Architectural Support for Dynamic Linking

Varun Agrawal, Abhiroop Dabral, Tapti Palit, Yongming Shen, Michael Ferdman
COMPAS
Stony Brook University

Abstract

All software in use today relies on libraries, including standard libraries (e.g., C, C++) and application-specific libraries (e.g., libxml, libpng). Most libraries are loaded in memory and dynamically linked when programs are launched, resolving symbol addresses across the applications and libraries. Dynamic linking has many benefits: It allows code to be reused between applications, conserves memory (because only one copy of a library is kept in memory for all the applications that share it), and allows libraries to be patched and updated without modifying programs, among numerous other benefits. However, these benefits come at the cost of performance. For every call made to a function in a dynamically linked library, a trampoline is used to read the function address from a lookup table and branch to the function, incurring memory load and branch operations. Static linking avoids this performance penalty, but loses all the benefits of dynamic linking. Given its myriad benefits, dynamic linking is the predominant choice today, despite the performance cost.

In this work, we propose a speculative hardware mechanism to optimize dynamic linking by avoiding executing the trampolines for library function calls, providing the benefits of dynamic linking with the performance of static linking. Speculatively skipping the memory load and branch operations of the library call trampolines improves performance by reducing the number of executed instructions and gains additional performance by reducing pressure on the instruction and data caches, TLBs, and branch predictors. Because the indirect targets of library call trampolines do not change during program execution, our speculative mechanism never misspeculates in practice. We evaluate our technique on real hardware with production software and observe up to 4% speedup using only 1.5KB of on-chip storage.

Categories and Subject Descriptors C.4 [**Performance of Systems**]: Performance of Systems—Design studies

Keywords Dynamic Linking; Hardware Memoization; Instruction Elision; Branch Prediction

ASPLOS '15, March 14 - 18 2015, Istanbul, Turkey
Copyright 2015 ACM 978-1-4503-2835-7/15/03…$15.00
http://dx.doi.org/10.1145/2694344.2694392

1 Introduction

All computer programs in use today rely on software libraries. Libraries can be linked dynamically, deferring much of the linking process to the launch of the program. Dynamically linked libraries offer numerous benefits over static linking: they allow code reuse between applications, conserve memory (because only one copy of a library is kept in memory for all applications that share it), allow libraries to be patched and updated without modifying programs, enable effective address space layout randomization for security [21], and many others. As a result, dynamic linking is the predominant choice in all systems today [7].

To facilitate dynamic linking of libraries, compiled programs include function call trampolines in their binaries. When a program is launched, the dynamic linker maps the libraries into the process address space and resolves external symbols in the program and between the libraries, populating the symbol addresses in a global lookup table. Subsequently, every call to a library function executes a trampoline that reads the corresponding function address from a lookup table and jumps to the target library function. The trampoline execution is an overhead of dynamic linking compared to static linking, increasing the number of instructions executed and adding pressure on the instruction cache and I-TLB for trampoline instructions, the data cache and D-TLB for storing addresses in lookup tables, and the branch predictor and branch target buffers.

Several techniques can mitigate the performance degradation associated with the overhead of dynamic linking. The entire penalty can be avoided by statically linking libraries. However, static linking loses the benefits of dynamic linking and is therefore not used in practice (the design of many complex applications does not even permit static linking). Hardware memoization techniques [8, 22] offer an alternative approach in which trampolines can be skipped by memoizing trampoline execution; however, these techniques lie on the critical path of instruction fetch. Some overheads of dynamic linking can be mitigated through profile-driven optimization [17] of data and instruction layout in memory [14], but these approaches are limited to improving the cache and TLB behavior; optimization does not eliminate trampolines or the branch predictor pressure they create.

We observe that much of the performance overhead of dynamic linking can be avoided. Entries in the dynamic linker lookup tables are updated only once, when each symbol is resolved, typically at the first execution of the corre-

sponding library call. All subsequent invocations of the trampolines are unnecessary and serve only to create performance overheads. Noting this, we propose a lightweight hardware mechanism that leverages existing branch predictor structures to optimize the function calls in dynamically linked libraries by speculatively skipping trampoline execution, providing the performance of static linking while retaining the benefits of dynamic linking. Our technique does not affect the processor's critical path, as it uses the existing branch target buffers and branch predictor update mechanisms. The approach works on all dynamically linked library techniques that we are aware of, across architectures (e.g., ARM and x86) and modern operating systems (e.g., Linux, Windows, BSDs).

To demonstrate the effectiveness of our approach, we use software to model the effect of the proposed hardware technique on real server hardware running production software, using existing performance counters to demonstrate the reduced pressure on the various microarchitectural structures. For an Apache web server running the SPECweb 2009 workload, for example, approximately 1% of the executed instructions are in trampolines for library function calls. Skipping the execution of the trampolines reduces instruction cache misses, data cache misses, and branch mispredictions. As a result, Apache performance (request processing latency) improves by 4%.

The software technique we use to emulate our proposed hardware solution incorporates assumptions that prevent its use in production systems: our software loads all executable code (including libraries) in a contiguous 2GB of the address space to limit relative jump offsets, changes page permissions of executable code on demand, uses profiling to locate library function call sites, copies executable code pages after they are patched, and increases load time for applications. To make a software technique practical, the software and compiler toolchains need to change significantly and security implications must be addressed. Given these limitations, we advocate a lightweight hardware solution that works transparently on existing systems, without requiring software modifications. However, with sufficient ecosystem change, a software mechanism that leverages our observations may also become viable.

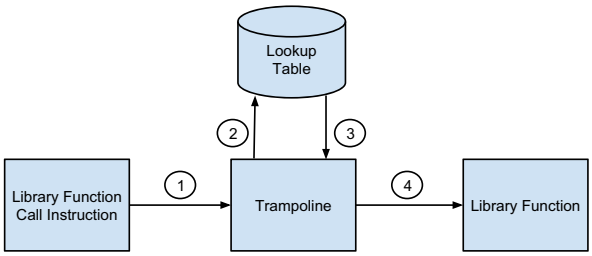

Figure 1. Library function call mechanism for dynamically-linked libraries.

2 Dynamic Linking

Dynamic linking is the predominant method of linking libraries in today's systems. In dynamic linking, external symbols used in a program and across libraries are pointed to small trampoline routines automatically inserted into the binaries by the compiler. The dynamic linker populates the corresponding symbol addresses into lookup tables referenced by the trampolines. When a program is launched, the dynamic linker maps the dynamically linked libraries into the application's memory space and allocates the corresponding lookup tables. Figure 1 shows the function call mechanism for dynamically linked libraries; every call to a library function jumps to a trampoline that reads the address of the target library function from a lookup table and then jumps to the function defined in the library's binary.

Figure 2 shows the instructions that are executed when calling the *printf* function in a dynamically linked library on x86 and ARM architectures with ELF binaries. The trampolines in these systems are defined in the procedure linkage table (PLT) sections. The trampoline for *printf* is called *printf@plt*. All the call/branch sites in the application and library binaries that call *printf* actually call the *printf@plt* trampoline. The trampoline comprises an x86 indirect *jmp* instruction (or two *add*s and a *branch* for ARM). The indi-

```
.text:      ...
            call    printf@plt
            ...
.plt:       ...
printf@plt:
            jmp     *(printf@got.plt)
            push    0x10
            jmp     .plt
            ...
.text:      ...
printf:     push    %rbp
            ...
            ret
```

(a) x86-64

```
.text:      ...
            bl      printf@plt
            ...
.plt:       ...
printf@plt:
            add     ...
            add     ...
            ldr     pc, [printf@got.plt]
            ...
.text:      ...
printf:     push    {fp,lr}
            ...
            pop     {fp,pc}
```

(b) ARM

Figure 2. Library function call examples for a dynamically-linked ELF executable.

rect branch instruction reads the global offset table (GOT) location *printf@got.plt* and branches to the address loaded from it. Resolution of function symbols is typically done in a lazy fashion. Initially, the dynamic linker populates all entries in the GOT to branch to the dynamic linker's own resolver function. Upon the first invocation of each library function, the resolver updates the GOT lookup table entry with the real library function's address, ensuring that subsequent executions of the trampoline find the address of *printf* when reading *printf@got.plt*.

Trampoline instructions are not densely packed in the PLT section. For each trampoline, several additional instructions follow the indirect branch. These additional instructions are used only on the first trampoline (resolver) execution to indicate to the resolver the name of the function being resolved. Moreover, compilers typically allocate PLT entries in the order in which the corresponding functions are defined in the source. Because only a small, random subset of available library functions are typically called by a program, the PLT sections are sparsely used and accesses to the PLT and GOT entries exhibit practically no spatial locality.

2.1 Benefits of Dynamic Linking

Dynamic linking is considered indispensable in the software and OS communities. The performance overheads are well worth the benefits, some of which we outline below.

Flexibility Dynamically linked libraries greatly enhance portability, particularly with *standard* libraries that provide a common system interface.

Dynamically linked libraries allow bug and security fixes to be incorporated systemwide by simply restarting applications with updated library binaries, without requiring modifications to each application.

With dynamically linked libraries, users can customize application execution by specifying which libraries to use and by selecting different implementations of specific functions from different libraries.

Memory conservation Libraries grow over time with the addition of new interfaces and functionality. Although most of the functions of any given library are not used by any single application that links against that library, the full library code must still be loaded in memory. Dynamic linking allows for a single read-only copy of the library binary to be mapped into all processes that need it, potentially saving gigabytes of memory on systems with large numbers of processes [19].

Security Shared libraries enable vital security measures, such as address space layout randomization [21]. The position-independent nature of dynamic libraries allows programs to randomly map shared libraries into application virtual address spaces. This means that the locations of program and library code change with each process invocation, limiting the ability of malicious users to mount attacks.

2.2 Penalties Incurred by Dynamic Linking

Dynamic linking incurs many penalties that reduce system performance. The source of these penalties is the trampolines; a trampoline is executed for every call to a library function, performing an extra memory load and an extra indirect branch. Complex applications that use many libraries (e.g., Apache, Memcached, MySQL and Firefox) frequently perform library calls from within the application and between shared libraries, resulting in significant overheads from trampoline execution and its side effects.

Instructions Calling library functions puts pressure on the instruction cache [20], as one cache line is occupied for each trampoline. On x86-64 systems using ELF binaries, trampolines are 16 bytes, allowing only four trampolines per 64-byte instruction cache line [9]. However, because the trampolines are sparsely spread throughout the PLT sections, an instruction cache line is effectively dedicated to each library call made by the application and each library. For example, if an application and four libraries against which it is linked frequently call *write*, five instruction cache lines are dedicated to the trampolines for this *write* library function (one in each PLT). The trampoline instructions put similar pressure on the instruction translation lookaside buffer (I-TLB).

Data working set Every trampoline performs a load operation that reads the library function pointer from the linker's lookup table. As in the instruction cache, the lookup tables (e.g., GOT) occupy data cache lines and data TLB entries. Although function pointers are more densely packed in memory than are the trampolines, storing eight pointers in each 64-byte cache line, the storage is similarly sparse and one cache line is needed for each library function called from the application and one for each calling library.

Branch prediction accuracy Branch predictor storage is extremely limited due to its critical nature; conflicts in the branch predictor structures lead to direction and target mispredictions, reducing system performance [15]. Statically linked function calls require one branch for their invocation; dynamically linked libraries require two branches for each library call—one to branch to the trampoline and a second one to branch to the actual function. Thus, dynamically linked libraries occupy two entries in the branch predictor tables and branch target buffers (BTB) per call, creating greater pressure on these structures. As a result, dynamic linking has an increased conflict rate and reduced accuracy compared to static linking.

Although the benefits of dynamic linking overshadow the penalties, the penalties are substantial, particularly in complex applications. The hardware technique we propose in this work ameliorates those penalties by skipping trampoline execution entirely, avoiding the allocation of instruction cache lines for trampolines, data cache lines for lookup tables, and branch predictor entries for trampoline branches. By reducing the number of executed instructions and conse-

quently the pressure on these microarchitectural structures, our approach retains all the benefits of dynamic linking and achieves the performance of static linking.

2.3 Challenges of a Software Solution

In dynamic linking, library function addresses are unknown when a program's code is compiled, preventing the program binary from directly calling the target functions. This necessitates the use of trampolines, which read a resolved target address from the GOTPLT and branch to the appropriate location. A dynamic resolver, which executes on the first invocation of each library function, identifies the target address and installs it into the GOTPLT. Although it is functionally effective, this approach hampers performance because the trampoline remains in the call path of each subsequent library function invocation.

At first glance, it seems possible to create an alternative dynamic resolver that avoids trampoline overheads. Rather than dynamically populating the GOTPLT with the target function's address, the dynamic resolver could directly modify the instruction at the call site with the target address. Future executions of the same call site would then call the needed library function directly, avoiding the performance overheads associated with executing the trampoline. However, this naive approach presents a number of challenges.

First, although library functions called from the application binary are similar to functions called from other libraries, they would be affected differently by the naive software solution. Real systems load libraries above the heap and far from their call sites in the virtual address space. As a result, the distance between the target address and the call site is much more than 2GB. For example, the x86 ISA encodes relative call targets using an offset from the call site; the instruction encoding is limited to an offset of 2GB. As a result, either the ISA has to change to support larger offsets or the software, including the conventional process memory map, must change to allow dynamically linked libraries to be loaded between the text and heap sections.

Another challenge arises from unconventional tricks for calling functions, such as using a *jump* instead of a *call* instruction to invoke a function. In a conventional scenario, a dynamic resolver can examine the stack to determine the location of the call site to patch. However, when the unconventional approach is used, the stack does not contain the address to patch (the address of a jump), instead pointing to the address of the preceding call; patching this call instruction would result in incorrect execution. A software solution would therefore need to address this possibility; either additional mechanisms would have to be added to disassemble the call site pointed to by the stack to verify that it is the appropriate location to patch, or toolchain changes would have to be made to enable the compiler to communicate a list of valid patch sites to the resolver.

The software solution could also create enormous memory overheads. Dynamic linking uses a copy-on-write (COW) mechanism to share a single executable read-only copy of each library between all applications in the system. However, the libraries are mapped at a different address in each application's virtual address space. Further, when a process is forked, the executable code is shared by the parent and child processes. Patching the call sites both in the libraries (when they call other libraries) and in the application binary will have different effects based on whether child processes are forked before or after patching. Patching before fork will retain the COW property, retaining just one copy of each code page. However, patching after fork will force allocation of a new physical memory page for each process. Busy servers with thousands of processes and thousands of library function call sites could waste multiple gigabytes of RAM on these unnecessary page copies. Avoiding this memory overhead with a software solution would require patching all call sites before the fork, which precludes the traditional lazy resolver mechanisms and requires modifying the compiler toolchain to include lists of all library function call sites to patch.

Even if the memory overheads associated with lazy resolution in this scenario were acceptable, the software solution would still have significant performance implications because an application can have thousands of library functions (including library functions calling other libraries), a fraction of which are actually used. Lazy resolution supports fast application load times by resolving functions only when they are called for the first time, regardless of which call site triggers the resolution. Thus, a single library function will be resolved only once in the traditional dynamic linking approach. By contrast, the naive software solution to avoid dynamic linking overheads would need to perform binary patching for each call site, even if all call sites pointed to one library function. The software solution may therefore incur significant performance overheads, especially considering that for each call site patch, the OS must be involved to unprotect the code page and make it writable and the resolver must keep track of all patched call sites for the rare case of library unload.

Even if these implementation challenges could be overcome, the software approach introduces a number of security challenges. The benefits of address-space layout randomization would be eroded because shared libraries would have to be placed within 2GB from call sites. Additionally, because the loader must unprotect code pages to patch call sites, the system creates an opportunity for attackers to inject malicious code [2].

2.4 Similar Techniques

The mechanism of dynamic linking is a special case of a lookup table, in which the address of a function is looked up in a table (here, the PLT) and control flow jumps to that

function address. We describe two related techniques that use lookup tables to change program control flow.

2.4.1 GNU Indirect Functions

New processors include specialized instructions that optimize particular computations. Many commonly used functions can benefit from these specialized instructions if they are available. In order to support different hardware, and thus allow access to these specialized instructions, libraries usually include several different implementations of a function, choosing the best implementation based on the hardware available when the application is loaded. The GNU linker provides a feature, called *ifunc*, which chooses from among the different candidate implementations using a resolver function that determines the hardware capabilities. Because the compiler is not aware of which function will be selected at load time, calls to *ifuncs* are made using the PLT in the same way that dynamically linked functions are called. Available in GNU libc since version 2.10, this mechanism is used extensively for common library routines (e.g., string manipulation functions) and is also available to software developers, who use it to incorporate platform-specific *ifunc* implementations directly in executable binaries.

2.4.2 C++ Virtual Functions

Declaring C++ member functions *virtual* allows objects to access them even if the object's pointer is passed as a pointer of the superclass. Virtual function addresses are dereferenced from the object's data structure, which then performs a direct call to the function address. This mechanism is similar to calls to dynamically linked functions, as both look up the target function in a table and then branch to it. However, the instruction sequence for address resolution and invocation of C++ virtual functions differ significantly from that of dynamic library calls.

3 Architectural Support for Dynamic Linking

Calls to functions in dynamically linked libraries occur with sufficiently high frequency to warrant hardware support in high-performance systems. In this work, we describe a speculative hardware mechanism to improve the performance of dynamic linking by skipping trampoline execution.

On an x86-64 system, trampolines comprise a complex indirect branch instruction that performs both load and branch operations. When a *call* instruction is fetched, the branch predictor provides the address of the trampoline in the PLT section. The trampoline virtual address is then translated to its physical address using the I-TLB and the trampoline instruction is fetched from the L1 instruction cache. The indirect branch instruction performs a load operation that uses the D-TLB to translate the GOT entry address and fetches the function pointer from the L1 data cache. Finally, the retrieved pointer is translated using the I-TLB to determine the address of the actual function.

Although the trampoline indirect branch instruction performs two expensive operations [13], the result of these instructions does not change after the first trampoline invocation (the pointer loaded from the GOT and the library function address do not change during program execution). The behavior of the proposed mechanism relies on this fact, skipping the trampolines whenever possible while maintaining an architectural state identical to the unmodified system.

3.1 Hardware Overview

Our mechanism uses existing hardware of the processor front end to trick instruction fetch into skipping the fetching of the trampoline (in turn, tricking the back end into skipping its execution). This is accomplished by storing the address of each library function in the BTB entry that would normally store the address of the trampoline corresponding to that function. The processor front end does not require any modifications to perform this operation, as the target address is set using the standard branch predictor feedback mechanism from the back end [24]. In the back end, the branch resolution and misprediction identification mechanism are modified to check whether the real branch target is the address of a trampoline; if it is, the processor treats the predicted address as correct if the predicted target matches the target of the trampoline's branch. To facilitate this comparison, a retire-time alternate BTB (ABTB) structure stores a mapping of trampoline addresses to library function addresses.

To ensure correctness in the rare event that the trampoline targets change, we use a bloom filter [1] to store the addresses of the GOT entries that represent trampoline targets corresponding to the entries in the ABTB. If an entry in the GOT is modified and the GOT entry hits in the bloom filter, then the target of the ABTB may be incorrect. In this case, we clear the entire ABTB table and rely on the standard branch misprediction logic to ensure correct execution (in practice, this happens only once per library call, at the start of a program's execution, making this overhead irrelevant).

Entries are added to ABTB and the bloom filter during the retire stage of the processor pipeline. We use the characteristic of a trampoline (call instruction followed by indirect branch instruction) to populate the ABTB and bloom filter. The ABTB and bloom filter entries are context specific and can be directly tied to the mechanisms and optimizations available for maintaining (or flushing) the BTB and TLB entries on context switch.

3.2 Speculative Trampoline Skip

Front end In the front end, we use the existing branch predictor mechanism to skip trampoline execution. When fetching a library function call instruction, the branch predictor provides the target of the call instruction. During normal execution, the target would be the function's trampoline; we change the branch prediction update mechanism to make the actual function address the target of the call instruction. The

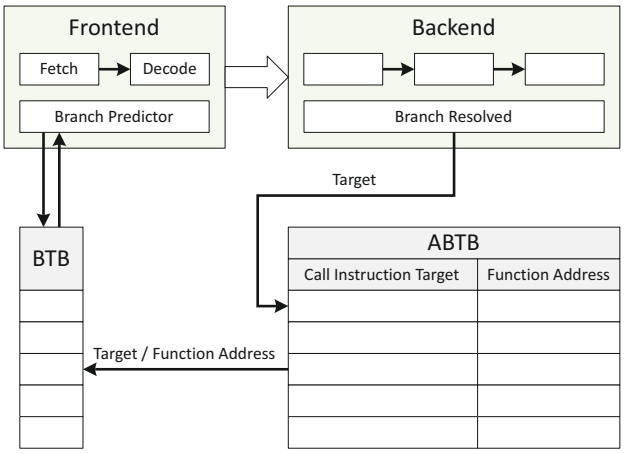

Figure 3. Speculative trampoline skip mechanism.

function address is read from a BTB entry that is indexed using the call instruction's program counter. The target is speculative; the branch predictor update mechanism corrects any mispredictions. Modifying the BTB entry in this way means the trampoline does not get executed, reducing the pressure on the instruction cache, data cache, I-TLB, and D-TLB, as well as on the BTB and branch predictor.

Back end In the back end, our hardware solution modifies the branch predictor update mechanism to use the target of the library call instruction and not the trampoline address. Branch prediction is speculative; the processor executes the instructions as they are fetched, whether the branch predictor provides the function address or the trampoline address. When the target of the call instruction is resolved, the branch predictor update mechanism tests whether the target was predicted correctly. If a misprediction is identified, the BTB entry is corrected. As long as the address in the GOT is not modified, the target of the trampoline does not change and the modified branch target prediction remains correct.

Figure 3 shows our hardware mechanism with an ABTB table. The ABTB maps the real target of the call instruction (the trampoline's address) to the target of the trampoline (the library function address). When the back end resolves the target of a branch instruction, the address is looked up in ABTB. If the target is not found in ABTB, the branch predictor operates without modification, updating the BTB with the real branch target. However, if the target matches an ABTB entry, the branch resolution mechanism uses the result of the ABTB lookup as the correct branch target.

Entries in ABTB are valid as long as the corresponding entries in the GOT section are not modified. A small bloom filter that contains the GOT addresses corresponding to the entries in ABTB is sufficient to detect when any GOT entry is updated. When the processor retires a store instruction to an address that hits in the bloom filter (or an invalidation for such an address is received from the coherence subsystem), all entries in ABTB and the bloom filter are cleared. At program startup, when GOT entries are initially populated, clearing the ABTB prevents the program from skipping the trampolines until the ABTB is populated with the correct final targets, at which point the addresses are never updated again in practice.

Populating the ABTB Library function calls have a distinguishing pattern: a call instruction is followed by an indirect branch. When a call instruction is retired, the resolved target of the call instruction is noted. On retiring the subsequent instruction, if that instruction is an indirect branch, the preceding call's target is used as the index into the ABTB and the target of the indirect branch instruction is used as the value. The bloom filter is updated with the source address from which the indirect branch's address is loaded. In all other cases, no new entries are added to the ABTB.

3.3 Handling Misspeculation

All speculative hardware mechanisms need to ensure that the processor recovers the correct execution state gracefully in case of misspeculation. In most cases, this recovery is expensive, as it requires flushing the pipeline to remove instructions following the misspeculation. In our proposed hardware, misspeculation happens when the result of the BTB doesn't match the result of a matching ABTB entry. There are four possible cases where this can happen:

Conflicts in BTB entries When there are conflicts in the BTB entries, the branch predictor mispredicts and the processor fetches the wrong instructions following the library function call. To recover from such misspeculation, the hardware mechanism described in Section 3.2 fetches instructions from the library function instead of its trampoline. Because the program binary is unmodified, conflicts in BTB are *not* significantly affected by our approach. In fact, skipping trampolines reduces pressure on the BTB, reducing the probability of conflicts occurring in the first place.

GOT entry of library function modified The address of a library function is resolved on the first call to that function from a given module (application or library). It is then stored in the GOT entry for the module corresponding to that function. On the first invocation of a call instruction, the branch predictor cannot predict correctly, so the first call always results in a branch misprediction. Once the function address is resolved and stored in the GOT, it remains unmodified for the life of the program (unless the module is unloaded, which does not occur in practice).

In our hardware, the second invocation of a call instruction to a library function will also mispredict, as the trampoline has never branched to the function before. This is also true in the base system, as the branch predictor can predict the call instruction target (trampoline) correctly, but cannot predict the trampoline target correctly. On subsequent invocations, however, the branch targets are correctly predicted

both by the base system and by our proposed hardware. In summary, we do not introduce any branch mispredictions that were not present in the base system.

Missing ABTB entry after a hit in bloom filter As discussed in Section 3.2, all ABTB entries are removed if the address of a store operation hits in the bloom filter. Modification of function addresses after the first invocation is rare, and thus, entries in ABTB stabilize rapidly and do not get cleared in practice. However, when the ABTB is cleared, the system behaves in exactly the same way as a base system without hardware support for dynamic linking, hence incurring exactly the same number of misspeculations.

Missing ABTB entry after context switch In an OS context switch, the processor starts executing a different program having a different virtual to physical address mapping. Addresses in the ABTB are virtual and become invalid after a context switch in the same way that the TLB entries do. If, however, a mechanism is employed to retain TLB entries on a context switch (e.g., using address space IDs), the same mechanism applies to the ABTB, avoiding misspeculation.

3.4 Alternate Implementation

The mechanism we describe introduces no architecturally visible changes to the hardware. If, however, architecturally visible changes can be introduced, the hardware cost of the proposed technique can be reduced by avoiding the bloom filter and requiring the software to explicitly invalidate the ABTB. This process is analogous to architectures that do not enforce coherence between the instruction and data caches, requiring an explicit instruction cache flush when instructions in memory are updated [16].

4 Methodology

To measure the impact of our technique, we conduct experiments on a real system and mimic the behavior of the proposed hardware technique by modifying the dynamic linker software so that it changes the targets of call instructions from the trampoline routines to the function address. Although this approach enables us to measure the effects of the proposed hardware, it is functionally similar to static linking, in that it loses all of the benefits of dynamic linking. Removing or updating a library could result in dangling call instruction targets. Thus, this implementation, which is designed solely for the purpose of evaluation, doesn't support unloading or replacing libraries; on the other hand, the hardware we propose implicitly supports these operations.

4.1 Hardware

Our experiments were conducted on a Dell server with two Intel® Xeon® E5450 [11] CPUs and 32 GB of memory. The CPUs run at 3.0 GHz. L1 and L2 caches are private to each core and a 12MB L3 cache is shared among all cores. To avoid interference during measurement, we bind server and client processes on two different sets of cores.

Table 1. Software Versions

CentOS	6.5 x86-64
Linux Kernel	2.6.32-431.el6.x86_64
glibc	2.12.1
gcc	4.4.7
Intel® Pin®	x86 64 bit, Rev. 65163
Intel® VTune™ XE 2013	Update 17 (build 353306)

4.2 Software

The software configuration is shown in Table 1. We use Intel® VTune™ to measure L1D misses, L1I misses, DTLB misses, ITLB misses, and branch mispredictions.[1] Together, these measurements show how skipping library call trampolines affects the pressure on the various hardware components of the CPU. Performance counters are aggregated across all cores that run the processes under study.

4.3 Experimental Dynamic Linker

Intel® Pin® is used to collect library call information, identifying all library call sites and their corresponding trampoline targets after they are resolved during program execution. When a call instruction followed by an indirection branch instruction is observed, the pintool records the instruction addresses and the branch targets.

We modify the dynamic linker in glibc to support skipping library call trampolines. The modified dynamic linker uses the pintool output to determine the locations of the call sites and patches them directly in the text section. Every recorded "call and indirect branch" sequence is replaced with a call to the corresponding target. To achieve this goal, our modified linker removes application security restrictions by making the entire address space writable. To ensure consistent behavior across the various application runs (with and without the pintool, the base case and the experimental case), we disable address space randomization and use a custom allocator in glibc to load all libraries within the 32-bit reach of the patched call instructions in the original application and library binaries.

4.4 Workloads

We measure the performance of four applications in our experiment; three server applications and one desktop application. For server applications, we measure the Apache web server, Memcached, and MySQL. For the desktop application, we measure the Firefox web browser.

Apache web server We use Apache web server version 2.2.25 with the prefork MPM module. The PHP module is installed

1. The CPU has two performance counter registers. We use two counters per analysis run, one for the instruction count and one for the target metric. Multiplexing all counters within the same analysis run increases interference from VTune, especially when measuring DTLB misses.

as a shared library. To measure the server's performance, we use SPECweb 2009 1.20 [23] as the workload.

To measure performance, we instrument the Apache server to measure the latency of each request type from the arrival of the first packet of a request until the transmission of the last reply packet. To plot a smooth distribution curve of the measured latencies, we run the experiment for 10 hours at close to peak load. For every 10,000 requests, we observe 5 to 6 outlier measurements of significantly longer request latency caused by perturbations in the system (e.g., the performance counter interrupts). The outliers exist in both the base and experimental case and we omit them from the plots for clarity.

Memcached We use Memcached [6] version 1.4.15 with Libevent version 2.0.21. The client and dataset are taken from CloudSuite [4]. Parameters for Memcached are taken from the CloudSuite web site; we use a client with four threads and a target of 100,000 requests per second.

To measure performance, we instrument the Memcached server to measure the latency of the *get* and *set* requests. Due to the short execution duration of the requests, the timestamp counter register (RDTSC [10]) is used to collect timestamps without triggering system calls. We benchmark for one hour to get data for smooth and stable plots.

MySQL We use the TPC-C benchmark from OLTP-Bench [3] to measure the response time of the server for different request types. We present results only for the most popular requests types, *New Order* and *Payment*.

Firefox To measure desktop application performance, we use the official x86-64 Linux Firefox version 30.0 binary [5]. The Peacekeeper [18] browser benchmark suite runs an array of HTML5 and JavaScript performance tests that are representative of typical browser usage. We use the cumulative and individual scores for each test run to evaluate performance improvement with the proposed hardware.

5 Results

We evaluate the performance of an x86-64 processor running our workloads under a dynamic linker that emulates the proposed hardware by patching the binaries in memory to skip trampolines. Results show that our proposed hardware increases overall performance by eliminating the execution of expensive trampoline instructions and reducing pressure on the caches, TLBs, and branch predictor.

5.1 Opportunity

All programs use dynamically linked libraries; however, the extent to which each application depends on libraries is different. For our hardware approach to affect the performance of a program, the number of library function calls must be significant. In Table 2, we show the percentage of executed instructions that lie in a function trampoline. In a complex application like Apache, the number of library calls is much higher than in Memcached, which provides relatively simple functionality. The number of library calls in desktop applications such as Firefox is even lower, as execution is dominated by small computation kernels.

Table 2. Instructions in trampoline per kilo instruction

Workload	Trampoline Instructions (PKI)
Apache	12.23
Firefox	0.72
Memcached	1.75
MySQL	5.56

Table 3 shows the number of distinct trampolines accessed during the measurement period. Apache has a large number of trampolines, exercising many library calls across many libraries. Firefox uses an even larger number of distinct library calls, but they are exercised less frequently. On the other hand, even though Memcached has a higher frequency of library calls, there are only 33 distinct trampolines used, owing to the limited functionality of the server.

Table 3. Number of trampolines used by program execution

Workload	Distinct Trampoline Count
Apache	501
Firefox	2457
Memcached	33
MySQL	1611

We further break down the distinct trampolines by their relative execution frequency in Figure 4. The steep cutoffs visible for Apache and Memcached indicate that a very specific set of library calls was made for every request serviced, whereas the Firefox curve is much less steep, as the program accesses many different libraries to execute the diverse functionality exercised by the benchmark tests. For Memcached, the majority of library calls are made to fewer than 10 library functions.

5.2 Microarchitectural Benefits

In addition to the instructions executed within the library call trampolines, Section 2.2 discusses the additional microarchitectural penalties of dynamic linking. We use performance counters to measure the effect of our technique on the instruction cache, data cache, I-TLB, D-TLB, and branch prediction behavior. The results are presented in Table 4. As expected, the greatest impact on the microarchitecture is observed when running Apache, which makes the largest number of library calls to the largest number of library functions among the three workloads. Cache miss rates, TLB miss rates, and branch mispredictions drop across all workloads when trampolines are skipped. The improvements in Apache are so significant that the second-order performance impact of these microarchitectural

Table 4. Performance counters (values are per kilo instruction)

Performance Counter	Apache		Firefox		Memcached		MySQL	
	Base	Enhanced	Base	Enhanced	Base	Enhanced	Base	Enhanced
I-$ Misses	109.31	104.22	10.70	10.38	51.99	51.42	25.21	24.93
I-TLB Misses	1.78	1.18	0.87	0.79	0.03	0	2.41	2.36
D-$ Misses	7.96	7.56	2.66	2.67	12.25	12.16	8.48	8.46
D-TLB Misses	4.03	4.62	1.54	1.75	4.74	4.73	2.86	2.77
Branch Mispredictions	13.46	12.32	4.84	4.77	5.48	5.30	14.44	14.40

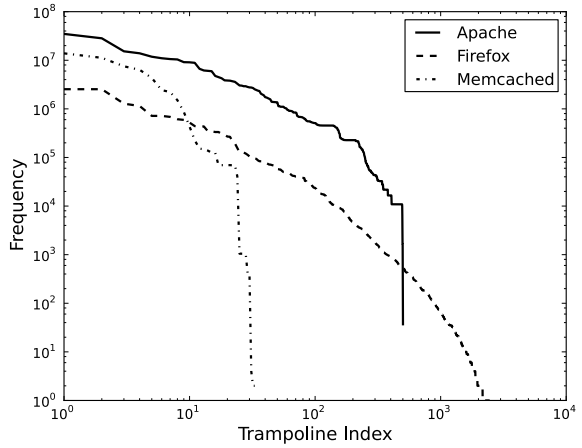

Figure 4. Frequency of trampolines.

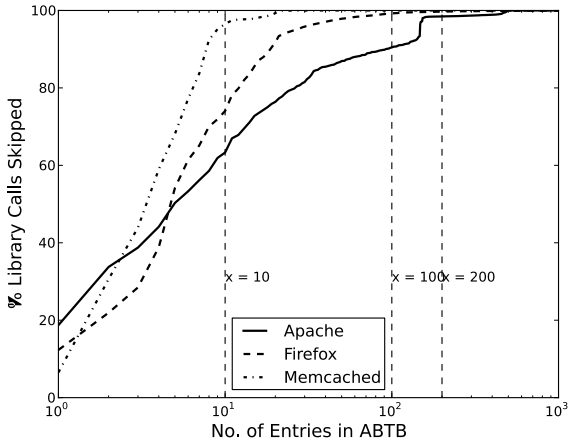

Figure 5. Percentage of library function call trampolines skipped for different sizes of ABTB.

improvements is actually greater than the first-order impact of skipping the trampoline instructions. In Memcached, although there are only a small number of distinct library function calls, skipping the trampolines is sufficient to eliminate all I-TLB conflict misses.

5.3 Hardware Cost

Every microarchitectural enhancement must be judged by the tradeoff between its benefit and its cost. Our experiments demonstrate the benefits of the proposed hardware support for dynamic linking, in terms of reducing the number of executed instructions and alleviating microarchitectural pressure. We now estimate the size of the ABTB needed to achieve these benefits.

Every entry in the ABTB consumes 12 bytes, six bytes for the call instruction's target and six bytes for the function address.[2] To estimate the required size of ABTB, we collect traces of our workloads and count the number of library call trampolines that can be skipped for different ABTB sizes. Figure 5 shows the distribution (number of entries in log scale) of the percentage of library call trampolines skipped for different sizes of the ABTB. We see that with just 16 entries (192 bytes), we can skip more than 75% of the trampolines in any of the three workloads. With a 256-entry ABTB, totaling less than 1.5KB, nearly all actively used trampolines can be skipped.[3]

Figure 5 provides an interesting additional insight into the instruction patterns of the workloads. A steep slope in the plot indicates that there is a repeating sequence of library functions that are called frequently. A small ABTB can skip all trampolines within that repeating sequence, as long as it is large enough to cover the full sequence. This implies the existence of ABTB "working sets" that can be used to reason about application behavior.

5.4 Performance

In Figure 6, we present the cumulative distribution function (CDF) of the percentage of requests served as a function of the response time for each request type in the SPECweb 2009 workload. We measure the response time of the server in microseconds; the CDF shows the fraction of requests served within the particular response time. When the Apache server runs without executing the trampolines (enhanced), the tail latencies are unaffected while the average response times are improved up to 4% over the base case.

2. x86-64 supports distinct virtual addresses up to 48 bits.

3. We do not consider additional savings made possible by offset encoding.

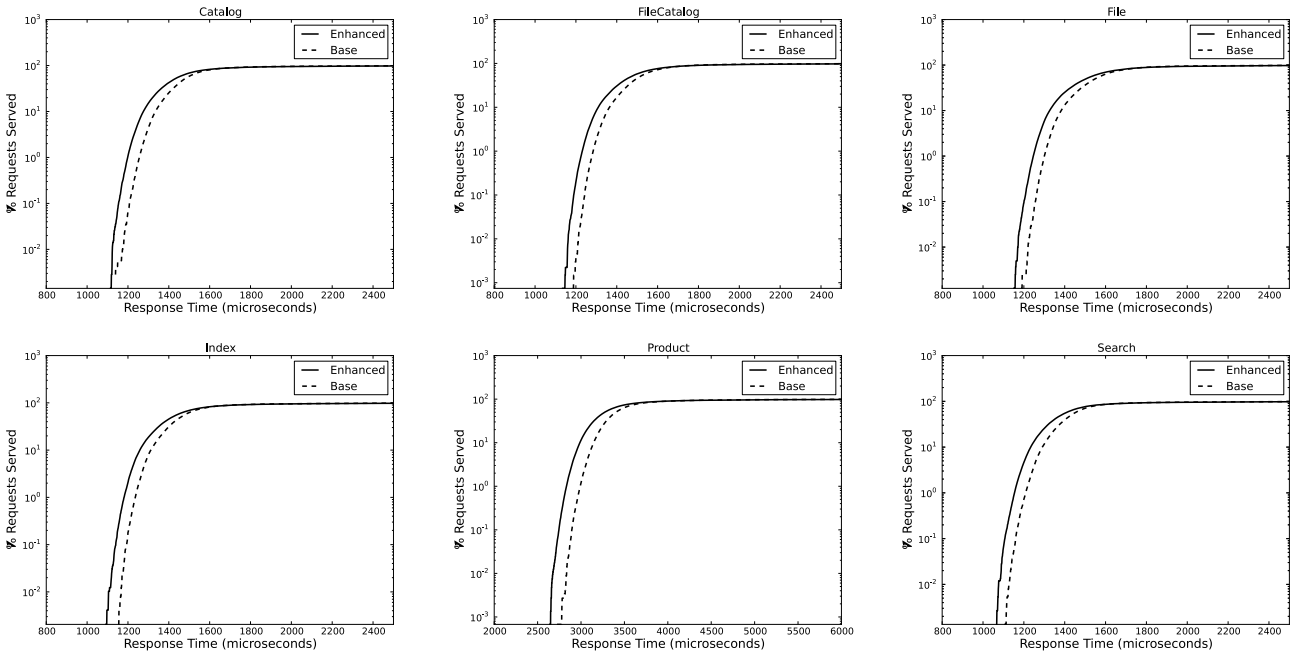

Figure 6. Cumulative distribution of the percentage of Apache requests served within the specified response time.

Table 5. Firefox Peacekeeper scores (higher is better)

Workload	Base	Enhanced
Rendering (fps)	49.31	50.64
HTML5 Canvas (fps)	37.47	37.94
Data (ops)	22,499	22,727
DOM operations (ops)	16,547	16,850
Text parsing (ops)	214,897	216,625

Table 5 presents Firefox web browser performance scores (higher is better). Performance improved for all workload categories included in the Peacekeeper benchmark. In rendering and DOM operations, the main tasks performed by web browsers in normal operation, we observed improvements of more than 2.7% and 1.8%, respectively. Text parsing performance improved by 0.8% due to the heavy use of string operations from shared libraries.

Figure 7 presents the performance improvement for Memcached under the proposed hardware technique. The two plots show histograms of request processing time for the Memcached GET and SET requests. Because the service experiences a wide range of request latencies, we plot the buckets within the largest peak, which accounts for the majority of requests (we omit other minor peaks for clarity, as they follow similar trends). For both request types, the peaks of the histograms for the enhanced version (skipping the trampolines) are shifted to the left, indicating an average reduction in request processing time.

Figure 8 shows the CDF of requests served vs. response time of the requests for MySQL. This data is summarized in Table 6, which demonstrates that more time is required to serve a given percentage of requests in the base system than in the enhanced system.

Table 6. Response Time of MySQL Requests in milliseconds (lower is better)

Requests	New Order		Payment	
	Base	Enhanced	Base	Enhanced
50%	43.5	43.0	17.9	17.7
75%	57.3	56.9	27.9	27.2
90%	72.8	72.3	37.2	35.9
95%	87.1	86.8	44.4	43.0

5.5 Memory Savings

Multithreaded server software shares code pages across threads, allowing a software approach to patch the call instructions in shared pages with target addresses (in our test suite, Firefox, Memcached, and MySQL fall into this category). However, many server applications use the *prefork* model to serve requests. For example, the main Apache web server spawns child processes to handle requests, but does not itself perform any request processing. A software approach to patching code pages for prefork software precludes the OS copy-on-write mechanism, increasing memory consumption. Our hardware solution has no memory overheads in the prefork case, allowing code pages to remain unmodified and shared across processes.

If call site patching is applied to all processes in a system, the patched code pages in all libraries would be copied to each running process, resulting in memory overhead that

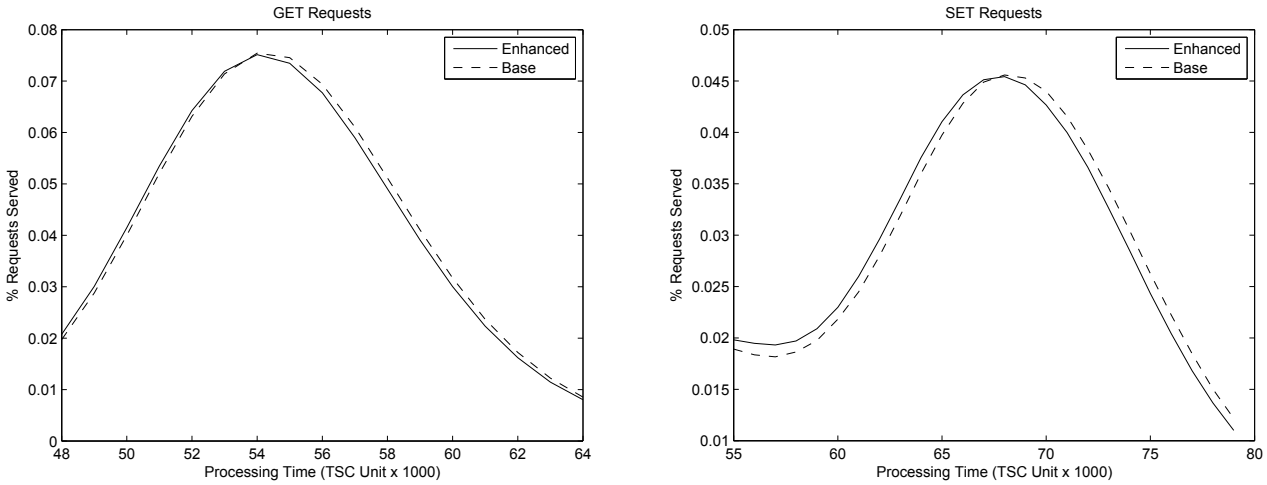

Figure 7. Histogram of Memcached requests served for a given request processing time.

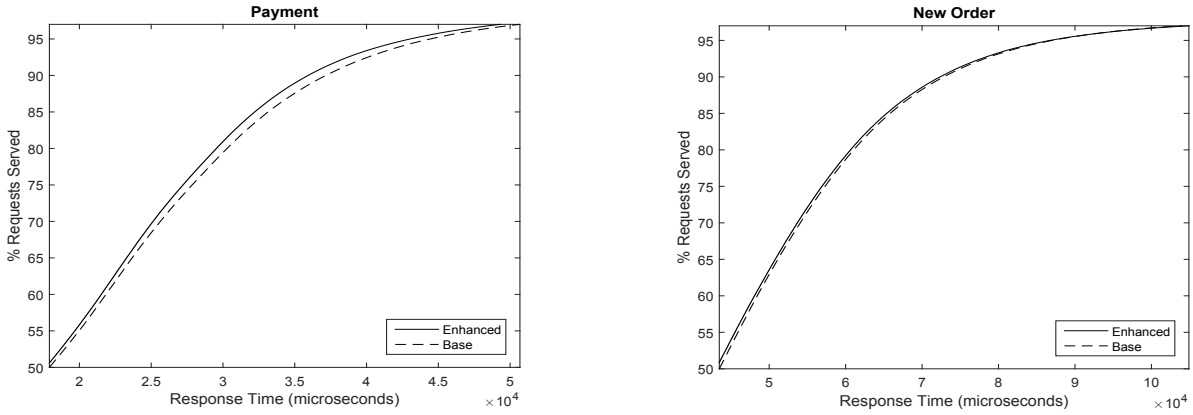

Figure 8. CDF of MySQL requests served within the specified response time.

can easily exceed multiple gigabytes. Even if patching is only applied to processes of the server application, significant overhead can still occur. For example, busy Apache servers have hundreds or even thousands of processes running concurrently. With dynamic patching (Apache binary, PHP, and supporting libraries) on the first invocation of each library call, approximately 280 code pages will be copied, resulting in 1.1MB of wasted memory for each process, or on the order of 0.5GB of RAM for a typical busy server.

6 Related Work

Sodani and Sohi proposed an instruction reuse technique to improve application performance by skipping the execution of instructions that repeat their work [22]. In their technique, a buffer stores the results of instructions indexed by the inputs of the instruction and its PC. During instruction fetch, inputs of an instruction are used in a lookup, skipping execution if a previous result is found. In a modern out-of-order processor, there can be more than 100 instructions in flight and the inputs of the fetched instruction are unavailable. Moreover, reading values from memory is expensive and is therefore not done by this technique. Thus, this approach cannot be used to skip dynamically linked library trampolines. Even if it is modified to consider memory inputs, simply fetching the trampoline instructions and performing the memory lookup would lose most of the microarchitectural benefits of the hardware we propose.

Huang and Lilja propose a hardware memoization technique that records the results of basic blocks and reuses entire blocks [8]. To store results, an entry needs to be large, supporting storage of multiple inputs and outputs. Library call trampolines are small basic blocks (one instruction on x86-64), making this technique excessively expensive for the benefits achieved. Moreover, both memoization techniques lie on the critical path of the instruction fetch and may impact the system's clock frequency [12], whereas the hardware support for dynamic linking that we propose operates primarily at retire time, off the critical path.

Kistler and Franz propose continuous optimization of programs using profile-guided optimization [14]. They propose load-time optimization to make use of as much infor-

mation about the system as possible. Profiling helps restructure code and data in memory to increase the likelihood of cache hits. This approach can benefit dynamic linking by collocating frequently used PLT and GOT entries close to each other to reduce their footprint in the caches and TLBs. However, it does not reduce the number of instructions executed or the number of branch predictor and target buffer entries used, as the trampoline instructions must still be fetched and executed.

7 Conclusions

Although static linking yields higher performance, dynamic linking has become the predominant choice for integrating libraries into today's complex applications, due to its myriad benefits, from flexibility in development and deployment, to memory conservation across processes and system security. The software community has embraced dynamic linking, largely ignoring the costs that arise from the use of trampolines that perform function pointer table lookups and execute additional indirect branches.

In this work, we showed that a simple hardware mechanism, working in tandem with existing branch predictor structures, can improve performance by speculatively skipping trampoline execution. The mechanism relies on the fact that the lookup result and indirect branch target of the trampoline do not change after their first invocation. We evaluated our proposed hardware technique by emulating the behavior of the hardware on a real system, demonstrating that using only 1.5KB of storage, a retire-time (off critical path) mechanism can eliminate the overheads of dynamic linking, reducing instruction cache, data cache, I-TLB, D-TLB, BTB, and branch predictor pressure to achieve performance improvements of up to 4%.

8 References

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[2] Willem De Groef, Nick Nikiforakis, Yves Younan, and Frank Piessens. Jitsec: Just-in-time security for code injection attacks. In *Benelux Workshop on Information and System Security (WISSEC)*, 2010.

[3] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[4] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012. recognized as Best Paper by the program committee and recognized as Top Pick of 2013 by IEEE Micro.

[5] Firefox. https://www.mozilla.org/en-US/firefox/new/.

[6] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[7] Michael Franz. Dynamic linking of software components. *Computer*, 30(3):74–81, March 1997.

[8] Jian Huang and David Lilja. Exploiting basic block value locality with block reuse. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, pages 106–, Washington, DC, USA, 1999. IEEE Computer Society.

[9] Intel Corporation. *extregistered 64 and IA-32 Architectures Optimization Reference Manual*. Intel, March 2009.

[10] Intel Corporation. *extregistered 64 and IA-32 Architectures Software Developer's Manual*. Intel, December 2009.

[11] Intel Xeon Processor E5450 (12M Cache, 3.00 GHz, 1333 MHz FSB). http://ark.intel.com/products/33083/Intel-Xeon-Processor-E5450-12M-Cache-3_00-GHz-1333-MHz-FSB.

[12] Daniel A. Jiménez. Reconsidering complex branch predictors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 43–, Washington, DC, USA, 2003. IEEE Computer Society.

[13] Hyesoon Kim, José A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. Vpc prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 424–435, New York, NY, USA, 2007. ACM.

[14] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, July 2003.

[15] Pierre Michaud, AndréSeznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 292–303, New York, NY, USA, 1997. ACM.

[16] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherence. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, pages 230–242, New York, NY, USA, 1989. ACM.

[17] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.

[18] Peacekeeper - The universal Browser Test. http://peacekeeper.futuremark.com/.

[19] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304, New York, NY, USA, 2011. ACM.

[20] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V Adve, and Luiz Andre Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[21] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[22] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 194–205, New York, NY, USA, 1997. ACM.

[23] SPEC - Standard Performance Evaluation Corporation. http://www.spec.org/.

[24] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA, 1991. ACM.