Transient Caches and Object Streams

Stephen M. Blackburn and Kathryn S. McKinley

Australian National University
Steve.Blackburn@anu.edu.au

The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Memory latency limits program performance. Object-oriented languages such as C# and Java exacerbate this problem, but their software engineering benefits make them increasingly popular. We show that current memory hierarchies are not particularly well suited to Java in which *object streams* write and read a window of short-lived objects that pollute the cache. These observations motivate the exploration of *transient caches* which assist a parent cache. For an L1 parent cache, transient caches are positioned similarly to a classic L0, providing one cycle access time. Their distinguishing features are (1) they are tiny (4 to 8 lines), (2) they are highly associative, and (3) the processor may seek them in *parallel* with their parent. They can assist any cache level. To address object stream behavior, we explore policies for read and write instantiation, promotion, filtering, and valid bits to implement no-fetch on write.

Good design points include a *parallel L0* (PL0) which improves Java programs by 3% on average, and C by 2% in cycle-accurate simulation over a two-cycle 32KB, 128B line, 2-way L1. A *transient qualifying cache* (TQ) improves further by a) minimizing pollution in the parent by filtering short-lived lines without temporal reuse, and b) using a write no-fetch policy with per-byte valid bits to eliminate wasted fetch bandwidth. TQs at L1 and L2 improve Java programs by 5% on average and up to 15%. The TQ even achieves improvements when the parent has half the capacity or associativity compared to the original larger L1. The one-cycle access time, a write no-fetch policy, and filtering bestow these benefits. Java motivates this approach, but it also improves for C programs.

1. Introduction

Because memory latency has and will continue to limit processor performance, researchers have focused a lot of attention on understanding and improving cache memories and their behavior The vast majority of this work uses classic C and Fortran workloads with techniques that eliminate and tolerate latency by exploiting spatial and temporal locality. Programmers are, however increasingly turning to Java and C#, because these languages offer software engineering benefits that help them produce higher quality software faster. A recent Gartner report predicts that by 2008, 80% of software will be written in Java or C# [10].

This paper examines Java workloads and their interactions with modern caches to derive new cache designs that work well for Java and C benchmarks. We first characterize *object streams*. A typical Java program creates many short-lived objects [7, 37] that start with an initializing write, followed by a few reads and writes. Accesses to these objects create a window of irregular accesses to several active cache lines that march through memory, similar to a stream in conventional languages [6, 15, 28, 29, 32] but with a wider and less regular window of accesses.

Like conventional streams, object streams displace useful data as they march through memory. Unlike conventional streams, object streams cannot be naively discarded because they are irregular and some fraction of the objects will be reused. Object streams have the following properties: (1) initializing writes, (2) subsequent reads and writes with short reuse distance, and (3) then most, but not all, see no further reuse.

To solve these problems, we propose a family of *transient* caches: tiny, highly associative caches that assist a *parent* cache by providing single cycle reuse to very short lived (transient) data. A transient cache is *not* just another level in the cache hierarchy because a) the processor seeks it in parallel, b) it is tiny (less than 2% the capacity of its parent), c) it is highly associative, and d) it may assist any cache level. Unlike spatial and temporal caches [14, 18, 22, 36], a transient cache is *not* intrusive since it does not require changes to the parent cache or adding other structures to keep auxiliary state.

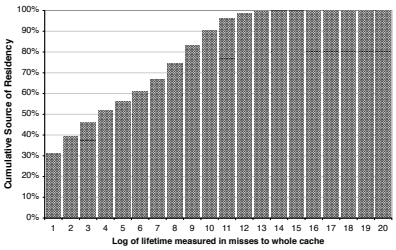
The key insight is that due to temporal locality, a tiny transient cache can bestow benefits on a large fraction of all data accesses. while employing expensive features such as full associativity and per-byte valid bits sparingly, coupled with single cycle access times. Transient caches improve performance in any of four ways. They a) provide low latency access to hot data, b) filter short-lived lines to minimize cache displacement, c) eliminate wasted fetch bandwidth, and d) reduce write traffic.

We implement this cache in a cycle accurate simulator using SimpleScalar [5, 13] and MicroLib [30] with the SPECjvm98, DaCapo, and SPECcpu2000/C benchmark suites. We experiment with a variety of configurations, showing many improve performance over the base cache. For a two-cycle, 32KB, 2-way, L1 data cache with 128 and 64 byte lines we demonstrate that a 512 byte (4 and 8 lines respectively), one-cycle transient cache improves performance over a base 32KB two-cycle parent cache by an average of 5% on Java programs, and 3% on C programs. The transient cache never degrades performance, and improves it up to 15%.

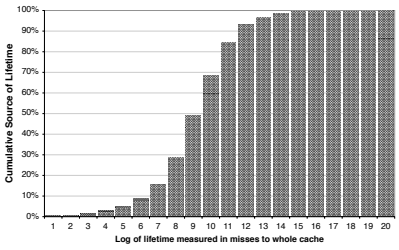
This paper (1) characterizes of the object stream behavior in Java programs, and (2) proposes a new cache design that exploits this behavior. Transient caches are a unique combination of features and policies that protect the main cache from pollution, eliminate write-miss bandwidth, and provide high performance and efficiency for Java and C workloads.

2. Object Streams

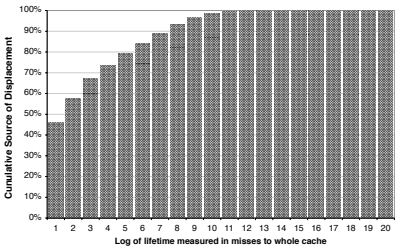
Ample anecdotal and empirical evidence shows that the high allocation rate of modern object-oriented programs degrades the performance of modern memory hierarchies [7, 8, 37]. We refer to the disruptive behavior as *object streams*, by analogy to data streams. This section quantifies this effect to motivate our design. Section 4 further characterizes Java memory behavior, and shows differences between Java and C programs.



(a) Source of L1 Data Cache Residency as a Function of Lifetime



(b) Source of L1 Data Cache Live-time as a Function of Lifetime



(c) Source of L1 Data Cache Displacement as a Function of Lifetime

Figure 1. Short Lived Data Displaces the Cache

To characterize object streams, we develop an analysis using the Cachegrind simulator [26] and modifications to Jikes RVM [2]. We modify the memory manager *only for this analysis* to never reuse memory or move objects. We modify Cachegrind to track cache line, noting a) instantiations, b) evictions, and c) the last use before eviction. We aggregate statistics to compute: 1) the total time each line is resident in the cache, 2) the total live time (instantiation to last use), and 3) the total time dead (last use to eviction). We compute a *displacement metric* to identifies lines that displace live lines, the program reuses. The displacement metric records the number of times the victim line is re-instantiated before the displacing line is evicted. We measure time in terms of misses to a a 32KB, 2-way set associative, 128 byte line, L1 data cache (an IBM PowerPC [28] design we use throughout the paper)

All freshly allocated objects are instantiated with a write, and most new objects (and thus cache lines) are short lived [7]. Figure 1 quantifies this effect on cache behavior. It shows the geometric mean of the residency, lifetime, and displacement metrics for the SPECjvm98 benchmark suite. Each figure is a histogram of lifetimes on a $\log_2$ scale, accumulating all lines with a lifetime less than or equal to 2^N in bucket N.

Figure 1(a) shows the distribution of all cache *residency* as a function of line lifetime. Just over half of all cache residency (50th percentile) is due to lines that live for no more than 16 (2^4) misses to the whole cache. Figure 1(b) shows the distribution of *live-time* (i.e., time the line is live: residency minus dead time for each line). Lines that are live for no more than 16 (2^4) misses to the whole cache account for less than 5% percent of the aggregate live-time of the cache. The 50th percentile for live time is not reached until we include lines which have seen 512 (2^9) misses to the whole cache. The impact of short-lived lines on residency is therefore skewed by a factor of 32 ($2^9/2^4$) compared to their lifetime.

Not only do short-lived lines occupy the cache for a disproportionate time, but Figure 1(c) shows that they evict important data at an even more disproportionate rate. More than half of displacements are due to lines which are live in the cache for four or fewer misses to the whole cache. In this case, the 50th percentile is out of proportion to cache liveness by a factor of 128 ($2^9/2^2$)!

3. Transient Caches

We define transient caches as a family of small, highly associative caches that assist a *parent* cache by servicing loads and/or stores to short lived (transient) data. Unlike an orthodox L0, a transient cache is *not* just another level in the hierarchy because a) the processor may seek it in parallel with its parent, b) it is tiny (less than 2% the capacity of its parent), c) it is highly associative, and d) it can assist any cache level. The basis of transient cache design is that temporal locality allows a tiny structure to bestow benefits on a large fraction of all data accesses. Section 4 shows that 60% of reads and 90% of writes hit in a transient cache with less than 2% the capacity of its parent L1. Thus, transient caches can apply expensive features such as full associativity and per-byte valid bits sparingly and couple them with single cycle access times to provide substantial benefits. Transient caches improve performance in any of four ways. They a) provide low latency access to hot data, b) filter short-lived lines to minimize displacement, c) eliminate wasted fetch bandwidth, and d) reduce write traffic.

3.1 Background

We first discuss inclusion, write policies, write buffers and L0 designs to provide the backdrop for our work.

Inclusion Policies A cache may be *inclusive* or *exclusive* with respect to the cache below it in the hierarchy. An inclusive cache includes the same line in both levels, while an exclusive policy includes a line at only one level. Exclusive cache hierarchies increase total capacity and associativity by avoiding duplication, but generate more data movement (all victims must be propagated) and require more complex snooping in an multiprocessor setting [38, 17].

Write Policies Write policies determine when and how the cache propagates changes. They determine (1) whether writes are immediately propagated through to lower levels of the hierarchy (*write-through*) or only propagated when a line is evicted (*write-back*), (2) whether a new line is instantiated upon a write miss (*write-allocate*) or not (*write-no-allocate*), and (3) whether the backing data is fetched before writing to a new line (*fetch-on-write*), or not (*no-fetch*). A no-fetch policy requires validity bits for each line to prevent reads to uninitialized data. Previous work found that write-allocate-no-fetch is an ideal policy for highly allocating languages [8]; we confirm that 24 Java and C programs almost never read unwritten bytes after a write miss (99.9%). However, validity bits increase cache area by more than 10% [8, 16] and so no modern cache implements a no-fetch policy [16].

Previous work [12] argues that a no-allocate policy is a poor choice for object oriented languages since the line subject to the write miss is typically accessed again immediately. However, since a no-fetch policy is impractical, the alternative to no-allocate is fetch-on-write. A fetch-on-write policy *always* fetches the backing data, while a no-allocate policy *only* fetches the backing data if there is a subsequent read. Furthermore, subsequent reads may be satisfied by store-to-load forwarding in modern out-of-order processors, bypassing the need for a fetch altogether. We find a write-through-no-allocate reduces L1 misses by 6% on average and up to 18% for C and Java benchmarks compared to both write-back and write-through-allocate, but this does not translate to any significant performance advantage in cycle accurate simulation.

Write Buffers The primary function of the write buffer is to mitigate the bandwidth demands of a write-through cache by coalescing multiple stores to the same line [16, 33]. Write-through no-allocate with a write buffer is a popular choice for L1 caches [28, 35, 9, 11]. A typical write buffer has 8-entries with valid bits that insure only valid data is written to lower levels of the memory hierarchy. The

mechanism	**L0**	PL0	RQ *read qualifying*	RM *read miss*	**WB *write buffer***	WC *write cache*	TM *r/w miss*	TQ *r/w qualifying*
parallel seek		Yes	Yes	Yes		Yes	Yes	Yes
filtering			Yes	Yes		Yes	Yes	Yes
satisfies reads	**Yes**	Yes	Yes	Yes		Yes	Yes	Yes
instantiate on parent read hit	**Yes**	Yes	Yes					Yes
instantiate on parent read miss	**Yes**	Yes	Yes	Yes			Yes	Yes
instantiate on write miss					**Yes**	Yes	Yes	Yes
write buffering					**Yes**	Yes	Yes	Yes
write-no-fetch						Yes	Yes	Yes

Table 1. Transient Cache Features and Names

write buffer works in parallel with its parent cache. Since it does not satisfy reads, such systems are inclusive: they duplicate write hits in the buffer and L1. A write-miss followed by a read to the same word requires the system to push the line out of the write-buffer, fold it together, and place it in the L1. A write-through allocate policy eliminates this problem [15].

L0 Cache Figure 2 shows an inclusive L0 design in which the processor first seeks in the L0 and fetches on a miss [20]. On an L0 read miss, the memory system forwards the request to the L1 which responds immediately on a L1 hit. On an L1 miss, the cache hierarchy populates both the L1 and the L0, for an access time of at least the L0 plus the L1 latency. L0 designs are typically motivated by power savings, but if searched in parallel rather than sequentially, an L0 cache can offer performance advantages [34]. A classic L0 is typically write back. In a *write-back allocate fetch-on-write* design, the L0 first propagates dirty victims to the L1 before fetching on a miss. A write buffer could mitigate write traffic in a write through design, but putting two similarly sized structures in front of the L1 is an odd choice. Most L0 implementations are too large and not sufficiently associative to be considered transient caches [20, 34].

3.2 Transient Cache Mechanisms

Table 1 summarizes the transient cache mechanisms and configurations we discuss below, including the two orthodox design points: a sequential seek L0 ("L0") and a write buffer ("WB").

Parallel Seeking. When servicing reads, the processor can seek a transient cache and its parent *sequentially* or in *parallel*. Parallel seeking requires an extra data path and more power, but offers performance advantages. Since a transient cache is tiny it can attain one cycle access time and deliver data much faster than its parent. Agarwal et al. use Cacti to show that up to a 1KB highly associative cache will be a single cycle in current and future technologies [1].

Filtering. A transient cache may *filter* instantiations to its parent to mitigate the displacement effect of short-lived data. Figure 4 shows the mechanism in which the only path from the L2 into the L1 is *via* the transient cache, and that path is gated by a "Keep?" function. A miss the parent is instantiated in the transient cache and only conditionally instantiated in the parent upon eviction from the transient cache. Transient caches are therefore neither strictly inclusive or exclusive. When a line is instantiated in the parent cache, it is placed in the MRU position in the appropriate set.

Filtering is implemented by conditionally setting a *keep-me* bit associated with each transient cache line. We use two heuristics. First, we always set the keep-me bit for write-instantiated transient cache lines that were resident in the parent cache at the time of the write. Otherwise, the transient cache initializes the keep-me bit to zero when it instantiates the line. Second, we want to promote lines into the parent if they will be reused soon. We use accesses to the transient cache to predict future reuse.

We govern this prediction with a *keep-me filter threshold*. On each read, we set the keep-me bit for a line if it is below the threshold position. We thus promote lines to the parent if the program reads them in the transient cache, but otherwise do not put them in the parent. A filter threshold of zero (F0) is conservative; if the line is ever read while in the transient cache after the reference that instantiates it, we promote it to the parent. Higher thresholds (F1 to the size of the transient cache) are more discriminating. Note that the filtering bit and policy are isolated to the transient cache logic and *do not* intrude on the parent cache, in contrast to prior approaches [14, 18, 22, 36].

Read Instantiation We explore three instantiation policies when the transient cache suffers a read misses: (a) always instantiate, (b) instantiate only if it also misses the parent, and (c) never instantiate. The L0, PL0, read qualifying (RQ) and read/miss qualifying (TQ) designs follow policy (a); the *miss cache* designs, read miss (RM) and read/write miss (TM), follow (b); and the write cache (WC) follows policy (c). Policy (a) maximizes the likelihood of a transient cache hit, but policy (b) may filter better, reducing parent cache pollution and thus improving parent cache hit rates. Policy (c) is the simplest design, and often effective (in Table 4 WC outperforms RM).

Write Instantiation We implement a write-through no-allocate policy in the transient cache designs which do not perform write buffering (L0, PL0, RQ and RM) which simplifies implementation and as outlined in Section 3.1 does not impact performance.

Write Buffering The WC, TM and TQ designs all subsume write buffer functionality. We implement a retire-at-N write buffer [33] with a most-recently-written (MRW) ordering. The WC design simply adds the capacity to service reads from the write buffer and parallel seeking without changing the instantiation policy or the MRW ordering. Since the role of the write buffer is to mitigate write traffic, its utility falls as write bandwidth to the next level in the cache rises.

Write-No-Fetch We use validity bites on transient cache lines to implement a write-no-fetch policy [8, 16]. We found that this reduced the number of fetches due to write misses by 99.9% across both Java and C benchmarks. Since we only apply the validity bits to the transient cache, the overhead is tiny (around 1% of the overhead of applying validity bits to the entire parent cache).

3.3 Design Options

We now briefly describe how the new transient cache designs combine these mechanisms, summarized in Table 1. The RQ design adds filtering to the PL0, and performs almost identically. The RM design only instantiates lines which miss the parent cache, and is thus strictly exclusive and therefore cannot reduce access times to lines resident in the parent cache.

The WC extends the WB by servicing reads which hit in the write buffer, using validity bits to implement a write-no-fetch policy, and filtering instantiations to the parent cache. Since reads

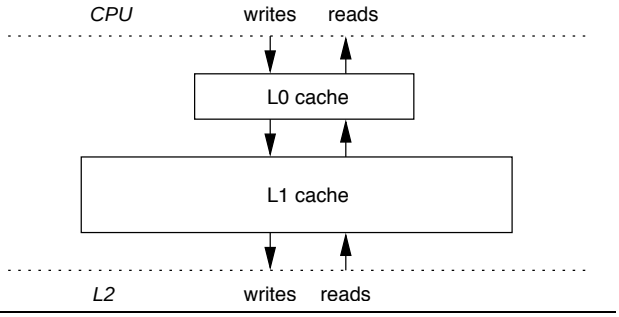


Figure 2. L0: Classic L0 Data Cache Design

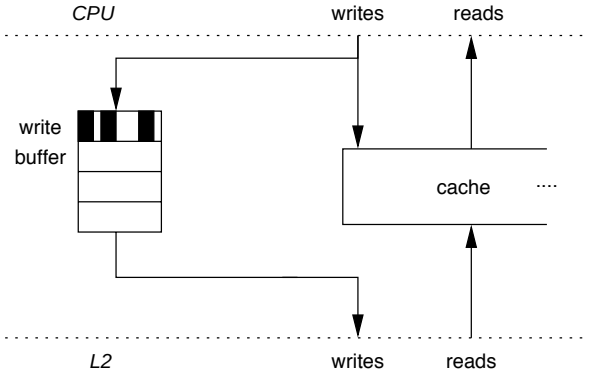


Figure 3. WB: Classic Write Buffer for Write Through L1

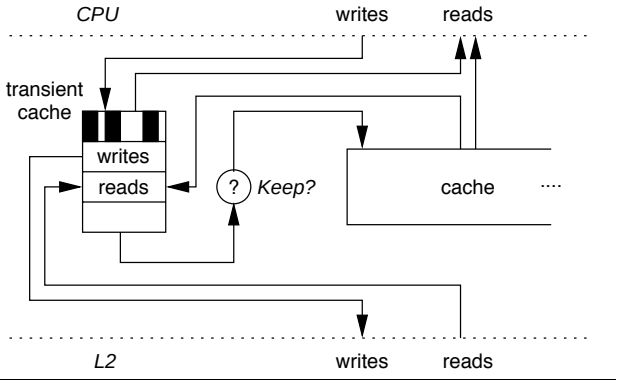


Figure 4. A TQ Cache for a Write Through L1

often immediately follow writes, the WC services many read requests. The write-no-fetch policy dramatically reduces the number of fetches due to write misses. By using read access patterns to guide filtering, the WC reduces parent cache instantiations due to write misses.

3.4 TQ and TM Transient Cache Implementation.

The transient qualifying (TQ) and miss (TM) caches differ only in their instantiation policies, and are otherwise are identical. We therefore describe the TQ implementation in detail, and conclude with the single modification needed for the TM.

Figure 4 shows the data paths for a TQ design assisting the L1 between the CPU and the L2. It has a valid bit per byte and a keep-me bit per cache line. The TQ services reads in parallel with the parent cache. The TQ is structured as two logically distinct parts: a write component and a read component. The purpose of this division is to preferentially keep write instantiated lines longer than a traditional cache, making them available to satisfy reads from object streams, but to do so without polluting the parent cache. The TQ puts read misses in the read component, and write misses in the write component. The TQ services reads from both components. A read to an invalid byte triggers a fetch from the next level of the memory hierarchy, correctly implementing the write no-fetch policy. The proportion of lines devoted to each is a policy choice, and may be adaptive.

Write Component. The write component functions as a write buffer and is maintained as a MRW queue. Its victims cascade directly into the top of the read component. It uses write misses to help predict short-lived lines; thus it differentiates a write hit to the parent cache by setting the line's keep-me bit, and otherwise initializes the keep-me bit to zero. On a write miss, the TQ instantiates a new line in the MRW position of the write component. It may need to first stall if the previous retirement is not complete. It sets the valid bits, and initiates the retirement of the last line in the write component. The miss also causes the last line in write component (which is guaranteed to be retired) to fall into the read component, and the last line in the read component is then promoted to the parent cache or discarded. It only promotes valid lines to the parent.

Lines that are never written do not appear in the write component. If the program writes a line in the TQ, the TQ promotes the line to the MRW position in the write component and shifts down the intervening lines. It also sets the necessary valid bits in the new line. When a line progresses to the last entry in the write component, the TQ retires it, writing it to the next level of memory hierarchy. This policy reflects a standard write buffer retirement policy [33]. Reads to lines that are being retired must stall [33]. However, unlike a write-buffer, the TQ keeps the line after retirement, thus making it available to satisfy reads until it falls out of the read component.

Read component. The read component is a least-recently-read (LRR) queue. When a read misses the TQ, it fetches and instantiates the line at the top of the read component and evicts the bottom entry of the read component and sets the valid bits. For lines that fall into the read component from the write component, it maintains their valid bits. When the program writes a line in the read component, it instantiates the line in the MRW position of the write component and slides the other lines down in the cache. The write component eventually retires all written lines. This policy means read component lines are never dirty and need never be written back. When the TQ evicts from the read cache, it promotes the line to the parent if its keep-me bit is set. Otherwise, it discards the line.

The TM differs from the TQ only in its read miss instantiation policy. The TM will only instantiate a line which misses the TM if that line also misses the parent cache. This removes the need for a data path from the parent cache to the TM.

4. Results

This section presents our methodology, cycle accurate results of various transient cache and L0 configurations, and an analysis of how it achieves these improvements. We organize the cycle count comparisons into four parts: (1) upper bounds for improvement, and comparisons of L1 write policies, L0, and PL0 designs; (2) comparisons across all the read, write, and qualifying L1 transient caches, and variations of the filtering thresholds; (3) reducing the line size to 64B; and (4) L1 and L2 transient qualifying (TQ) caches. We show many configurations that improve performance (from 3 to 5% on average and up to 15% on Java, and 7% on C), and are *robust* (i.e., never degrade performance), and they still improve over the original cache, even when hampered by a smaller parent or parent with less associativity (we reduce the size by half, and associativity from 2-way to direct-mapped).

Methodology. We implement our system in MicroLib [30], a cycle-accurate simulator infrastructure. Since the MicroLib simu-

Parameter	Value
Processor Core	
Fetch, Decode, issue width	4
Load-Store queue	8
Memory System	
L1 TC latency	1 cycle
L1 TC capacity	512 bytes (4 × 128 or 8 × 64)
L1 TC associativity	fully
L1 Data latency	2 cycles
L1 Data capacity	32 KB
L1 Data line size	128 byte
L1 Data associativity	2-way
L1 Data allocate policy	write no-allocate
L1 Data ports	2
L1 MSHR	8
L1 Write policy	write back
L1 Write buffer size	0
L1 Instruction capacity	32 KB
L1 Instruction latency	2 cycles
L2 TC latency	2 + 1 cycles
L2 TC capacity	2KB (16 × 128 or 32 × 64)
L2 TC associativity	fully
L2 unified latency	2 + 11 cycles
L2 unified capacity	512 KB
L2 unified line size	128 btye
L2 unified associativity	8-way
Bus	
Frequency	0.5 × CPU
Width	128 bits
SDRAM (memory cycles)	
Banks	4
Rows	8192
Columns	1024
RAS to RAS	2
RAS to CAS	5
CAS latency	3
RAS precharge	5
RAS active	8
RAC cycle	11

Table 2. Base and Transient Cache Processor Configurations

lator could not execute Java programs (which dynamically generate code), we ported the MicroLib memory model sub-system to Dynamic SimpleScalar which supports systems calls and instructions used by Java as well as dynamic code generation [5, 13] in Jikes RVM [2]. We further modularized the MicroLib to generalize and configure existing cache components and include transient caches. Building this infrastructure to faithfully simulate C and Java programs was a substantial engineering feat, that possibly was a barrier to such work to date. We will make our simulation infrastructure publicly available.

Table 2 shows the hardware configurations we use for our cycle accurate simulation. These represent a current aggressive out-of-order processor model and corresponding memory system. We closely modeled the memory system of the IBM PowerPC 970 [28, 35]. We use this as our base configuration, but *without* its 8-entry 64B write buffer [27]. We use in a uniprocessor setting and Section 5 briefly discusses multiprocessors issues.

We use gcc v3.3 to compile 10 of the SPECcpu2000 benchmarks for the PPC. We use the minnespec reduced input set [21], available from SPEC. For executing 14 SPECjvm98 and DaCapo [4] benchmarks, we use Jikes RVM 2.4.1 [2] for PPC. For our cycle accurate simulations, we limit execution to 2 billion instructions, which does not effect the reduced input set SPECcpu2000 benchmarks, but reduces the length of 13 out of 14 Java benchmarks.

We configure Jikes RVM to use a high performance generational copying 4MB bounded nursery and a mark-sweep older space, with a heap size of three times the minimum in which the application executes [3]. We use *replay compilation* which deterministically applies the optimizing compiler to frequently executed methods chosen by the adaptive just-in-time (JIT) compiler in previous (offline) runs. This configuration produces a realistic mixture of optimized and unoptimized code, but eliminates variations due to sample-driven application of the dynamic optimizing compiler. We report this *first run* for the DaCapo benchmarks, because of implementation issues which we will resolve for the final version. We report only application time using the *second run*, a better metric, on the SPECjvm benchmarks. We run one iteration of the application with replay compilation, and then turn off the compiler. Eeckhout et al. previously showed that including the JIT compilation can dominate the application time, and obscure differences between benchmarks.

Bounds, Write Policies, and Conventional Policies. Table 3 separates the results and geometric means for C and Java programs. The second column shows the simulated cycle count in millions of the base PPC system. We normalize *all* results in the paper to this base for ease of explanation. Column three presents an upper bound—a perfect 2 cycle L1. These configurations show that Java programs lose slightly more performance due to cache effects than C programs: 11% average potential improvement for Java, and 9% for C. Importantly, this places a hard upper bound on the effect of L1 miss reductions—a 100% reduction in misses is required to yield 10% in performance. However, column four shows that if we could halve the access time of the 32KB cache to one cycle, we could get within 2% of a perfect caches for both C and Fortran. One interesting result is that for programs like jess, a one-cycle cache provides *more* improvement than a perfect two-cycle cache. These results demonstrate that there is room for improvement, and that one-cycle access has the potential to bestow good performance improvements. Indeed, Table 7 (explained below) shows the improvements comes from one-cycle access times as well as filtering. Our bottom line 5% average improvement is substantial relative to these bounds.

Columns five through seven show write through (WT) and write through allocate (WTA); and a write buffer (WB) relative to the write back no-allocate base. These choices do not impact performance much. This insensitivity is due in part to high bandwidth between the L1 and L2, modeling the PPC, and the effectiveness of the load-store queue in satisfying short distance read after write.

The last two columns shows the relative performance compared to PPC of adding a write-through no-allocate L0 with conventional policies, including sequential seek, and a PL0 whose only difference is a parallel seek. These caches are just 4 × 128B lines. A serial L0 degrades performance systematically between 2 and 10% for C and Java programs, which is consistent with prior work [20]. However, a PL0 improves average performance by 3% and up to 12% on Java, and 2% on C. Unfortunately, the PL0 is not completely robust with respect to the original, as it slows down javac and pseudojbb.

L1 Transient Cache Policies and Filtering. This section shows that two L1 transient cache configurations (TM with default filtering, and TQ with more aggressive filtering) attain the same average performance as the PL0, but are more performance robust. Columns three though eight in Table 4 show the new transient cache designs: write cache (WC), read miss cache (RM), read qualifying (RQ), transient (read/write) miss (TM), and transient (read/write) qualifying (TQ), each applied only to the L1, with a default write back Le. All contain just 4 × 128B lines. TQ and TM use two lines each for their write and read components. These results use a conservative keep-me which puts the line in the parent if it is ever read again after entering the assisting cache. The last three columns change this policy to keep less: the line must be read at least once after it is not in the most recently used (MRU) cache position.

The first result to note is that the RQ with default filtering and PLO both perform and behave well and almost identically. Since the RQ is slightly more robust, and generalizes over PL0, we use

program	Base L1: 128B, 32KB, 2-way, 2 cycle						4-lines, ∼512B	
	cycles(M) PPC	perfect	1 cycle	WT	WTA	WB	serial L0	parallel PL0
Java								
_201_compress	1701	0.81	0.83	1.00	1.00	1.00	1.02	0.88
_202_jess	2061	0.95	0.88	1.00	1.00	1.02	1.06	0.95
_205_raytrace	2260	0.88	0.89	1.00	1.00	1.01	1.09	0.97
_209_db	2593	0.51	0.92	0.99	0.99	1.00	1.06	0.99
_213_javac	2430	0.96	0.96	1.00	1.00	1.00	1.03	1.01
_222_mpegaudio	1520	0.99	0.85	1.00	1.00	1.01	1.06	0.92
_228_jack	2667	0.99	0.95	1.00	1.00		1.03	0.99
antlr	369	0.96	0.92	1.00	1.00	1.00	1.04	0.96
fop	2424	0.96	0.93	1.00	1.00		1.05	0.98
hsqldb	2665	0.95	0.94	1.00	0.99		1.04	0.97
pmd	2518	0.92	0.92	1.00	1.00		1.05	0.99
ps	2085	0.94	0.90	1.00	1.00		1.08	0.97
xalan	2328	0.79	0.91	1.00	1.00	1.00	1.03	0.95
pseudojbb	2115	0.98	0.96	1.00	1.00	1.01	1.04	1.01
max	2667	0.99	0.96	1.00	1.00	1.02	1.09	1.01
min	369	0.51	0.83	0.99	0.99	1.00	1.02	0.88
geomean	**1961**	**0.89**	**0.91**	**1.00**	**1.00**	**1.00**	**1.05**	**0.97**
C								
bzip2	1255	0.80	0.91	1.00	1.00	0.99	1.08	0.97
crafty	46	0.96	0.96	1.00	1.00	1.00	1.02	0.99
gap	755	0.94	0.92	1.00	1.00	1.00	1.03	0.97
gcc	21	0.98	0.96	1.00	1.00	1.00	1.06	0.99
gzip	952	0.82	0.92	0.99	1.00	0.99	1.06	0.96
parser	1799	0.88	0.93	1.00	1.00	1.00	1.07	0.98
twolf	962	0.95	0.94	1.00	1.00	1.00	1.06	0.99
vortex	1365	0.97	0.91	1.00	1.00	1.00	1.10	0.98
max	1799	0.98	0.96	1.00	1.00	1.00	1.10	0.99
min	21	0.80	0.91	0.99	1.00	0.99	1.02	0.96
geomean	**462**	**0.91**	**0.93**	**1.00**	**1.00**	**1.00**	**1.06**	**0.98**

Table 3. L0 Variations: 4 lines (∼512B); default L1: 128B line, 32KB, 2-way, 2-cycle; L2: write back .

RQ in the remainder of the discussion. An RQ with aggressive filtering is a poor choice. The write component (in WC, TM and TQ) only instantiates on a write miss, which compared to a read component, (a) gives lines more time to age before filtering, and (b) more closely matches the properties of object streams, which are initiated with a write. Thus WC, TM and TQ perform better with aggressive filtering than RQ. Similarly, RM does not bestow temporal locality on write instantiated lines, and although it improves performance by 1% on average, it is not performance robust and at best improves by 4%.

With default filtering the TM outperforms the TQ because it has reduced traffic through it, giving the write and read lines sufficient time to provide locality, and if they do, the line goes into the parent. When the TQ uses the less aggressive filtering, its increased traffic compared to the TM causes some lines to get replaced before they can provide locality, but since the filtering is not too aggressive they quickly come back into the TQ. More aggressive filtering reverses this effect; the TM performs less well with aggressive filtering because its traffic is already reduced by only instantiating on a miss to its parent. TM with default filtering and TQ with more aggressive filtering are both better design points than PL0/RQ because they are more robust, although all have the same average performance.

Reduced Line Size. The data in Table 5 show that these designs are more effective when the cache line size is 64B which enables better filtering through the extended opportunity for aging gained by the increase in line count. All numbers are normalized to the original PPC. As in Table 5, these configurations are applied only to the L1, with a default write-back L2. The second column (64B), shows that compared to the PPC, this cache performs the same as 128B lines, but improves db by 6%. however, all the transient caches (RQ, TM, TQ, TQ2/6) offer further improvements for db, as well as the other benchmarks. TQ2/6 splits the write:read components 2:6 instead of the default 4:4. The improvements range from 3% for TM, 4% for RQ, up to an average of 5% for the TQ designs. The best result improves performance by 13% with these configurations.

L1 & L2 Transient Caches and Protection Against Reduced Total Size and Associativity. The first results to note in Table 6 are in columns seven (128B line) and eleven (64B line), marked 2-way, 32KB. These columns show that a TQ L1 coupled with a TQ L2 offers a 1% further performance improvement over just a L1 TQ, but increasing the best result to 12% (and 15%). All improvements in this table are still normalized against the original PPC cache.

Columns five and six of Table 6 show the effect of halving the L1 cache capacity (L1 – 16KB) and halving the L1 associativity (L1 – Direct Mapped D-M). These configurations also use a two-cycle parent cache latency. Not surprisingly, they slow down the system without transient caches (4% on average). However, transient caches at the L1 and L2 on these two reduced configurations *still improve performance over the base 32KB, 2-way L1!* Columns eight and nine show that improvement is 2% on average for 128B lines, and the last two columns show that improvement is 4 and 5% on average. We found these results very encouraging.

Why the Transient Cache Works. Table 7 breaks down the details of the benchmark behaviors on the base system and compares it to one of the best transient cache configurations (64B lines; L1 TQ8 F1; L2 TQ32 F1) whose performance improvement is repeated here in column eight.

These results also reveals some differences and similarities between Java and C program behavior. Column two shows the instructions per cycle (IPC). Both C and Java have close to the same range of IPC values, but the average for Java is 10% lower than C, which is not surprising considering that even with aggressive inlining, Java programs typically have smaller methods and smaller

program	PPC	Default Filtering (F0) WC	RM	RQ	TM	TQ	Agressive Filtering (F1) RQ	TM	TQ
Java									
_201_compress	1701	0.92	0.96	0.88	0.90	0.90	1.40	0.90	0.90
_202_jess	2061	0.98	0.99	0.96	0.98	0.97	1.54	0.98	0.97
_205_raytrace	2260	0.98	0.99	0.97	0.98	0.97	1.65	0.98	0.97
_209_db	2593	0.99	1.00	0.99	0.97	0.99	1.50	0.99	0.97
_213_javac	2430	0.99	0.99	0.97	0.98	1.02	1.36	1.02	0.97
_222_mpegaudio	1520	0.99	0.99	0.93	0.99	0.96	1.85	0.99	0.96
_228_jack	2667	1.00	1.01	0.99	0.98	1.00	1.30	1.00	0.98
antlr	369	0.98	0.99	0.97	0.98	0.97	1.36	0.98	0.97
fop	2424	0.99	1.00	0.98			1.41	0.99	0.98
hsqldb	2665	0.98	1.00	0.97	0.98	0.97	1.34	0.98	0.98
pmd	2518	0.99	0.99	0.98	0.99	0.99	1.43	0.99	0.98
ps	2085	0.99	0.98	0.97	0.98	0.97	1.86	0.98	0.97
xalan	2328	0.97	0.99	0.96	0.98	0.96	1.26	0.97	0.97
pseudojbb	2115	1.02	1.03	1.01	0.98	1.01	1.29	1.01	0.98
max	2667	1.02	1.03	1.01	0.99	1.02	1.86	1.02	0.98
min	369	0.92	0.96	0.88	0.90	0.90	1.26	0.90	0.90
geomean	**1961**	**0.98**	**0.99**	**0.97**	**0.97**	**0.98**	**1.46**	**0.98**	**0.97**
C									
bzip2	1255	0.98	0.98	0.97	0.98	0.97	1.50	0.98	0.98
crafty	46	1.00	0.99	0.99	0.99	0.99	1.17	0.99	0.99
gap	755	0.99	1.00	0.97	0.99	0.98	1.31	0.99	0.98
gcc	21	0.99	1.00	0.99	0.99	0.99	1.25	0.99	0.99
gzip	952	0.98	0.99	0.96	0.97	0.97	1.38	0.97	0.97
parser	1799	0.98	0.99	0.98	0.98	0.99	1.36	0.98	0.99
twolf	962	0.99	0.99	0.99	0.99	0.99	1.46	0.99	0.99
vortex	1365	0.98	0.99	0.98	0.98	0.97	1.51	0.98	0.97
max	1799	1.00	1.00	0.99	0.99	0.99	1.51	0.99	0.99
min	21	0.98	0.98	0.96	0.97	0.97	1.17	0.97	0.97
geomean	**462**	**0.99**	**0.99**	**0.98**	**0.98**	**0.98**	**1.36**	**0.98**	**0.98**

Table 4. L1 Transient Cache Policies: 4 lines (∼512B); default L1: 128B line, 32KB, 2-way, 2-cycle; L2: write back

basic blocks than C, and indirect method calls. These features make it less predictable and therefore Java must relies more on hardware speculation to achieve instruction level parallelism.

C and Java differ substantially on their relative mix of load and store instructions (columns three and four). Java performs slightly more memory operations on average than (38% versus 34% of total operations), but only 25% of memory operations in Java are stores, whereas in C, 33% are stores. This result is consistent with C's reuse of temporary storage, whereas Java allocates a new object, writes, reads, and discards it. This result also indicates why filtering write instantiated misses is relatively more effective on Java than C–in C these lines are relatively more likely to be misses than in Java. Columns five through seven show miss rates and store forwarding, where C and Java behave similarly, although Java shows enormous variability in L2 miss rate. Together with the load/store behavior, these results indicate that for the most part, it is the particular access patterns, rather than the gross statistical properties of these workloads that make Java amenable to improvements from transient caches (except for the outlier, db which has a 13.6% L2 miss rate).

The *rel m/r* columns show the relative L1 and L2 miss rates of the parent cache with a transient cache as a fraction of the base PPC cache. The transient cache both usually improves the L1 miss rate, but sometimes degrades the L1 miss rate, and often degrades the L2 miss rate. Performance is tolerate to this variation because of the high hit rates in the in the L1 transient cache for both reads (63% for Java and C) and writes (92% for Java and 89% for C). These hits of course provide single cycle access time.

The L2 transient cache has a substantially different behavior. Writes hit in the L2 TQ 85% of the time, and reads hit 28% for Java, but only 1% for C programs. These results are consistent with the wider window of access due to object streams that may not get captured due to filtering at the L1 TQ, but are then quickly brought closer to the CPU. These results suggest exploring the addition of victim cache functionality of the L2 transient cache, by including a pathway for instantiating L1 victims. The columns labeled *filter*, show that the percent of lines the filtering mechanism actually excludes from the parent cache is low (2% for Java, 3% for C in the L1) and higher for the L2 (4% for Java, and 10% but up to 30% for C). These results show that even F1, the more aggressive filtering, is pretty conservative and leaves open the door for an accurate but more aggressive policy.

We also measured how the behavior of the no-fetch policy implemented with validity bits in the transient cache (not in the table). Lines instantiated on a write misses almost never ($< 0.1\%$) require a fetch for the reaming bytes for both C and Java. Even for C programs that have slightly fewer write hits in the TQ, programs virtually never read unwritten data from a write miss line.

In summary, these results show that in our PPC model most of the benefits come from the direct benefits of one-cycle access times, and there are modest gains from improved cache efficiency due the filtering policies that prevent pollution of the parent caches. In a system with more modest L1 to L2 bandwidth, the benefits of write buffering (which was neutral in our model) and write-no-fetch may be more pronounced.

5. Related Work

This section differentiates our work from prior work on cache design and object streams.

Level-0 Caches. Kin et al. show that adding a tiny 128 to 256 byte direct-mapped L0 cache in front of a traditional L1 reduces energy consumption substantially [20]. This savings comes at reduced performance. A fully-associative L0 improves performance over a direct-mapped L0, but uses more power. The combination does not attain energy reductions or performance improvements

program	64B	Default Filtering (F0)				Agressive Filtering (F1)		
		RQ	TM	TQ8	TQ2/6	TM	TQ8	TQ2/6
Java								
_201_compress	1.00	0.86	0.88	0.88	0.87	0.88	0.88	0.87
_202_jess	1.00	0.95	0.97	0.94	0.94	0.97	0.94	0.94
_205_raytrace	1.01	0.98	0.99	0.97	0.97	0.99	0.97	0.97
_209_db	0.94	0.92	0.93	0.90	0.90	0.93	0.93	0.90
_213_javac	1.00	0.96	0.96	0.95	0.95	1.01	0.96	0.95
_222_mpegaudio	1.00	0.92	0.98	0.94	0.93	0.98	0.94	0.93
_228_jack	1.00	0.99	1.00	0.97	0.97	1.00	0.99	0.97
antlr	1.00	0.96	0.97	0.96	0.96	0.97	0.96	0.96
fop		0.97			0.96		0.97	
hsqldb	1.00	0.97	0.98	0.97	0.97	0.98		0.97
pmd	1.01	0.97	0.99	0.98	0.97	1.00	0.98	0.97
ps	0.99	0.96	0.97	0.96	0.96	0.97	0.96	0.96
xalan	1.02	0.99	1.00		1.00		0.99	1.00
pseudojbb	1.00	1.00	1.01	0.97	0.98	1.01	1.00	0.97
max	1.02	1.00	1.01	0.98	1.00	1.01	1.00	1.00
min	0.94	0.86	0.88	0.88	0.87	0.88	0.88	0.87
geomean	**1.00**	**0.96**	**0.97**	**0.95**	**0.95**	**0.97**	**0.96**	**0.95**
C								
bzip2	0.99	0.93	0.96	0.95	0.95	0.96	0.95	0.95
crafty	1.00	0.98	0.99	0.99	0.99	0.99	0.99	0.99
gap	1.01	0.98	1.00	0.98	0.98	1.00	0.98	0.98
gcc	1.00	0.98	0.98	0.98	0.98	0.98	0.98	0.98
gzip	1.00	0.96	0.97	0.96	0.96	0.97	0.96	0.96
parser	1.03	1.01	0.99	0.99	0.99	0.99	0.99	0.99
twolf	1.01	0.99	0.99	0.99	0.99	1.00	0.99	0.99
vortex	0.99	0.96	0.96	0.96	0.97	0.96	0.96	0.97
max	1.03	1.01	1.00	0.99	0.99	1.00	0.99	0.99
min	0.99	0.93	0.96	0.95	0.95	0.96	0.95	0.95
geomean	**1.00**	**0.98**	**0.98**	**0.98**	**0.98**	**0.98**	**0.98**	**0.98**

Table 5. L1 Transient Caches with 64B line: 8 lines (∼512B); L1: **64B line**, 32KB, 2-way, 2-cycle; L2: write back

over the base cache. Our work uses a similarly sized structure, but attains performance improvements instead of degradations due to seeking it in parallel (which is less power efficient) and filtering. In addition, the per-word valid bits save fetch bandwidth.

A technical report by Srinivasan et al. suggests seeking in parallel on a relatively large (4KB 2-way) one-cycle L0 with conventional L0 inclusive replacement policies [34]. The one-cycle access time goal is the same, but transient caches add per-byte tags, full associativity, and filtering to attain similar or higher improvements with a much smaller structure (1/8 the size).

Spatial and Temporal Caching. Cache designers have long differentiated temporal and spatial locality to improve performance with stream buffers [24] and prefetching [15]. A number of researchers have proposed tracking locality and exploiting spatial and temporal locality differently to improve performance by improving efficiency in each structure [14, 18, 22, 36]. Most of these do not use cycle accurate simulation. All of these approaches differ substantially from our work because they require modifying all levels of the hierarchy to include history bits on cache lines and/or tables that track program history. Transient caches do not strictly define a type of locality. Transient caches keep limited state only in the transient cache itself, are localized, require only additional logic for the parent, and require *no additional state* in the parent.

Object Streams. Because the collector is a key part of Java performance, researchers have examined its performance separately and together with the application [3, 19, 31]. For example, generational collectors with copying nurseries provide good locality as well as object stream behavior [3]. The collector itself has object stream behavior coupled with a higher miss rate than the application because when invoked, it touches reachable data that may not be in the program's current working set [19, 31].

Diwan et al. first noted that highly allocating ML programs with contiguous nursery allocation benefit from write through allocate no-fetch policy, as found in DECStation 5000/200 [8]. Jouppi found that even C programs benefited from this structure [16], and per-word valid bits in a write-buffer could attain some of the benefits of write through no-fetch while avoiding the prohibitive cost of valid bits in the cache itself. Transient caches subsumes these benefits and satisfy reads for one-cycle access times. Hölzle and Ungar [12] confirm Diwan et al. results for object-oriented programs, and suggest write-no-allocate is a bad idea, which we contradict. They suggest that object oriented programs behave similarly to C and thus require no special architectural features, whereas we find differences that new memory system features can exploit.

Another approach to object streams is using special instructions. For example, PowerPC **`dcbz`** instruction clears a cache line and the memory allocator and compiler can use it when it can guarantee the entire line will always be written before read. This approach exposes the problem to both the underlying VM and the architecture, and thus limits its applicability. The transient cache designs achieve the same effect without a special instruction, can achieve this effect when the compiler can not make the guarantee (e.g., in C programs with free-list allocators), and applies judicious filtering which further improves over a special instruction.

Cache Coherency in Multiprocessors. Marden et al. find a *processor consistency* model, in which a write-miss proceeds without stalling and only fetches on a read miss to an invalid word, attains better performance than *sequential consistency* [23]. Implementing our design in multiprocessors requires a similar consistency model. Mounes-Toussi and Lilja consider write buffer design for a cache-coherent multiprocessor, and show that a write through L1 cache with a write buffer offers a good performance point, and is not inconsistent with attaining cache coherence [25]. Thus a multiprocessor design, with valid bits and a processor consistency model that does not require a fetch on a write-miss, is compatible with our

	128B line size								64B line size			
	PPC	Perfect		Reduced L1		L1 TQ4 F1; L2 TQ16 F1				L1 TQ8 F1; L2 TQ32 F1		
				2-way	D-M	2-way	2-way	D-M		2-way	2-way	D-M
programs	cycle(M)	L1	L2	16KB	32KB	32KB	16KB	32KB	default	32KB	16KB	32KB
Java												
_201_compress	1701	0.81	0.99	1.11	1.02	0.88	0.93	0.89	1.00	0.86	0.89	0.86
_202_jess	2061	0.95	0.99	1.04	1.05	0.97	1.01	1.01	1.00	0.95	0.97	0.96
_205_raytrace	2260	0.88	0.97	1.02	1.02	0.97	0.99	0.98	1.01	0.96	0.98	0.97
_209_db	2593	0.51	0.62	1.04	1.05	0.91	0.90	0.92	0.94	0.85	0.83	0.85
_213_javac	2430	0.96	0.98	1.04	1.07	0.97	0.98	0.99	1.00	0.95	0.95	0.96
_222_mpegaudio	1520	0.99	1.00	1.01	1.06	0.96	0.96	0.96	1.00	0.94	0.94	0.94
_228_jack	2667	0.99	0.98	1.04	1.04	1.00	1.01	1.02	1.00	0.99	0.99	1.00
antlr	369	0.96	0.98	1.01	1.02	0.97	0.98	0.97	1.00	0.95	0.97	0.97
fop	2424	0.96	0.97	1.02	1.03		0.99	0.99		0.97	0.97	
hsqldb	2665	0.95	0.97	1.01	1.02	0.98	0.99		1.00	0.96	0.97	0.97
pmd	2518	0.92	0.95	1.04	1.04	0.99	1.00	1.00	1.01	0.97	0.98	0.99
ps	2085	0.94	0.98	1.09	1.05	0.98	1.02	1.00	0.99	0.96	1.00	0.98
xalan	2328	0.79	0.85	1.01	1.03		0.98	0.96	1.02	0.97	0.99	0.98
pseudojbb	2115	0.98	0.97	1.05	1.05	1.00	0.98	1.01	1.00	0.99	0.97	1.00
max	2667	0.99	1.00	1.11	1.07	1.00	1.02	1.02	1.02	0.99	1.00	1.00
min	369	0.51	0.62	1.01	1.02	0.88	0.90	0.89	0.94	0.85	0.83	0.85
geomean	**1961**	**0.89**	**0.94**	**1.04**	**1.04**	**0.96**	**0.98**	**0.98**	**1.00**	**0.95**	**0.96**	**0.95**
C												
bzip2	1255	0.80	0.85	1.02	1.05	0.99	1.00	1.00	0.99	0.92	0.93	0.93
crafty	46	0.96	0.98	1.05	1.02	0.98	1.00	1.00	1.00	0.98	1.00	0.99
gap	755	0.94	0.96	1.01	1.03	0.97	0.98	0.98	1.01	0.97	0.97	0.98
gcc	21	0.98	1.00	1.03	1.01	0.99	1.02	1.00	1.00	0.99	1.00	1.00
gzip	952	0.82	1.00	1.05	1.03	0.97	1.01	0.98	1.00	0.96	1.01	0.97
parser	1799	0.88	0.94	1.02	1.02	0.96	0.98	0.98	1.03	0.96	0.98	0.97
twolf	962	0.95	1.00	1.04	1.05	0.99	1.02	1.01	1.01	0.99	1.02	1.00
vortex	1365	0.97	0.99	1.02	1.03	0.97	0.99	0.99	0.99	0.96	0.97	0.97
max	1799	0.98	1.00	1.05	1.05	0.99	1.02	1.01	1.03	0.99	1.02	1.00
min	21	0.80	0.85	1.01	1.01	0.96	0.98	0.98	0.99	0.92	0.93	0.93
geomean	**462**	**0.91**	**0.96**	**1.03**	**1.03**	**0.98**	**1.00**	**0.99**	**1.00**	**0.97**	**0.99**	**0.98**

Table 6. L1 & L2 Transient Caches: default L1: 32KB, 128B line, 2-way, 2-cycle; L2: write through, 512KB, 128B line, 8-way

design. However, we leave a detailed exploration of the transient cache design for cache-coherent multiprocessors to future work.

6. Conclusion

This paper examines the interaction of modern programming languages and modern architectures. We introduce the term *object stream* to describe how Java programs march through memory creating a stream of short-lived objects that programs write and read in quick succession, and then often never access again. These results motivate a new cache design with a policy that instantiates write and read misses, but only promotes them to the parent cache if they demonstrate a degree of reuse beyond their initial miss. Because these lines live in the transient cache, rather than for instance, at the head of a stream buffer, they can have richer reuse patterns than stream buffers. Extensive experimental results show that the transient cache is so effective that the parent cache can shrink to half its original capacity or half its associativity, and yet the transient cache can still achieve performance improvements compared with the original larger cache.

By choosing a base architecture with a two-cycle L1 latency, we have been conservative about the potential value of a single-cycle transient cache. We performed (but did not report here) experiments for three-cycle and five-cycle L1 latencies, where as exepected: these mechanisms perform even better. It is unclear whether or not technology trends will lead to higher latency L1 caches, but the trends at the L2 do seem to be for larger higher access times. Our design complicates processor pipeline design a bit, since a transient cache load hit is a single cycle and the L1 in two or more, and always predicting the shorter latency for loads is slightly problematic since about 62% hit. Although we focused our policy exploration on the L1 here, we also showed configurataions that are effective at L2, where it may become increasingly important. Trends toward higher latency caches and memory, and toward languages such as Java and C# suggest that these types of mechanisms for mitigating the effects of object streams will be increasingly important.

References

[1] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in java. In *ACM Conference Proceedings on Object–Oriented Programming Systems, Languages, and Applications*, Denver, CO, Nov. 1999.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the ACM Conference on Measurement & Modeling Computer Systems*, pages 25–36, NY, NY, June 2004.

[4] S. M. Blackburn, K. S. McKinley, J. E. B. Moss, S. Augart, E. D. Berger, P. Cheng, A. Diwan, S. Guyer, M. Hirzel, C. Hoffman, A. Hosking, X. Huang, , R. Garner, A. Khan, P. McGachey, D. Stefanović, and B. Wiedermann. The DaCapo benchmarks. Technical report, 2004. http://ali-www.cs.umass.edu/-DaCapo/Benchmarks.

[5] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[6] T. Chen and J. Baer. Effective hardware based data prefetching. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[7] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming*, June 1999.

[8] A. Diwan, D. Tarditi, and J. E. B. Moss. Memory subsystem performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 13(3):244–273, Aug. 1995.

[9] D. B. et al. The microarchitecture of the Pentium 4 processor on 90nm technology. *The Intel Technology Journal*, 8(1), Feb. 2004.

[10] J. Fenn and A. Lindon. Gartner's hype cycle special report. Technical report, 2004. http://www.gartner.com/.

[11] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and

	Base L1 Cache (%)						Transient Caches: L1 TQ8 F1; L2 TQ32 F1								
		load	store	L1	L2	store	rel	L1 (%)				L2 (%)			
program	IPC	inst	inst	m/r	m/r	fwd	perf	rel m/r	filter	r hit	w hit	rel m/r	filter	r hit	w hit
Java															
_201_compress	1.175	35.5	7.0	5.6	0.4	1.7	0.86	0.83	4.0	73.4	88.0	1.08	2.4	13.1	85.7
_202_jess	0.970	29.4	9.4	1.6	0.5	4.2	0.95	0.93	1.1	64.1	93.7	1.40	7.4	47.3	90.4
_205_raytrace	0.885	33.9	12.9	2.9	0.6	7.1	0.96	1.15	0.9	55.2	93.6	0.91	3.4	26.5	90.9
_209_db	0.771	29.9	5.8	4.8	13.6	0.5	0.85	1.08	1.3	52.1	87.7	1.09	2.6	4.5	88.2
_213_javac	0.823	27.3	13.7	1.2	0.5	6.8	0.95	0.91	1.3	67.0	91.7	1.33	2.8	36.1	90.7
_222_mpegaudio	1.315	33.2	6.6	0.3	0.0	2.7	0.94	1.30	0.9	62.1	93.1	12.19	5.5	42.9	89.8
_228_jack	0.750	25.9	11.5	1.6	0.6	4.7	0.99	0.75	2.5	69.2	92.2	1.77	4.8	35.5	77.2
antlr	0.869	27.3	11.8	1.2	1.1	8.0	0.95	0.93	2.1	68.1	93.4	1.48	3.7	37.6	87.8
fop	0.825	27.9	13.0	1.4	0.8	11.6	0.97	0.91	2.2	66.6	94.1	1.15	3.3	38.3	86.5
hsqldb	0.750	26.5	10.4	1.8	0.9	4.0	0.96	0.96	2.0	63.5	91.9	1.56	3.8	34.6	85.5
pmd	0.794	28.7	11.6	2.1	1.3	4.7	0.97	0.93	2.0	58.6	92.0	1.47	3.8	34.2	85.4
ps	0.959	28.6	6.2	2.1	0.9	2.8	0.96	0.79	2.9	46.1	89.5	1.31	5.5	30.0	88.5
xalan	0.859	22.4	9.0	3.1	4.6	2.0	0.97	1.19	3.5	66.1	92.0	1.49	6.9	27.7	75.8
pseudojbb	0.946	21.4	10.4	1.3	0.9	8.9	0.99	0.93	2.5	71.4	94.1	1.17	5.3	39.7	80.8
max	1.315	35.5	13.7	5.6	13.6	11.6	0.99	1.30	4.0	73.4	94.1	12.19	7.4	47.3	90.9
min	0.750	21.4	5.8	0.3	0.0	0.5	0.85	0.75	0.9	46.1	87.7	0.91	2.4	4.5	75.8
geomean	**0.894**	**28.2**	**9.6**	**1.8**	**0.7**	**3.9**	**0.95**	**0.96**	**1.9**	**62.6**	**91.9**	**1.53**	**4.1**	**28.4**	**85.8**
C															
bzip2	1.246	21.7	9.4	3.4	3.2	2.5	0.92	0.83	8.0	66.9	91.5	0.85	29.8	0.6	84.6
crafty	0.948	17.7	8.4	2.2	0.5	1.6	0.98	0.68	1.9	53.0	90.9	1.07	34.9	0.7	87.0
gap	0.930	24.0	13.0	0.8	0.8	6.7	0.97	1.07	2.0	71.3	90.1	1.26	24.6	0.8	88.6
gcc	0.858	23.4	14.2	1.6	0.2	14.5	0.99	0.86	2.8	68.6	86.8	1.94	29.6	1.0	91.5
gzip	1.139	22.4	8.3	7.6	0.1	2.2	0.96	1.01	5.3	64.8	89.8	1.09	2.9	0.8	79.2
parser	1.111	23.5	11.9	1.9	0.9	13.0	0.96	1.19	1.5	69.9	95.2	1.68	10.9	1.9	83.0
twolf	0.847	28.0	11.2	2.7	0.0	3.8	0.99	1.19	2.5	53.5	81.1	2.37	0.1	3.5	69.1
vortex	0.915	23.8	17.9	1.0	0.1	1.0	0.96	0.70	1.2	62.3	89.4	1.16	21.2	0.7	95.2
max	1.246	28.0	17.9	7.6	3.2	14.5	0.99	1.19	8.0	71.3	95.2	2.37	34.9	3.5	95.2
min	0.847	17.7	8.3	0.8	0.0	1.0	0.92	0.68	1.2	53.0	81.1	0.85	0.1	0.6	69.1
geomean	**0.990**	**22.9**	**11.4**	**2.1**	**0.2**	**3.8**	**0.97**	**0.92**	**2.6**	**63.4**	**89.3**	**1.35**	**9.7**	**1.0**	**84.4**

Table 7. Transient cache and read/write statistics for L1 TQ8 F1; L2 TQ32 F1; or why the transient cache improves performance

P. Roussel. The microarchitecture of the Pentium 4 processor. *The Intel Technology Journal*, Q1, July 2001.

[12] U. Hölzle and D. Ungar. Do object-oriented languages need special hardware support? In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 283–302, London, UK, 1995. Springer-Verlag.

[13] X. Huang, J. E. B.Moss, K. S. McKinley, S. Blackburn, and D. Burger. Dynamic simplescalar: Simulating java virtual machines. Technical Report TR-03-03, University of Texas at Austin, Department of Computer Sciences, Feb. 2003.

[14] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time spatial locality detection and optimization. In *International Symposium on Microarchitecture*, pages 57–64, 1997.

[15] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990.

[16] N. P. Jouppi. Cache write policies and performance. In *International Symposium on Computer Architecture*, pages 191–202, San Diego, CA, May 1993.

[17] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in two-level on-chip caching. In *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*, pages 34–45, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[18] M. Karlsson and E. Hagersten. Timestamp-based selective cache allocation. In *the Workshop on Memory Performance Issues*, Gatteborg, Sweden, June 2001.

[19] J. Kim and Y. Hsu. Memory system behavior of Java programs: methodology and analysis. In *Proceedings of the ACM Conference on Measurement & Modeling Computer Systems*, pages 264–274, Santa Clara, California, June 2000.

[20] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *International Symposium on Microarchitecture*, pages 184–193, 1997.

[21] A. Kleinosowski and D. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research, 2002.

[22] J. Lee, J. Lee, and S.Kim. A selective temporal and aggressive spatial cache system based on time intervan. In *IEEE International Conference on Computer Design*, Austin, TX, Sept. 2000.

[23] M. Marden, S. Lu, K. Lai, and M. Lipasti. Comparison of memory system behavior in java and non-java commercial workloads. In *Proceedings of the Workshop on Computer Architecture Evaluation using Commercial Workloads*, Boston, MA, Feb. 2002.

[24] S. A. McKee and W. A. Wulf. A memory controller for improved performance of streamed computations on symmetric multiprocessors. In *International Conference on Parallel Processing*, Honolulu, HI, Apr. 1996.

[25] F. Mounes-Toussi and D. J. Lilja. Write buffer design for cache-coherent shared-memory multiprocessors. In *IEEE International Conference on Computer Design*, pages 506–511, Austin, TX, 1995.

[26] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, Nov. 2004.

[27] F. P. O'Connell. Personal communication, Jan. 2006.

[28] F. P. O'Connell and S. W. White. Power3: The next generation of PowerPC processors. *IBM Journal of Research and Development*, 44(6), 2000.

[29] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *International Symposium on Computer Architecture*, pages 24–33, Chicago, IL, Apr. 1994.

[30] D. G. Perez, G. Mouchard, and O. Temam. MicroLib: A case for the quantitative comparison of micro-architecture mechanisms. In *International Symposium on Microarchitecture*, pages 43–54, Portland, OR, Dec. 2004.

[31] A. Rajan, S. Hu, and J. Rubio. Cache performance in java virtual machines: A study of constituent phases. In *IEEE International Workshop on Workload Characterization*, Nov. 2002.

[32] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *International Symposium on Microarchitecture*, Monterey, California, Dec. 2000.

[33] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *International Symposium on High-Performance Computer Architecture*, pages 144–155, Feb. 1997.

[34] V. Srinivasan, M. Charney, G. Tyson, and E. Davidson. Recovering single cycle access of primary caches. Technical Report CSE-TR-425-00, University of Michigan, Apr. 2000.

[35] J. Stokes. PowerPC 970. Technical report, 2005. http://arstechnica.com.cpu/-03q1/ppc970/ppc970-8.html/.

[36] E. S. Tam, J. Rivers, V. Srinivasan, G. Tyson, and E. Davidson. Active management of data caches by exploiting reuse information. *ACM Transactions on Computer Systems*, 48(11):1244–1259, Nov. 1999.

[37] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, April 1984.

[38] Y. Zheng, B. Davis, and M. Jordan. Performance evaluation of exclusive cache hierarchies. In *2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–96, 2004.