DEP: Detailed Execution Profile

Qin Zhao[1], Joon Edward Sim[2], Weng-Fai Wong[1,2]
[1]Singapore-MIT Alliance
[2]Department of Computer Science
National University of Singapore
{zhaoqin,esim,wongwf}@comp.nus.edu.sg

Larry Rudolph
Singapore-MIT Alliance
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
rudolph@csail.mit.edu

ABSTRACT

In many areas of computer architecture design and program development, the knowledge of dynamic program behavior can be very handy. Several challenges beset the accurate and complete collection of dynamic control flow and memory reference information. These include scalability issues, runtime-overhead, and code coverage. For example, while Tallam and Gupta's work on extending WPP (Whole Program Paths) showed good compressibility, their profile requires 500MBytes of intermediate memory space and an average of 23 times slowdown to be collected.

To address these challenges, this paper presents DEP (Detailed Execution Profile). DEP captures the complete dynamic control flow, data dependency and memory reference of a whole program's execution. The profile size is significantly reduced due to the insight that most information can be recovered from a tightly coupled record of control flow and register value changes. DEP is collected in an infrastructure called Adept (A dynamic execution profiling tool), which uses the DynamoRIO binary instrumentation framework to insert profile-collecting instructions within the running application. DEP profiles user-level code execution in its entirety, including interprocedural paths and the execution of multiple threads.

The framework for collecting DEP has been tested on real, large and commercial applications. Our experiments show that DEP of Linux SPECInt 2000 benchmarks and Windows SysMark benchmarks can be collected with an average of 5 times slowdown while maintaining competitive compressibility. DEP's profile sizes are about 60% that of traditional profiles.

Categories and Subject Descriptors

D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Run-time environments*

PACT'06, September 16–20, 2006, Seattle, Washington, USA.

General Terms

Measurement, Performance

Keywords

Profile, Dynamic Instrumentation, Memory Reference, Control Flow

1. INTRODUCTION

Tallam et. al [14] proposed an extension to Larus' work on WPP (Whole Program Paths) [6], called eWPP (extended WPP). eWPP is WPP with additional encoding of memory dependency information. However, eWPP requires a large overhead in terms of intermediate space needed (500MBytes of buffers) and execution time slowdown (23 times on average). eWPP requires a two-pass approach in order to handle interprocedural paths. This paper describes *DEP* (Detailed Execution Profile) which is collected in a single pass. It incurs considerably less overhead in terms of intermediate space needed (10MBytes per thread) and execution time slowdown (five times on the average). The way in which DEP is collected handles inter-procedural paths at virtually no extra cost. Last but not least, DEP can be applied to multi-threaded applications and encodes more information than eWPP because both dynamic control flow and memory references can be recovered.

Dynamic control flow supplies the information needed by path-based optimizations [3] and analysis [7], providing important details such as frequently executed basic block sequences. Memory reference information encompass details such as memory dependences and memory access patterns which are required by many memory hierarchy researches [12].

However, there are specific challenges in collecting both control flow and memory references *at the same time.* The storage space and time needed to obtain the complete profile may not scale with the number of instructions executed dynamically in the program. Also, completeness of the profile may be an issue if one does not have access to the library sources. Furthermore, profiling multi-threaded applications is made complicated by the sharing of code and data between threads.

This paper shows how these problems are addressed by DEP which is collected by *Adept* (A dynamic execution profiling tool), a runtime binary instrumentation infrastructure. The observation that memory is accessed through register-relative addresses leads to the intuition that a tightly-coupled record of control flow and register value changes is sufficient for the recovery of memory reference. Thus, Adept

collects DEPs through dynamic binary instrumentation by inserting code to record register values at the appropriate program points. Section 7 shows that this insight reduces both profile size and runtime overhead significantly.

The main advantages of DEP are as follows:

- A DEP has complete coverage of the program, including shared libraries, because it is collected by a runtime binary instrumentation system.
- Thread-private DEPs can be collected easily for multi-threaded applications.
- Adept and DEP does not rely on any special operating system, compiler, hardware, or modification of programs, so the approach can be applied in most modern architectures.
- The collection of DEP is very efficient, incurring a 5 times slowdown on average.
- DEP, as far as we know, is the first profile which compactly represents memory reference and control flow information.

The remainder of the paper is organized as follow: Section 2 discusses the format of DEP. Section 3 describes the collection framework, Adept. Section 4, 5 and 6 discuss the collection, optimization and analysis aspects of Adept. Experimental results are discussed in Section 7. This is followed by the related work and conclusion.

2. DETAILED EXECUTION PROFILE

In this section we describe how the control flow and memory reference information are represented in DEP before showing how the reference trace can be restored from the DEP representation. It should be noted that the control flow and the memory reference profiles are stored separately. In this way, the control flow profile can be used independently and different compression algorithms can be applied to each profile. For the rest of the paper, we shall refer to the control flow and memory reference components of a DEP as DEP_C and DEP_M, respectively.

2.1 Control Flow Profile: DEP_C

Control flow is usually represented as a sequence of basic blocks executed by an application. One naïve way of doing this is to note down the starting address of each basic block. Assuming a 32-bit machine, four bytes would be needed to tag a basic block. However, a full 4-byte tag is an overkill and would consume too much space. Using a 3-byte tag would cause mis-aligned memory reference in our profile buffer and would degrade performance. On the other hand, using a 2-byte tag limits us to only 65,536 basic blocks. In addition, smaller tags require some amount of static analysis of the application so as one is able to assign each basic block an unique tag. This is difficult in the case of indirect branches and dynamically generated code.

In DEP we use a 2-byte tag to represent most basic blocks with extra bytes for special cases. No static analysis is required. The 4-byte starting address of a basic block is split into two tags: *H-tag* for high 2 bytes of the starting address and *L-tag* for the low 2 bytes. During the profiling, changes to the H-tag are tracked. If two consecutive basic blocks have the same H-tag, only L-tag is entered into the profile buffer. When they have different H-tags, a special value of `0x0000` followed by the new H-tag are stored into the profile buffer to indicate the change. In the case of a L-tag being `0x0000`, two `0x0000` is appended into the buffer. In addition, the memory location `0x00000000` to `0x0000ffff` is reserved for Adept's use. In this way, two consecutive `0x0000`, `0x0000` uniquely represents the L-tag `0x0000`.

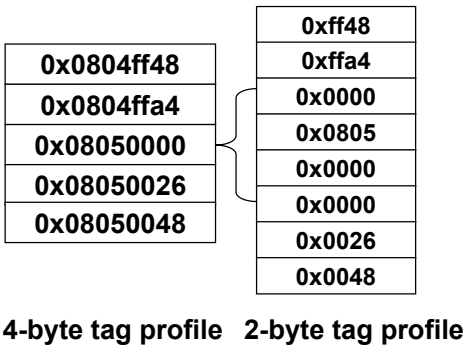

Figure 1: An example of H-tag and L-tag.

Figure 1 shows an example of how control flow is recorded. Let us assume that five basic blocks are executed in the following sequence: `0x0804ff48`, `0x0804ffa4`, `0x08050000`, `0x08050026`, and `0x08050048`. The first two and the last two basic blocks are only represented by their L-tags since there is no change in their H-tags, assuming the basic block preceding the first has a H-tag of `0x0804`. The third basic block has a different H-tag from the second basic block. `0x0000` followed by `0x0805` represents this change in the H-tag. Furthermore, because its L-tag is `0x0000`, an extra `0x0000` is inserted to avoid confusion. A 4-byte tag profile would use 20 bytes for the five entries, while DEP_C uses 16 bytes for eight entries. This way we are able to reduce the total profile size to as much as 60% that of the original 4-byte tag profile. In Section 7 we shall further show that this optimization does not compromise the compressibility of the profile.

Recovering the full 4-byte basic block execution trace from the DEP profile is straight-forward. Starting from the beginning of the profile, which must be `0x0000` and a H-tag, the 4-byte tag is the concatenation of current H-tag and L-tag.

It is also possible to use a single bit to represent the branching decision of the control flow [11]. This leads to a denser control flow profile but it would be harder to recover the full 4-byte basic block execution trace. In addition, such a stream of fairly random bits do not compress well. The comparison between the compressibility of the bit control flow profile and DEP_C is reported in Section 7.

Finally, it should be noted that the amount of space required by the control flow is small compared to memory reference trace, which will be discussed in the next section.

2.2 Memory References Profile: DEP_M

A complete memory reference should be represented as {`pc`, `addr`, `size`, `type`} to indicate the program counter of the memory reference instruction, the address of memory reference, the size of the data being accessed, and if it is a read or a write. Considering that the number of data

memory references can be very large even for a small program run, the execution and memory overhead involved for storing memory references can be very significant. This is especially the case for a CISC ISA like the Intel IA-32 architecture that has a small number of registers and most instructions are allowed to access memory directly. So it is a challenge to collect the complete memory reference information.

This is where DEP makes its contribution. The key insight is that in many modern architectures, a memory reference is addressed via several registers and an offset encoded in the instruction's operands. So with the registers' values and the memory reference instruction, the complete memory reference can be recovered. DEP_C is used to pin-point the memory reference instruction. DEP_M stores the values of the registers used for the address calculation. By storing only the necessary values, the profile size is minimized.

We also observe that a register's value may not change for every memory reference. For example, the stack pointer is usually updated at the entry and the exit of a function, and different local variables are referenced via a fixed stack pointer with different offsets. Another common scenario is when members of an object are referenced from a fixed object base by using different offsets. Making use of this observation reduces the profile size as well.

Below is a typical code sequence at beginning of a function. It first saves and updates the frame pointer `ebp`, and then allocates space for local variables and initializes them.

```
push ebp // store esp if necessary
mov  esp    → ebp;
sub  esp 16 → esp;
mov  0      → [esp + 4]; //store esp
mov  0      → [esp + 8];
```

There are three memory references: one `push` at the beginning and two `mov`s at the end. In the corresponding DEP_M, at most two values of `esp` are stored. The first value stored is for the `push` instruction if the `esp` has been updated and was not previously stored. The second value stored is for the the `mov` instruction (as shown by the comments above) since `esp` is updated by the third `sub` instruction. Although the `esp` is also updated in the first instruction by the `push`, the new value will not be stored since there is no memory reference that uses it.

Recovering the exact memory reference information from a DEP is more complicated than recovering the control flow information. Section 6.2 shows how the complete memory reference information can be recovered from DEP_C and DEP_M.

3. ADEPT: A DYNAMIC EXECUTION PROFILING TOOL

We have implemented a tool we called Adept to collect the DEP profile described above. Another contribution of this paper is that Adept is very efficient and usable in collecting different kinds of profiles. In this section, we will describe Adept framework in detail. Adept uses the DynamoRIO [5] binary instrumentation framework as a vehicle to perform its dynamic instrumentation on running applications. It is conceivable that Adept may be implemented using other dynamic binary instrumentation frameworks. The inserted profiling code is executed along with user code and collects

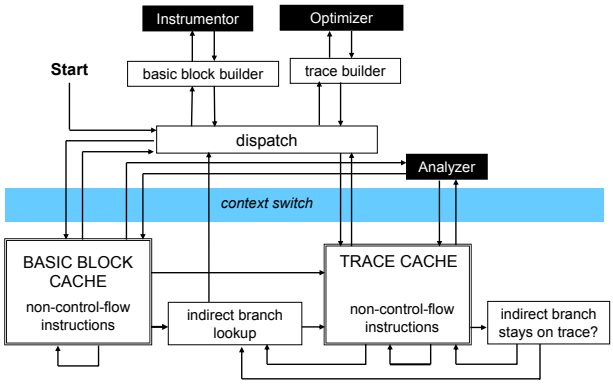

Figure 2: DynamoRIO and the Adept extensions.

the DEP into a profile buffer. We utilized the page protection mechanism to trigger an analyzer whenever the buffer is full. Any customized analysis can be implemented in the analyzer. After returning from analyzer, the buffer is reset and the application's execution continues. To reduce the profiling overhead, an optimizer is used to optimize frequently executed profiled code during the execution.

3.1 DynamoRIO

DynamoRIO is an efficient, and transparent runtime code manipulation system. It can execute and manipulate large, real world applications running on IA-32 hardware, under both Linux and Windows. Figure 2 shows the main components of DynamoRIO. The darkened components represents the extensions for Adept described in the later sections. DynamoRIO executes applications by copying the user code, one basic block at a time, into a basic block cache, and then executing the code from there. The copied code is the same as the original application code, with the exception that control transfer operations are modified so that DynamoRIO retains control of execution. DynamoRIO reduces its copying overhead by caching the basic blocks for future re-execution. To reduce the number of context switches between DynamoRIO and the application, it directly links blocks that are joined with direct branch as soon as they are encountered. A fast in-cache lookup is used to transit between blocks that are linked with an indirect branch. If some code fragments and paths are "hot" enough, they are stitched together as single-entry multi-exits traces and upgraded into a trace cache. DynamoRIO provides a programming interface for a client to manipulate the code in execution. Before emitting code into either caches, DynamoRIO calls hooks that are implemented by the client to allow any customized modifications of the application code.

3.2 Adept's extensions

Adept is built on top of DynamoRIO and consists of three added components: the instrumentor, the analyzer, and the optimizer.

- The *instrumentor* is invoked each time before a basic block is emitted into basic block cache. It instruments all of the user's code for collecting profiles such as DEP into a buffer.
- The *analyzer* is triggered periodically whenever the

profile buffer is full. It can be implemented as any customized online analysis on the profile, or it can compress or write the profile onto the hard disk for future off-line analysis. At the end of the analyzer, the profile buffer is reset and application execution continues with profiling.

- The *optimizer* is an optional component we implemented to improve the performance of Adept. It is called when basic blocks are upgraded into a trace. At that point, certain redundant profile code can be removed, thereby reducing the profiling overhead in frequently executed code (traces).

DynamoRIO maintains private code caches for each thread. Therefore, Adept is naturally able to handle multi-threaded code by maintaining runtime profile information for different threads simultaneously. Thanks to DynamoRIO, Adept can profile events for dynamically generated code and self-modified code without any difficulty since any user code will be copied into code cache before being executed.

4. COLLECTING DEPS IN ADEPT

The instrumentor is the key component of Adept, which is invoked before a basic block is emitted into cache and after it is built. The implementation of instrumentor determines how and what type of runtime information is collected. Our implementation is designed to collect the DEP of a program's entire execution in user space including dynamically loaded modules and dynamically generated code. All of the instrumented code, buffer and variables are thread private in order to support the profiling of multi-threaded applications.

4.1 Control Flow: Obtaining DEP_C

As described in the previous section, a basic block is tagged by a two bytes H-tag and a two bytes L-tag. The instrumentor inserts a stub code at beginning of each basic block to check if the H-tag of new basic block is different from the previous one. If the H-tag has changed, a value of `0x0000` and the new H-tag is stored. The L-tag is then written into the profile. For ease of understanding we will use C-like pseudo code to show the stub code for checking and profiling and ignore details such as register stealing. `H_tag` and `L_tag` are known constants when a basic block is built, and `prev_H_tag` is a variable holding previous H-tag value. The variable `cf_buf` is the control flow profile buffer and `cf_cnt` is the buffer counter. The pseudo-code of control flow profiling is shown below:

```
short cf_buf[CF_BUF_SIZE];

if(prev_H_tag != H_tag) {
   prev_H_tag       = H_tag;
   cf_buf[cf_cnt]   = 0x0000;
   cf_buf[cf_cnt+1] = H_tag;
   cf_cnt           = cf_cnt + 2;
}
cf_buf[cf_cnt] = L_tag;
cf_cnt         = cf_cnt + 1;
```

If the L-tag is `0x0000`, the last two statements are changed to

```
cf_buf[cf_cnt]   = 0x0000;
cf_buf[cf_cnt+1] = 0x0000;
cf_cnt           = cf_cnt + 2;
```

4.2 Memory References: Obtaining DEP_M

In order to minimize the updating of register values, Adept tracks the status of each register, *i.e.,* whether or not they have been updated since the last time it was recorded. Before executing any memory reference instruction, the status of the register used for address calculation is checked, and its value is written into profile buffer if necessary.

For each register, Adept uses a shadow variable to represent if it is `UPDATED` or `RECORDED`. The shadow variables are also thread private so that Adept can trace status of different threads' execution correctly. For each instruction that writes to a register, the instrumentor inserts an instruction to set its corresponding shadow variable as `UPDATED`. Before each memory reference instruction, code is inserted to perform a check to see if the register has been updated. If it has been updated, the register's value is written into the profile buffer and the shadow variable is set as `RECORDED`. Otherwise, the memory reference instruction is executed directly. We are therefore able to track register value changes and only record its value when necessary. As a result, we are able to produce a profile that is less than half of a straight forward memory address profile.

```
unsigned int mem_buf[MEM_BUF_SIZE];

if(esp_var == UPDATED) {
    mem_buf[mem_cnt] = esp;
    mem_cnt          = mem_cnt + 1;
    esp_var          = RECORDED;
}
esp_var = UPDATED;
push ebp;
ebp_var = UPDATED;
mov  esp     → ebp;
esp_var = UPDATED;
sub  esp 16 → esp;
if(esp_var == UPDATED) {
    mem_buf[mem_cnt] = esp;
    mem_cnt          = mem_cnt + 1;
    esp_var          = RECORDED;
}
mov  0 → [esp + 4];
if(esp_var == UPDATED) {
    mem_buf[mem_cnt] = esp;
    mem_cnt          = mem_cnt + 1;
    esp_var          = RECORDED;
}
mov  0 → [esp + 8];
```

The above is the instrumented version of the function entry code example given in Section 2.2. The original assembly code is shown in bold face. The other pseudo-code are instrumentation code. Checking and updating code have been inserted before the three memory reference instructions. The first three (original) instructions – `push`, `mov`, and `sub` – write to registers. Therefore, there is one register status updating statements for each instruction.

As is apparent, in order to minimize the size of the profile, there are a lot of dynamic checking that need to be performed. These can significantly degrade performance. In Section 5, we will describe how performance can be improved by optimizations that remove redundant checking.

4.3 The Profile Buffer

The profile buffer is used to temporarily store the collected profile for future analysis. The size of buffer depends on

the number of threads of an application since there is one buffer for each thread. A single threaded application can use a larger buffer to reduce the number of context switches and analyzer invocations. For multi-threaded applications, however, the buffer for each thread has to be smaller.

Each profile buffer is divided into two parts so as to store DEP_C and DEP_M separately. We found that a 80-20 split – with 80% of the buffer devoted to hold the memory reference profile – works reasonably well in practice.

In order to avoid checking to see if the buffer is full at each basic block, we used the page protection mechanism to trigger the analyzer when either parts of the buffer is full. Section 6 will elaborate on the details of this trick.

5. OPTIMIZING THE COLLECTION PROCESS

As discussed in Section 4, the instrumented user code includes many checks to avoid storing redundant information, degrading the application's performance. The optimizer is a component of Adept that streamlines the profiling of frequently executed code. Recall that in DynamoRIO "hot" basic blocks are stitched together to form traces. The optimizer is called when a set of basic blocks are upgraded to a trace. By taking advantage of the single-entry, multi-exits property of a trace, some checking and profiling code can be safely removed. The subsections below give some details on how this is achieved.

5.1 Optimizing Control Flow Profiling

The original instrumented control flow profiling code first checks if the H-tag has changed, and updates the H-tag if necessary. Then the L-tag is written into the profile buffer. In a trace, except the first basic block, the other basic blocks will not be executed unless their predecessor has been executed. So the check of the H-tag can be performed only once statically as the trace is built. The checks in the profile code are removed while the corresponding update and profile action is kept unchanged. This way, the overhead of dynamically checking H-tags is reduced.

We shall illustrate this optimization using Figure 1 as an example. Let us assume that the five basic blocks constitute a trace. A check for changes in the H-tag needs only be inserted in the first basic block. At the third basic block, we know that the H-tag is different, so a new H-tag together with the L-tag is written into the profile buffer. The other blocks simply store their L-tag into the profile buffer. The optimized profile code is as follows:

```
// bb 0x0804ff48
if(prev_H_tag != 0x0804) {
   prev_H_tag      = 0x0804;
   cf_buf[cf_cn]   = 0x0000;
   cf_buf[cf_cn+1] = 0x0804;
   cf_cnt          = cf_cnt + 2;
}
cf_buf[cf_cnt]   = 0xff48;
cf_cnt           = cf_cnt + 1;
...
// bb 0x0804ffa4
cf_buf[cf_cnt]   = 0xffa4;
cf_cnt           = cf_cnt + 1;
...
// bb 0x08050000
prev_H_tag       = 0x0805;
cf_buf[cf_cnt]   = 0x0000;
cf_buf[cf_cnt+1] = 0x0805;
cf_cnt           = cf_cnt + 2;
cf_buf[cf_cnt]   = 0x0000;
cf_cnt[cf_cnt+1] = 0x0000;
cf_cnt           = cf_cnt + 2;
...
// bb 0x08050026
cf_buf[cf_cnt]   = 0x0026;
cf_cnt           = cf_cnt + 1;
...
// bb 0x08050048
cf_buf[cf_cnt]   = 0x0048;
cf_cnt           = cf_cnt + 1;
...
```

5.2 Optimizing Memory Reference Profiling

As in the case in control flow profiling, there are a lot of checking in the profile code for memory references. Moreover, there are many register status update instructions. The optimizer scans the instructions in a trace looking for the following optimization opportunities. If an instruction tries to set a register's shadow variable which has already been set to an `UPDATED` status, this instruction can be removed. Furthermore, if a register's status is known, the checking code can be removed. The update and profile instructions are left unchanged if the register's status is `UPDATED`.

```
if(esp_var == UPDATED) {
   mem_buf[mem_cnt] = esp;
   mem_cnt          = mem_cnt + 1;
   esp_var          = RECORDED;
}
esp_var = UPDATED;
push ebp;
ebp_var = UPDATED;
mov  esp     → ebp
sub  esp 16 → esp
mem_buf[mem_cnt] = esp;
mem_cnt          = mem_cnt + 1;
esp_var          = RECORDED;
mov  0 → [esp + 4]
mov  0 → [esp + 8]
```

The optimized version of our previous memory reference profiling code is shown above. Several redundant profiling instructions has been removed. First, the second '`esp_var = UPDATED;`' is removed since the status of `esp_var` is already at `UPDATED`. Next, the conditional checking instructions for '`mov 0 → [esp + 4]`' is removed because the `esp`'s status is known to be at `UPDATED`. Finally, all the instrumentation code for the last instruction is removed because the value of `esp` has already been stored.

Other optimization opportunities exist. For example, all the update statements for `esp_var` can be removed because they are in the same basic block, and only the final '`esp_var = RECORDED;`' determines `esp_var`'s status. We did not implement these because the context in which they can be applied are not general enough. The expected returns did not warrant the effort.

6. ANALYZING DEP

The analyzer is invoked periodically when one of the two parts of the profile buffer is full. The analyzer can implement any customized analysis or simulation.

In this section, we describe the analyzer invocation mechanism, and some ways of using DEPs collected by Adept. In particular, we shall describe how the complete memory

reference information can be recovered from DEP and give an example of how DEP can be used to detect data race between threads of a multi-threaded application.

6.1 Optimizing the Analyzer's Invocation

To avoid the overhead of checking if the buffer is full on every buffer counter update, we use the operating system's page protection mechanism to trigger the analyzer. The last page of each part of the profile buffer is set to be 'not accessible'. Any attempt to write to a full profile buffer will cause a page access violation signal to be raised. In Linux, the `SIGSEGV` signal is sent to the process. In order to retain control of execution and to keep track of self-modifying code execution, DynamoRIO uses its own signal handler to intercept all the signal sent to a process before taking the corresponding action. We extended DynamoRIO's signal handler to check if the received `SIGSEGV` signal is caused by writing to a full profile buffer. If so, the analyzer is called with two parts of the profile as its parameters. On the return of the analyzer, the profile buffer is reset, and the register used as the profile counter is also reset to zero. The application's execution resumes, and the write instruction causing `SIGSEGV` signal is re-executed, only this time writing to start of profile buffer. A similar analyzer invocation mechanism has been implemented in the Windows version.

6.2 Recovering memory reference trace

In order to recover the memory reference trace, a traversal of all dynamically executed basic blocks is unavoidable. As these basic blocks are traversed, two pieces of information for each register need to be maintained: (a) the up-to-date values for the register which are obtained by reading DEP_M and (b) an `EXPIRED/CURRENT` status flag associated with each register indicating whether the value of the register is currently valid. The status flag information can be efficiently represented using a bit array.

The naïve approach of recovering the memory reference trace from a DEP is straight-forward and is akin to program interpretation. By traversing basic blocks indicated by dynamic control flow, we decode the instructions in each basic block sequentially. If an instruction alters the value of a register, the flag of the corresponding register is set to be `EXPIRED`. On the other hand, if the instruction references memory, we can calculate the address using the register values being maintained. However, if the status of any of the registers required for the address calculation is `EXPIRED`, the correct value has to be read from DEP_M before address calculation can proceed. Thereafter, the status of the registers are set to be `CURRENT`. In this way, we are able recover the complete memory reference represented as {`pc`, `addr`, `size`, `type`}.

This approach, as one might expect, is very time consuming. One of the main overhead is in the decoding of each instruction every time its basic block is encountered. As an optimization, instructions of each basic block are predecoded. Before the traversal, we decode each instruction and derive the following information from each memory reference instruction: (a) the registers used for address calculation and (b) the register values which has expired between the current memory reference instruction and its predecessor. These information can be efficiently represented by means of bit vectors associated with each of these instructions. Let us illustrate this with an example. Suppose the assembly code below are the first few instructions of a particular basic block.

```
sub eax 16 -> eax
add edx 32 -> edx
mov [eax, ecx, 4]16 -> ebx
```

For the third `mov` instruction, the registers used bit vector is `00000101b`, i.e., `eax` and `ecx` is needed. The bit vector for expired register values is `00001001b`, i.e., `eax` and `edx` has been overwritten prior to the current instruction. During the traversal of the dynamically executed basic blocks, whether these two values needs to be obtained from DEP_M depends on the value of the registers' status flags. Assuming that the status flags bit array is `00000000b`, only one value needs to be read from DEP_M to update `eax`. The status flags bit vector is updated to `00001010b` after processing this memory reference instruction. The key idea here is that the bit vectors can be obtained prior to traversal and the first two instructions can be ignored when the basic blocks are traversed during the actual address calculation.

6.3 DEP-based Data Race Detection

Different threads of a process share the same address space. A thread is able to access any part of memory that another thread is using. Although programmers can use locks to ensure atomicity, there is no way to enforce it in practice. A bug such as a buffer overflow or neglecting to properly lock critical regions of data can result in improper accesses to shared data. Moreover, if a data race is caused by improper arguments passed to the utility function in a dynamically loaded module and the un-authorized access happens in the utility function, it is even harder to isolate the problem through a source code level analysis approach. For instance, assuming two strings `src` and `dst` are passed to `strcpy`, and the length of `src` is longer than `dst`, a buffer overflow occurs on `dst` string could very well overwrite some data in a critical region. Such a bug may cause incorrect behavior due to data races. In such a scenario, it is hard to detect the data race using buffer overflow detection tools.

A customized Adept analyzer can be implemented for the data race detection over specific memory ranges. The analyzer maintains the software locks status. The status is updated when the function for lock request and release is encountered in the control flow. Information such as which software lock was accessed, and the result of the operation, for example whether a `trylock` successfully obtained the lock or not, can be recovered from the DEP. With the status of the locks, it is easy to check if any memory reference in the specified memory region was performed without requesting the proper lock.

The data race detection described above is but one example of how Adept and DEP can be used. Our framework opens up many more possibilities for both on-line and off-line performance tuning and debugging.

7. EXPERIMENTAL EVALUATION

In this section, we shall evaluate the performance of Adept, as well as the size, the compressibility, and the recovery overhead of DEPs. We ran the experiments on a dual-core 3.2GHz Intel Pentium D 840 processor with 2 GBytes of RAM which dual-boots to Linux Fedora Core 4 and Windows XP Professional SP2. For our experiments in Linux, we used the SPEC CPU2000 integer [1] benchmarks, which

Benchmark	Native (sec)	BB_pc+Mem_addr (sec)	DEP (sec)
164.gzip	181	943	625
175.vpr/place	91	462	319
176.gcc	77.3	618	513
181.mcf	182	355	282
186.crafty	114	983	613
197.parser	203	1081	880
252.eon.cook	53.1	262	176
253.perlbmk	183	1309	1064
254.gap	85.4	672	563
255.vortex	147	1552	1249
256.bzip2	178	979	648
300.twolf	305	1316	1053
Access	283	1147	1107
PowerPoint	349	517	526
WinWord	257	451	477

Table 1: Execution Time of Benchmark.

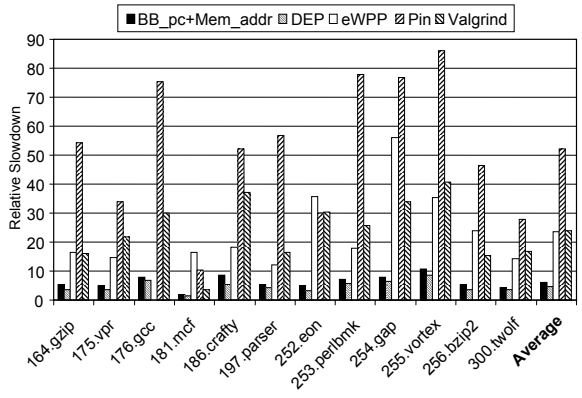

Figure 3: Relative slowdown of various profiling frameworks.

were compiled with gcc 4.0 using the '-O3' flag. In the Windows experiments, we selected several benchmarks from SysMark 2004 SE [2]. SysMark is a commercial benchmarking suite for Microsoft Windows. It measures performance based on real applications such as the Microsoft Office suite. SysMark runs and evaluates interactive applications by using the IBM Rational Visual Test suite to perform a series of mouse and keyboard actions simulating human operations. We chose the Microsoft Access, PowerPoint and Word benchmarks as we were able to run them individually instead of using the standard SysMark interface which runs all the benchmarks in a single pass.

In the following section, we shall use the following notations to describe various profile formats. BB_pc shall refer to the 4-byte basic block profile, consisting of the starting addresses of each basic block. CF_bit refers the bit vector control flow profile that represents conditional branches using bits and 4-byte target addresses for indirect branches. For memory references, Mem_addr refers to the memory addresses only profile, without the other components of `pc`, `size` and `type`. The definitions of DEP_C and DEP_M were given in Section 2.

7.1 Runtime Overhead

The first set of experiments assess the runtime overhead of Adept. In these experiments, we set the size of profile buffer for each thread to be 10 MBytes, consisting of 2 MBytes for control flow and 8 MBytes for memory references. The analyzer merely computes the total size of the profiles.

In order to evaluate the runtime overhead of collecting DEP, we implemented a version of Adept that collects a profile format that consists of a 4-byte basic block tag and the memory reference address (BB_pc+Mem_addr). Table 1 shows three execution times for each benchmark, namely the native execution time (Native), the time it took to collect a profile in the BB_pc+Mem_addr format, and the time taken to collect DEPs. The timings shown are taken by running the SPECint 2000 with reference inputs in Linux and SysMark benchmarks in Windows XP.

The collection of DEP profiles performs better than the collection of BB_pc+Mem_addr profiles.. There are two reasons for this. Firstly, for a given buffer size, the smaller profile record of DEP will trigger fewer numbers of calls to the analyzer compared to a more traditional profile format (BB_pc+Mem_addr). Secondly, to collect BB_pc+Mem_addr, a higher penalty has to be paid for the calculation of the memory address. This penalty includes stealing and restoring registers, the address calculation, storage of the address, and the update of the profile counter. The overhead of register profiling is lower as there is less need to steal registers and no need to perform address calculation. On the other hand, there is an extra overhead on checking for changes in the H-tag, and the checking and updating of register status. However, the optimizer removes many redundant checks and register updates in frequently executed code. This significantly reduced both the dynamic execution overhead and the code size.

From Table 1, we can see that PowerPoint and WinWord has a smaller relative slowdown with profiling. This is because PowerPoint and WinWord are interactive applications. So a large fraction of time is spent in waiting for events and I/O. Here, collecting BB_pc+Mem_addr outperforms collecting DEPs. We attribute this to the fewer number of 'hot' traces compared to the SPECint benchmarks. Consequently, there are fewer opportunities for the optimizer to remove redundant profile code.

We also compare our relative slowdown with that of three important previous works: Pin [8], Valgrind [9] and eWPP [14]. The Pin dynamic instrumentation framework could be used for building customized program-analysis tools through a provided rich set of APIs. Using these APIs, we implemented a control flow and memory reference profiler. The profiler inserts function calls into each basic block of a trace and before each memory reference. These function calls simply count the number of basic block executed and the number of memory references. Cachegrind is a cache profiler and simulator distributed with Valgrind. The cache simulator is invoked every 16 events or at the end of each basic block for the events that occurred in a basic block. To evaluate the profiling overhead, we modified Cachegrind so that basic block execution events and memory reference events can be captured and counted. Extended Whole Program Paths (eWPP) [14] is a representation for recording control flow and dependence information. It uses a two-phase profiling approach. In the first phase, a filter identifies all memory dependence. The second collection phase produces the eWPP. Using the CPU time of collection phase and the native execution times – both of which were reported in their paper –

Benchmark	BB_pc	DEP_C	CF_bit	Mem_addr	DEP_M
164.gzip	82,825	43,836 (52.93%)	4,908 (5.93%)	283,005	94,694 (33.46%)
175.vpr	55,998	29,480 (52.64%)	4,985 (8.90%)	234,068	86,703 (37.04%)
176.gcc	26,148	14,533 (55.58%)	4,081 (15.61%)	78,537	36,447 (46.41%)
181.mcf	40,869	20,464 (50.07%)	1,406 (3.44%)	97,634	33,989 (34.81%)
186.crafty	87,797	58,334 (66.44%)	20,027 (22.81%)	480,970	153,997 (32.02%)
197.parser	198,090	123,532 (62.36%)	55,414 (27.97%)	673,455	384,212 (57.05%)
252.eon	19,709	13,664 (69.33%)	9,950 (50.48%)	189,745	90,773 (47.84%)
253.perl	14,289	10,928 (76.48%)	6,839 (47.86%)	50,325	31,886 (63.36%)
254.gap	110,289	73,030 (66.22%)	48,966 (44.40%)	433,493	258,693 (59.68%)
255.vortex	72,969	44,746 (61.32%)	17,742 (24.31%)	278,306	195,841 (70.37%)
256.bzip2	49343	26,056 (52.81%)	7,785 (15.78%)	240,034	112,709 (46.96%)
300.twolf	162,951	94,251 (57.84%)	18,391 (11.29%)	603,199	268,381 (44.49%)
Average		(60.33%)	(23.23%)		(47.79%)

Table 2: Uncompressed Profile Size in MBytes. Runs conducted using *ref* inputs. The percentages shown for DEP_C and CF_bit indicates how their profile sizes compare with BB_pc while for DEP_M the percentages are calculated by comparing with Mem_addr

Bench	# thr	BB_pc			DEP_C			Mem_addr			DEP_M		
		min	max	avg	min	max	avg	min	max	avg	min	max	avg
Access	15	0.00484	112445	7555	0.0033	67855	4561	0.017	284622	19112	0.011	191790	12872
PowerPnt	45	0.005	12936	406.8	0.0037	7969	254.7	0.017	52153	1808	0.012	24454	875.1
WinWord	20	0.0069	16186	831.3	0.0049	9928	502.3	0.024	57236	2891	0.015	32126	1620

Table 3: Uncompressed Profile Sizes for multithreaded Windows benchmarks in MBytes.

we computed its relative slowdown. eWPP shows an average 23 times slowdown compared to native execution. This does not take the filter-phase into account.

As shown in Figure 3, Adept has the best performance with the lowest relative slowdown. Pin and Valgrind have significant overheads due to frequent analyzer invocations and context switches. Valgrind generally performed better than Pin because the analyzer is invoked only once every 16 events or at the end of a basic block. Pin on the other hand invokes the analyzer on every memory reference, as well as on the entry of basic blocks. eWPP performs better than Pin and Valgrind because its static instrumentation inlines the analysis code, doing away with context switches. The major overhead of eWPP lies in the runtime disambiguation checks. The main advantage of Adept lies in the delayed invocation of the analyzer because of the buffering of events.

7.2 Profile Size and Compressibility

We evaluated three types of control flow profile formats, namely BB_pc, DEP_C, and CF_bit. The results for Linux and multi-threaded Windows benchmarks are shown in Tables 2 and 3. In terms of profile sizes, DEP_C is 60% of BB_pc on the average, while CF_bit is 23%. For memory references, our DEP_M profile is half the size of the Mem_addr profile.

To assess compressibility, we profiled the SPEC integer benchmarks ran with the `test` input sets. Two compression algorithms, namely Sequitur [10] and bzip2 [13], were then used to compress the profiles stored on disk. The profiles that are too large for Sequitur to handle were split into smaller files and compressed separately.

Figure 4 shows the relative sizes of different compressed control flow, which are the quotients of the compressed profile sizes divided by the size of the uncompressed BB_pc profile for the same benchmark. Comparing profile formats, we find that DEP_C are nearly the same size as BB_pc when compressed by Sequitur and smaller when compressed by bzip2. In practice, DEP_C has an important advantage. During execution, keeping 2-byte tags implies that longer traces can be stored in the profile buffer before the analyzer is triggered. This translates to better performance especially if compression is done online. It is noteworthy that although CF_bit representation is small prior to compression, it does not compress well, and has the largest compressed file.

Figure 5 shows the relative sizes of various compressed memory reference representations which are the quotients of the compressed memory profile sizes divided by the size of the uncompressed Mem_addr profile for the same benchmark. DEP_M shows good compressibility with both compression algorithms, and is on average half the size of the compressed Mem_addr.

In general, DEP showed good compressibility in both compression algorithms. Sequitur generally performs better than bzip2, but bzip2 is able to handle larger data.

7.3 Recovering Memory References

We implemented the memory reference recovery in the analyzer invoked by Adept when the profile buffer is full. We evaluate the performance of reference recovery by comparing two scenarios: (1) the entire complete memory reference profile i.e. {`pc`, `addr`, `size`, `type`} is collected by Adept and during invocation of analyzer, the profile buffer is scanned once from the beginning to the end; and (2) DEP is collected by Adept and a memory reference recovery is performed when the profile is full.

Figure 6 compares the difference between the time spent in the analyzer by the two scenarios described above, nor-

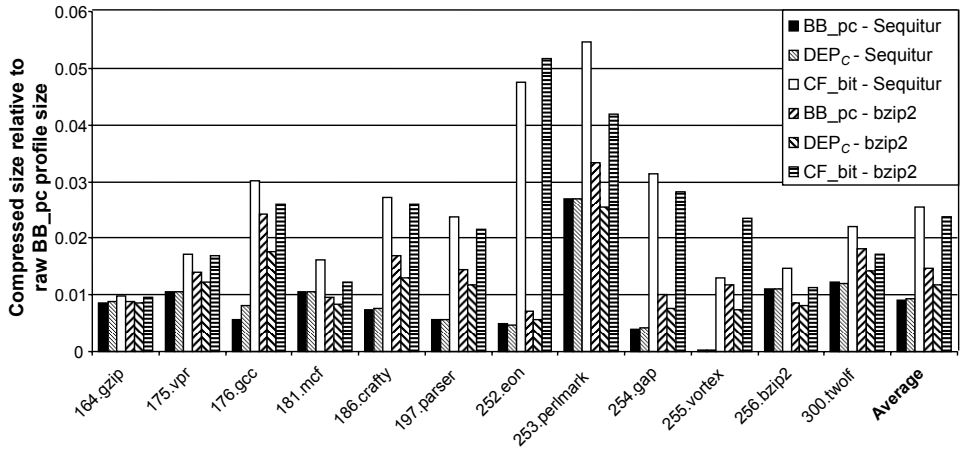

Figure 4: Relative size of compressed control flow profile.

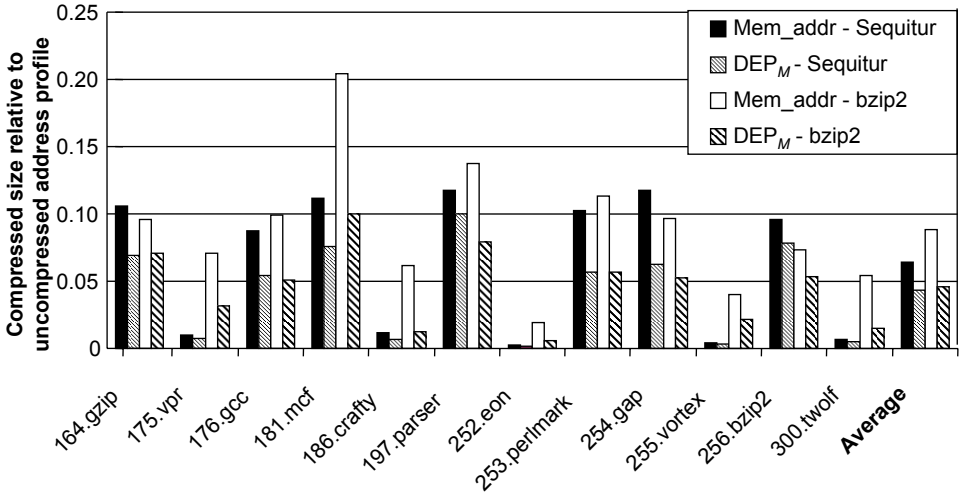

Figure 5: Relative size of compressed memory reference profile.

malized to the benchmark's native execution time, i.e. it is the difference of the running times of Adept with and without the above changes to the analyzer, divided by the native execution time of the benchmark. The benchmarks were run with the `ref` input set.

Scenario 1 spends an average of 7.5 times of native execution time in the analyzer. For each memory reference, there is one conditional branch to check whether the profile boundary is reached and at least four values, i.e., {`pc`, `addr`, `size`, `type`}, are read from memory. Recovering the memory references from DEP takes an average of 21 times of native execution time, or almost thrice the time of the scenario 1. As one would expect, there is a trade-off between profile sizes and recovery time.

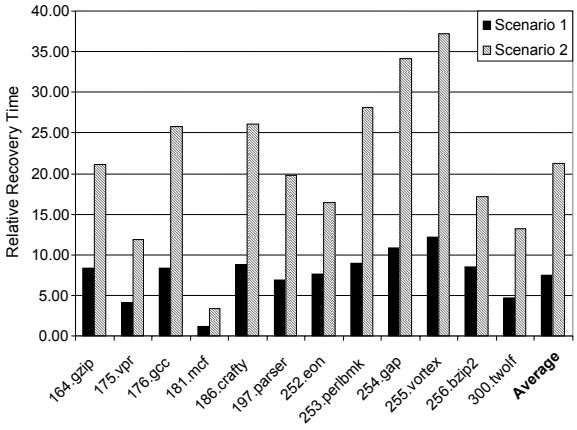

Figure 6: Overhead of Profile Recovery.

8. RELATED WORKS

The two works most closely related to DEP would be Whole Execution Traces (WET) [15] and extended Whole Program Paths (eWPP) [14]. Whole Execution Trace aims to be be a unified representation of different kinds of pro-

files that will serve as the basis for the study of the inter-relationships between these profiles. WET is implemented on Trimaran which is a simulation environment. Therefore, the real run-time overhead incurred in the collection of WET is not known yet. WET employs static instrumentation and thus it may not support multi-threaded applications easily.

eWPP seeks to encode memory dependences trace information in WPP. It captures the dependences by inserting disambiguation checks before use statements. However, in order to handle interprocedural paths, eWPP takes a two-pass approach. The first pass captures dynamic memory edges and the second pass inserts the disambiguation checks. The first pass accounts for a large part of the run-time overhead whereas Section 7 has shown that collection of DEP incurs little runtime overhead. Finally, eWPP is not easily extensible to multi-threaded applications because of the problem of sharing.

Other works have concentrated on compact, analyzable representation of dynamic control flow. Larus introduced WPP (Whole Program Paths) which is an extension of Ball and Larus [4] work on path profiling. The key idea is that a sequence of acyclic intraprocedural paths can be compactly represented by compressing this sequence with Sequitur [10]. Zhang and Gupta [16] extended this work by making subsets of information more accessible by the removal of redundant sub-paths. Manos et al. [11] proposed an efficient way to represent program paths as arithmetically encoded bit traces.

9. CONCLUSION

In this paper, we proposed DEP, a detailed program execution profile, which captures the major program execution information, including control flow, memory reference, and data dependency.

DEP is collected by Adept, a dynamic execution profiling tool, which is used to perform on-line or off-line analysis of large and/or multi-threaded applications. Adept does not require any special operating systems, compiler or hardware support. It is completely transparent to applications as it works on binary executables requiring no knowledge of the source code. Adept builds the mapping between collected information and original application. It is able to obtain runtime profiles from dynamically loaded modules, dynamically generated code, or even self-modified code.

Our experiment results show that Adept can collect a large amount of profile information with a low overhead, average less than 5 times slowdown compared to native execution. Furthermore, we were able to save up to 40% of space, compared to traditional profiles without compromising compressibility.

One of the properties of DEP that we would like to point out is that we do not need the complete trace to recover segments of program execution. That is, dividing the trace into segments and given the initial context of a segment, the program behavior of that particular segment can be easily reproduced. This may be especially useful for replay mechanisms or fast forwarding simulations.

10. REFERENCES

[1] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite, 2000. http://www.spec.org/osg/cpu2000/.

[2] Sysmark 2004 SE. SYSmark benchmark Second Edition, 2004. http://www.bapco.com/.

[3] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 72–84, 1998.

[4] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, 1996.

[5] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, September 2004. http://www.cag.csail.mit.edu/rio/.

[6] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269, 1999.

[7] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, 2003.

[8] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, pages 190–200, 2005.

[9] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004. http://valgrind.org/.

[10] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[11] Manos Renieris, Shashank Ramaprasad, and Steven P. Reiss. Arithmetic program paths. *SIGSOFT Softw. Eng. Notes*, 30(5):90–98, 2005.

[12] Shai Rubin, Rastislav Bodík, and Trishul M. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 140–153, 2002.

[13] Julian Seward. bzip2. http://www.bzip.org/.

[14] Sriraman Tallam, Rajiv Gupta, and Xiangyu Zhang. Extended whole program paths. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 17–26, 2005.

[15] Xiangyu Zhang and Rajiv Gupta. Whole execution traces. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–116, 2004.

[16] Youtao Zhang and Rajiv Gupta. Timestamped whole program path representation and its applications. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 180–190, 2001.