Architecture Support for Improving Bulk Memory Copying and Initialization Performance

Conference Paper *in* Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT · September 2009

DOI: 10.1109/PACT.2009.31 · Source: DBLP

4 authors:

Xiaowei Jiang
Intel
20 PUBLICATIONS **562** CITATIONS
SEE PROFILE

Yan Solihin
University of Central Florida
124 PUBLICATIONS **3,640** CITATIONS
SEE PROFILE

Li Zhao
Intel
61 PUBLICATIONS **1,269** CITATIONS
SEE PROFILE

Ravi R Iyer
Intel
141 PUBLICATIONS **3,577** CITATIONS
SEE PROFILE

Architecture Support for Improving Bulk Memory Copying and Initialization Performance

Xiaowei Jiang, Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
Raleigh, USA
{*xjiang,solihin*}*@ncsu.edu*

Li Zhao, Ravishankar Iyer
Intel Labs
Intel Corporation
Hillsboro, USA
{*li.zhao, ravishankar.iyer*}*@intel.com*

***Abstract*—Bulk memory copying and initialization is one of the most ubiquitous operations performed in current computer systems by both user applications and Operating Systems. While many current systems rely on a loop of loads and stores, there are proposals to introduce a single instruction to perform bulk memory copying. While such an instruction can improve performance due to generating fewer TLB and cache accesses, and requiring fewer pipeline resources, in this paper we show that the key to significantly improving the performance is removing pipeline and cache bottlenecks of the code that follows the instructions. We show that the bottlenecks arise due to (1) the pipeline clogged by the copying instruction, (2) lengthened critical path due to dependent instructions stalling while waiting for the copying to complete, and (3) the inability to specify (separately) the cacheability of the source and destination regions. We propose FastBCI, an architecture support that achieves the granularity efficiency of a bulk copying/initialization instruction, but without its pipeline and cache bottlenecks. When applied to OS kernel buffer management, we show that on average FastBCI achieves anywhere between 23% to 32% speedup ratios, which is roughly $3\times$–$4\times$ of an alternative scheme, and $1.5\times$–$2\times$ of a highly optimistic DMA with zero setup and interrupt overheads.**

***Keywords*-memory copying; memory initialization; cache affinity; cache neutral; early retirement**

I. Introduction

Bulk (large-region) memory copying and initialization is one of the most ubiquitous operations performed in current computer systems by both user applications and Operating Systems (OSes). Critical OS functions such as buffer management and virtual memory management rely heavily on memory copying and initialization. For example, kernel buffer copying (through calling `memcpy`) and user-kernel buffer copying (through calling `copy_from_user` and `copy_to_user` [18]) are performed extensively to support TCP/IP processing and file I/O operations. Many TCP/IP intensive applications (e.g. Apache web server) indirectly spend a significant portion of execution time performing memory copying. A typical OS also performs bulk initialization and copying to support various virtual memory management functions such as page allocation, copy-on-write, and swapping pages in and out of the physical memory. User applications also perform bulk memory copying and initialization for buffer management and string handling, typically through calling `memcpy` and `memset` [9].

In current systems, there are at least three ways memory copying is supported at the instruction level (Figure 1): *explicit loop* – using a loop of loads and stores, *implicit loop* – using an instruction that is expanded into a series of loads and stores at instruction decode time (e.g., `rep movsd` in x86 [12], [14] and `mvcl` in IBM S/390 [1]), and *vector extension*– using vector load and store instructions.

Explicit Loop

PowerPC:

```
loop: lwz  R1, 0(R2)
      addi R2, R2, 4
      stw  R1, 0(R3)
      addi R3, R3, 4
      bdnz loop
```

Vector Extension

SSE2:

```
loop: movaps xmm0,0(esi)
      add    esi, $16
      movaps 0(edi),xmm0
      add    edi,$16
      sub    ecx,$16
      jnz loop
```

Implicit Loop

IBM S/390:

```
la    R2, dst_addr
la    R3, dst_len
la    R4, src_addr
la    R5, src_len
mvcl  R2, R4
```

x86:

```
mov esi, src_addr
mov edi, dst_addr
mov ecx, len
rep  movsd
```

Figure 1. `memcpy` implementation in various instruction sets. For PowerPC (SSE2), the source address is in R2 (esi), the destination address is in R3 (edi), and the region length is in the count register (ecx).

Comparing the three schemes, it is clear that the explicit loop and implicit loop implementations suffer from a *granularity inefficiency* problem, in that an excessive number of TLB accesses and cache accesses occur during copying. Instead of checking the TLB once for each page involved, each load and store instruction incurs a TLB access. Similarly, rather than accessing the cache once for each cache block involved, each load and store instruction incurs a cache access. In this regard, vector instructions

This work was supported in part by NSF Award CCF-0347425 and by gifts from Intel. The authors would like to thank anonymous reviewers and Abhik Sarkar for their feedback.

achieve a better (but not perfect) granularity efficiency since each load or store can work on data as wide as the width of the vector registers, e.g. 128 bits in SSE2. Future vector instructions may increase the width even more (e.g. Intel Larrabee will likely support 512-bit loads/stores [22]).

However, an efficient memory copying and initialization requires more than just granularity efficiency. In fact, surprisingly, we found that granularity efficiency alone does not improve the performance of memory copying by much. The overall performance of memory copying is *impacted by how the code that follows the copying performs, more than by the efficiency of the memory copying operation itself*. This is illustrated in Figure 2, which shows limit speedups of various copying-intensive benchmarks such as Apache web server (sender and receiver side), iperf network benchmarking tool (sender and receiver side), and iozone I/O performance tool, and their average. The "perfect granularity" bars show speedups when each kernel buffer copying function is replaced with a single copying instruction, assuming such an instruction can handle buffers with an arbitrary width and alignment. Furthermore, in the perfect granularity, we assume that the TLB and caches are only accessed as few times as possible, i.e. the TLB is checked once for each page involved while the cache is accessed once for each cache block involved. The "perfect pipeline" bars show limit speedups when the memory copying instructions do not restrict younger instructions from executing and retiring, except when they are data-dependent on the copying instruction. Finally, the "perfect cache" bars show limit speedups when the code that follows memory copying operations does not suffer from cache misses when it accesses the copying regions, and does not suffer from extra cache misses when it accesses non-copying regions. "Perfect cache" represents the case in which memory copying produces all the useful prefetching effect for the code that follows the copying but none of the harmful cache pollution effects.

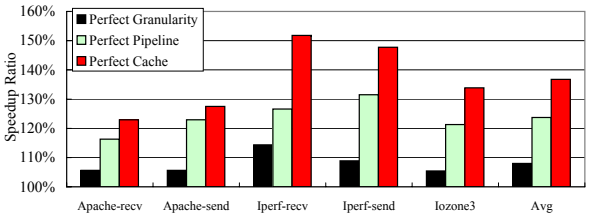


Figure 2. Limit speedups of applications with bulk copying engine, assuming perfect instruction granularity, perfect pipeline, and perfect cache. Details of applications and machine parameters can be found in Section V.

Interestingly, the figure shows that perfect granularity alone contributes to only 8% speedup on average. However, when the copying instructions do not restrict the execution of younger, non-dependent instructions, the average speedup jumps threefold to 24%. When the copying instructions do not incur cache pollution and introduce a beneficial prefetching effect to younger instructions, the average speedup jumps almost fivefold to 37%. The figure illustrates that while granularity efficiency is important, the *real key to significantly improving the performance of copying/initialization is removing pipeline and cache bottlenecks of the code that follows copying operations*.

Contributions. This paper seeks to provide an instruction and architecture support for Fast Bulk Memory Copying and Initialization (FastBCI). FastBCI consists of a scalar instruction that can perform copying/initialization involving a large memory region. Unlike an implicit loop approach, the instruction is not expanded into loads and stores. Rather, it is executed on an engine that performs the copying/initialization and keeps track of its progress.

Furthermore, we found two pipeline bottlenecks that severely restrict the potential performance of a copying instruction. The first is that copying/initializing a large memory region takes a long time, and as a result, the copying instruction stalls at the head of the reorder buffer (ROB), eventually stalling the entire pipeline. To get around this, we design a mechanism that allows the copying instruction to retire early prior to completion, but safely. Moreover, we also discuss correctness criteria needed for retiring copying instructions early, and present one efficient implementation.

The second pipeline bottleneck we discovered arises from a situation in which dependent instructions often stall waiting for the completion of a copying instruction, and they increase the critical path of instruction execution. Clearly, the larger the copying region is, the longer it takes for the copying instruction to complete, and the more likely some dependent instructions stall in the pipeline. To avoid such stalls, we propose a mechanism to track which blocks have completed their copying, and allow dependent instructions to wake up. As a result, as copying progresses, more younger instructions avoid stalls and wake up, execute, and retire.

Figure 2 also illustrates the importance of the cache effect of memory copying on the code following the copying operations. To address this issue, we allow cache affinity options to be specified as a part of the copying instruction parameters. We allow three option values that can be applied individually to the source and destination copying regions: *cacheable* (the region is brought into the cache as it is accessed), *non-cacheable* (the region is not brought into the cache and is evicted if all or parts of it are cached), and *cache neutral* (no new blocks from that region are brought into the cache but existing blocks are retained in the cache).

Overall, this paper makes the following contributions:

- It shows that the key to improve the performance of copying/initialization is solving the pipeline and cache bottlenecks of the code that follows copying operations.
- It analyzes the microarchitecture and multiprocessor correctness requirements for mechanisms that avoid the pipeline bottlenecks through allowing a copying instruction to retire early.
- It proposes an architecture support for efficient bulk copying and initialization, achieving granularity effi-

ciency while avoiding the pipeline and cache bottlenecks. To evaluate its performance, we use a cycle-accurate full system simulator with an out-of-order processor and two-level caches, running Linux OS. For the benchmarks we evaluated, on average FastBCI achieves a 23.2% speedup with pipeline improvement, very close to the perfect pipeline case.

- It proposes and evaluates cache affinity options including cacheable, non-cacheable, and cache neutral, as parameters of copying instructions. It shows that cache affinity options affect performance tremendously, with different regions favoring different affinity options. On average, the best affinity option outperforms the worst affinity option by 10%. Interestingly, we found that the cache neutral option gives the most robust performance for the benchmarks that we evaluate.

The rest of the paper is organized as follows: Section II discusses related work, Section III describes architecture support for FastBCI, Section IV describes its microarchitecture design, Section V describes the evaluation environment, while Section VI discusses evaluation results. Finally, Section VII summarizes our findings.

II. Related Work

DMA and memory-side copy engine. An alternative way to accelerate memory copying is to use DMA. Traditional DMA engines (e.g. PCI DMA [17]) allow copying between device and memory but not from and to memory. More recent DMA engines support bulk data movement between two memory regions (e.g. Intel I/OAT [3]). However, such a DMA engine still requires (1) a long setup latency, (2) an interrupt-based completion, which requires an expensive pipeline flush and interrupt handling, and (3) OS involvement. Thus, from performance stand point, such overheads cannot be easily amortized if the copying region is relatively small, e.g. a few kilobytes or less. Moreover, there are other practical limitations such as requiring physical addresses.

Some recent DMAs support bulk data movement between the main memory and on-chip scratchpad memory (e.g. Cell processor [23] and others [19]). Cell DMA can operate at the user level and does not require a long setup latency. However, it requires scratchpad memory, which is not available in many general purpose processors. In addition, it still relies on interrupt notification on transfer completion, has no fine-grain dependence checking (incurring pipeline bottleneck), and is asynchronous. We compare FastBCI's performance with zero setup-latency DMA in Section VI.

Zhao et al. [21], [20] proposed an instruction support and hardware engine for performing memory copying. The engine includes an off-chip memory-side component called Copy Engine (CE) which performs copying, and an on-chip component called Copy Control Unit (CCU). When a copying instruction is encountered, CCU accepts virtual addresses, translates them into physical addresses, and sends them off to the CE. The CE acquires ownership of the copying regions by sending invalidations to caches, and then performs the copying by *directly* reading from and writing to the main memory. While copying is in progress, any loads or stores made by the processor to such addresses are stalled until the copying is fully completed. In contrast to FastBCI, CE does not address pipeline and cache bottlenecks for code that follows copying operations, which Figure 2 points out as the key for improving copying performance. For example, CE stalls all dependent loads/stores until the copying is fully completed. Being memory-side, CE also always requires caches to flush blocks in the copying region prior to the start of copying. Such a policy negates the potential beneficial effect of bringing blocks in the copying region into the cache, and cause extra cache misses when the following code accesses the copying regions in the future. In contrast, FastBCI deals with both pipeline and cache bottlenecks by keeping track of which cache blocks have completed their copying, allowing dependent loads/stores to proceed even when copying for other blocks has not completed (*non-blocking* policy), as well as supporting three cache affinity options. As a result of these differences, our evaluation (Section VI) shows that FastBCI achieves roughly three times the speedups achieved by CE (23.2% vs. 7.9% on average) across benchmarks we tested.

Cache Affinity Flexibility. In current systems, while copying instructions lack such flexibility, instructions that control cache affinity are already provided. For example, Power and Intel architectures [2], [14] have instructions to flush or prefetch specific cache blocks. IA64 [7] load/store instructions can be augmented with cache hints that specify the cache level into which data should be fetched. Load/store instructions in SSE4 [4] (e.g. `movntdq` and `movntdqa`) allow non-cacheable loads and stores using a streaming load and a write-combining buffer. The fact that various instruction-level supports are provided in several processors highlights the feasibility and importance of such flexibility on performance. In FastBCI, cache affinity flexibility is provided for both the source and destination regions, and each region can be specified individually as cacheable, non-cacheable, or cache neutral.

Cache affinity flexibility may also be provided beyond the instruction level. For example, direct cache access (DCA) is an architecture that allows the Network Interface Card (NIC) to directly place inbound packets in the processor's cache [26]. Finally, there are also techniques to accelerate TCP/IP processing (such as on-chip network interface [24] and Remote DMA [11]). Such architectures and techniques have been shown to accelerate TCP/IP processing. Compared to them, FastBCI is a more general support in that it deals more with accelerating memory copying and initialization rather than TCP/IP processing only. However, FastBCI can also be used to accelerate TCP/IP processing.

Instruction-Support for Bulk Memory Copying. As men-

tioned briefly in Section I, in current architectures, instructions that can operate on wide data are typically vector or vector-extension instructions [4], [10], [16]. Vector loads/stores can work on data as wide as the width of the vector registers. However, vector loads/stores require data to be aligned at the respective width [15], making them unsuitable for use in scalar data which may not be aligned at vector width.

It is likely that instructions that can perform a large-region copying have been implemented in some CISC (Complex Instruction Set) architectures. Our FastBCI instructions can be thought of as an example of such CISC-flavored instructions. However, we showed in Figure 2 that such instructions do not improve performance much unless pipeline and cache bottlenecks of code that follows are removed. To our knowledge, FastBCI is the first to propose mechanisms to remove such bottlenecks for bulk copying instructions.

III. Architecture Support for Bulk Copying/ Initialization

A. Overview of Bulk Copying and Initialization Instructions

To achieve maximum granularity efficiency, ideally there should be a single instruction to copy or initialize a large memory region. Such instructions can be one of instructions already used in current architectures (e.g., `mvcl` or `rep movsd` in Figure 1), or be new ones. For ease of discussion, but orthogonal to the design of FastBCI, we assume the following three-register operand instructions are available:

```
BLKCPY  Reg_SRC, Reg_DEST, Reg_LEN
BLKINIT Reg_INITVAL, Reg_DEST, Reg_LEN
```

where BLKCPY copies `Reg_LEN` number of bytes from a region that starts at address `Reg_SRC` to a region that starts at address `Reg_DEST`, and BLKINIT initializes `Reg_LEN` number of bytes in a region that starts at address `Reg_DEST` using an initialization string value stored in `Reg_INITVAL`.

In actual implementations, the instruction format may differ depending on the constraints of a particular ISA. The source and destination regions in copying may or may not overlap, but for our discussion we assume they are not overlapped. Furthermore, BLKINIT can be thought of as a special case of BLKCPY where the source region is a register value rather than a memory address. Architecture support for a bulk copying instruction can also be used to support a bulk initialization instruction. Hence, from this point on, *our discussion will center around memory copying*.

In traditional implicit loop implementations, the copying instruction is fetched and decoded into a loop of loads and stores. Each load and store then incurs a TLB access to check the validity of the page and obtains its physical address, and incurs a cache access to read value into a register or write from a register. Such an approach has poor granularity efficiency. For example, if a load/store granularity is 8 byte, for each 4KB-sized page copied, we would incur $2 \times \frac{4096}{8} = 1024$ TLB and cache accesses. If instead, the copying instruction is executed in a special functional unit (an "engine"), we can improve the granularity efficiency by checking the TLB only once for each page involved, and performing the copying at the granularity of cache blocks. With the latter approach, copying a 4KB region only incurs 2 TLB accesses for the source and destination pages (a reduction of 99.8%) and 128 cache accesses assuming 64-byte cache block size (a reduction of 87.5%) [1]. Hence, FastBCI adopts the latter approach.

B. Cache Bottlenecks

As shown in Figure 2, one of the keys to improving memory copying performance is solving the cache performance bottlenecks of the code that follows memory copying operations. The huge performance improvement in the "perfect cache" is by assuming that copying produces beneficial prefetching effect (i.e. later accesses to the source and destination regions result in cache hits), but none of the cache pollution effect (i.e. later accesses to blocks that were in the cache prior to copying result in cache hits). While "perfect cache" is optimistic, it highlights the upperbound performance improvement of maximizing the prefetching effect while minimizing the cache pollution effect.

Whether the prefetching or pollution effect is stronger depends on the temporal reuse patterns of the application that uses copying. Caching new blocks from the copying regions will cause cache pollution if the processor needs them less urgently than the blocks they replace, but will cause a helpful prefetching effect if the processor needs them more urgently than blocks they replace. For example, in a network application in which an outgoing packet is copied from a buffer to the network interface memory-mapped buffer, the destination region will not be accessed by the processor, hence it is likely beneficial to keep the region uncached. But if an incoming packet is copied from the memory-mapped buffer to a destination buffer, the destination region will soon be accessed by the processor, and hence it is likely beneficial to fetch it into the cache during copying.

Considering this, we propose to augment the bulk copying instruction with parameters that can specify *separately* the cache affinity policy for the source and destination regions. In FastBCI, the affinity policy specifies whether a region should be *cacheable* (the region is brought into the cache as it is accessed), *non-cacheable* (the region is not brought into the cache and is evicted if all or parts of it are cached), or *cache neutral* (no new blocks from that region are brought into the cache but existing blocks are retained in the cache). A cacheable affinity intends to allow an application to maximize the prefetching effect of copying, whereas a non-cacheable affinity intends to allow an application to minimize the cache pollution effect of copying. A cache

[1] The cache data path width of a processor determines the actual reduction in the number of cache accesses.

neutral affinity achieves a balance in the sense that it does not produce any prefetching or pollution effect. The affinity options can be embedded as parameters into the copying instruction using unused operands or opcode bits.

There are several factors that determine which affinity option will perform best for a particular pair of source or destination regions. One factor is the likelihood of the region to be needed by the processor right after copying is completed. The higher the likelihood, the more attractive the cacheable option becomes. Another factor is the temporal locality of data that was already cached prior to copying. The higher the temporal locality, the more cache pollution it has if a copying region is fetched into the cache. Another important factor is the cache capacity compared to the working set of the application. A larger capacity can tolerate cache pollution better, and increase the likelihood that a cache-allocated region remains in the cache when the processor accesses it. Due to the combination and interaction of these factors, we find that typically, the source and destination regions require different affinity options to achieve the best performance, which argues for the need to provide flexible and separate affinity options for source and destination regions. In contrast, current approaches do not offer cache affinity options. The loop approach always allocates both regions in the cache. DMA or Copy Engine [21], on the other hand, always makes both regions non-cacheable since DMA and Copy Engine are memory side devices. Our evaluation results (Section VI) demonstrate that using the same cacheable or non-cacheable affinity option for both the source and destination regions usually result in suboptimal performance (they yield 7% and 10% lower application speedups compared to the optimal options, respectively).

Note that the affinity flexibility also eases the interaction with the I/O system. If copying involves a memory-mapped I/O region, specifying a *non-cacheable* affinity helps in ensuring that the region is not cached at the end of copying, and avoids cache coherence problems with I/O devices.

To support the non-cacheable and cache neutral options for the copying region, the cache controller needs to be modified slightly. For a non-cacheable source region, if a block is already cached, it is read and then evicted from the cache. If it is not found in the cache, it is read from memory but not allocated in the cache. For a cache neutral source region, if a block is already cached, it is left where it is. If it is not found in the cache, it is fetched and read but not placed in the cache. For a non-cacheable destination region, if a block is already cached, it is overwritten and immediately evicted and written back. If it is not already cached, an entry in the write combining buffer can be created with the copied value, so bandwidth is not wasted for fetching the block. For a cache neutral destination region, if a block is already cached, it is overwritten. If it is not already cached, an entry in the write-combining buffer is created containing the copied value. It is possible to let the affinity policy be applied differently to different cache levels, but in this paper we assume it is applied to all cache levels.

C. Pipeline Bottlenecks

Let us now investigate the pipeline performance of bulk copying instructions. When the instruction is encountered in the pipeline, let us assume it is sent off to a special‘functional unit” (or engine) that performs the actual copying. The granularity efficiency of the instruction allows the copying to be performed without occupying much pipeline resources. Unfortunately, while the copying instruction itself is executed efficiently, the code that follows copying may not benefit much from it. Figure 3(a) and (b) illustrate this problem using a bulk initialization instruction. In Figure 3(a), the bulk initialization instruction validates a page it wants to write to, then performs writes to each block. Some writes may take longer than others due to cache misses (shown as longer lines). Unfortunately, the instruction may take a long time to complete due to writing to a large number of blocks, and hence it stays in the ROB of the processor for a long time. Since instruction retirement is in order, younger instructions cannot retire either, and eventually the pipeline is full, at which point no new instructions can be fetched. We refer to this bottleneck as the *ROB-clog* bottleneck.

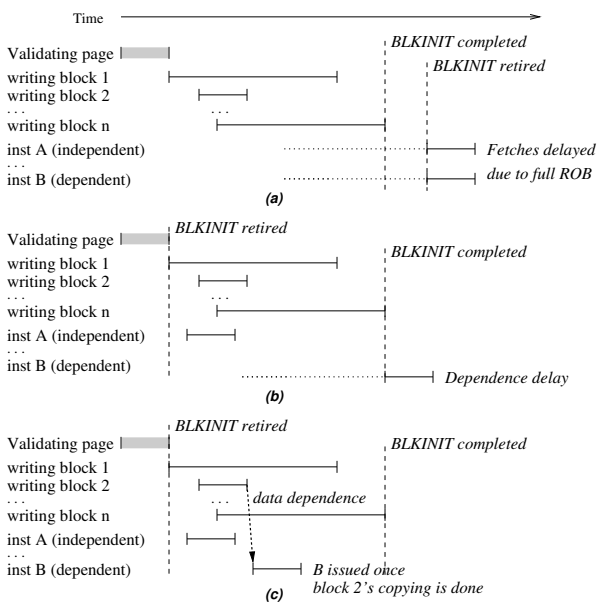


Figure 3. The effect of ROB-clog (a) and dependence (b) bottlenecks on pipeline performance. Execution without bottlenecks (c).

The pipeline bottleneck described above can be avoided if we allow the bulk copying instruction to retire early, e.g. right after page validation is completed (Figure 3(b)). Assuming such an early retirement is safe to do (a big if for now), another pipeline bottleneck may occur. A younger instruction (instruction B in the figure) stalls in the pipeline waiting for the initialization completion. The stall can again propagate and increases the critical path of execution, or

even clog the pipeline. We refer to this bottleneck as the *dependence* bottleneck.

If the above two bottlenecks are avoided, then instructions can flow through the pipeline without being blocked by the copying instruction or by instructions that depend on the copying instruction. In Figure 3(c), instruction B, which depends on the completion of initialization of block 2, can wake up and execute once block 2 is initialized. This improves performance as instruction B will no longer stall for a long time in the pipeline.

To get an idea of the importance of the two pipeline bottlenecks, Figure 4 shows the breakdown of cycle per instructions (CPI) due to processor execution and regular stalls such as structural, data dependence, branch misprediction (bottom section), and due to copying-specific stalls: ROB clog bottleneck (middle section), and dependence bottleneck (top section). The figure shows that on average, 12.2% of the CPI stall is due to copying-specific stalls (7.8% from ROB clog bottleneck and 4.4% from dependence bottleneck). To understand why copying-specific CPI stalls are high, we find that for the tested applications, a copying on average takes 9,086 cycles to complete. A high-performance processor pipeline can support a few hundred in-flight instructions, which is only enough to keep the processor busy for a few hundred cycles, before stalling for the remaining thousands of cycles. In addition, the average distance between a copying instruction to the first dependent instruction is only 16 instructions. Thus, even if we solve the ROB clog bottleneck, stalls due to dependence delay occurs soon after.

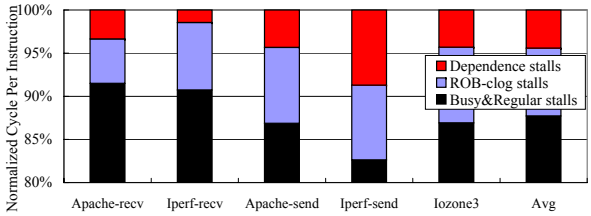


Figure 4. Cycle Per Instruction (CPI) breakdown due to regular processor execution&stalls, and stalls due to ROB clog and dependence bottlenecks. Details of applications and machine parameters can be found in Section V.

1) Dealing with Pipeline Bottlenecks: Comparing the ROB-clog bottleneck and the dependence bottleneck, the dependence bottleneck can be dealt with more easily because it involves dependence checking mechanisms rather than instruction retirement mechanisms. However, solving the dependence bottleneck without solving the ROB-clog bottleneck unlikely improves performance much because the ROB is more likely to be clogged by the copying instruction before it is clogged by (younger) dependent instructions. Hence, first we must solve the ROB clog bottleneck before we can solve the dependence bottleneck.

There are several ways to remove the ROB-clog bottleneck. The first option is to remove the guarantee that instruction retirement implies the completion of the instruction execution, i.e. by making the bulk copying instruction *asynchronous*. This allows the instruction to be retired ahead of its completion, freeing the ROB. However, using asynchronous copying instruction complicates programming because programmers must explicitly test the completion of the copying instruction (e.g., by polling). Moreover, polling for completion introduces overheads, and these overheads preclude a cost-effective solution for avoiding the dependence bottleneck problem since it requires fine-grain dependence tracking.

Another possible solution to allow the copying instruction to be retired prior to its completion is by treating the retirement as speculative, and using a checkpoint and rollback mechanism to rollback and re-execute it when a dependent instruction uses data that has not been produced by the copying instruction. Unfortunately, a checkpoint and rollback mechanism is quite expensive to support. In addition, if the distance between a copying instruction and dependent instructions is short (on average, only 16 instructions apart), this will cause frequent and persistent rollbacks.

With FastBCI, we adopt a new, non-speculative approach. The key to our approach is while we allow the copying instruction to retire ahead of its completion, we provide an *illusion of completion* to other instructions. There are several requirements to providing an illusion of completion:

1) Exception completeness requirement: any exception that a copying instruction may raise must have been handled prior to retiring the instruction, and no conditions that may raise a new exception can be permitted.
2) Consistency model requirement: a retired copying instruction must be consistent with load/store ordering supported by the architecture.

In addition to the two requirements needed for providing an illusion of completion, we also employ an additional *non-speculative* requirement that simplifies the implementation of copying: the effect of copying should not reach the memory system before the copying instruction is verified to be non-speculative, i.e. after exceptions and branch mispredictions in older instructions have been handled.

2) Non-Speculative Requirement: The non-speculative requirement deals with when the copying instruction should be allowed to start its execution. One alternative is to let it start right away as soon as its operands are ready. However, this means that we must provide an ability to roll it back if it turns out to be speculative, e.g. it lies in the wrong branch path. Supporting such a rollback capability is difficult since the instruction may write to a large number of cache blocks and these writes must be either buffered or canceled.

Hence, FastBCI adopts a simpler alternative that avoids the reliance on roll back altogether: it allows the Bulk Copying and Initialization Engine (BCIE) to be programmed, but it holds off the BCIE from performing the actual copying until the copying instruction has been verified to be non-speculative. This means all older instructions must have been

verified exception free and have their branch mispredictions handled. To achieve that, we *conservatively* wait until the copying instruction reaches the ROB head before firing off the BCIE. While this may introduce some delay in starting the copying, a delay of up to tens of cycles is insignificant compared to the performance gain obtained by FastBCI.

3) Exception Completeness: It is straightforward to observe that without exception completeness and ordering, the illusion of completion for a copying instruction is broken. For example, an exception raised by a copying instruction after it retires leaves the system in an inconsistent state, i.e. it appears as if the copying instruction has not completed or has failed, but some of its younger instructions have retired.

To provide the illusion of completion, FastBCI allows a copying instruction to be retired only after it is verified to be exception-free or all its exceptions have been handled. When a copying instruction is executed, the Bulk Copy and Initialization Engine (BCIE) is simply programmed with the required parameters, but is not fired off. After the instruction is verified to be exception free, or its exceptions have been completely handled, then the instruction can be retired and the actual copying can start (the BCIE is fired off). Hence, the effect of copying does not reach the memory system until after the instruction can no longer raise an exception.

Whether a copying instruction will raise an exception or not can be verified by checking the TLB. If the translation exists and page permission allows the respective operations (read permission for the source region and write permission for the destination region), then the copying instruction will not generate an exception. Note, however, that both the page permission and the physical address of pages in the source and destination regions must remain valid for the entire copying operation. This requires the pages to be locked to prevent the OS from swapping the pages out to disk, or perform an operation that changes the permission of the pages, until the completion of copying.

Due to the requirements to handle all exceptions prior to retiring a copying instruction, and locking pages until the completion of copying, the implementation can be made easier if the maximum region size is limited. In FastBCI, we limit the maximum copying region size to be the smallest page size, i.e. 4KB. Hence, page validation incurs at most four TLB misses (two pages for each of the source and destination regions if they are not page-aligned). In addition, the granularity efficiency loss compared to an unbounded copying is negligible since most of the granularity efficiency gain is obtained going from word/double-word-sized load/store to a page-sized copying granularity.

Finally, there are events that can potentially break the illusion of completion, such as interrupts, context switches, thread spawning, etc. This requires the processor to arrive at a consistent state because the interrupt handling or the new thread may read or modify data in the copying regions. If any of these events occurs when a copying instruction is already retired but not completed, the event handling is delayed until copying is fully completed.

4) Illusion of Completion in Consistency Models: The illusion of completion for a copying instruction that is retired early also applies in the context of a multiprocessor system. Let us consider both the cache coherence issue as well as the memory consistency model.

The engine that performs copying (BCIE) communicates with cache controllers to send read/write requests for each block involved. Coherence requests are generated by cache controllers in response to the requests, as in a regular multiprocessor system. Therefore, cache coherence is handled properly by existing coherence protocols without changes.

FastBCI allows copying to different blocks to proceed out of order and allows younger loads/stores to issue before an older copying instruction fully completes. Therefore, FastBCI fits naturally with relaxed consistency models such as *weak ordering* and *release consistency*. In these models, it is assumed that programs are properly synchronized, and ordering among loads and stores are not enforced except at synchronizations or memory fences. A copying instruction can be thought of loads and stores surrounded by synchronizations/fences. Therefore, the copying instruction is prevented from being issued when an older synchronization/fence has not performed. In addition, a younger synchronization/fence is prevented from being issued until all older copying instructions have fully performed. Such a restriction conforms to these consistency models.

More restrictive consistency models rely on strict ordering of ordinary loads and stores, such as *sequential consistency* (SC) or *processor consistency* (PC), hence another approach needs to be employed. We start from an observation that the main problem with the copying instruction execution is that it is not atomic, i.e. when a younger load/store issues, the reads/writes of an older copying instruction have not fully completed. To conform to SC or PC, the illusion of completion of a copying instruction also needs to be maintained with respect to older and younger loads/stores.

To achieve the illusion of completion, after page validation, the processor issues read (read-exclusive) requests for source (destination) in order to obtain read (write) permission for the source (destination) region. Since there may be many block addresses in one region, one way to achieve this efficiently is to send the address ranges to other processors. Other processors react by downgrading the affected cache blocks to a clean-shared state (if modified or exclusive) in response to a read request, or invalidating them (if valid) in response to a read-exclusive request. After all appropriate ownerships are obtained by the processor which will execute the copying instruction, the copying/initialization is started.

If later other processors access an address in the copying regions that requires coherence state upgrade, an intervention or invalidation request is sent to the processor that executes the copying instruction. Upon receiving the request, if the

copying instruction is already retired but not fully completed, the processor must provide the illusion of completion. If copying for the requested block is completed (a block-granularity copying progress tracking is assumed), the request is processed. Otherwise, the request is either buffered until when the block's copying is completed, or negatively acknowledged so that the request can be retried. Which solution is more appropriate depends on various factors, for example the negative acknowledgment approach is more applicable to a distributed shared memory multiprocessor.

IV. Microarchitecture of BCIE

In this section, we discuss the architecture design of the engine that executes the copying instruction. The architecture of the engine, which we refer to Bulk Copying and Initialization Engine (BCIE) is illustrated in Figure 5. It has a *main control unit* (MCU). MCU controls other components in the BCIE, interfaces with TLB for page validation and interface with caches to perform copying.

BCIE needs to track the progress of page validation (to determine when the instruction can be retired) and copying (to determine the completion of copying and to support fine-grain dependence checking). Page validation progress is tracked by using *Page Validation Status Registers* (PVSRs), while copying progress is tracked using *Copy Status Registers* (CSR) at the page level and using *Copy Status Table* (CST) at the block level for outstanding pages.

PVSRs keep the range of addresses that have not yet been validated separately for the source and destination regions. Read permission is checked for the source region while write permission is checked for the destination region. Since copying region may not be page-aligned, a source and destination region may span up to two pages each.

Experimentally, we have not observed the need for supporting concurrent execution of multiple copying instructions. 82% of back-to-back copying instructions are separated by over 5000 instructions, and only 9% of them are separated by less than 2000 instructions. Hence, a copying instruction is likely already completed before the next one arrives at the pipeline. Therefore, we keep the hardware simple by keeping only one set of PVSRs, CSR, and CST.

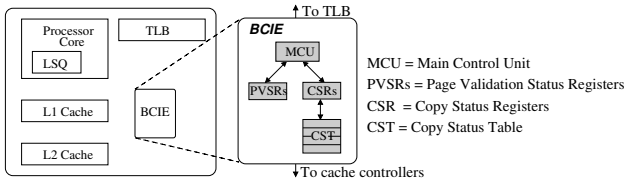


Figure 5. Bulk Copying and Initialization Engine (BCIE).

A copying instruction flows through the pipeline in three sequential steps: in the *start-up* step, the instruction is issued and BCIE is programmed; in the *validation* step, page validation is carried out; and in the *finishing* step, copying is performed. FastBCI does not require copying regions to be aligned. For now, we assume that they are aligned at cache block boundaries, and defer the discussion for non-cache block aligned regions to Section IV-A.

The Start-up Step. The goal of the start-up step is to determine when the copying instruction can be issued. Like regular instructions, a copying instruction waits until all its register operands are ready before issuing. It also waits if the BCIE is occupied by an older copying instruction. After register dependences and structural hazard are resolved, the copying instruction is issued to the BCIE to program it with the instruction parameters, but BCIE does not fire off until after the validation step is completed (Section III-C2 and III-C3). For example, the PVSR and CSR are initialized with the initial address ranges of the source and destination regions specified in the instruction.

The Validation Step. In the validation step, the permission of pages in the range of addresses in the PVSRs are checked in the TLB to validate whether reads (writes) are allowed for the source (destination) regions. The physical addresses are obtained as well. PVSRs keeps the range of addresses that have not been validated, and they are continually updated each time a page is validated. If an exception occurs, the exception bit of the instruction in the ROB entry is raised. After the exception is handled, the copying instruction is re-executed. The completion of page validation is detected when PVSRs contain an empty address range.

Recall that validated pages must remain valid until the copying is completed (Section III-C3). One way to avoid the risk that the page is swapped out or its permission changed is to lock the page in the physical memory. Normally, to lock a page, the OS must be invoked to manipulate its page lock data structure [18]. When an OS wants to swap out a page, it checks whether the page is locked. Meanwhile, an invalidation is sent to the TLB. To avoid the OS involvement in page locking, we augment each TLB entry with a *lock bit* and set the bit for the duration of the copying operation. If the OS wants to swap out a page, the TLB will receive invalidation, and at this time TLB checks the lock bit of the entry. If set, the invalidation fails, and the OS reacts by setting the lock bit of the page in its data structure. Similarly, when a TLB entry with a lock bit set is evicted, the OS must also be invoked so it can lock the page. With this technique, most copying operations only involve the TLB but not the OS. To support this technique, the OS is modified slightly to react to a failed invalidation and TLB replacement of entries with lock bit set by performing the page locking.

The Finishing Step. After page validation, the actual copying can be started. The instruction also becomes retireable. BCIE uses CSR and CST to keep track of the copying progress at page and cache block level, respectively. If the copying regions cross page boundary, the copying is broken into smaller *subcopying* operations, each of which only involves one source page and one destination page (a

page pair). Since we limit the maximum region size to be the size of a page, a copying operation is broken down into at most three subcopying operations, as illustrated in Figure 6. Whenever a subcopying operation completes, the CSR is updated by bumping the source and destination base addresses up to the next page pair.

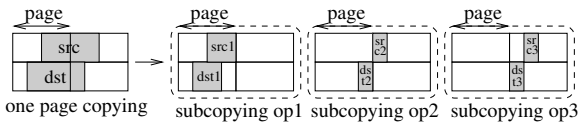


Figure 6. Handling regions that are not page aligned.

The page pairs in the subcopying operation are tracked independently from one another in the CST. Each page pair is allocated an entry in the CST, hence the CST has three entries to keep track of up to three page pairs. Each entry in the CST has a valid bit (V), the virtual and physical base addresses of the source and destination pages, mask bits indicating blocks that belong to the source or destination region in the pages (Mask-Src and Mask-Dst), and bit vectors that track copying progress at the cache block level for both the source and destination pages (Figure 7(b)). We refer to the bit vector as *Copy Progress Bit Vector* (CPBV). The CPBVs (CPBV-Src and CPBV-Dst) have as many bits as cache blocks in a page, e.g. 64 bits for a system with 4KB page and 64-byte cache block size. CPBVs are initially cleared to zero when a page pair is allocated in a CST entry. When a block in the source has been copied to the corresponding block in the destination, the corresponding bit in source and destination's CPBV is set to '1'. When all bits in the CPBVs that are not masked out have a value of '1', the copying for the page pair is completed. The page pair is deallocated from the CST and the CSR is updated to reflect it. When the region length field in the CSR becomes zero, the entire copying has completed, and the BCIE is freed.

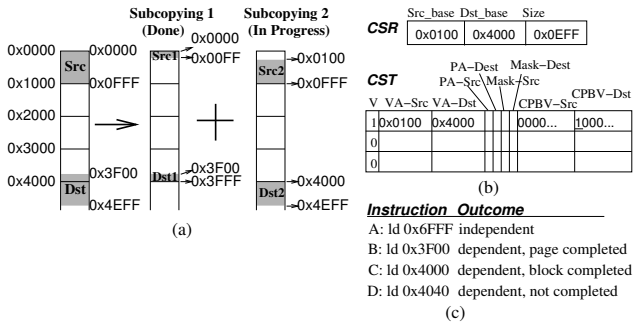


Figure 7. Illustration of fine grain dependence checking.

By checking the CPBVs of each CST, the MCU keeps track of blocks that still need to be copied, and generate appropriate read and write requests to the cache controller. The cache controller performs copying according to the cache affinity options specified. The MCU may simultaneously send several read/write requests to the cache controller, and based on whether the cache blocks are found in the cache, some blocks may be copied sooner than others. The CPBV allows blocks in a page to be copied in parallel and completed out of order. In addition, having multiple CST entries also allows blocks from different page pairs to be copied in parallel and complete out of order. However, completed page pairs are reflected on the CSR sequentially.

The CSR, CST and its CPBVs also serve as a way to check for memory dependences between a copying instruction and younger load/store instructions. For a younger load, its address must be checked against the destination region, while for a younger store, its address must be checked against source and destination regions. The check is performed when the BCIE has an outstanding copying instruction, and in parallel with regular load/store queue accesses. Since BCIE only has a few small registers and a small table, the dependence checking latency can be hidden by the overlapping it with load/store queue access. Moreover, in most cases, only the CSR needs to be checked to resolve dependences (the CST is only checked occasionally).

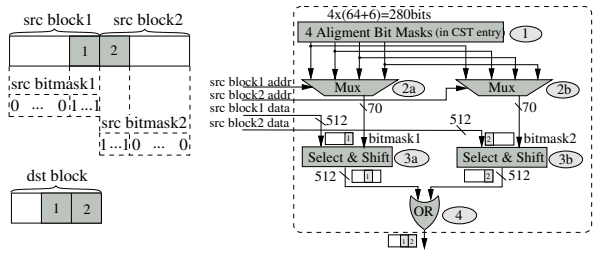
Figure 7 illustrates an example of how dependences with younger loads can be checked by the BCIE. Figure 7(a) shows a copying operation with the source region that is page aligned and the destination region that is not page aligned, and with the region size of 4KB. The copying is broken up into two subcopying operations, and let us assume that subcopying operation 1 is completed while subcopying operation 2 is in progress. Consequently, the address range stored in the CSR (Figure 7(b)) only includes those of "Src 2" and "Dst 2". An entry for subcopying operation 2 is allocated in the CST, and it contains bit vectors for the source (CPBV-Src) and destination (CPBV-Dst) which track which cache blocks in the source and destination pages have completed copying/initialization.

Suppose there are four younger load instructions (A, B, C, and D) illustrated in Figure 7(c). Instruction A loads from an address that is not in the copying region, so it is an independent instruction. B loads from the first byte of "Dst 1" and hence has a dependence with the copying instruction. However, address `0x3F00` does not intersect with the address ranges in the CSR because copying for the address has been completed by subcopying 1 operation. Instruction C loads from an address that is in the address range stored in the CSR. So it is checked further against the entry in CST. Since address `0x4000` is the first byte in "Dst 2" page, it falls into the first cache block of that page. The first bit in CPBV-Dst is a '1' which indicates that the block has completed copying, so its dependence is resolved and instruction C can issue and execute. Finally, instruction D loads from address `0x4040` which is the 65th byte of "Dst 2" page. It corresponds to the second cache block of that page, and since the second bit in CPBV-Dst is a '0', the block has not completed copying. Hence, instruction D waits until the dependence is resolved. Note that the per-block copying progress tracking and dependence checking removes

the dependence pipeline bottleneck. In contrast, DMA and Copy Engine do not have such a mechanism, so instructions B, C, and D would all stall until copying is fully completed.

A. Handling Cache-block Alignment Issues

We have discussed the BCIE design when the regions involved in copying/initialization are not page aligned. If they are not aligned at cache block boundaries, an additional mechanism is needed to handle them in order to still permit copying/initialization at the cache block granularity. Due to the space limitation, we only discuss the high-level idea.



(a). Formating a destination block (b). Logic to handle alignment issues

Figure 8. Handling regions that are not block-aligned.

When the source and destination regions are not aligned at cache block boundaries, it is likely that we need to read two cache blocks from the source in order to produce one cache block (or a part of it) for the destination. Thus, to produce a destination block, the appropriate bytes must be selected from both source cache blocks and combined together. Figure 8(a) shows that to select appropriate bytes, we employ two bitmasks, one for each source cache block and one for each destination block in the subcopying region, with each bit indicating whether a corresponding byte should be selected from the block ('1') or not ('0'). We found that the bitmasks show a repeated pattern and which bitmasks to use for a cache block can be determined by the its address. Hence, at most four distinct 64-bit alignment bitmasks for the source cache blocks are required. To reduce storage overheads, rather than storing four 64-bit bitmasks for the destination cache blocks, we store four 6-bit shift values that indicate how much data from the source block needs to be shifted to generate the needed part of a destination block. The four source alignment bitmasks and the four destination shift value are generated at the beginning of finishing step, stored in the CST entry and used repeatedly.

The logic that performs the byte selection from source blocks and generates the destination block is shown in Figure 8(b). After bitmask generation (Circle 1), the four bitmasks are input into a mux, and one of them is selected based on the address of the source blocks (Circle 2a and 2b). The output is the selected source alignment bitmask (64 bits) plus a 6-bit destination shift value. Then, source blocks are input into the Select&Shift logic, which uses the bitmask to select the appropriate bytes from the source block, and the 6-bit shift value to shift the result (Circle 3a and 3b). Finally, the two blocks are ORed together to produce the destination block (Circle 4).

V. Evaluation Methodology

Benchmarks. To measure the overall performance improvement of FastBCI, we apply it for buffer management functions in Linux kernel (version 2.6.11) [13], such as `copy_from_user`, `copy_to_user` and `memcpy`. We use three buffer-intensive benchmarks: *Apache HTTP Server v2.0* [8] with *ab* workload [8], a network benchmarking tool *Iperf* [6] with 264KB TCP window, and I/O performance tool *iozone3* [5] with tests performed on a 4 MB file residing on an `ext3` filesystem. We run the benchmarks from beginning to the end except for iperf in which we skip the first 2 billion instructions and simulate the next 1 billion instructions. For apache and iperf, we separate the performance measurement of the sender side and the receiver side because they exhibit different performance behaviors.

Machine Parameters. We build FastBCI model on top of a full system simulator Simics [25] with *sample-micro-arch-x86* processor module and *g-cache-ooo* cache module. We simulate a desktop-like machine with an 4-wide out-of-order superscalar processor with 4GHz frequency and x86 ISA. The processor has a 32-entry load/store queue, 128-entry reorder buffer, bimodal branch predictor, 16-entry MSHR, and non-blocking caches. The I-TLB and D-TLB have 64 fully associative entries. The L1-instruction and L1-data caches are 32KB write-back caches with 64-byte block size, and an access latency of 2 cycles. The L2 cache is unified, 1MB size, 8-way associativity, 64-byte block size, and has an 8-cycle access latency. The memory access latency is 300 cycles, and the system bus has a 6.4 GB/s peak bandwidth. The BCIE has a 3-entry CST, and the total size overheads for PVSRs, CSR, and CST are less than 1KB. PVSRs, CSRs and CST, are accessible in 1 cycle, and each modulus calculation in BCIE takes 1 cycle latency as well.

VI. Evaluation

To evaluate FastBCI performance in real applications, we apply the bulk copying instruction for Linux kernel (v2.6.11) buffer management. Figure 9 shows various speedup ratios of various benchmarks under different schemes: traditional loop-based implementation (*Traditional*), copy engine with synchronous bulk copying instructions [21] (*CopyEng*), an optimistic DMA with 0-cycle setup overhead and 0-cycle interrupt handling cost (*0-DMA*), our FastBCI scheme with cacheable affinity for all regions (*FastBCI*), and our FastBCI scheme with the best cache affinity options (*FastBCI+CAF*), obtained by exhaustively trying all cache affinity options and choosing the best performing one for each benchmark. To make the comparison more fair, *CopyEng* and *0-DMA* assume cacheable copying regions [2].

[2]As memory side devices, Copy Engine DMA actually require uncached regions. Using uncached regions, *CopyEng* and *0-DMA*'s average speedups are lower: 5.7% and 12.8%, respectively.

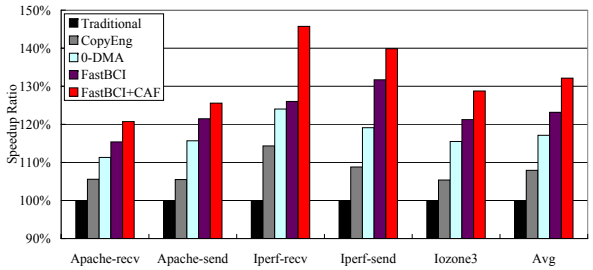


Figure 9. Speedup ratios of various schemes.

Figure 9 shows that FastBCI outperforms traditional implementation, copy engine, and an optimistic DMA in all cases. The copy engine gives 7.9% speedup over conventional loop-based approaches on average due to its granularity efficiency over traditional loop-based implementation. However, the bulk copying instructions quite frequently clog the ROB, and hence the speedups of the copy engine are limited. 0-DMA performs better than copy engine, but even with highly optimistic assumptions, it does not perform as well as FastBCI, reaching an average speedup of 17.1%. Note, however, that the average copying region sizes are relatively small: ranging from 300 bytes to 1.9KB. Thus, in reality, a DMA engine will not be able to amortize its setup and interrupt completion handling overheads well. Through early retirement and per-block copying progress tracking and dependence checking, FastBCI significantly outperforms other alternatives for all benchmarks. The average speedup is 23.2% (roughly 3× of *CopyEng* and 1.5× of *0-DMA*). With the best cache affinity options applied to each benchmark, the speedups further improve to an average of 32.1% (roughly 4× of *CopyEng* and 2× of *0-DMA*). Overall, the results demonstrate that pipeline and cache bottlenecks are the biggest performance roadblocks to a bulk copying instruction. Solving them gives a significant performance improvement over just naively providing better granularity efficiency through a bulk copying instruction support.

Cache Affinity Flexibility Effectiveness. The speedup provided by *FastBCI* is very close to the "perfect pipeline" case in Figure 2 (23.2% vs. 24%). The speedup provided by *FastBCI+CAF*, however, is less close to the "perfect cache" case in Figure 2 (32.1% vs. 37%). Thus, it is important to understand better the relationship of cache affinity performance with the behavior of the application.

Figure 10(a) shows the speedup ratios of apache achieved by FastBCI for various cache affinity options over a traditional loop implementation. The notation on the X axes of A_B corresponds to A being the affinity option for the source (*C* = cacheable, *NEU* = cache neutral), and B being the affinity option of the destination. Note that the non-cacheable option is not shown because it is consistently outperformed by the cache neutral option for both source and destination regions.

Figure 10(a) shows that cache affinity options affect the receiver and sender sides of apache differently, so we

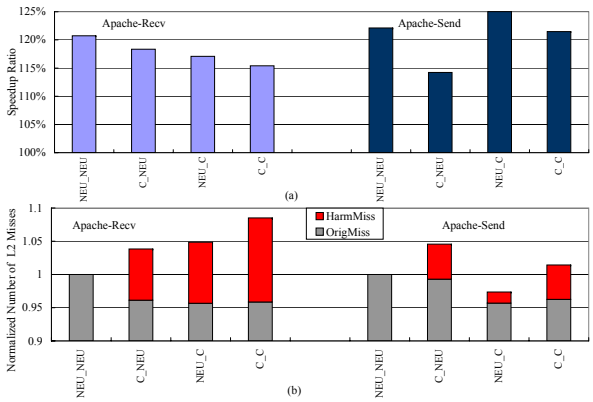


Figure 10. Speedup ratios (a) and the number of L2 cache misses (b) of Apache with various cache affinity options.

will discuss them separately. At the receiver, (NEU_NEU) outperforms other affinity options (21% speedup vs. 18% speedup in C_NEU, 17% in NEU_C, and 15% in C_C). Conceptually, with cacheable source and destination regions, during copying they are brought into the cache. If a region has good temporal locality, future cache accesses will find blocks of that region already fetched into the cache, reducing future number of cache misses. However, the figure shows that the more we make regions cacheable, the worse the speedup becomes.

To understand why this is so, we collect the number of L2 cache misses for the code not including misses caused by the copying instructions themselves (Figure 10(b)). The number of misses is broken into two components: *OrigMiss* shows the number of original misses that remain after a cacheable option is applied, and *HarmMiss* shows the number of new misses that are caused by using the cacheable option, which represents the harmful cache pollution incurred by bringing copying region into the cache. The number of L2 cache misses is normalized to the NEU_NEU case.

Figure 10(b) shows that while caching the source or destination region provides some prefetching effect, as evident by the reduction in original misses, it also produces cache pollution. Since the cache pollution effect is stronger, the new extra misses outnumber the reduction in original number of misses. To understand why this is the case, in TCP/IP processing, the receiver side receives packets by copying them from an OS kernel buffer (which is not reused after the copying) to a user buffer (which is reused by the application after the copying). Hence, caching the source region mostly pollutes the cache. Caching the destination seems fruitful but unfortunately the system call involved (`recv` or `read`) consists of a long chain of kernel routines before and after the copying. Such kernel routines accesses a lot of data within the kernel stack that have better temporal locality than the copying region. So before the user has a chance to reuse the destination, many of the destination region blocks are already replaced. As a result, caching the destination also leads to cache pollution.

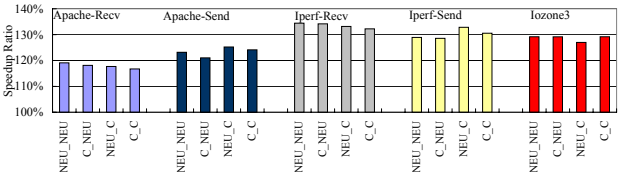


Figure 11. Speedup ratios over loop-based implementation for 8MB L2 cache with various cache affinity options.

Now let us consider the sender side. At the sender side, NEU_C outperforms all other options (25% speedup vs. 22% speedup in NEU_NEU, 21% in C_C, and 14% in C_NEU). The figure shows that the reason NEU_C performs the best is because its number of original L2 cache misses is reduced by 4.4% without adding many new harmful misses (only 1.7%). The reason is that in TCP/IP processing at the sender side, data is copied from a user buffer to an OS kernel buffer. The source region is not reused so caching it causes cache pollution (as in C_NEU). However, the destination region is immediately reused because the TCP/IP stack performs further processing on the newly copied packets, so caching it is beneficial (as in NEU_C).

Although space limitation prevents us from showing the results for iperf and iozone, we observe the same performance trends between iperf and apache, in particular because they rely on the same OS kernel functions for TCP/IP processing, both at the sender side and at the receiver side. One difference is that iperf shows much higher speedups and performance differentials between different cache affinity options. iozone3 performs file write operations rather than TCP/IP operations. It copies data from the user heap to a destination region in kernel slabs that will later be transferred to the disk through disk DMA. There is little reuse in the source region and there is also little reuse in the destination region because it is only accessed for a short time period before being invalidated from the cache to prepare for DMA transfers. As a result, iozone3's cache affinity option performance is similar to the sender side of apache and iperf.

Sensitivity to a Larger L2 Cache Size. Figure 11 shows the performance of FastBCI with various cache affinity options for a larger, 8MB L2 cache size. The figure shows almost identical relative speedup patterns as in a 1MB L2 cache. What is markedly different is that the relative difference in speedups of different cache affinity options has diminished significantly. The reason for this is that an 8MB cache can tolerate cache pollution much better than a 1MB L2 cache, hence cacheable source or destination, or both source and destination, perform very close to the best affinity option. Overall, in both 1MB and 8MB L2 caches, the cache neutral option for both source and destination regions shows a robust performance that is comparable to the best affinity options.

VII. Conclusions

We have shown that the key to significantly improving the performance of bulk copying/initialization instructions is removing pipeline and cache bottlenecks of the code that follows copying operations. We have proposed and presented a novel architecture support that achieves granularity efficiency of a bulk copying/initialization instructions without their pipeline and cache bottlenecks (FastBCI). To overcome the pipeline bottlenecks, FastBCI relies on a non-speculative early retirement of the copying instruction, and fine-grain tracking of copying progress. To overcome the cache bottlenecks, FastBCI allows three cache affinity options that can be applied individually to the source and destination regions: cacheable, non-cacheable, and cache neutral. When applied to kernel buffer management, we showed that on average FastBCI achieves anywhere between 23% to 32% speedup ratios, which is roughly $3\times$–$4\times$ of an alternative scheme, and $1.5\times$–$2\times$ a highly optimistic DMA engine with zero setup and interrupt overheads.

References

[1] ESA/390 Principles of Operation. *IBM Corporation*, 2001.

[2] PowerPC User Instruction Set Architecture. *IBM Corporation*, page 161, 2003.

[3] Intel I/O Acceleration Technology. 2006.

[4] Intel Streaming SIMD Extensions 4 (SSE4) Instruction Set Innovation. *Intel Corporation*, 2006.

[5] IOzone Filesystem Benchmark. *http://www.iozone.org/*, 2006.

[6] Iperf. *http://dast.nlanr.net/Projects/Iperf*, 2006.

[7] Itanium Architecture Software Developer's Manual. *Intel Corporation*, 2006.

[8] Apache HTTP Server Project. *http://httpd.apache.org*, 2007.

[9] GNU C Library. *http://www.gnu.org/software/libc/*, 2007.

[10] NVIDIA CUDA. *http://www.nvidia.com*, 2007.

[11] RDMA Consortium. *http://www.rdmaconsortium.org*, 2007.

[12] AMD Opteron Processor Family. *http://www.amd.com*, 2008.

[13] Linux Kernel. *http://www.kernel.org*, 2008.

[14] Pentium/Core/Core2 Processors. *http://www.intel.com*, 2008.

[15] A. Eichenberger, P. Wu and K. O'Brien. Vectorization for SIMD Architectures with Alignment Constraints. *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[16] D. Abts et. al. The Cray Blackwidow: A Highly Scalable Vector Multiprocessor. *In Supercomputing*, 2007.

[17] D. Miller, R. Henderson and J. Jelinek. Dynamic DMA mapping. *Linux Documentation*, 2006.

[18] D. P. Bovet and M. Cesati. Understanding the Linux Kernel. *O'Reilly Media, Inc*, 2005.

[19] J. Gummaraju et. al. . Architectural Support for the Stream Execution Model on General-Purpose Processors. *Proceedings of International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007.

[20] L. Zhao, L. Bhuyan, R. Iyer, S. Makineni and D. Newell. Hardware Support for Accelerating Data Movement in Server Platform. *IEEE Trans. Comput.*, 56:740–753, 2007.

[21] L. Zhao, R. Iyer, S. Makineni, L. Bhuyan and D. Newell. Hardware Support for Bulk Data Movement in Server Platforms. *Proceedings of the 2005 International Conference on Computer Design (ICCD)*, 2005.

[22] L.Seiler et. al. Larrabee: a many-core x86 architecture for visual computing. *SIGGRAPH '08: ACM SIGGRAPH*, 2008.

[23] M. Gschwind. Chip Multiprocessing and the Cell Broadband Engine. *In ACM Computing Frontiers*, pages 1–8, 2006.

[24] N. Binkert, A. Saidi and S. Reinhardt. Integrated Network Interfaces for High-Bandwidth TCP/IP. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[25] P. Magnusson et. al. Simics: A full system simulation platform. In *Computer*, Feb. 2002.

[26] R. Huggahalli, R. Iyer and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. *International Symposium on Computer Architecture (ISCA)*, 2005.