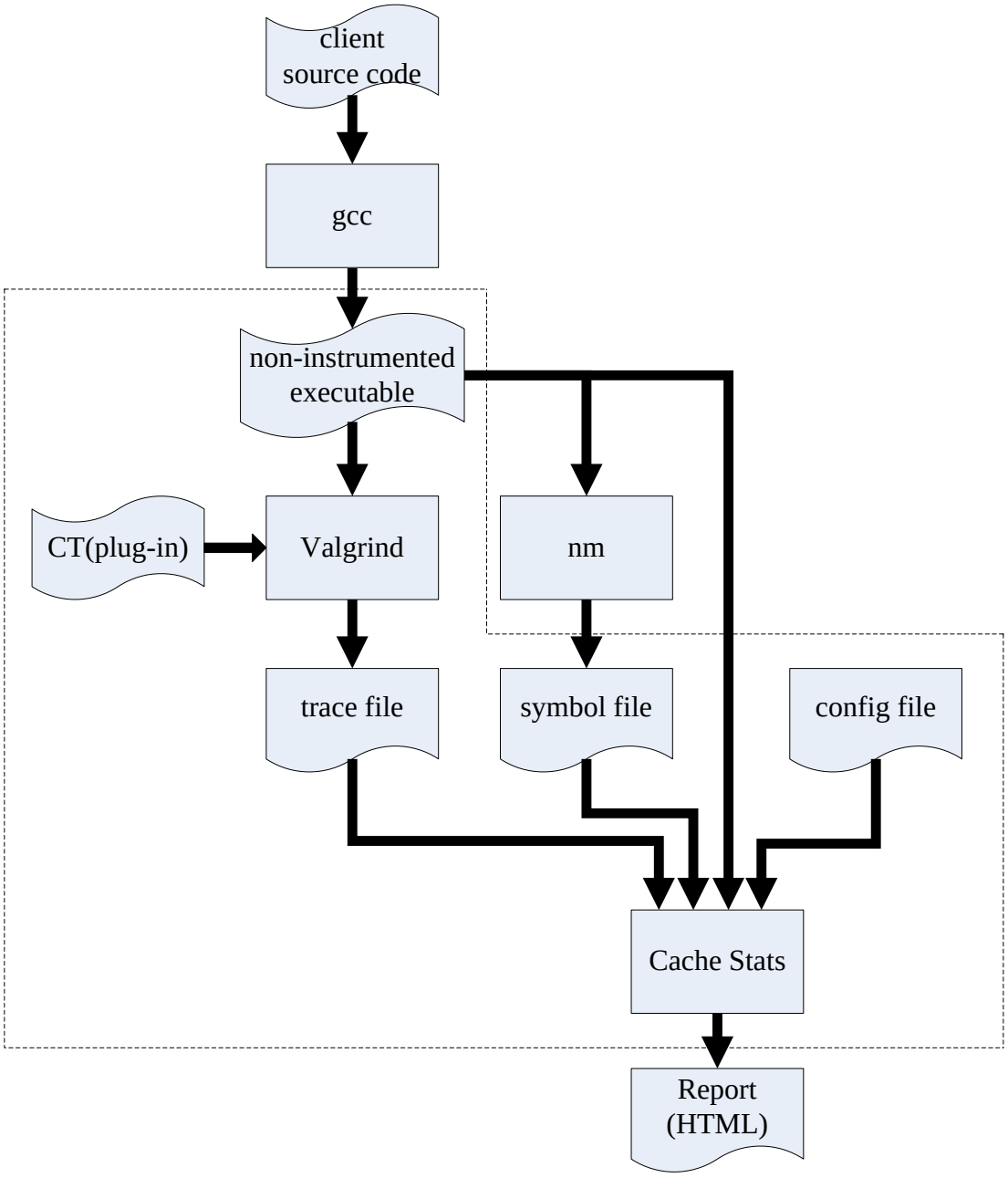

Figure3.2 Valgrind & Cache Stats simulation flow chart

3.1.7 Statistic Report

The output statistics are generated as an HTML file. Cache Stats counts the total access time, load/store time, heap allocation time, and hit rate for LI and L2 cache. The results are show in

tables, presenting the following cache conflicts statistics:

- Cache lines sorted by miss rate of L1 data cache and L2 cache
- Symbols sorted by miss rate of L1 data cache and L2 cache
- Allocated memory area conflicts by size of L1 data cache and L2 cache

Table 3.1 demonstrates a snapshot of the statistic in the output file. Many attributes are listed: symbol names (or the line number of the allocation in the original source code), sizes, segments (heap, static, stack or mmap), hit rates, addresses, and cache lines. They are listed in order according to their miss rate, and this helps programmers to figure out where case slowdowns happen.

Accesses	Name	Size	Segment	Address	Cacheline	Hit Rate	Overall Miss
1680402	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	307624	MMAP	0x40002004	ALL	89.491562	15.146231
1085979	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	116112	HEAP	0x84324ac	ALL	89.830282	9.472913
927141	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	116112	HEAP	0x83f1ddc	ALL	90.508024	7.548413
284659	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	70312	HEAP	0x83d4d6c	ALL	80.847611	4.676286
340106	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	13832	HEAP	0x83ede74	ALL	85.937619	4.102290
40338206	STACK	285212671	UNKNOWN	0xaf000000	ALL	99.881933	4.085050
508874	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	72128	HEAP	0x8473bf4	ALL	92.573211	3.241639
403964	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	55896	HEAP	0x84a4c3c	ALL	93.556604	2.232599
339792	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	51288	HEAP	0x84d07dc	ALL	93.610209	1.862315
203656	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	35056	HEAP	0x84fa184	ALL	94.220647	1.009554
203046	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	32752	HEAP	0x854120c	ALL	94.203284	1.009554
171124	/ufs/vault/vince/cache_suite/instrumented_binaries/trace_smg2000/utilities/memory.c:115 ptr = calloc(count, elt_size);	32752	HEAP	0x851e64c	ALL	94.010776	0.879093

Table 3.1 Cache conflicts of symbols sorted by miss rate of L1 cache of smag2000 (part)

3.1.8 Comparison Items

The first step in comparing results was using a stripped-down version of Cache Stats called Cache Sim. Cache Sim does not report detailed conflicts, it only shows cache hit and miss rates. Because of this, it can be used to roughly compare both implementations in a shorter time than Cache Stats does. The compared items include total access time (separated to read time and write time), hit rate of L1 data cache and L2 cache (read time, write time, hit time and miss time are reported for each cache) and heap allocation time (malloc, realloc, calloc and free). The access time may be slightly dissimilar because their linked libraries and compilation are different. Heap allocation also has this deviation, but hit rates and heap allocation time are supposed to be similar.

In order to remove some of the variations, “nofit” versions of the benchmarks compiled with FIT’s tool-chain were simulated. These executables are as close to the FIT instrumented binaries as possible (figure 3.3). Statically linked versions of the binaries can be compared for even closer similarity, but Valgrind is unable to intercept the allocation functions of statically linked binaries, so a full comparison cannot be done using this method.

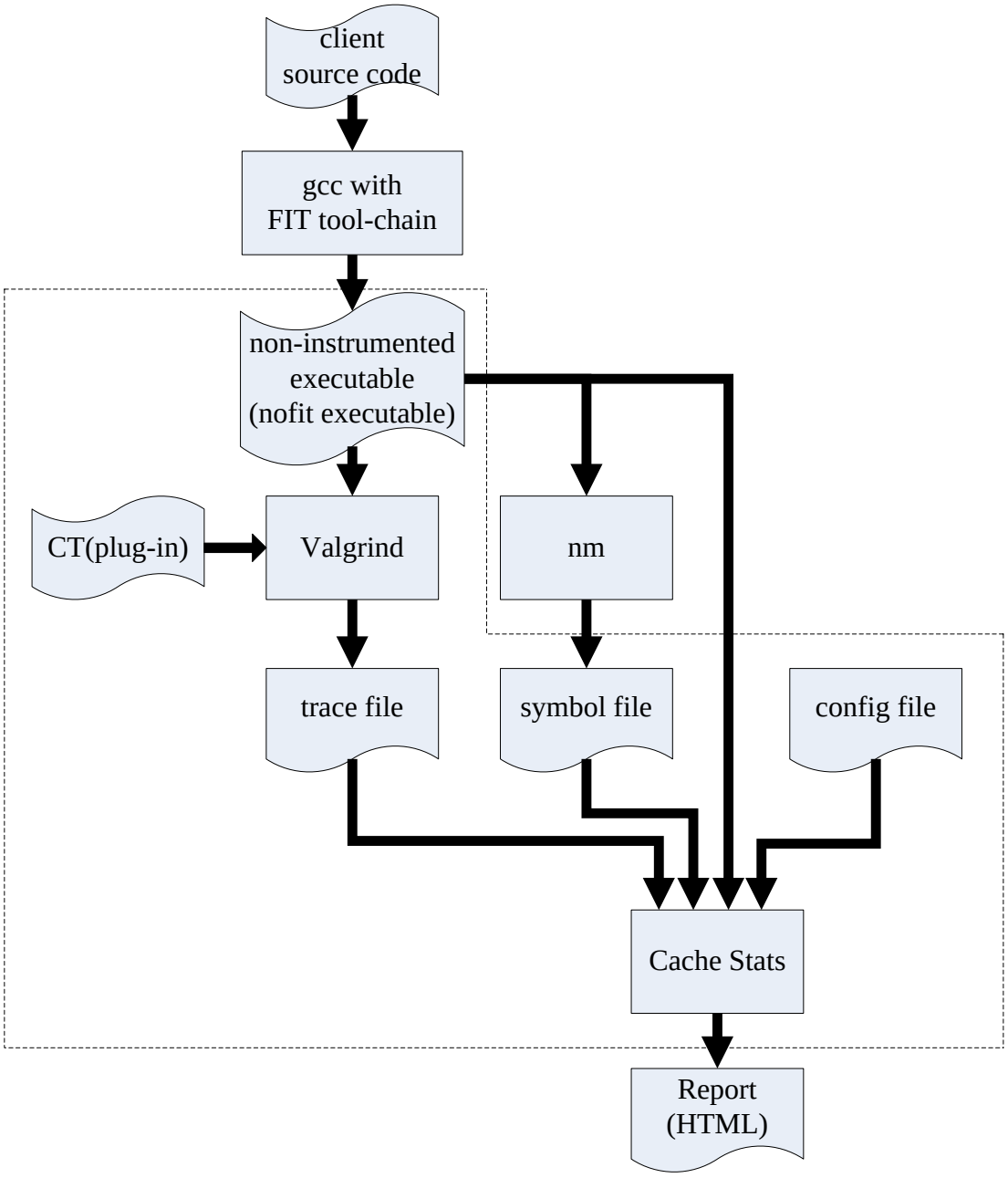

Figure3.3 Simulating nofit executable under Valgrind & Cache Stats

After the results from Cache Sim were compared, the more detailed statistics from the Cache Stats simulation were compared. The symbols sorted by miss rate are the crucial ones to guarantee that Valgrind's instrumentation is accurate. Orders, access times and miss rates of

symbols causing high conflict rates are compared to assure the hypothesis that Valgrind has similar accuracy to FIT. Finally, simulation time is compared to tell which one is faster and more practical for memory conflict analysis. NBS will report the total CPU time used by the instrumentation tool and Cache Stats together when finished. Also, the time of generating trace files by each implementation was compared.

3.2 Comparing Simulation Results

3.2.1 Memory Access and Conflict Analysis

Memory access times of each application are listed in table 3.2. Three simulation results are listed by row: FIT's executable and Cache Stats (figure 3.1), nofit executable under Valgrind and Cache Stats (figure 3.3), and original executable under Valgrind and Cache Stats (figure 3.2). The percentage column compares Valgrind's time to FIT's time. The formula is:

$$error(\%) = \left| \frac{\text{Valgrind's access time} - \text{FIT's access time}}{\text{FIT's access time}} \right|$$

For L1 cache, error percentages of access times between FIT and Valgrind (nofit) are all less than 2% and most of hit rates (table 3.3) between FIT and Valgrind (nofit) implementations are similar. Error percentages for L2 cache are higher, but it can be explained by comparing the conflict analysis. The details should be inspected manually. Comparing the table "Symbols, Sorted by Miss Rate" of both L1 cache and L2 cache of Valgrind and FIT's report, some C

libraries' functions will be listed in FIT's report but not Valgrind's because they are dynamically linked in Valgrind's executables. During FIT's compilation with its tool-chain, functions in the C library must be statically linked; thus they will be recorded in symbol files. On the contrary Valgrind linked them dynamically, and thus they are not in the symbol files. When Cache Stats finds a memory reference in the trace file but not in the symbol file, it will be registered as an "unknown" access rather than a conflict with a known symbol. These differences are not always significant; the primary usage is internal variables used by the C malloc arena allocation routines.[3.] However, if the nofit version's report is compared with FIT's, all the static linked symbols show. Sometimes contiguous symbols' order may swap because FIT and Valgrind's compilations are slightly different. Regardless of these static linked symbols, their results are almost the same. Therefore, the comparison concludes that two implementations have similar accuracy as profiling programs' memory references.

3. Sometimes, all symbols listed are from the static linked library, such as crafty. If a program's report is like this, it is not easy to find out where to reorganize the code to reduce conflicts.

		L1 data cache						L2 cac		
Application	Input	Accesses		Reads		Writes		Accesses		Rea
		#	%	#	%	#	%	#	%	#
mesa	FIT	177848025764		113346233483		64501792281		15930577718		15529713099
	Valgrind(nofit)	178460835890	0.34457	113643831165	0.26256	64817004725	0.48869	676797090	95.75158	321714977
	Valgrind	171349091752	3.65421	108706478421	4.09344	62642613331	2.88237	1025448293	93.56302	578345072
crafty	FIT	117201833163		82043624597		35158208566		19247468340		18731706574
	Valgrind(nofit)	117202036892	0.00017	82043746521	0.00015	35158290371	0.00023	5688195487	70.44705	5376136054
	Valgrind	128995405620	10.06262	85784478174	4.55959	43210927446	22.90424	5462578994	71.61923	5132833235
equake	FIT	90549626083		72756421441		17793204642		3457108757		3338937035
	Valgrind(nofit)	91292301374	0.82019	73203619276	0.61465	18088682098	1.66062	3284675098	4.98780	3166366094
	Valgrind	91907869413	1.50000	73047145207	0.39959	18860724206	5.99959	7420650954	114.64905	7156342253
gzip	FIT	48914844430		39575091830		9339752600		4161861193		3944394992
	Valgrind(nofit)	48914854806	0.00002	39575098112	0.00002	9339756694	0.00004	5259827896	26.38163	5153625592
	Valgrind	48614322014	0.61438	39032084801	1.37209	9582237213	2.59626	5273284085	26.70495	5171316865
vpr	FIT	50786613454		34996527844		15790085610		3021334831		2805885322
	Valgrind(nofit)	51294265148	0.99958	35300233362	0.86782	15994031786	1.29161	2708436582	10.35629	2487736117
	Valgrind	44449221037	12.47847	31270352584	10.64727	13178868453	16.53707	2643069919	12.51979	2410056443
smg2000	FIT	55904423		40552268		15352155		1402310		1105877
	Valgrind(nofit)	55922831	0.03293	40563947	0.02880	15358884	0.04383	1176657	16.09152	970763
	Valgrind	50572383	9.53778	37663924	7.12252	12908459	15.91761	1465372	4.49701	1213007

Table 3.2 Cache access time

		L1 data cache			L2 cache		
Applications	Input	Hit	Miss	Hit rate (%)	Hit	Miss	Hit rate
mesa	FIT	176602018331	1246007433	99.29	15762539089	168038629	98.94
	Valgrind(nofit)	177710775563	750060327	99.57	509363467	167433623	75.26
	Valgrind	170166473893	1182617866	99.30	857958254	167490039	83.66
crafty	FIT	111404408009	5797425154	95.05	19217187310	30281030	99.84
	Valgrind(nofit)	111516588962	5685447921	95.14	5661204424	24243497	99.57
	Valgrind	123537947692	5457457828	95.76	5430092144	27365684	99.49
equake	FIT	87253646484	3295979599	96.36	571700421	2884810848	16.53
	Valgrind(nofit)	88007173464	3285127910	96.40	400404050	2884723860	12.18
	Valgrind	84487218459	7420650954	91.92	928241816	6492409138	12.50
gzip	FIT	44752983237	4161861193	91.49	4111621567	50239626	98.79
	Valgrind(nofit)	43655026910	5259827896	89.24	5234252626	25575270	99.51
	Valgrind	43341037939	5273284085	89.15	5246874935	26409150	99.50
vpr	FIT	48170843704	2615769750	94.85	2335211744	686123087	77.29
	Valgrind(nofit)	48585828566	2708436582	94.72	2004461220	703975362	74.01
	Valgrind	42236338705	2212882329	95.02	1942177973	700891946	73.48
smg2000	FIT	54737085	1167338	97.91	1211022	191288	86.35
	Valgrind(nofit)	54746174	1176657	97.89	981805	194852	83.44
	Valgrind	49107011	1465372	97.10	1164015	353299	76.71

Table 3.3 Hit rates

3.2.2 Heap Allocation

Table 3.4 shows heap allocation times of each program. FIT and Valgrind handle malloc with different methods, and the recognition of malloc by their respective instrumentations may not be the same. Valgrind's tool, CT, uses replacement functions and *VG_malloc()* to allocate memory when *malloc()*, *calloc()* or *realloc()* happen. However, FIT's tool-chain compiler handles heap allocation by itself (FIT's core), and its instrumentation code simply inserts the trace writing code when heap allocations happen. Consequently, some allocations are not recognized equally by each

tool. The C library and loader internally call *malloc()* directly and Valgrind is not capable of reporting this properly. Because of this some early *malloc()* calls go unreported. Also, the Valgrind routines handle allocations in a different way and sometimes cause some simplifications. In smg2000, *realloc()* is called many times, but only with NULL as the first argument. FIT reports this as a *realloc()* followed by a *malloc()* (which is what the standard C library does), but Valgrind notices the NULL argument and thus reports these allocations as plain *malloc()*s, without the intervening *realloc()* call. Because of these differences, the malloc count may be slightly different, but by less than 10 calls, for every application. Despite these slight differences, the heap allocation statistics can be considered essentially the same between both implementations.

Applications	Input	malloc	calloc	realloc	Free
mesa	FIT	21	49	0	59
	Valgrind	13	49	0	59
crafty	FIT	47	0	0	2
	Valgrind	39	0	0	2
equake	FIT	1335075	0	0	0
	Valgrind	1335067	0	0	0
gzip	FIT	283466	0	0	283454
	Valgrind	283458	0	0	283454
vpr	FIT	73243	88	12	69269
	Valgrind	73235	88	11	69269
smg2000	FIT	40112	35314	1260	75418
	Valgrind	40104	35314	0	75418

Table 3.4 Heap allocation times

3.2.3 Simulation Time

Table 3.5 shows the time of every simulation, and table 3.6 compares their speed. Columns (3) and (4) in the table simply show how long it takes to generate the trace. From columns (5) to (8), two programs were executed at the same time, and the simulation time may have higher deviation. In table 3.6, it used FIT's time divided by Valgrind's. If the factor is greater than one, it means Valgrind's implementation is faster, and vice versa. Basically, Valgrind is faster than FIT in vpr, gzip and smg2000 whose access time is comparatively less than the other three. Valgrind takes longer time than FIT in larger access times applications, mesa, crafty, equake. Because Valgrind instrumented clients on the fly, it may require more memory. Valgrind only allocates a certain amount of space to hold cached JIT code, bigger programs might end up overflowing this cache and taking longer amounts of time to run. The results show that as the run time increases, Valgrind's slowdown compared to FIT's will also increase.

Another parameter to remember is that FIT spends an extra long time to compile and then instrumented a code (in (2), and it takes more than ten minutes for a 30 lines C code). Therefore, it may actually take a longer time for FIT's simulation if you factor in compilation and instrumentation time.

Applications	(1) Application execution time	(2) FIT instrument	(3) FIT trace	(4) Valgrind trace	(5) FIT & Cache Sim	(6) Valgrind & Cache Sim	(7) FIT & Cache Stats	(8) Valgrind & Cache Stats
mesa	0.05485 (3.291 min)	1.726*	29.297	29.449	193.429	181.896	189.183	398.514
crafty	0.3458 (2.075 min)	0.203 (12.2 min)	21.518	19.620	122.157	126.323	129.874	290.060
equake	0.04237 (2.542 min)	0.184 (11.08 min)	13.950	16.054	82.452	83.968	109.059	114.868
gzip	0.01830 (1.098 min)	0.141 (8.45 min)	12.421	7.328	44.768	30.684	**	80.623
vpr	0.03622 (2.173 min)	0.238 (14.25 min)	9.265	7.407	29.317	45.798	64.680.	44.179
smg2000	0.00006 (0.22 sec)	0.242 (14.52 min)	0.00968 (34.84 sec)	0.00803 (28.89 sec)	0.12736 (7.642 min)	0.01408 (2.465 min)	0.2026 (12.156 min)	0.10468 (6.821 min)

Table 3.5 Simulation time (hours)

*mesa took so long because it used up more than a gigabyte of RAM and was heavily swapping to disk when instrumented.

**gzip run under FIT and Cache Stats took a long time (more than 6000 hour). This may be caused by some bugs of Cache Stats and will be fixed in the future work.

(1) Application execution time without any simulation or instrumentation.

(2) The time of every application instrumented by FIT.

(3) (4) The time to trace every application and write the trace file by FIT or Valgrind.

(5) (6) The time to trace the application and run Cache Sim together

(7) (8) The time to trace the application and run Cache Stats (Conflict analysis) together

Applications	(3) ÷(4)	(5) ÷(6)	(7) ÷(8)
mesa	0.99	1.06	0.47
crafty	1.01	0.96	0.45
equake	0.99	0.98	0.95
gzip	1.70	1.46	N/A
vpr	1.86	0.64	1.46
smg2000	1.21	3.10	1.78

Table 3.6 Slowdown factor

4. Conclusion and Future Work

The simulations demonstrated that the dynamic binary instrumentor, Valgrind, has similar accuracy to the static binary instrumentor, FIT, but their speeds are dissimilar. Valgrind is faster on shorter running programs, especially when FIT's instrumentation time is considered. However, as the run time increases, this advantage degrades, and Valgrind may perform even worse than FIT (a new version of Valgrind has been released just after the project was finished that claims to have improved the speed. It may help to get better performance.). Moreover, since Valgrind uses dynamic binary instrumentation, it is not like FIT that has to compile and instrument binary codes separately before execution. Valgrind's JIT compiler instruments the client's binary executable on the fly and the libraries can be dynamically linked. This makes the instrumentation process of the

Valgrind implementation quicker than FIT's, and the total client-related file size of the Valgrind implementation is also smaller. Furthermore, Valgrind's tools are compiled separately from the clients' code, and it can save the instrumentation time while the client is still under development and modified frequently. What is more, Valgrind can instrument a binary which can be coded by any programming language, but FIT can only instrument a client's source code in C. Some benchmarks, such as 176.gcc (too large) and 252.eon (written in C++) of SPEC CPU2000, can not be simulated if using FIT as Cache Stats' profiler. Therefore, Valgrind is considered to be more practical and generic for a memory reference analysis and tracing tool.

The whole program (the Valgrind tool and Cache Stats) will be released as an open source memory conflict inspection tool. It will be used to help programmers to find out memory conflicts which degrade performance and waste power, and it can also simulation different memory architectures by modifying the config file. Programmers can reorganize their codes without extra hardware to improve the speed and decrease the power consumption. This can be especially useful for resource-restricted embedded systems.

However, the whole simulation time still seems very long. Both kinds of simulations require many days to simulate some benchmarks. The current architecture of the simulation process is that the instrumented tool writes the trace information to the pipe, and then the simulator reads the data and then simulates it. If the simulator and instrumentor can be combined, such as embedding the Cache

Stats in the Valgrind’s tool and then simulation and analysis report are all done in the tool, it may makes the tool become faster and more practical.

Furthermore, the whole analysis tool currently can only be run in X86 Linux machine and analyze C code. Nonetheless, Valgrind owns the benefit to analysis a binary code without the source code, and the action of expending its application to other platform is in progress. Therefore, Cache Stats may be developed to analyze codes other than C in multi-platform in the future. The end goal is that programmers and system developers will have a helpful tool to build up their applications and systems.

5. Reference

[1] David A. Patterson, John L. Hennessy, “Computer Organization & Design: The Hardware/Software Interface ”, Morgan Kaufman, 2nd edition, pp540-544, 1997

[2] John L. Hennessy, David A. Patterson, “Computer Architecture: A Quantitative Approach”, Morgan Kaufman, 3rd edition, pp. 392-402, 2003

[3] Cache Stats has been developed by Vince Weaver in Computer System Lab of Cornell University and not yet released
http://www.csl.cornell.edu/~vince/

[4] Nicholas Nethercote, “Dynamic Binary Analysis and Instrumentation”, PhD Dissertation, University of Cambridge, pp.1-34, November 2004.

[5] The Flexible Instrumentation Toolkit, FIT
http://www.elis.ugent.be/fit

[6] Valgrind
http://www.valgrind.org

[7] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94), pp. 196-205, Orlando, Florida, USA, June 1994.

[8] Bruno De Bus, Dominique Chanet, Bjorn De Sutter, Ludo Van Put, Koen De Bosschere, “The design and implementation of FIT: a flexible instrumentation toolkit.”, Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering, pp 29-34, June 2004.
http://portal.acm.org/citation.cfm?doid=996821.996833

[9] Nicholas Nethercote and Julian Seward, “Valgrind: A Program Supervision Framework”, Electronic Notes in Theoretical Computer Science 89 No. 2, 2003.

[10] Julian Seward and Nicholas Nethercote, “Using Valgrind to detect undefined value errors with bit-precision”, Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, April 2005

[11] “Valgrind Technical Documentation”, pp.109-119,June 7, 2006
http://www.valgrind.org/docs/manual/valgrind_manual.pdf

[12] Valgrind source code, “m_main” in Valgrind/coregrind

[13] Valgrind source code, “pub_tool_execontext.h” and “m_execontext.c” in Valgrind/coregrind/

[14] Valgrind source code, “libvex_ir.h” in Valgrind/VEX/pub

[15] Voyager Software, Network Batch System
http://www.vgersoft.com/nbs/

[16] Standard Performance Evaluation Corporation, SPEC CPU2000
http://www.spec.org/cpu2000/

[17] Lawrence Livermore National Laboratory (LLNL)
http://www.llnl.gov/asci/purple/benchmarks/limited/smg/

Appendix A: Trace File Format of Cache Stats

Trace File Version 10

By Vince Weaver

This is documentation on what the programs instrumented with the "fit_vmw_trace" instrumentation does:

The instrumented program first opens a named fifo (pipe) for writing called /tmp/trace.PID where PID is the process ID of the running instrumented program. The value of the PID can be accessed in a shell script by running the instrumented program in the background (ie followed by a &) and then using the shell substitution $! (such as /tmp/trace.$!).

The first thing written is a 16 byte long header:

byte0 = sizeof(long) in bytes (ie 4 on a 32 bit machine, 8 on 64)
byte1 = 1 if big-endian, 0 if little-endian
byte2 = version of trace file
byte3 - byte15 reserved (0 for now)

Then what follows is as follows, repeating until done, and all are unsigned long in type:

ADDRESS	(0)	
LOAD/STORE	(Load=0, Store=1)	
ADDRESS	(Address being loaded/stored)	
PC	(Program Counter of the Load/store instruction)	
SIZE	(Length in bytes of the value being loaded/stored)	
MALLOC_INFO	(1)	[happens before malloc call]
SIZE	(size of malloc)	
CALLSITE	(PC of where malloc happens)	
MALLOC_ADDRESS	(2)	[happens after malloc call]
ADDRESS	(address pointing to allocated memory)	
CALLOC_INFO	(3)	[happens before calloc call]
COUNT	(number of areas allocated)	

	SIZE	(size of areas allocated)
	CALLSITE	(PC of where calloc happened)
CALLOC_ADDRESS	(4)	[happens after calloc call]
	ADDRESS	(address pointing to allocated memory)
REALLOC_INFO	(5)	[happens before realloc call]
	OLD_ADDRESS	(address of region being realloc)
	SIZE	(size of new region)
	CALLSITE	(PC of where realloc happened)
REALLOC_ADDRESS	(6)	
	ADDRESS	(address pointing to allocated memory)
FREE_INFO	(7)	
	ADDRESS	(address of region to be freed)
FREE_FINISHED	(8)	
BLOCK_BEGIN	(9)	
	BLOCK	(block number of this block)
	ADDRESS	(address of beginning of block)
	SIZE	(length of block in bytes)
CALL	(10)	
	RETURN	(return address)
RETURN	(11)	[a "ret" instruction happened]

Appendix B: Source Code

```
/*--------------------------------------------------------------------*/
/*--- Cachetool: The cachetool interface.        ct_main.c ---*/
/*--------------------------------------------------------------------*/
/*
   This file is a tool built up in Valgrind for tracing heap allocation
   and memory reference. An output trace file will be created and then
   fed into a cache simulator to find out cache conflicts.

   Copyright (C) 2005-2006
   Vince Weaver vince _at_ csl.cornell.edu
   I-Chun Li        yohowo _at_ csl.cornell.edu

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.    See the GNU
   General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
   02111-1307, USA.

   The GNU General Public License is contained in the file COPYING.
*/

#include "pub_tool_basics.h"            //contains the basic types and other things needed everywhere.
                                        //also included libvex.h, a library for dynamic binary instrumentation and translation.
#include "pub_tool_libcassert.h"        //clib assert replacement
#include "pub_tool_tooliface.h"         //core/tool interface
```

```
#include "pub_tool_replacemalloc.h" //library functions replacement:malloc
#include "pub_tool_mallocfree.h"       //handle free
#include "pub_tool_libcprint.h"          //printing
#include "pub_tool_hashtable.h"        //Generic type for a separately-chained hash table.(for realloc)
#include "pub_tool_libcbase.h"          //VG_(memset)
#include "pub_tool_options.h"            //command line options
#include "pub_tool_libcfile.h"           //file I/O
#include "pub_tool_execontext.h"       //PC
//#include "pub_tool_stacktrace.h"

/*file I/O*/
/*trace file 16-byte header
   The first thing written is a 16 byte long header:
   byte0 = sizeof(long) in bytes    [ie 4 on a 32 bit machine, 8 on 64)
   byte1 = 1 if big-endian, 0 if little-endian
   byte2 = version of trace file
   byte3-byte15    reserved (0 for now)
*/
unsigned char header[16]={4,0,11,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned long output[10];
int file_ptr;
char FileName[256];    //trace file's name
#define VKI_O_LARGEFILE         0100000

/* Record malloc'd blocks */
VgHashTable ct_malloc_list = NULL;//initialized in void ct_pre_clo_init()

//get PC
unsigned long get_alloc_callsite(void *address);

//from /memcheck/mac_share.h
/* For malloc()/new/new[] vs. free()/delete/delete[] mismatch checking. */
typedef
    enum {
        ct_AllocMalloc = 0,
        ct_AllocNew       = 1,
```

```
        ct_AllocNewVec = 2,
        ct_AllocCustom = 3
    }
    ct_AllocKind;

struct _ExeContext {
        struct _ExeContext * next;
        /* Variable-length array.    The size is VG_(clo_backtrace_size); at
          *           least 1, at most VG_DEEPEST_BACKTRACE.    [0] is the current IP,
          *           [1] is its caller, [2] is the caller of [1], etc. */
        Addr ips[0];
};

/* first two fields must match core's VgHashNode. */
typedef
    struct _ct_Chunk {
        struct _ct_Chunk* next;
        Addr                data;                   // ptr to actual block
        SizeT               size : (sizeof(UWord)*8)-2; // size requested; 30 or 62 bits
        ct_AllocKind allockind : 2;    // which wrapper did the allocation
        ExeContext*     where;                  // where it was allocated
    }
    ct_Chunk;

/* memory allocation functions */
/* table is ct_malloc_list*/
void *ct_new_block ( ThreadId tid, \
                             Addr p, SizeT size, SizeT align, Bool is_zeroed, \
                             ct_AllocKind kind, VgHashTable table);
void *ct_malloc ( ThreadId tid, SizeT n );
void *ct_calloc ( ThreadId tid, SizeT nmemb, SizeT size1 );
void *ct_realloc ( ThreadId tid, void* p, SizeT new_size );

/* free memory functions */
void ct_handle_free ( ThreadId tid, Addr p, ct_AllocKind kind );
void ct_free ( ThreadId tid, void *p );
```

```
void die_and_free_mem ( ThreadId tid, ct_Chunk* mc,
                        ct_Chunk** prev_chunks_next_ptr);

/* Allocate its shadow chunk, put it on the appropriate list. */
static void add_ct_Chunk ( ThreadId tid, Addr p, SizeT size, ct_AllocKind kind, VgHashTable table)
{
    ct_Chunk* mc;

    mc                  = VG_(malloc)(sizeof(ct_Chunk));
    mc->data            = p;
    mc->size            = size;
    mc->allockind = kind;
    mc->where           = VG_(record_ExeContext)(tid);

    VG_(HT_add_node)( table, (VgHashNode*)mc );
}

/* Allocate memory and note change in memory available */
/* ThreadId tid
    Addr p: allocation address
    SizeT size: allocation size
    SizeT align: alignment VG_(clo_alignment)
    Bool is_zeroed: initialization for shadow area value
    ct_AllocKind kind: use only ct_AllocMalloc here
    VgHashTable table: ct_malloc_list
*/
__inline__
void* ct_new_block ( ThreadId tid,
                                Addr p, SizeT size, SizeT align, Bool is_zeroed, ct_AllocKind kind, VgHashTable table)
{
    // Allocate and zero if necessary
    if (!p){
        p = (Addr)VG_(cli_malloc)( align, size );
        if (!p) {
            return NULL;
        }
```

```
        if (is_zeroed) VG_(memset)((void*)p, 0, size);//initialize value to zero
    }

    add_ct_Chunk( tid, p, size, kind, table );
    return (void*)p;
}

unsigned long get_alloc_callsite(void *address) {
    ct_Chunk* mc;
    ct_Chunk** prev_chunks_next_ptr;

        /* the "chunk" has the execution context which has the stack */
        /* backtrace which knows where we were called from                    */
    mc = (ct_Chunk*)VG_(HT_get_node) ( ct_malloc_list, (UWord)address,
                                                (void*)&prev_chunks_next_ptr );

//     VG_(pp_StackTrace)(mc->where->ips,2);
//     VG_(printf)("Malloc from %x\n",mc->where->ips[1]);

    return mc->where->ips[1];
}

/* malloc */
void *ct_malloc ( ThreadId tid, SizeT n ) {

    void *temp_pointer;

    temp_pointer=ct_new_block ( tid, 0, n, VG_(clo_alignment),
                                /*is_zeroed*/False, ct_AllocMalloc,
                                ct_malloc_list);

    //VG_(printf) ("Malloc of size %d to address %p\n",n,temp_pointer);
    output[0]=1;            //MALLOC_INFO
    output[1]=n;      //allocation size
    output[2]=get_alloc_callsite(temp_pointer);      //PC
    output[3]=2;      //MALLOC_ADDRESS
```

```
    output[4]=(unsigned long)temp_pointer;    //allocation address
    VG_(write)(file_ptr,output,5*sizeof(unsigned long));

    return temp_pointer;
}

/* calloc */
void *ct_calloc ( ThreadId tid, SizeT nmemb, SizeT size1 ){

  void *temp_calloc_ptr;
    temp_calloc_ptr=ct_new_block ( tid, 0, nmemb*size1, VG_(clo_alignment),
                                                                    /*is_zeroed*/True,
ct_AllocMalloc,ct_malloc_list);

    //VG_(printf) ("Calloc of size %d to address %p\n",nmemb*size1,temp_calloc_ptr);
    output[0]=3;                          //CALLOC_INFO
    output[1]=nmemb;          //type argument from calloc
    output[2]=size1;          //size argument from calloc
    output[3]=get_alloc_callsite(temp_calloc_ptr);      //PC
    output[4]=4;                //CALLOC_ADDRESS
    output[5]=(unsigned long)temp_calloc_ptr;        //allocation address
    VG_(write)(file_ptr,output,6*sizeof(unsigned long));

    return temp_calloc_ptr;
}

/* free requires 3 function:
    ct_free -> ct_handlefree -> die_and_free_mem    */

void die_and_free_mem ( ThreadId tid, ct_Chunk* mc,
                        ct_Chunk** prev_chunks_next_ptr)
{
    /* Remove mc from the malloclist using prev_chunks_next_ptr to
        avoid repeating the hash table lookup.    Can't remove until at least
        after free and free_mismatch errors are done because they use
        describe_addr() which looks for it in malloclist. */
```

```
    *prev_chunks_next_ptr = mc->next;

    VG_(free) ( mc );
}
__inline__
void ct_handle_free ( ThreadId tid, Addr p, ct_AllocKind kind )
{
    ct_Chunk*    mc;
    ct_Chunk** prev_chunks_next_ptr;

    mc = (ct_Chunk*)VG_(HT_get_node) ( ct_malloc_list, (UWord)p,
                                       (void*)&prev_chunks_next_ptr );

    die_and_free_mem ( tid, mc, prev_chunks_next_ptr);
}
void ct_free ( ThreadId tid, void* p )
{
    //VG_(printf) ("free at:    %p\n",p);
    output[0]=7;
    output[1]=(unsigned long)p;
    output[2]=8;
    VG_(write)(file_ptr,output,3*sizeof(unsigned long));

    ct_handle_free( tid, (Addr)p, ct_AllocMalloc );
}

/* realloc */
void* ct_realloc ( ThreadId tid, void* p, SizeT new_size )
{
    ct_Chunk    *mc;
    ct_Chunk **prev_chunks_next_ptr;
    UInt         i;

    mc = (ct_Chunk*)VG_(HT_get_node) ( ct_malloc_list, (UWord)p,
                                       (void*)&prev_chunks_next_ptr );
```

```
output[0]=5;
output[1]=(unsigned long)p;
output[2]=(unsigned long)new_size;
output[3]=mc->where->ips[1];;
output[4]=6;

if (mc->size == new_size) {/* size unchanged */

    mc->where = VG_(record_ExeContext)(tid);
    //VG_(printf) ("realloc of the same size %d to address %p\n",new_size,p);
    output[5]=(unsigned long)p;
    VG_(write)(file_ptr,output,6*sizeof(unsigned long));
    return p;

} else if (mc->size > new_size) {/* new size is smaller */

    mc->size = new_size;
    mc->where = VG_(record_ExeContext)(tid);
    output[5]=(unsigned long)p;
   // VG_(printf) ("realloc of smaller size %d to address %p\n",new_size,p);
    VG_(write)(file_ptr,output,6*sizeof(unsigned long));
    return p;

} else {/* new size is bigger */

    Addr p_new;

    /* Get new memory */
    p_new = (Addr)VG_(cli_malloc)(VG_(clo_alignment), new_size);

    /* Copy from old to new */
    for (i = 0; i < mc->size; i++)
        ((UChar*)p_new)[i] = ((UChar*)p)[i];

    /* Free old memory */
    die_and_free_mem ( tid, mc, prev_chunks_next_ptr );
```

```
         /* this has to be after die_and_free_mem, otherwise the
              former succeeds in shorting out the new block, not the
              old, in the case when both are on the same list.    */
         add_ct_Chunk ( tid, p_new, new_size,
                                      ct_AllocMalloc, ct_malloc_list );

         output[5]=(unsigned long)p_new;
        // VG_(printf) ("realloc of bigger size %d to address %p\n",new_size,p_new);
         VG_(write)(file_ptr,output,6*sizeof(unsigned long));

         return (void*)p_new;
     }

}

/* Tell Valgrind this function has one parameter */
/* write load information to trace file*/
static VG_REGPARM(1) void print_Load (Addr a) {

     //VG_(printf)(" Load    of address: %p\n",a);
     output[0]=0;                                                    //address info
     output[1]=0;                                                    //0 for load
     output[2]=(unsigned long)a;    //load address
     output[3]=0;                          //PC
     output[4]=4;                          //?
     VG_(write)(file_ptr,output,5*sizeof(unsigned long));
}
/* write store information to trace file*/
static VG_REGPARM(1) void print_Store (Addr a) {

     //VG_(printf)(" Store of address: %p\n",a);
     output[0]=0;                                                    //address info
     output[1]=1;                                                    //1 for store
     output[2]=(unsigned long)a;    //store address
     output[3]=0;                          //PC
```

```
    output[4]=4;                        //?
    VG_(write)(file_ptr,output,5*sizeof(unsigned long));
}


/*------------------------------------------------------------*/
/*--- Our instrumenter                                     ---*/
/*--- Translates the Basic Block passed in as "bb_in"      ---*/
/*---       into a new "instrumented" basic block "bb"     ---*/
/*------------------------------------------------------------*/
//from ac_main.c

static IRBB* ct_instrument ( IRBB* bb_in, VexGuestLayout* layout,
                                    IRType gWordTy, IRType hWordTy ) {

    Int          i, access_size;
    IRStmt*      st;
    IRExpr*      data;
    IRExpr*      access_address;
    IRExpr*      guard;
    IRDirty*     di;
    Bool         isLoad;
    IRBB*        bb;

    /* Create a new basic block */
    /* We'll put all of the original instructions, plus our      */
    /* instrumentations into it, and return it back to valgrind */

    /* create an empty basic block */
    bb          = emptyIRBB();

    /* copy over configuration from the original basic block */
    bb->tyenv   = dopyIRTypeEnv(bb_in->tyenv);
    bb->next    = dopyIRExpr(bb_in->next);
    bb->jumpkind = bb_in->jumpkind;
```

```
/* Walk through each statement, */
/* from first (0) to last (bb_in->stmts_used) */

for (i = 0; i <    bb_in->stmts_used; i++) {

     st = bb_in->stmts[i];

     /* clear these variables */
     access_size         = 0;
     access_address    = NULL;
     guard    = NULL;
     isLoad = True;

     switch (st->tag) {

          /* Ist_Tmp means we are copying data into a */
          /* "Temporary" register                                    */
          case Ist_Tmp:
                    data = st->Ist.Tmp.data;
               /* We only care if it's a load instruction */
                    if (data->tag == Iex_Load) {
                         access_address    = data->Iex.Load.addr;
                         access_size        = sizeofIRType(data->Iex.Load.ty);
                         isLoad = True;
                              }
                              break;

          /* Ist_Store means we are storing data */
          case Ist_Store:
                         data    = st->Ist.Store.data;
                    access_address = st->Ist.Store.addr;
                    access_size = sizeofIRType(typeOfIRExpr(bb_in->tyenv, data));
                         isLoad = False;
                    break;
```

```
    /* We ignore these */
    case Ist_Put: /* We are copying some "guest state" */
    case Ist_PutI:/* We are copying some "guest state" */
    case Ist_Exit:/* We are conditionally leaving a basic block    */
    case Ist_NoOp:
    case Ist_IMark:
    case Ist_MFence:
            break;

    /* We are in a "dirty" function? */
    case Ist_Dirty:
        if (st->Ist.Dirty.details->mFx != Ifx_None) {
            /* We classify Ifx_Modify as a load. */
            isLoad = st->Ist.Dirty.details->mFx != Ifx_Write;
            access_size        = st->Ist.Dirty.details->mSize;
            access_address    = st->Ist.Dirty.details->mAddr;
            guard    = st->Ist.Dirty.details->guard;
          }
          break;

    /* Print an error if an unknown statement type */
    default:
        VG_(printf)("\n");
        ppIRStmt(st);
        VG_(printf)("\n");
        VG_(tool_panic)("unhandled IRStmt");
        break;
}

/* If we were a load or store, add a call to print it */
if (access_address) {

       if (isLoad) {
    /* Create a new "instruction" called "di" */
    /* This is a dirty instruction, meaning it has side effects */
    /* the "0" means we don't expect a return value */
```

```
            /* the "N" means we can pass many arguments          */
            /* We pass 1 argument, the name of the function, */
            /* a pointer to the function, and an "argument vector" */
            /* which in this case only has one, the address */
                di = unsafeIRDirty_0_N( 1, "print_address", &print_Load,
                                                    mkIRExprVec_1(access_address));
              }
            else {
                di = unsafeIRDirty_0_N( 1, "print_address", &print_Store,
                                                     mkIRExprVec_1(access_address));
                }
            /* If the call has arisen as a result of a dirty helper which
                references memory, we need to inherit the guard from the
                dirty helper. */ /* ??? */
            if (guard) {
                di->guard = dopyIRExpr(guard);
                }

            /* put the helper call into the new Basic Block */
                /* before the load or store */
            addStmtToIRBB( bb, IRStmt_Dirty(di) );
        }

        /* Make sure the original instruction gets added to the basic block. */
        addStmtToIRBB( bb, st );
    }
    return bb;
}

static void ct_post_clo_init(void) {

    SysRes sysr;

    sysr=VG_(open)(FileName,
                VKI_O_CREAT|VKI_O_TRUNC|VKI_O_WRONLY|VKI_O_LARGEFILE,
                VKI_S_IRUSR|VKI_S_IWUSR);
```

```
    if (sysr.isError) {
        VG_(printf)("file %s can not be opened\n",FileName);
    }
    else {
        VG_(printf)("trace file %s opened\n",FileName);
    }

    file_ptr=sysr.val;

    VG_(write)(file_ptr,header,16);
    VG_(printf)("header written\n");
}

    /* Parse the command line options */
static Bool ct_process_cmd_line_option(Char* arg) {

          // 12 is length of "--tracefile="
    if (VG_CLO_STREQN(12, arg, "--tracefile=")) {
        VG_(sprintf)(FileName,"%s",&arg[12]);
    }
    else {
        return False;
    }

    return True;
}


static void ct_print_usage(void) {
    VG_(printf) ("      --tracefile=<file>    filename to use for tracefile\n");
}

static void ct_print_debug_usage(void) {
    VG_(printf)("       (none)\n");
}
```

```
static void ct_fini(Int exitcode) {
    VG_(close)(file_ptr);
    VG_(printf) ("\n\nCachetool:Exiting! \n\n");
}

static void ct_pre_clo_init(void)
{
    VG_(details_name)                 ("Cachetool");
    VG_(details_version)              (NULL);
    VG_(details_description)          ("generates traces for the cachteool program");
    VG_(details_copyright_author)("Copyright (C) 2006 - Vince Weaver/yohowo");
    VG_(details_bug_reports_to)    (VG_BUGS_TO);

    /* set up default output file */
    VG_(sprintf)(FileName,"trace2.out");

    VG_(basic_tool_funcs)             (ct_post_clo_init,
                                       ct_instrument,
                                       ct_fini);

    VG_(needs_command_line_options)(ct_process_cmd_line_option,
                            ct_print_usage,
                            ct_print_debug_usage);

    VG_(needs_malloc_replacement)(    ct_malloc,        /* malloc()        */
                                      NULL,             /* new()            */
                                      NULL,             /* vec_new()       */
                                      NULL,             /* memalign()      */
                                      ct_calloc,        /* calloc()        */
                                      ct_free,          /* free()           */
                                      NULL,             /* delete()        */
                                      NULL,             /* vec_delete() */
                                      ct_realloc,       /* realloc()       */
                                      16);              /* redzone block size? */
```

```
    //initialize the hash table,from mac_share.c
    ct_malloc_list    = VG_(HT_construct)( 80021 );      // prime, big

}

VG_DETERMINE_INTERFACE_VERSION(ct_pre_clo_init, 0)

/*--------------------------------------------------------------------*/
/*--- end                                                          ---*/
/*--------------------------------------------------------------------*/
```