Improving BTB Performance in the Presence of DLLs

Stevan Vlaovic, Edward S. Davidson and Gary S. Tyson

Advanced Computer Architecture Lab

The University of Michigan

{vlaovic, davidson, tyson}@eecs.umich.edu

ABSTRACT

Dynamically Linked Libraries (DLLs) promote software modularity, portability, and flexibility and their use has become widespread. In this paper, we characterize the behavior of five applications that make heavy use of DLLs, with a particular focus on the effects of DLLs on Branch Target Buffer (BTB) performance. DLLs aggravate hot set contention in the BTB. Standard software remedies are ineffective because the DLLs are shared, compiled separately, and dynamically linked to applications. We propose a hardware technique, the DLL BTB, that adds a small second buffer to the BTB and dedicates it to storing DLL target addresses. We show that the DLL BTB performance is similar to a BTB with a victim buffer, but the DLL BTB requires no parallel lookups or datapaths between the original BTB and the added buffer.

1.0 Introduction

Use of Dynamically Linked Libraries (DLLs) has become widespread, as they enable software providers to change software incrementally when enhancing functionality and adapting to new systems and platforms. Instead of having one large monolithic binary executable, a smaller kernel executable is used, with specific DLLs loaded at run-time to enable execution in the environment of interest. The use of DLLs is increasing commensurate with the increasing size and complexity of modern applications and environments.

DLLs thus allow for more modularity, flexibility, and portability. For instance, in the Win32 API, the three most important DLLs are KERNEL32.DLL (which consists of functions for managing memory, processes and threads), USER32.DLL (which implements user-interface tasks such as window creation and message sending), and GDI32.DLL (which consists of functions for drawing graphical images and displaying text). In order to change the way an image is displayed involves changing only GDI32.DLL and recompiling its source code. Note that, since DLLs are dynamically linked, finding all of the applications that call GDI32 and statically relinking the library is not required, nor is recompilation of monolithic executables that make use of this modified function. This reduces the number of errors introduced, and makes it easier to update older executables. Portability in Windows NT is accomplished by the *H*ardware *A*bstraction *L*ayer, as implemented in HAL.DLL, which has the responsibility of interfacing directly to the hardware. Running Windows on different platforms requires rewriting the hardware abstraction layer, or HAL.DLL, while other DLLs, system software and applications can remain (theoretically) unchanged.

Although the benefits of DLLs on software engineering are clear, their impact on performance is not. DLL-reliant applications have not been as widely studied, are inherently much larger, and have much more complex interaction with the hardware than standard benchmarks used by computer architects. We will show that the use of DLL calls exacerbate contention in the Branch Target Buffer (BTB). Since the libraries are dynamically linked and shared among different processes, standard software remedies do not alleviate contention. We then show how a hardware remedy, the DLL Target Buffer (DTB) - a second buffer added to the BTB that is dedicated to storing DLL call targets - can significantly reduce BTB contention.

Section 2 below discusses some previous results in this area. Section 3 characterizes our target applications, while Section 4 discusses how DLLs affect BTB performance. Section 5 describes the DLL BTB design we propose here to reduce BTB contention. Section 6 discusses the results of our DLL BTB evaluations. Section 7 presents some concluding remarks.

2.0 Related Work

Lee et al. [6] provide some insight into applications that use DLLs by discussing some of their results with Etch, a general purpose tool for rewriting arbitrary Win32 binaries on x86 platforms without requiring modification of the source code. Their study compares some popular desktop applications to some SPECINT95 benchmarks in terms of application characteristics, cache behavior, TLB behavior, and branch prediction accuracy. Even though Etch is limited to user level traces only, some important findings about DLLs are outlined. This work and [7] point to the prominent role of DLLs in Win32 environments.

Numerous studies have emphasized branch prediction [13], with a few targeting the Branch Target Buffer [3,8,9]. [3,9] target statically compiled code, but show that BTB design is important. Perleberg and Smith [8] investigate a hierarchical BTB design, but conclude that it is too expensive for the modest improvement it offers in performance. [9] uses a multilevel BTB, but for the purpose of reducing wire delay.

Calder et al. [2] showed that libraries tend to have similar control flow characteristics across different applications. They then reduced the overall branch misprediction rate by up to 28% by simply linking in a preoptimized library. This work gave insight into indirect branches and their negative impact on performance which led Calder [1] to propose new methods of reducing indirect function call overhead in C++ programs. Earlier, Wall [12] had suggested using hardware to predict the targets of indirect function calls.

In contrast with previous work, our work specifically focuses on dynamically linked object files (DLLs) and the applications that use them. We propose a multilateral [10, 11] BTB design, the DLL BTB, consisting of two buffers: a standard BTB with DLL call targets filtered out called the Filtered BTB, and a smaller dedicated buffer called the DLL Target Buffer (DTB), into which these call targets are placed. We contrast the DLL BTB with another multilateral design, a Victim BTB, where a victim buffer [4] is added to a standard BTB. The DLL BTB achieves similar performance on the target applications with a more straightforward design.

3.0 Five Win32 Applications

Target applications should broadly represent the domain of interest. We have chosen five popular Windows NT applications: *Id's Doom, Microsoft Explorer 5.0, Microsoft Visual Studio 5.0, Netscape 4.0*, and *Winamp 2.5e*. *Doom* is one of the early first-person type combat games and is available as shareware. The run of *Doom* included recording a session of a *Doom* game, and then replaying it on the simulator. *Explorer 5.0* is *Microsoft*'s web browser; our input is a set of three `.htm` pages. The first is the CNN web page, the second is an ESPN web page, and the third is University of Michigan's EECS homepage. *Microsoft Visual Studio 5.0* (Msdev) is a code development environment, with 5.0 being the previous release. Our run of *Visual Studio* involved the compilation of *go* from the SPEC95 benchmark suite. *Netscape 4.0* is another web browser, but a few versions old. The same web pages that were loaded on *Explorer* were also used for *Netscape*. *Winamp 2.5e* is the latest release of a popular mp3 player; its input was "Cool Down Daddy" by Jellyroll. Table 1 highlights some dynamic characteristics of the applications. Basic Block Size is the average size in instructions over the entire benchmark trace (applications and DLLs).

TABLE 1. Application Trace Characteristics

App.	**Insts ($x10^6$)**	**Data Refs ($x10^6$)**	**DLL Calls ($x10^6$)**	**Basic Block Size**
Doom	761	510	11.9	7.61
Explorer	408	247	4.03	7.07
Msdev	697	432	14.3	5.36
Netscape	865	500	18.5	7.27
Winamp	935	772	8.77	9.08

In order to see how these applications differ from standard benchmarks with regard to instruction cache miss rate, Figure 1 compares our five applications to that of CPU2000. The average miss rate over all the CPU2000 benchmarks is calculated using the reference input set and a direct mapped cache. As seen from the graphs our applications have poorer instruction cache performance than CPU2000, and hence may be expected to have poorer BTB performance as well. Doom has the highest instruction cache miss rate for small caches; for a 4KB direct mapped cache, it misses nearly 9% of the time. However, when the cache is increased to 256KB, the miss rate nearly disappears. Msdev has a modest miss rate for small instruction cache sizes, but doesn't approach zero as quickly as the other applications. For a 1MB direct mapped instruction cache, Msdev still has a 0.5% miss rate.

Branch predictor performance is quite tightly coupled with BTB performance, since the BTB is accessed and updated on every *predicted* taken branch. In our experiments, we updated the BTB only on each branch that is *actually* taken, which puts only those branch addresses and targets into the BTB that are needed. The more accurate the branch predictor, the better the chance of putting useful data into the BTB. In order to see this effect, we measured the branch prediction accuracy with GAg [13], a global predictor that uses a shared history vector and a two-bit saturating counter per history state. Perleberg and Smith [8] show the correlation between instruction cache performance and BTB performance; essentially, the larger the working set in the instruction cache, the larger the working set of branches that needs to be captured by the BTB.

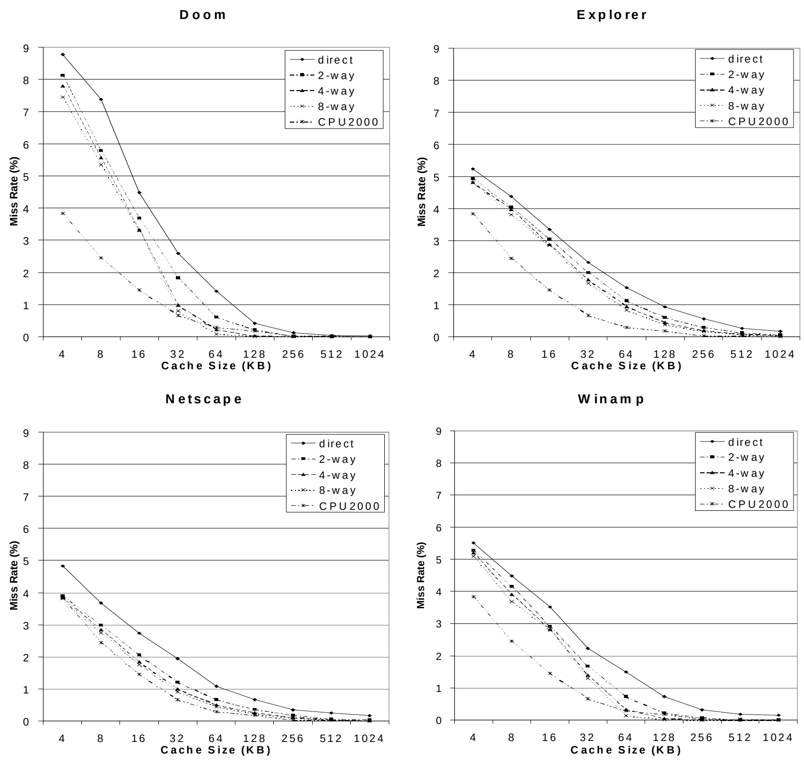

Figure 1. Instruction Cache Miss Rate (line size = 32B)

In Figure 2, the BTB is 1K entry, 4-way with a return address stack of size 8. Since Figure 2 uses the same configuration as our later experiments, we can see that branch prediction accuracy will impact the results presented in Section 6.

In these branch prediction curves, the *direction* line indicates the percentage of time that the branch predictor choses the correct direction. *Address hits* shows the number of times that the branch predictor found an entry with that branch address in the table (but not necessarily with the correct destination). The final curve is the percentage of time that the predictor produced the correct target address (i.e. in this case there would not be any branch penalty in a real processor). Netscape has the most predictable branches, in terms of target addresses, out of all the applications. For a 16K entry GAg predictor, the correct address is delivered nearly 85% of the time. For the same size predictor, Winamp achieves only 60% correct addresses even though the direction prediction accuracy is close to 95% and the address hit ratio is 85%.

4.0 DLL Effects on the BTB

Since it is widely available and in some sense a representation of commonly used applications, the SPEC benchmark suite has generally been used to evaluate new computer architectures. These applications have generally been developed for Unix platforms, and hence do not contain multiple object files that are linked at run-time. This is also true of the latest release, CPU2000. Because of this, they have quite different Branch Target Buffer needs than applications running on Wintel configurations. With a 1K entry, 4-way BTB and perfect branch prediction, the average misses per one thousand instructions is shown in Table 2 for CPU2000, our Win32 applications, and the Win32 applications with DLL calls filtered out.

For example, for indirect branches, a 1K entry, 4-way BTB will miss in the BTB 1.10 times per one thousand instructions for CPU2000, while for our Win32 applications this figure rises to 2.66 times. Note that when the DLL calls are filtered out, the number of misses per one thousand instructions for Indirects falls back close to that of CPU2000. Removing DLL calls had a significant beneficial effect on Calls, reducing the misses per one thousand instructions by 31%. For Conditionals and Unconditionals, there is also a small reduction. The

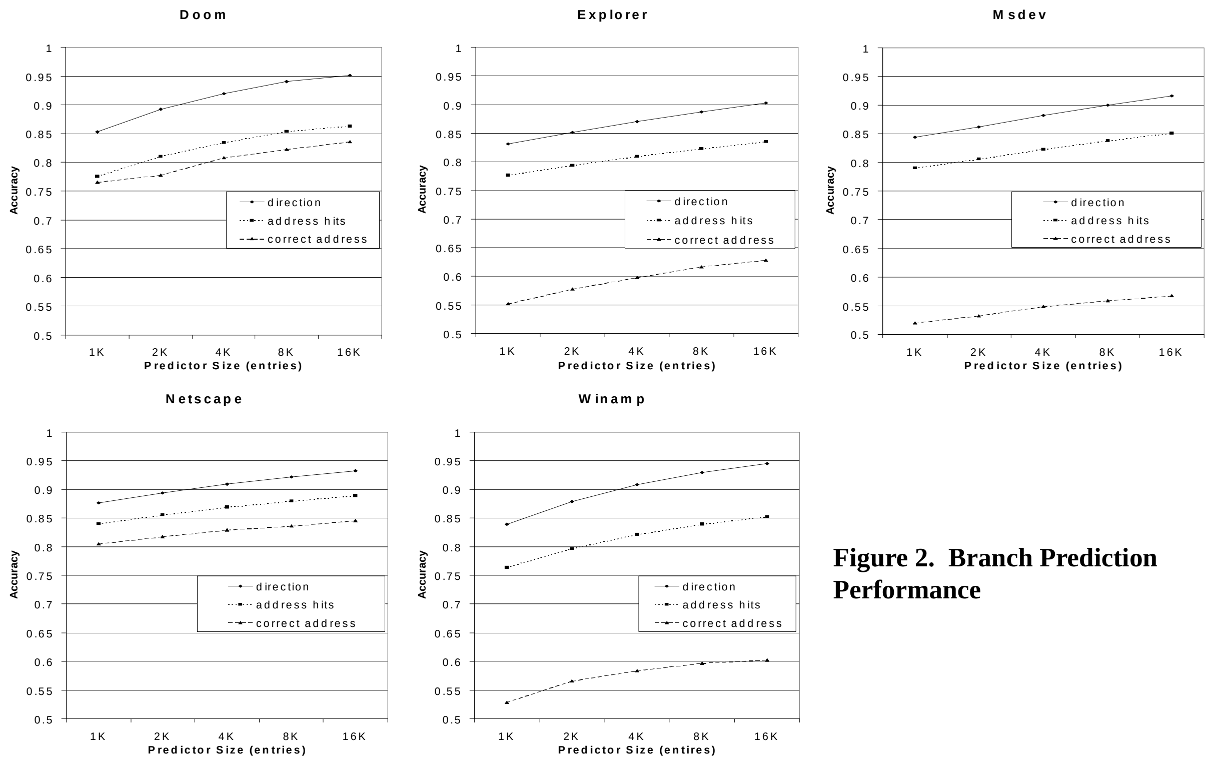

Figure 2. Branch Prediction Performance

results for CPU2000 are averaged over all benchmarks run for 1 billion instructions each.

TABLE 2. Misses per 1000 Instructions in a 1K entry, 4-way BTB (Perfect Branch Prediction)

	CPU2000	Win32	Win32 (no DLL)
indirect	1.10	2.66	1.11
call	1.07	5.22	3.60
conditional	0.73	4.94	4.40
unconditional	1.59	4.90	4.53

4.1 The Windows DLL Linkage

In order to implement DLLs, a number of software mechanisms need to be in place to facilitate multiple binaries. At some point, the actual DLL has to be found and mapped into the process's space. Since DLLs (like regular executables) are just files, the directories on the magnetic media are searched until the appropriate DLL is found. In the Win32 API, a DLL has a preferred location that it maps to; if it doesn't conflict with any previously loaded binary, the loader maps the DLL to that location. Otherwise, it has to be loaded elsewhere, and any absolute addresses contained within the DLL must be corrected to reflect the new location.

Lee et al. [6] provide some insight into why DLL calls are more expensive than statically linked functions:

- DLL calls are implemented as indirect function calls
- DLLs are shared among applications, so improving instruction locality through scheduling is difficult, therefore it is difficult to remap branches to reduce contention for the BTB
- DLLs are aligned on page boundaries which forces the caller and callee to reside in different pages in the address space and contributes to hot set contention in the BTB

Unfortunately, since DLLs must be aligned on page boundaries, it is difficult to reduce the number of conflicts by virtual mapping alone. Alternatively, code layout algorithms can be applied to individual DLLs with self conflicts, and inter-DLL conflicts can be taken into consideration. But because the DLLs are shared between processes, this might reduce the contention only for one process, while increasing it for other processes. We have therefore focused on a hardware remedy for conflict misses in the BTB.

Using our base of a 1K entry, 4-way set associative BTB configuration, we wanted to see what would happen if we filtered DLL calls out of the branch stream. We measured the total number of misses in each BTB set for the entire run and then measured the total number of misses per set with DLL calls removed, i.e the number of misses that would be seen by a Filtered BTB. The differ-

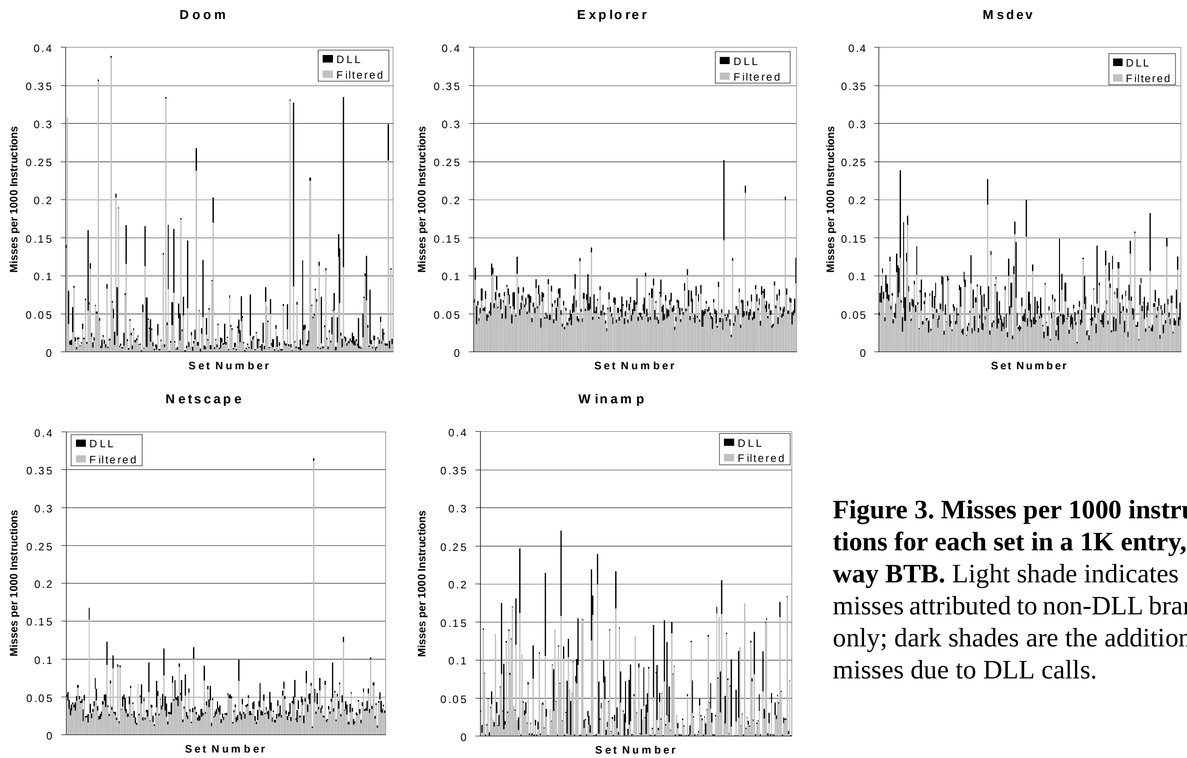

Figure 3. Misses per 1000 instructions for each set in a 1K entry, 4-way BTB. Light shade indicates misses attributed to non-DLL branches only; dark shades are the additional misses due to DLL calls.

ence between these two metrics is a measure of the misses attributed to DLL calls.

This information is shown in Figure 3 for all 5 benchmarks. Many DLL calls are mapped to the same set which creates contention for those sets, seen as a spike on the graph. This shows how DLL calls clearly aggravate hot set contention. Although the behavior of the Filtered BTB is also “spikey”, the DLL calls clearly aggravate the problem.

One way to ameliorate these conflicts is to introduce a victim buffer. The victim buffer does help, but since the DLL calls do cause a large percentage of the misses (for relatively few static calls), we will show how reducing the misses for these calls dramatically improves performance.

4.2 DLL Usage

The number of DLL calls will have a direct impact on indirect call overhead. If there are few DLL calls, then the benefit of removing these indirects will be minimal. Lee et al. [6] showed that some applications can spend as much as 50% of their overall running time in DLLs. Unfortunately, they did not measure the actual number of DLL calls or their impact on performance. Their application suite included the standard desktop applications: MSWord, Excel, Netscape, and Photoshop. In this paper we have measured the impact of DLL calls on a more diverse set of applications.

5.0 The DLL BTB

Our general approach to improving BTB performance is to take advantage of the fact that once a DLLs is loaded, the target of a call to it *never* changes; hence DLL calls are fully predictable. DLL calls tend to increase the address predictability percentage, as a simple last-address-seen mechanism in the BTB is sufficient to capture these indirect addresses. However, since DLL calls increase BTB misses considerably, it might be best to remove these easily predictable indirects from the BTB, handle them with some other mechanism and use the BTB to obtain better coverage of other harder to predict target addresses.

An alternative approach is to add a victim buffer [4] to the BTB, which has been proven to reduce hot set conflicts in caches. However, although the victim cache design has been shown to reduce conflicts, it’s implementation cost can be excessive. The swaps it requires involve extra datapaths, and both the victim buffer and the BTB have to be searched for every access. Such a Victim BTB should alleviate the hot spot contention due to the clustered DLL call sites.

From Figure 3, we know that much of the hot set contention is due to the additional DLL calls, so we propose a DLL BTB filtering mechanism that performs nearly as well as the Victim BTB without the additional datapath or parallel search requirements.

In the DLL BTB, we assign all DLL call targets to a small fully associative buffer, the DLL Target Buffer (DTB). All other branch targets are entered into the BTB, called the *Filtered* BTB when it is used within a DLL BTB configuration.

Filtering out DLL calls can easily be done at run-time by using the import table of the application. The import table is included at the end of the text segment of every application, and contains the addresses of all imported symbols (functions and variables). DLL calls map indirectly through this table. Since we know the boundary addresses of this table, a simple compare can identify which calls are DLL calls.

The results presented in the next section compare a DLL BTB and a *Victim* BTB with a *Base* BTB. We use a DTB of size 128 entries, which turns out to be a good choice for most of these applications; the Victim buffer is set to the same size. The BTB in the Base and Victim configurations and the filter BTB in the DLL BTB configuration are each 1K entry, 4-way set associative. Each implementation has a 32 entry Return Address Stack (RAS), with the Filtered BTB and the DTB sharing a common RAS. Returns from DLL calls are pushed onto this stack even though DLL call targets are in the DTB.

6.0 Quantitative Results

We have developed a PC simulator based on *Bochs* [5], that models not just the CPU, but the entire target platform in enough detail to support the execution of a complete operating system and the applications that run on it. Currently, we are using out-of-the-box Windows NT 4.0 (Build 1381) as the operating system for our Virtual PC. This simulator runs as a user-level process on a standard PC by modeling the platform components completely in software.

Since Windows NT can execute on the functional simulator, we can study complete commercial applications. This approach allows access to all operating system events in addition to the standard instruction, data, and branch traces of user code. The limitation of this method is that the simulator is only functional, not cycle accurate, which precludes detailed timing analysis. However, our simulator can be used as a front end for more detailed simulations. With our traces we can do workload, memory, and branch prediction studies.

TABLE 3. Taken Control Flow Instructions by Category (per 1000 instructions)

	Doom	Expl.	Msd.	Net.	Win.
ind.	9.97	7.99	15.1	12.7	5.50
ind. calls	8.42	6.20	8.55	10.1	4.00
calls	22.1	21.2	28.9	23.0	14.5
ret.	23.8	22.3	29.0	23.4	15.5
cond.	48.4	39.2	47.0	58.7	53.2
uncon.	36.5	36.0	58.5	34.9	25.8
other	24.3	45.1	52.6	21.3	16.3
total	131	141	187	138	110

Table 3 highlights some of the differences between the five applications by showing the breakdown of the dynamic count of *taken* control instructions of each type. Note that these are all *taken* control instructions. Whereas in a real implementation, we would access the BTB on every *predicted taken* control instruction. If we predict the branch to be taken, and it ends up not being taken, the BTB gets updated with the wrong information. Although with increasingly accurate branch predictors these effects become less important, our results are in effect assuming perfect branch prediction (best-case scenario). The ramifications of how the BTB gets updated can be seen in Figure 2. In these graphs, the BTB remains fixed at 1K entries, but as the predictor size increases, the correct target address generation improves.

In Table 3 and throughout the paper, Calls include both direct calls and Indirect Calls; while Unconditionals include jumps and Returns. DLL calls are implemented as indirect calls. Our applications have significantly more indirect calls than the SPEC95 or CPU2000 benchmarks. A few points are worth noting about Table 3. First, the number of calls and returns are not the same. Some of this discrepancy is due to the nature of the trace, as well as interrupts and exceptions. We started the trace after the system was already up and running, but before we launch the application (this leads to some unbalanced calls and returns). The Other category includes instructions such as LOOP and REP. Based on certain flags, instructions LOOP (loop) and REP (repeat) will stay in the execution core and continue executing (in essence, creating instructions) until the flag condition clears.

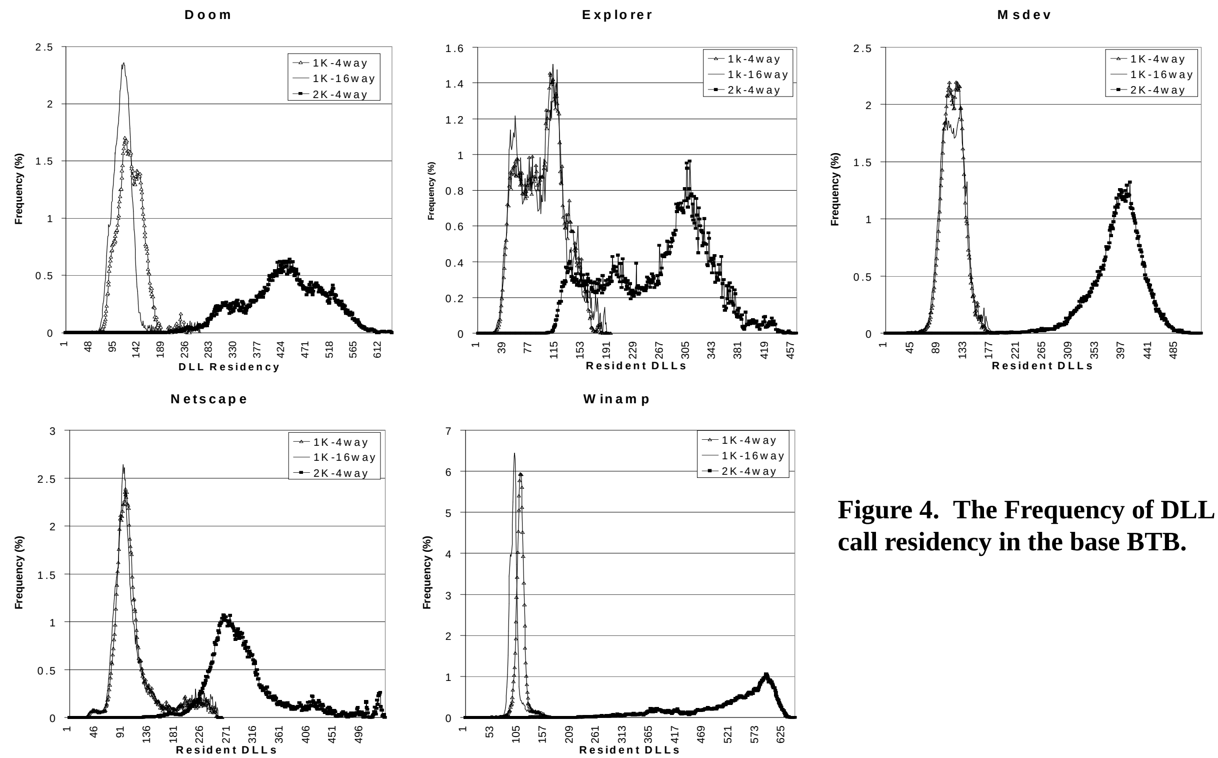

Figure 4. The Frequency of DLL call residency in the base BTB.

6.1 Initial Observations

Of the applications studied, Doom uses DLLs a little less frequently than the others. This is because Doom is not a true 32-bit application, but requires the use of NTVDM.EXE to convert the Win16 calls to Win32 calls.

The Explorer benchmark trace is about half the size of the other benchmark traces. In fact, for the loading of the same three web pages, it took Explorer only 400 million instructions, while Netscape required over 700 million. Although the overall figures are lower, the various ratios are fairly consistent.

The Microsoft Visual C++ environment tends to make liberal use of DLLs, as we see in the following section. Compiling a Win32 application, as opposed to a console application (as *go* was compiled), would have seen an even greater increase in the number of DLLs used by Msdev. Win32 applications require more system support (and hence more DLL calls) to operate.

Finally, Winamp uses the most floating point instructions out of the applications studied. Interestingly, with a 16K entry branch predictor for Winamp, we correctly predict the direction 95% of the time, but correctly predict the right address only 60% of the time. We will see that this drop is due to contention in the 1K entry BTB.

6.2 BTB Residency by Category

To construct Table 4 we simulated a standard size (1K entry, 4-way) Base BTB (*B*) and a DLL BTB (consisting of a Filtered BTB (*F*) and a DTB (*D*)). Table 4 shows a categorization of the accesses to the BTB. The sum of the entries in a column is the total number of BTB accesses per one thousand instructions. The first four rows in the table handle all taken control instructions except DLL calls. The first row represents the total number of times that a non-DLL call hits in both the Base BTB (*B*) and the Filtered BTB (*F*) in our DLL BTB. The second row shows the total number of times that a non-DLL branch was found in *B* but not *F*. Although intuitively this number should be zero, there are some cases where LRU replaces different entries in the two systems. The third row highlights the additional coverage that the Filtered BTB captures over the Base BTB. The fourth shows taken non-DLL calls that miss in both the Base BTB and the Filtered BTB. The last four rows are analogous to the first four, but handle only DLL calls. For instance, the fifth row shows the total number of times that a DLL call target was found in both the Base BTB (*B*) and the DTB (*D*) in our DLL BTB. The seventh row highlights the additional coverage of DLL function calls that our dedicated 128 entry DTB buffer provides. The sixth row gives us insight into whether the 128 entry size is sufficient. For most of the applications, it seems to provide good coverage.

TABLE 4. Likelihood of BTB Residency by Category (per 1000 instructions)

	Doom	Expl.	Msd.	Net.	Win.
$B \cap F$	100.5	114.0	148.7	105.4	86.23
$B \cap \overline{F}$	0.001	0	0.022	0	0
$\overline{B} \cap F$	1.750	0.777	1.356	0.699	1.651
$\overline{B} \cap \overline{F}$	13.44	16.77	16.40	10.43	12.54
$B \cap D$	13.93	7.708	15.48	19.74	6.289
$B \cap \overline{D}$	0.028	0.034	0.080	0.029	0.029
$\overline{B} \cap D$	0.808	0.480	0.532	0.383	1.073
$\overline{B} \cap \overline{D}$	0.838	1.684	4.559	1.317	2.013

For Doom, in Table 4, the Filtered BTB captures only 1.7% of the extra branches whereas the DTB captures 5.6% more DLL function calls. For Msdev, which has the greatest reliance on DLLs, we get a 4% increase in coverage of DLL calls, but only a 0.9% increase in coverage from other branch targets. Since we still miss nearly 30% of all DLL calls with a 128 entry buffer, we might need to use a larger DTB. However, we do capture more DLL calls than the Base BTB. From the instruction cache miss rates in Figure 1, it is clear that Msdev has a larger working set than the other applications. For Netscape, coverage of DLL calls is roughly 93%, which implies that a buffer size of smaller than 128 can be used. Winamp is unique in that the DTB captures 17% more DLL call targets than the standard configuration. For each application, most of these DLL calls actually *pollute* the standard BTB, as seen in Figure 3, and hence we can gain added performance by filtering out these calls.

6.3 DLL Residency

To get a better indicator of coverage, we count the number of DLL calls resident in the base BTB *on every BTB update*, i.e.every taken control flow instruction. Figure 4 shows the frequency distribution of the number of DLL call targets in the BTB over all BTB accesses. This data is shown for each application for three different configurations: the standard 1K entry, 4-way base BTB plus a 1K entry, 16-way and a 2K entry, 4-way configuration for comparison. This information can be used as guide to select the size of the DTB. This curve is a distribution of the number of DLL calls resident in the standard BTB whenever the standard BTB is updated. These curves do not have a bimodal distribution; for every application, the two 1K entry curves end at about the middle of the x-axis.

The DLL residency graph for Explorer shows a wider distribution than for Doom, but the average is close to the same for similar configurations. The varying distribution is due to the shorter instruction trace. Note the higher average of resident DLLs for the 2K entry BTB in which case a larger DTB would definitely be useful. The DLL residency graph for Msdev shows a similar distribution to Doom for the 1K entry sizes, but for the 2K configuration, the average is much higher than for the other benchmarks. On average, nearly 400 DLL calls are resident at any given time. This is almost 15% of the total number of entries. Since DLL calls only comprise 6% of all dynamic taken branches, an average residency of 400 seems excessive. Winamp has the least variance when it comes to the average number of DLL call targets resident in the standard BTB. This would imply that Winamp has one of the smaller instruction footprints, and tends to stay in the same DLLs. As shown in Figure 2, it does have a small instruction footprint.

6.4 DLL BTB Performance

The performance numbers that we present in this section are represented in terms of misses per one thousand instructions. By presenting the data in this fashion, the impact of the hardware optimizations immediately becomes clear. For example, if the penalty to miss in the BTB is 23 cycles and the average instructions per cycle is (IPC) 1.25, saving one miss per one thousand instructions, corresponds to a 23 cycle reduction from 800 to 777 cycles (or about 2.5%). The numbers shown in Table 6 are for the base configuration: a 1K entry, 4-way BTB.

Most of the applications have similar values for the base configuration, except for Msdev. This application has roughly twice as many indirects, and twice as many calls, that miss in the standard BTB. The DLL BTB outperforms the Victim BTB for all of the applications for indirects and indirect calls; however, most of the applications already have low values for these categories. Because this is not true for Msdev, the DLL BTB outperforms the Victim BTB in overall performance (shown in Figure 5).

Generally, the categories with the largest misses per one thousand instructions are Calls and Conditionals. The Victim system has an edge over the DLL BTB throughout these categories, since any hot set contention can be ameliorated by the Victim BTB, whereas only those hot sets caused by DLLs can off-load entries into the DTB.

TABLE 5. Misses Per Instruction (x1000) for 1K entry, 4-way configuration

	Doom			Explorer			Msdev			Netscape			Winamp		
	base	**dll**	**vic**	**base**	**dll**	**vic**	**base**	**dll**	**vic**	**base**	**dll**	**vic**	**base**	**dll**	**vic**
ind.	1.41	0.83	1.02	2.45	2.01	2.20	5.86	5.20	5.56	1.63	1.27	1.42	1.95	0.97	1.48
ind. calls	1.30	0.76	0.95	1.70	1.33	1.52	3.68	3.23	3.50	1.09	0.83	0.95	1.73	0.86	1.31
calls	3.84	2.94	2.55	6.11	5.57	5.39	8.16	7.30	7.38	3.63	3.20	3.09	4.36	3.12	2.96
rets	4.21	4.21	4.21	2.48	2.48	2.48	2.64	2.64	2.64	1.35	1.35	1.35	3.42	3.42	3.42
con.	4.72	3.95	2.84	5.15	4.80	4.28	4.50	4.19	4.01	4.46	4.12	3.50	5.85	5.01	3.65
unc ond.	5.80	5.47	5.16	4.79	4.64	4.54	6.25	5.93	5.98	2.74	2.60	2.57	4.90	4.64	4.33
oth.	0.69	0.40	0.46	0.45	0.41	0.38	0.43	0.37	0.36	0.24	0.22	0.21	0.45	0.28	0.31

This improvement is an average of 5.3% for calls and 22% for conditionals.

Table 6 shows how the DLL BTB did alleviate some of the hot set contention, and the impact of the number misses per one thousand instructions for the seven categories. To obtain another measure of interference of resources, we measured the average lifetime of a BTB entry. This is measured in terms of the number of instructions, and is done for the Base BTB, the Filtered BTB, and the DTB. These results are shown in Table 6.

Table 6. Average lifetime of a BTB entry (1K 4way)

	Base	Filtered	DTB
Doom	316,198	543,184	333,049
Explorer	67,431	76,072	153,530
Msdev	75,766	93,439	91,674
Netscape	110,163	130,225	199,045
Winamp	715,711	1,346,477	445,162

This table illustrates that, as we would expect, DLL calls have a coarser locality (larger average lifetime) than other branches. The Doom and Winamp applications highlight how much interference can occur between DLL calls and other branches. When DLL calls are removed from the instruction stream (i.e. the Filtered BTB), the lifetime of a BTB entry doubles in these two benchmarks. It is also in these two applications that the most benefit is gained from using the DLL BTB.

To see the effect on overall performance, we deduce the performance analytically. Since we know the total number of instructions and the total number of branches and branch hits, if we fix the IPC and the branch misprediction penalty, we can calculate the improvement in IPC. Using a Pentium Pro as our model, we fixed the BTB miss penalty to 23 cycles, and our base IPC to be 1.25. Figure 5 shows the overall improvement. This is actually a pessimistic model for our method since a penalty for either swaps of lookups for the Victim BTB is not assigned, and our branch predictor is perfect. For a 16-way BTB for the Winamp application, we actually see around a 10% performance improvement. In most cases, we track the Victim BTB closely; in fact, in the Msdev application the DLL BTB actually surpass the performance of the Victim BTB.

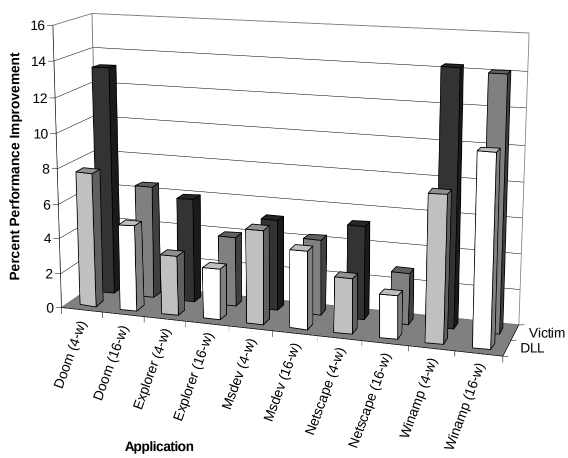

Figure 5. Performance Improvement over the Standard 1K entry System (darker shades 4-way, lighter shades 16-way)

7.0 Concluding Remarks

We have seen that, because of their nature, applications with multiple dynamic object files have different characteristics than their traditional statically linked counterparts. We have highlighted some of these differences in this paper. In one area that they deviate significantly, branch target prediction, we have presented some of the characteristics that can be leveraged to increase performance. Branch target prediction is especially important in today's (and tomorrow's) wide-issue superscalar machines.

The ratio of the number of indirect function calls to direct function calls is much higher in applications that use dynamically loaded libraries. Since these applications have multiple object files, the calling linkage between two functions residing in different object files requires an indirect function call. We have proposed the DLL BTB to separate out these calls and place the targets in a special buffer (DTB) so that they will not interfere with the targets of other types of branches. For a 1K entry Branch Target Buffer, the DLL BTB performs close to that of a Victim BTB for the applications we studied; however, the DLL BTB has the benefit of not requiring the extra datapath that the Victim requires. For larger BTB sizes, our 128 entry DLL target buffer does not improve performance as effectively; our results indicate that a larger DLL target buffer is required in this situation.

Although we primarily looked at one specific microarchitectural resource, the Branch Target Buffer, we can use this work as a jumping off point for other microarchitectural resource usage studies in the presence of multiple object files. Since Win32 applications are different from standard benchmarks and are more widely used in practice, further research tailored to this environment is well justified.

References

[1] B. Calder, D. Grunwald, "Reducing Indirect Function Call Overhead in C++ Programs," *Proceedings of the ACM Principles of Programming Languages,* ACM, 1994.

[2] B. Calder, D. Grunwald, A. Srivastava, "The Predictability of Branches in Libraries," *Proceedings of the 28th International Symposium on Microarchitecture.* pp. 24-34. IEEE, 1995.

[3] P.Chang, E. Hao, Y. Patt, "Target Prediction for Indirect Jumps," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 274-283, IEEE, 1997.

[4] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 364-373, IEEE, 1990.

[5] K. Lawton, "Welcome to the Bochs x86 PC Emulation Software Home Page!" http://www.bochs.com.

[6] D. C. Lee, P. J. Crowley, J-L Baer, T. E. Anderson, and B. N. Bershad, "Execution Characteristics of Desktop Applications on Windows NT," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 27-38, IEEE, 1997.

[7] S.E. Perl, R.L. Sites, "Studies of Windows NT Performance Using Dynamic Execution Traces," Digital Systems Research Center Research Report, RR-146, April 1997.

[8] C.H. Perleberg, A.J. Smith, "Branch Target Buffer Design and Optimization," *IEEE Transactions on Computers*, 42(4):396-412, 1993

[9] G. Reinman, T. Austin, B. Calder, "A Scalable Front-End Architecture for Fast Instruction Delivery," *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 234-245, IEEE, 1999.

[10] J. A. Rivers and E. S. Davidson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design," *Proceedings of the 1996 International Conference on Parallel Processing*, Vol. I, pp 151-162, August 1996.

[11] E.S. Tam, "Improving Cache Performance via Active Management," Ph.D. Dissertation, University of Michigan, June 1999.

[12] D. Wall, "Predicting Program Behavior Using Real or Estimated Profiles," *Proceedings of the ACM SIGPLAN '91 Conference on Programing Language Design and Implementation*, pp. 59-70, ACM, June 1991.

[13] T. Yeh, "Two-Level adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Super-Scalar Processors," Ph.D. Dissertation, University of Michigan, 1993.