TRACING MULTITHREADED WINDOWS APPLICATIONS

Hsien-Hsiang Lin *Chung-Ta King*
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan, R.O.C.
king@cs.nthu.edu.tw

Abstract

Performance evaluation is critical in developing computer systems. To evaluate the performance and to study the characteristics of program execution, program tracing and monitoring software have been used widely. However, in the Windows/x86 platform, it is very difficult to build such a software tool, because the Windows system and its many applications are distributed without source code. In addition, the x86 CPU has a complicated instruction set architecture and is difficult to analyze. In this paper, we report a program tracing tool for the Windows NT/x86 platform, which aims at tracing multithreaded Windows applications without source code available. We have applied our tracing system to a number of benchmark programs. Their execution characteristics obtained from out tool are presented.

1. INTRODUCTION

The personal computers (PCs) are the mainstream computing machines nowadays. Most PCs are equipped with x86-compatible processors, running the Microsoft Windows operating system. When people write programs on such platforms, they usually want to know the characteristics of their programs and how they perform. With this information, performance of the programs can be better tuned. Similarly, when a hardware designer is faced with various design tradeoffs, he also wants to know how target applications run on the hardware. Unfortunately, there are very few tools to analyze the program characteristics on Windows systems. The difficulty stems from the facts that the Windows operating system is distributed without source code, with many internal activities undocumented. The complicated instruction set of the x86 processor also posts a great barrier to develop such a tool.

Various techniques have been proposed to analyze program behavior and study their execution characteristics [1,2,7,10,12]. One important class of techniques is to insert pieces of codes into the source program through compilers, assemblers, or linkers [6,7]. When the instrumented program is executed, execution statistics or traces are output and collected. Unfortunately, these techniques can only work on applications with source code. More aggressive instrumentation techniques call for the modification of the binary executables, called *binary rewriting* or *instrumentation* [7,8,15].

In this paper, we introduce a program tracing tool on the Windows system, assuming the source code of the applications is unavailable. Our tracing system is able to capture the trace of an application just from its executable files or run-time states. Since Windows 95/98/NT supports system-aware threads, our tool is also able to distinguish the behavior of different threads and trace them individually.

The rest of this paper is structured as follows. Section 2 surveys program tracing software developed by other research groups or commercial corporations. Section 3 introduces the methods used to build our program tracing system. The OS and CPU related issues are discussed. Section 4 shows the experimental results of applying our tracing tool to benchmark programs. We analyze their instruction mix, basic blocks, call graphs, and many other important features. Section 5 gives our conclusions and points out several possible future works.

2. RELATED WORKS

Many academic and commercial research groups have devoted many efforts to application characteristics analysis. Dozens of software tools have already been developed for tracing and profiling programs on different platforms and different operating systems [1,3,5,6,8,15]. Some of them can even trace system level activities, such as *Vtune* [5] by Intel and *PatchWrx* [1] by Digital. The ultimate goal of these software is to capture the behavior of a running application, e.g., instruction mix, basic block size, branch and call graph, etc. By analyzing these informations, software programmers may optimize algorithms used in their applications and hardware designers may improve the system organization to get more performance.

These tools have adopted many advanced techniques, for instance, instrumentation [8], binary rewriting [1,15], and run-time sampling [5]. Some hardware-based methods are also used, such as the performance monitoring counters in x86 processors [2,5,10] and PALcode of Alpha processors [1]. The *ptrace()* system call in UNIX provides tracing and debugging capability at the operating system level.

3. THE PROGRAM TRACING TOOL

This section describes our program tracing tool on the Windows NT/x86 platform. The ultimate goal is to collect complete traces of all threads created by a running application. Since most Windows applications are distributed in executables without source code available, we

restrict our program tracing software to collect traces only from the executable images and the run-time states. This allows us to work with most Windows applications. We will focus on Windows NT, and discuss in the following some important issues related to hardware and operating system, when building such a program tracing tool.

3.1. Issues Related to Operating System

Since all programs are executed on top of the operating system, the first thing we have to know is how Windows NT creates an environment within which program runs. The Windows NT has a microkernel-like and layered kernel [16]. It is implemented with several protected subsystems, e.g., Win32 and POSIX subsystems, which run in the user mode and are responsible for the interaction between the user programs and the kernel. Windows NT also provides a large number of APIs for application programs to call and to accomplish system-related operations, such as creating a new thread or allocating a memory location.

When a program is put into run, Windows NT first creates a new address space and some kernel objects for this new process. Then, it uses a mechanism, named *memory-mapped file*, to map the module images, including the executable file and the dynamic linked libraries (DLLs) used, into the process' address space. The most important system DLLs are *Ntdll.dll* and *Kernel32.dll*. Kernel32.dll contains APIs for creating and destroying processes and threads, managing memory, accessing files, and synchronizing threads, etc. These APIs in turn call those exported from Ntdll.dll for minor functions or for trapping into the NT kernel. After these system activities complete, NT turns control to the first thread of the newly created process and the application gets the first chance to run. Because the NT threads are system-aware, the system makes a context switch between threads no matter whether they are in the same process or not.

Windows NT also provides a set of DEBUG APIs [14]. By using these DEBUG APIs, a programmer can create a process for debugging (CreateProcess), wait for some debug events to happen (WaitForDebugEvent), read or replace the thread context (GetThreadContext and SetThreadContext), and read or replace the content of remote processes' address space (ReadProcessMemory and WriteProcessMemory). Our program tracing tool heavily uses these APIs for snooping the thread contexts of a debuggee process and for capturing the machine instruction that a remote thread is executing. The following sections will discuss more details about the implementation.

When creating a process for debugging, we can call the appropriate APIs to get the content of the foreign address space. Thus, understanding how the executable files or modules are organized is also a necessary. The Win32 executable file is named the *Portable Executable* (PE) file [13]. The PE file format contains an old-style MSDOS header, a PE file header, a PE optional header, and a set of section headers and section bodies.

Those headers are composed of several fields, some of which keeps information about that PE file, e.g., magic number, entry point, code size, etc., and some keeps a relative virtual address (RVA) to another field or section. Sections included in a PE file are classified into different types and usages. The *.text* section contains machine instructions, *.data and .BSS* are used for carrying initialized and uninitialized data, respectively. The *.idata* section keeps the information about all imported APIs and DLLs. The *.edata* section contains export information of this PE file, usually appearing when it is a DLL or has to export some functions.

3.2 Issues Related to Hardware

Since our goal is to catch every instruction in the execution path of every running thread, we need a disassembler to translate the x86 machine instructions into a readable format. Thus we need to understand how x86 instructions are decoded.

The x86 architecture has very complicated instruction format and addressing modes [4]. In a x86 instruction, only the *opcode* field, either one or two bytes, must exist, the others are optional. The *ModR/M* field is composed of three subfields: *Mod*, *Reg/Opcode*, and *R/M*. It is used to determine the operands, the addressing mode, and whether the next three fields exist. In some cases, the Reg/Opcode subfield is used as an extension of the opcode. If the *SIB* field exists, it specifies the base register, index register, and scaling factor of the index. There are also situations that an instruction contains an opcode with implicit operands, e.g., opcode *16h* represents the instruction "*push SS*'", where SS is the stack segment register. It is also possible that the opcode will be immediately followed by a *Displacement* or *Immediate* field.

The x86 architecture provides a mechanism called *soft interrupt* or *exception* to trigger the processor to handle specific events. This is accomplished with instructions such as *INT n*, where *n* is an interrupt vector number.

When tracing a running application, we use a large number of breakpoints and single-step exceptions to stop the process and record the thread states. While a thread is stopped by a breakpoint or a single-step exception, the address of the next instruction is reported instead of the instruction causing this exception.

When developing our tracing program, we use the instruction *INT 3 (0xCC*) to trigger the breakpoint exception and set the *TF* flag in the *EFLAGS* register to trigger the single-step trap. The processor clears the TF flag before calling the exception handler. For protection purpose, the operating system checks the privilege level after any single-step trap to see if single stepping could be continued at the current privilege level. It follows that a user-level code cannot single-step into the system-level code.

3.3. Trace Generation

Our basic idea of generating trace is to set each thread into the single-step mode as it runs. When completing an instruction, the thread will be stopped and we have the chance to record the state of the thread context and generate a trace record. After these actions are done, we let this thread go to the next instruction and we continue tracing it. Although this method is time-consuming, it really works very well to get the full trace of an application.

In this subsection, we present the principal data

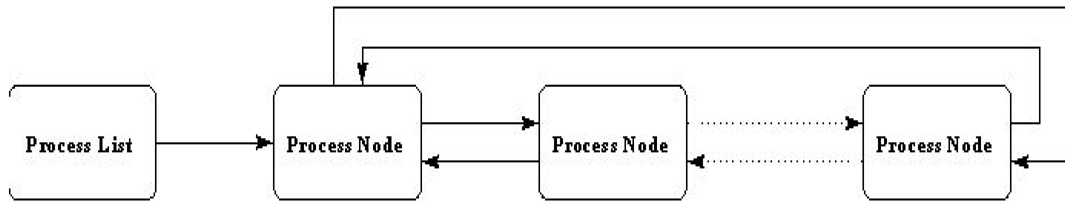

Figure 1: Abstraction of process management

abstractions used in our tracing program for managing the processes and threads created by a running application. The main algorithms in our tracing program and optimization methods are then described.

3.3.1 Data abstractions

A running application may create one or more processes, each of which may spawn several threads and map required DLLs into its address space. The data structures to represent these entities are described below.

Process, Thread, DLL

In most cases, an application may spawn several threads and even several processes. So, we use a doubly linked list to chain the processes created by the same running application and a global variable to hold the head node of that process list. The threads spawned by the same process and DLLs loaded by the process are chained respectively by two additional doubly linked lists, linked from that process node. The abstractions for process and thread management are shown in Figures 1 and 2.

Each data structure must keep some related information. A process node has to store the process id, the process entry point, the module image header, a flag to denote whether it is alive, a *HANDLE* to the process, and, most importantly, a *TRACE_INFO* record. A handle is a special data type to keep the connection to a kernel object in the Windows system. The TRACE_INFO record keeps information about basic blocks and instructions, which will be described below. Similarly, a thread node keeps the thread id, thread handle, a flag for live/die status, the number of instructions the thread has executed, and a thread trace record, which is used to store each thread state. A DLL node should keep the DLL name, DLL module header, base of the DLL address, etc.

Basic Block and Instruction

The TRACE_INFO structure in a process node contains two hash tables: a basic block hash and an instruction hash. They are used to deposit basic block records and instruction records respectively. We catch the basic block and the instruction information at run time as the program control goes through the code section of the executable file images or the dynamic linked libraries.

As a program is running, there is a very high probability that instructions or basic blocks are reexecuted. When an instruction or a basic block is first met, it is disassembled, analyzed, and then stored in the hash tables for reuse. Next time when they are met, the disassembling or analysis time can be reduced. Their information may be obtained by searching these hash tables.

Every basic block record must hold the start and end addresses, the number of instructions inside the basic block, the number of times it is executed, and the number of instructions with a repeat prefix. As mentioned before, a x86 instruction may exist with a prepended prefix field. There are several prefix codes, and one of them is the *repeat prefix.* The repeat prefix means that an instruction must be executed

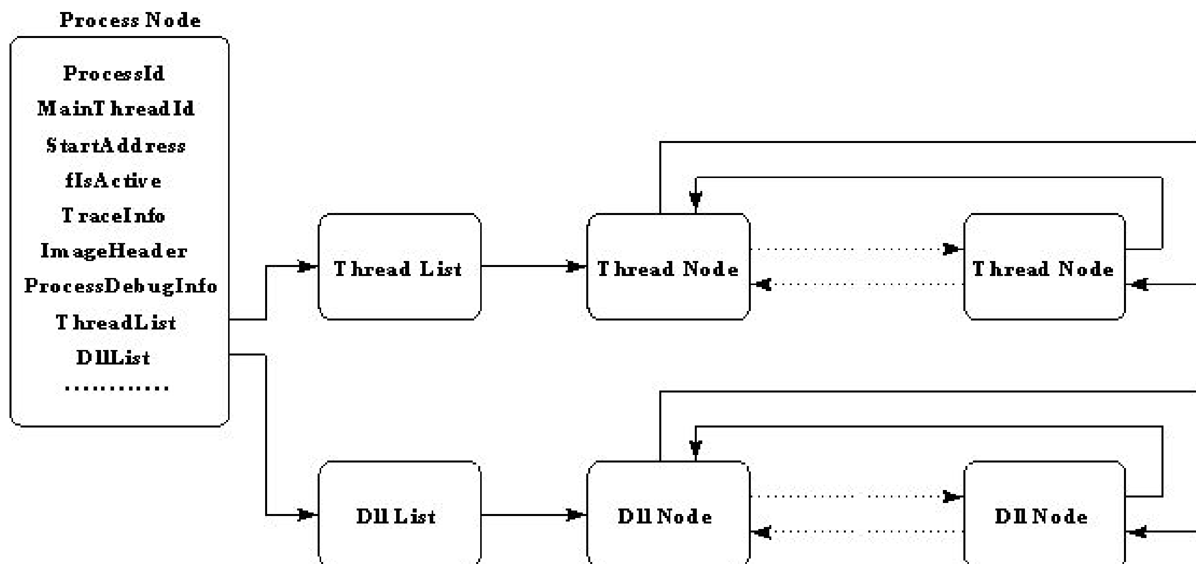

Figure 2: Thread and DLL lists within a process node

repeatedly until some condition is satisfied. We distinguish this kind of instructions from others, because they may be executed for an undetermined number of times. This may affect the instruction count in a basic block at run time. We count this kind of instructions just once in each basic block.

An instruction record holds the start and end addresses of this instruction, the machine code and its assembly equivalence, and an instruction type record. The purpose of the instruction type structure is to store the opcode and the type of this instruction, e.g., data movement or control transfer. We define the instruction types according to the classification in [4].

3.3.2. WIN32 debug events

In section 3.1, we mentioned that a debugger process could call some debug APIs to wait for the debuggee process to reply some debug events specified by the debugger. In general, the debuggee process will generate a debug event when one of the following conditions occurs:

1. An exception is generated.
2. A process is created or exits.
3. A thread is created or exits.
4. A DLL is loaded or unloaded into the address space.

While a debuggee process returns a debug event, all threads running in the current process will be blocked and the control will be transferred to the debugger, i.e., our tracing program. Unless the debugger returns the control and notifies the debuggee to continue, the debuggee will be blocked forever. Thus, debug events must be carefully handled.

We also implement several event handlers, and each of them handles one debug event. For example, a create process/thread event handler must allocate a process/thread node, fill its fields with its state, and insert it into the process/thread list. The breakpoint or single-step exception handlers have to snoop and record the current state of the thread which caused the exception, generate a trace record belonging to this thread and then set it to the single-step mode again. A debug loop will be presented in the next section to show how to repeatedly debug the debuggee process.

3.3.3. Debug loop

To trace a Windows application, we have to create a process to execute the application. That process is treated as a debuggee process and our tracing program enters a loop to wait and handle all the debug events from that process until the application finishes. Most actions are taken care of within the debug loop, e.g., to construct the process list and to generate trace records. As shown in Figure 3, this debug loop does nothing but dispatch each debug event to its corresponding handler.

In the debug loop, all event handlers except *HandleException()* are used to maintain process and thread lists. Inside the exception handler, we only intercept the breakpoint and single-step traps. When one of these traps is caught, our debugger program reads the remote process memory and thread context, makes a decision to disassemble the instruction at the current address or get its information from hash tables, stores a trace record, and sets the thread to the single-step mode again.

```
ProcessCount = 0;
while ( TRUE ) {
    WaitForDebugEvent( event );
    switch ( event ) {
        case EXCEPTION_EVENT :
            HandleException( event );
            break;
        case CREATE_PROCESS_EVENT :
            ProcessCount++;
            HandleCreateProcess( event );
            break;
        case EXIT_PROCESS_EVENT :
            HandleExitProcess( event );
            ProcessCount--;
            break;
        ..............................
    }
    if ( ProcessCount == 0 )
        break;
    else
        ContinueDebugEvent( debuggee process );
}
```

Figure 3: The debug loop in the tracing program

Every time when a create process event is received, the process count in the process list must be increased. An exit process event decreases the process count. If the process count equals to zero, it means all processes created by the running application are terminated. In this case the tracing is completed. We then dump all the trace results collected.

3.3.4. Optimization

When handling the exceptions, there are some considerations for optimization. First, every time when the exceptions handler is invoked, we only need to read a few bytes from the memory of the debuggee process. The cost to read from another process' memory is very high. To optimize memory reads, we implement a buffer, which behaves like a cache, and we call it the *soft cache*. Using soft cache, we can read more bytes once (like a cache line), and store them for the next use. This should reduce the number of times to read from another process' memory, because in most cases a stream of instructions is executed sequentially before the program control branches off. The cache line size is adjustable, and is set to 64 bytes by now.

Second, we can defer instruction disassembling and resume the threads as soon as possible. This will reduce the debuggee process' hanging time, and improve the tracing speed, especially when running on multi-processor machines. So, we also implement a large FIFO buffer called the *address trace buffer (ATB)*. When an exception is reported, the handler just stores the process id, thread id, and exception address into one ATB entry. When the number of ATB entries reaches a threshold value, an ATB consumer thread is invoked to flush the address records sequentially. After it finishes processing all the ATB entries, it is put into sleep again.

Classifications	Details
General	Total instructions Total basic blocks Instruction/basic block Static instructions Static basic blocks Maximum instr. in a basic block Maximum usage of a basic block
Module	Module base Instr. executed inside a module No. of basic blocks inside a module Static instructions inside a module Static basic blocks inside a module
Thread	Thread instruction counts Thread API calls Thread Call graph
Instruction	Instructions mix No. of each control transfer types No. of each prefix types Top N instructions

Table 1: Results reported by the tracing tool

3.4. Program Analysis

When applying our tracing program to a Windows application, the complete trace will be output. But the trace is not readable yet, because the trace format is designed to reduce the disk space and keeps only the necessary information. We also provide an analyzer program to reconstruct the execution. The results output from the analyzer are classified into four categories, which are listed in Table 1.

4. EXPERIMENT RESULTS

In this section, we present the results of applying our tracing system to the Windows benchmarks. The experiments were conducted on a personal computer with dual Intel 450-MHz Pentium III processors, 512-KB L2 cache, and 256-MB of RAM. The system ran the Windows NT 4.0 build 1381 patched with service pack 4. The benchmark programs include console and GUI programs. We distinguish these two kinds of applications, because they have different characteristics. In general, GUI applications need more DLL modules and create more cooperative threads, but have a smaller average basic block size.

4.1 Console Applications

This section discusses the characteristics of console applications running in the Windows NT. The console programs used in our experiments are multimedia applications based on the MediaBench suite [9]. The MediaBench suite was chosen, because a large number of applications running on the Windows systems are multimedia applications, doing image processing and speech compression, etc.

We apply our tracing system to four components of the MediaBench suite: JPEG, MPEG, EPIC, and ADPCM, each has its compression and decompression parts. The source code of these software was downloaded from the Internet [11] and compiled by the Microsoft Visual C++ 5.0. The programs and their input data used in the experiments are described below:

1. **JPEG:** JPEG is a lossy compression method for full-color and gray-scale images. This package contains two program *cjpeg* and *djpeg*. In our experiments, the input to cjpeg was a 101-KB PPM file and the input to djpeg was a 6-KB JPEG file. These two files contain the same picture but are encoded in different graphic formats.
2. **MPEG:** MPEG is a standard for digital video transmission. We used *mepg2enc* to encode a 4-frame video, and *mpeg2dec* to decode it.
3. **EPIC:** EPIC is an experimental image compression utility. We used *epic* to encode a 256x256 gray scale image, and *unepic* to reverse it.
4. **ADPCM:** ADPCM is a utility for speech compression and decompression. It takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1.

Table 2 shows the general information of these console applications collected by our tracing system. In the table, the row denoted ``Total Instructions" lists the number of total dynamic instructions executed and the row denoted ``Total Basic Blocks" lists the number of dynamic basic blocks executed. The row of ``Instr./B.B." shows the

	cjpeg	djpeg	mpeg2enc	mpeg2dec	epic	unepic	rawcaudio	rawdaudio
Total Instructions	22,860,466	7,009,207	2,457,654,596	185,631,762	56,153,896	7,432,749	14,309,580	15,400,176
Total Basic Blocks	4,110,920	500,810	325,540,386	27,383,916	9,832,032	1,332,268	2,732,952	3,193,584
Instr./B.B. Ratio	5.56	14.00	7.55	6.78	5.71	5.58	5.24	4.82
Static Instructions	22,553	22,268	26,679	17,080	17,637	18,678	12,237	12,201
Static Basic Blocks	4,363	4,132	5,074	3,606	3,495	3,690	2,518	2,514
Static Instr./B.B. Ratio	5.17	5.39	5.26	4.74	5.05	5.06	4.86	4.85
Max Instr./B.B.	160	198	154	79	79	79	79	79
Max Usage/B.B.	179,456	33,972	6,287,790	4,325,376	2,343,360	65,959	148,638	297,409
Modules	3	3	3	3	3	3	3	3
Threads	1	1	1	1	1	1	1	1

Table 2: General information of the console applications

	cjpeg		djpeg		mpeg2enc		mpeg2dec		epic		unepic		rawcaudio		rawdaudio	
Instruction Types	Counts	Ratio	Counts	Ratio	Counts	Ratio	Counts	Ratio	Counts	Ratio	Counts	Ratio	Counts	Ratio	Counts	Ratio
Data Movement	13,502,920	59.07%	4,392,735	62.67%	1,277,343,061	51.97%	59,022,670	31.80%	15,915,683	28.34%	2,124,576	28.58%	6,475,459	45.25%	6,442,847	41.84%
Binary Arithmetic	4,732,791	20.70%	1,379,458	19.68%	590,396,027	24.02%	48,538,862	26.15%	19,654,897	35.00%	2,138,397	28.77%	3,769,932	26.35%	4,130,359	26.82%
Decimal Arithmetic	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
Logical	396,169	1.73%	276,217	3.94%	260,236,270	10.59%	7,772,094	4.19%	705,049	1.26%	70,404	0.95%	937,368	6.55%	1,049,914	6.82%
Shift and Rotate	570,547	2.50%	228,328	3.26%	32,210,702	1.31%	1,334,204	0.72%	141,228	0.25%	21,312	0.29%	522,501	3.65%	350,423	2.28%
Bit and Byte	99,559	0.44%	53,965	0.77%	2,521,911	0.10%	1,783,387	0.96%	910,577	1.62%	239,754	3.23%	309,107	2.16%	607,038	3.94%
Control Transfer	3,163,295	13.84%	398,724	5.69%	237,346,796	9.66%	25,765,066	13.88%	8,008,932	14.26%	1,167,066	15.70%	2,190,665	15.31%	2,715,045	17.63%
String Operation	1,284	0.01%	1,165	0.02%	5,710	0.00%	9,221	0.00%	1,571	0.00%	2,413	0.03%	506	0.00%	506	0.00%
Input and Output	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
Enter and Leave	240	0.00%	61	0.00%	542,285	0.02%	540,739	0.29%	65,600	0.12%	56,803	0.76%	43	0.00%	43	0.00%
EFlags	50	0.00%	46	0.00%	151	0.00%	56	0.00%	50	0.00%	45	0.00%	35	0.00%	35	0.00%
Segment Register	122	0.00%	118	0.00%	393	0.00%	323	0.00%	129	0.00%	218	0.00%	375	0.00%	524	0.00%
Miscellaneous	95,746	0.42%	110,414	1.58%	22,922,651	0.93%	918,241	0.49%	1,736,632	3.09%	88,527	1.19%	78,982	0.55%	302,863	1.97%
Floating-Point	4	0.00%	4	0.00%	55,404,087	2.25%	40,010,219	21.55%	9,697,377	17.27%	781,738	10.52%	0	0.00%	0	0.00%
System	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
Multimedia Extention	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
Prefix	122,337	0.54%	26,592	0.38%	6,627,895	0.27%	2,274,894	1.23%	281,458	0.50%	327,065	4.40%	11,324	0.08%	309,160	2.01%

Table 3: Statistics of instruction types for the console applications

average ratio of instructions per basic block for each application. The rows on static statistics list the information on static instructions and basic blocks. Table 2 also presents the maximum number of instructions in a basic block, the number of times the most executed basic block runs, and the number of modules used and threads created.

The results in Table 2 show that the compression programs execute more instructions than the corresponding decompression programs, except the ADPCM package. All of these applications create only one thread and use three modules (the program file itself, ntdll.dll, and kernel32.dll). We observe that the ``Instr./B.B" is slightly larger than the ``Static Instr./B.B.". This tells that while executing, these programs tend to spend more time in larger basic blocks. Among them, djpeg has a relatively higher average instruction per basic block ratio (14).

Table 3 presents the statistics of instruction types, which are classified according to [4]. An instruction may belong to more than one type. For example, *pushfd* and *popfd* belong to both data movement and eflags types. From this table, we can see that the most executed types of instructions in these multimedia programs are data movement, binary arithmetic, and control transfer. These three types of instructions constitute 71.83% at least (mpeg2dec) and 93.61% at most (cjpeg) of the total instructions executed. Not all of these applications use floating-point instructions heavily. Instructions in the I/O and system types are privilege instructions, so none of these instructions can be captured. Some instructions, such as *lea*, *cpuid*, and *nop*, etc., are hard to be classified and are collected to the Miscellaneous type [4]. Finally, we can see that these applications do not take the advantage of the MMX technology of the Intel processors. In fact, the VC++ 5.0 compiler only supports the MMX instructions in inline assembly code. Thus, a programmer who wants to make use of the MMX instructions has to write the pieces of code in assembly.

Our tracing system also has the ability to collect per-module or per-thread statistics. That is, we can show how many instructions are executed by a thread or inside a module. Besides, we can intercept the API calls made by a thread, and construct the call graph of the thread. Here, we use the MPEG package to demonstrate the capability, and only show the per-thread results in Table 4. The Module Switches field in Table 4 indicates how many times the thread jumps across module boundaries. This usually occurs when the thread calls an API or DLL-exported function, and returns from them. Thus, the number of module switches must be smaller than or equal to twice the number of API calls, which we do not distinguish between the system APIs and the functions exported from application DLLs. The more the number of module switches implies the more difficult we could improve the instruction locality.

4.2 Windows GUI Applications

We also apply our tracing system to several Windows GUI applications. The GUI applications usually use more modules and create more than one thread. We selected five

Application	Thread Id	Instruction Count	API Calls	Module Switches
mpeg2enc	371	2,457,654,596	1,047	1,661
mpeg2dec	386	185,631,762	733	1,327

Table 4: Per-thread statistics of the MPEG package

	WordViewer	XL Viewer	PPViewer	MediaPlayer2	WinAmp
Total Instructions	195,179,435	91,119,113	47,515,883	302,222,227	187,738,794
Total Basic Blocks	38,136,209	21,606,904	9,435,287	68,328,872	34,064,366
Instr./B.B. Ratio	5.12	4.22	5.04	4.42	5.51
Static Instructions	262,867	246,471	199,576	48,875	79,792
Static Basic Blocks	60,823	58,378	46,538	46,538	17,895
Static Instr./B.B. Ratio	4.32	4.22	4.29	4.33	4.46
Max Instr./B.B.	266	130	107	488	653
Max Usage/B.B.	780,493	230,377	304,234	1,098,552	961,478
Modules	17	19	20	30	30
Threads	8	2	2	9	5

Table 5: General information of the GUI applications

GUI programs and traced their execution, three of which are document viewers for Microsoft Office Suite (Word, Excel, and Powerpoint), and the other two are MediaPlayer2 and Winamp v2.10. The programs and their input data are described below:

1. **WordViewer:** This is an application for displaying Word 97 documents. We used this viewer to open a 243-KB, 9-page Word document, paged down to the last page, and then closed it.
2. **XLViewer:** This program is for displaying Excel 97 documents. We used it to show a 163-KB, 8-sheet Excel document. The program displayed all its content, and then closed it.
3. **PPViewer**: This is a Powerpoint97 document viewer. We used it to play a 131-KB 29-page PowerPoint document with an interval of 2 seconds between adjacent pages.
4. **MediaPlayer2:** This is a video player program, which play many multimedia files of various formats. We traced this application by playing a 6-second MPEG video file. We observe from the traced output that MediaPlayer2 uses a huge amount of MMX instructions.
5. **Winamp:** Winamp is a MPEG layer 3 decoder and player. We used it to play a 5-second mp3 file, and traced the total execution.

The general information of these GUI applications is shown in Table 5. We see that GUI applications use more modules and create more threads. The average instructions per basic block ratio is smaller relative to the console programs. This means that the GUI applications have shorter basic blocks in average. We found that all these GUI applications have a similar pattern of a peak ratio occurring at three instructions per basic block. They seldom have basic blocks with more than seven instructions.

Statistics of instruction types for the GUI applications are shown in Table 6. Frequently used instruction types are data movement, control transfer, and binary arithmetic. That is slightly different from those in the console applications. These GUI applications use more control transfer instructions, perhaps because they have shorter basic blocks. We also can see that Winamp uses a huge amount of floating-point instructions, and MediaPlayer2 uses a lot of MMX instructions. MediaPlayer2 is the only application that takes the advantage of Intel MMX technology among those benchmarks we traced.

We also choose one GUI applications, MediaPlayer2, to show per thread statistics in Table 7. In the table, the threads are listed according to the sequence of their creation. We observe that most execution time spent in a few threads, and others just take a little part of the total work.

	WordViewer		XLViewer		PPViewer		MediaPlayer2		WinAmp	
Instruction Types	Counts	Ratio	Counts	Ratio	Counts	Ratio	Counts	Ratio	Counts	Ratio
Data Movement	75,200,336	38.53%	36,190,973	39.72%	18,981,972	39.95%	136,615,204	45.20%	61,921,148	32.98%
Binary Arithmetic	32,678,925	16.74%	14,069,090	15.44%	9,573,541	20.15%	45,164,329	14.94%	29,640,570	15.79%
Decimal Arithmetic	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
Logical	9,969,937	5.11%	4,097,480	4.50%	2,001,265	4.21%	12,160,497	4.02%	11,817,294	6.29%
Shift and Rotate	5,433,088	2.78%	1,253,986	1.38%	547,857	1.15%	3,107,942	1.03%	4,122,525	2.20%
Bit and Byte	8,801,461	4.51%	4,077,914	4.48%	1,715,477	3.61%	11,181,409	3.70%	7,179,192	3.82%
Control Transfer	35,287,230	18.08%	20,377,783	22.36%	8,814,727	18.55%	65,602,309	21.71%	30,521,984	16.26%
String Operation	1,298,534	0.67%	348,678	0.38%	83,791	0.18%	640,063	0.21%	219,767	0.12%
Input and Output	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
Enter and Leave	587,962	0.30%	245,553	0.27%	180,816	0.38%	1,638,046	0.54%	745,669	0.40%
EFlags	3,395	0.00%	8,895	0.01%	29,431	0.06%	24,095	0.01%	3,915	0.00%
Segment Register	21,087	0.01%	76,087	0.08%	9,979	0.02%	609,953	0.20%	392,733	0.21%
Miscellaneous	5,287,772	2.71%	1,446,466	1.59%	1,024,439	2.16%	8,112,064	2.68%	3,573,095	1.90%
Floating-Point	19,961	0.01%	63,508	0.07%	246,978	0.52%	131,865	0.04%	35,108,936	18.70%
System	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
Multimedia Extention	0	0.00%	0	0.00%	0	0.00%	15,194,082	5.03%	0	0.00%
Prefix	8,334,280	4.27%	4,232,265	4.64%	2,565,581	5.40%	8,057,786	2.67%	3,490,056	1.86%

Table 6: Statistics of instruction types for the GUI applications

Thread Id	Instruction Count	API Calls	Module Switches
285	240,966,638	5,679,349	5,847,406
280	109,091	3,055	6,111
333	10,366,908	50,651	88,673
73	3,361	59	119
330	219,473	1,602	3,332
288	21,374,253	15,623	24,757
319	27,818,620	58,824	20,415
167	1,362,655	8,726	9,061
343	1,228	19	34

Table 7: Per-thread statistics of MediaPlayer2

5. CONCLUSIONS

In this paper, we introduce a workable method to capture the full user-level trace of multithreaded Windows applications. Such an effort is very difficult, because most Windows applications and the Windows NT itself are only available without source code and the x86 instructions are hard to decode and analyze. Our program tracing tool may help users to observe the execution characteristics of applications running on Windows NT/x86-compatible platforms. We have also applies our tool to several real applications and study their execution characteristics. This demonstrates the usefulness of the tool.

Our tracing system also has some limitations. Techniques such as binary instrumentation and run-time sampling may be integrated into our tracing system to overcome or to reduce the existing restrictions. Run-time sampling may reduce the disk space for the collected trace and the time spent in getting the trace. But it may lower the accuracy of the results and affect the correctness in analyzing the program behavior. Its effect requires further investigation. Binary instrumentation, on the other hand, takes more efforts to analyze the binary files, e.g., the Win32 PE files, and to rearrange the machine instructions. It remains to see if this technique can be adopted for Windows NT.

BIBLIOGRAPHY

[1] J.P. Casmira, D.P. Hunter, and D.R. Kaeli. Tracing and Characterization of Windows NT-based System Workload. *Digital Technical Journal*, 10(1), December 1998.

[2] J.B. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Seltzer, and M. D. Smith. The Measured Performance of Personal Computer Operating System. *ACM Transaction of Computer Systems*, February 1996.

[3] R.F. Cemlik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical report, Sun Microsystems Lab., 1993.

[4] Intel Co. Intel Architecture Software Developer's Manual. Vol. 2: Instruction Set Reference.

[5] Intel Corporation. Intel Vtune™ Performance Analyzer. http://www.mmx.com/vtune/analyzer/.

[6] J. Fenlason, R. Stallman. GNU Documentation – The GNU Profiler. http://www.delorie.com/gnu/docs/binutils/gprof_toc.html

[7] J.R. Larus and T. Ball. Rewriting Executable Files to Measure Program Behavior. *Software Practice and Experience*, 24(2): 197-218, Feb. 1994.

[8] J.R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. *Proc. of 1995 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 291-300, June 1995.

[9] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. *Proc. 30th Int'l Symp. on Microarchitecture*, 1997.

[10] D. Lee, P. Crowley, J.Baer, T. Anderson, and B. Bershad. Execution Characteristics of Desktop Applications on Windows NT. *Proc. 25th Int'l Symp. on Computer Architecture*, 27-38, 1998.

[11] W.H. Mangione-Smith. MediaBench Suite. http://www.icsl.ucla.edu/~billms/

[12] S.E. Perl and R.L. Sites. Studies of Windows NT Performance using Dynamic Execution Traces. *Proc. of 2nd USENIX Symp. on Operating System Design and Implementation*, October 1996.

[13] M. Pietrek. *Windows 95 System Programming Secrets*. IDG Books, 1995.

[14] J. Richter. Advanced Windows. Microsoft Press, 3rd edition, 1997.

[15] T. Romer, F. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. *Proc. of the USENIX Windows NT Workshop*, August 1997.

[16] D.A. Solomon. Inside Windows NT: The official guide to the architecture and internals of Microsoft Premier operating system. Microsoft Press, 2nd edition, 1998.