HICAMP: Architectural Support for Efficient Concurrency-safe Shared Structured Data Access

David Cheriton * Amin Firoozshahian Alex Solomatnikov John P. Stevenson † Omid Azizi
Hicamp Systems, Inc.
{cheriton, aminf13, sols, jps, oazizi}@hicampsystems.com

Abstract

Programming language and operating system support for efficient concurrency-safe access to shared data is a key concern for the effective use of multi-core processors. Most research has focused on the software model of multiple threads accessing this data within a single shared address space. However, many real applications are actually structured as multiple separate processes for fault isolation and simplified synchronization.

In this paper, we describe the HICAMP architecture and its innovative memory system, which supports efficient concurrency safe access to structured shared data without incurring the overhead of inter-process communication. The HICAMP architecture also provides support for programming language and OS structures such as threads, iterators, read-only access and atomic update. In addition to demonstrating that HICAMP is beneficial for multi-process structured applications, our evaluation shows that the same mechanisms provide substantial benefits for other areas, including sparse matrix computations and virtualization.

Categories and Subject Descriptors B.3.2 [*Hardware*]: Memory Structures—associative memories, cache memories; C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors)—parallel processors

General Terms Design, Performance, Reliability

Keywords Parallel Programming, Parallel Architecture, Memory Model, Snapshot Semantics, Memory Sharing, Fault Isolation, Memory Deduplication, Iterator Register

1. Introduction

Programming language and operating system support for efficient concurrency-safe access to shared data is a key issue to address to achieve the effective use of multi-core processors. Most of the focus in this area has been on the multithreaded model, where multiple threads within a single address space concurrently access shared data in memory. However, this model introduces *non-deterministic* execution because of arbitrary interleaving of threads accessing shared memory [25]. The programmer thus needs to carefully orchestrate access to shared data using mechanisms such as locks, monitors or transactions to prevent corruption of shared state. Various programming language constructs and checking tools have been developed to reduce the associated problems. However, software bugs can still cause the entire application or system to fail and these bugs are hard to reproduce and thus difficult to track down.

A common approach is to partition the application system into multiple separate client processes and one or more shared state-managing processes, connected by an *inter-process communication* (IPC) mechanism such as sockets. For example, a web site is usually implemented as multiple web server processes accessing shared state stored in a database server, running as a separate process. To provide sufficient read bandwidth, many high traffic websites extend this model further by instantiating one or more memcached [15] processes that serve as a fast cache of common database queries, i.e., the common parts of dynamically generated web-pages[1]. A similar structure is found in many other web and database applications as well as embedded systems [5].

This multi-process structure provides strong fault isolation between the client processes and the servers, preventing the clients from corrupting the shared state and allowing restart of a failed client. This structure also allows client application code to be sequential, delegating concurrency control to these database/memcached servers, reducing the cost of development and improving the reliability of this software. It also allows the application to scale seamlessly across a cluster of networked processors, beyond the limits of a single physical server.

The disadvantage of this approach is the overhead: a client process needs to generate a message from its internal state, transmit the message over the socket to the server process, wait for the server's reply, extract information from server message and transform it back to the internal representation. Similar overhead is incurred on the server side as well as overheads for concurrency control, logging and buffer management. In fact, Stonebraker and Catte [28] estimate that only 13 percent of CPU cycles perform the actual work of a database when it is running completely in memory. As the number of cores on a chip increases, it becomes possible to run many processes on the same chip, yet these overheads, being memory-intensive, can be expected to become proportionally worse.

The **HICAMP (Hierarchical Immutable Content Addressable Memory Processor)** architecture provides an alternative to both shared memory and "database" models by implementing, in hardware, a memory abstraction of protected sharable segments

* also a professor at Stanford University

† also a Ph.D candidate at Stanford University

ASPLOS'12, March 3–7, 2012, London, England, UK.

[1] memcached is widely used by many popular high-traffic web-sites (Facebook [27], YouTube, Wikipedia, LiveJournal, etc.).

that can be "snapshotted" and atomically updated efficiently. This abstraction allows efficient, fault-tolerant, concurrency-safe access to shared data by multiple threads without the loss of hardware-enforced isolation, yet without the conventional overheads of an application structured as multiple processes. HICAMP also incorporates memory ***deduplication***, which further reduces copy overhead and improves cache and DRAM utilization.

In this paper, we describe the HICAMP architecture: how it supports key programming language constructs, including shared objects, iterators and atomic updates, how it simplifies operating systems for multi-core processors, and how it is amenable to a cost-effective implementation. We further describe how this novel memory architecture provides substantial benefits to other application domains, providing significant improvements in memory performance of applications ranging from web software to sparse matrix computations and virtual machine hosting. The latter allows HICAMP to run legacy software.

2. HICAMP Architecture

The HICAMP memory architecture introduces three key concepts:

1. *Content-unique lines:* memory is considered to be an array of small, fixed-size lines, where each line is addressed by a *Physical Line ID* (PLID), and has unique content that is immutable over its lifetime.
2. *Memory segments:* memory is accessed as a number of segments, where each segment is structured as a *Directed Acyclic Graph* (DAG) of memory lines, i.e. logically a tree but with potentially shared lines due to the content-uniqueness property of lines.
3. *Virtual segment map:* a mapping from *Virtual Segment IDs* (VSID) that identify memory segments for applications, to the root PLID of the DAG representing the segment contents.

The following subsections elaborate on these concepts.

2.1 Content-Unique Lines

The HICAMP main (DRAM) memory is divided into *lines*, each with a fixed size, such as 16, 32 or 64 bytes. Each line has unique content that is immutable during its lifetime. Uniqueness and immutability of lines is guaranteed and maintained by a duplicate suppression mechanism in the memory system. In particular, the memory system can either read a line by its PLID (similar to a reading a line in a conventional memory system using its address) or lookup a line by content (instead of writing), returning the PLID. The lookup-by-content operation allocates a line and assigns it a new PLID if a line with the specified content is not already present. When the processor needs to modify a line to effectively write new data into memory, it requests a PLID for a line with the specified (modified) content[2]. The *zero PLID* corresponds to *zero line* making it easy to identify any zero line references. It also provides a space-efficient representation of sparse data structures.

The PLIDs are a hardware-protected data type that can only be created in memory or registers from the return value of a lookup-by-content operation or by copying an existing PLID value. This protection is implemented by per-word tags that indicate whether the content contains a PLID[3]. This protection of PLIDs is necessary so that hardware can track the sharing of lines accurately. It has the further benefit of providing protected references; an application thread can only access content that it has created or for which the PLID has been explicitly passed to it.

2.2 Segments

A *segment* in HICAMP is a variable-sized, logically contiguous region of memory. It is represented as a DAG of fixed size lines with data elements stored at the leaf nodes, as illustrated in Figure 1. Figure 1a shows two memory segments representing two strings, labeled "First string" and "Second string" with the second one being a substring of the first. Note that the second string shares all the lines of the first string, given the latter is a substring of the former.

Each segment follows a canonical representation in which leaf lines are filled from the left to right. As a consequence of this rule and the duplicate suppression by the memory system, each possible segment content has a unique representation in memory. In particular, if the character string of Figure 1a is instantiated again by software, the result is a reference to the same DAG, which already exists. In this way, HICAMP extends the content-uniqueness property from lines to segments. Furthermore, two segments in HICAMP can be compared for equality by a simple comparison of their root PLIDs, independent of segment size. For example, two web pages represented as HICAMP strings (array of characters) can be compared in a single compare instruction.

When content of a segment is modified by creating a new leaf, the PLID of the new leaf replaces the old PLID in its parent line, requiring the content of parent line to change, and thus a different PLID for the parent line, thus requiring the parent of the parent to change, and so on. Consequently new PLIDs replace the old ones all the way up to the root of the DAG.

Each segment in HICAMP is *copy-on-write* because of the immutability of the memory lines. Consequently, passing a segment reference to another thread effectively guarantees a stable snapshot of the segment content at essentially no cost. Exploiting this property, concurrent threads can efficiently execute with *snapshot isolation* [6]; each thread simply needs to save the root PLIDs of all segments of interest. This isolation means that atomic updates do not have read-write conflicts, only write-write conflicts, which are substantially less common. In particular, it allows long running, read-only transactions to be executed without serialization overhead. For example, consider the task of calculating the balance of all bank accounts by bank database for audit or management purposes, by iterating over all accounts, using the state of the accounts at a given point in time. Concurrently, the state of accounts may be changing by ongoing customer transactions, a process which cannot be stalled. Modern databases support this form of query using the *consistent read* capability (effectively providing *snapshot semantics*) at the cost of copying and reverting database blocks to the start time of the long-running transaction. HICAMP supports a comparable facility in hardware largely eliminating this overhead.

A thread in HICAMP uses *non-blocking synchronization* to perform an atomic update of a segment by:

1. saving the root PLID for the original segment
2. modifying the segment and producing a new root PLID
3. using a *compare-and-swap* (CAS) instruction to atomically replace the original root PLID with the new root PLID, if the root

[2] A portion of the memory can operate in a conventional, non-deduplicated mode for memory regions that are expected to be modified frequently, such as thread stacks, which can be accessed with conventional read and write operations.

[3] The tags for words in the memory lines are stored in the ECC bits, using bits that are not needed for ECC at a line granularity (e.g. commodity DRAM chips with ECC contain 8 ECC bits per 64-bit of data yet for 16-byte lines 9 ECC bits are sufficient to implement standard *single-error-correcting-double-error-detecting* (SEC-DED) code [20]). Error detection can be further improved using the intrinsic properties of the HICAMP memory system as described in Section 3.1.

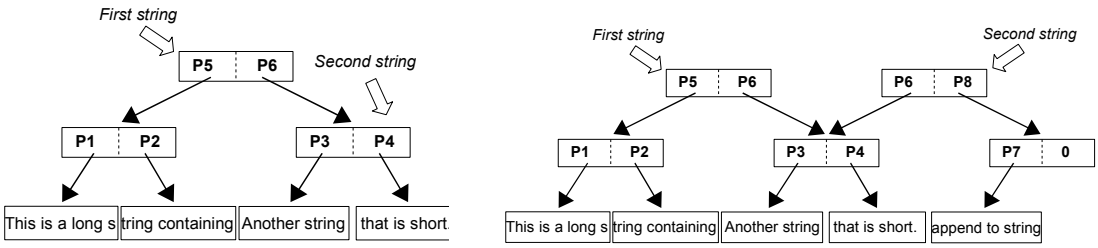


Figure 1: Two segment DAGs representing two strings

PLID for the segment has not been changed by another thread (and otherwise retrying as with conventional CAS).

In effect, the inexpensive (logical) copy and copy-on-write in HICAMP makes Herlihy's theoretical construction showing CAS as sufficient [21] actually practical to use in real applications. Because of the line-level duplicate suppression, HICAMP also maximizes the sharing between the original version of the segment and the new one. If the second string in Figure 1a is modified to add the extra characters "append to string", the memory then contains the segment corresponding to the second string, sharing all the lines of the original segment, simply extended with new lines to store the additional content and the extra parent lines necessary to form the DAG, as shown in Figure 1b.

2.3 Virtual Segment Map

A HICAMP segment is referenced in software using a *Virtual Segment ID* (VSID) that is mapped to the segment's root PLID, through a virtual segment map. Each virtual segment map is implemented either as a HICAMP array or in the conventional part of the memory, indexed by VSID and containing the following fields: `[rootPLID,height,flags]`. Flags are used to indicate read-only access, merge-update (described later) or to indicate a weak reference, i.e. a reference that should be zeroed when the segment is reclaimed, rather than prevent its reclamation.

Each software object corresponds to a segment. Software asks the memory system for a segment whenever it wants to create an object. When software needs one data structure to refer to another (e.g. object O1 needs to refer to O2), then an object's VSID is stored as the reference (O2 is represented by VSID S2; O1 stores S2 as one of its data members). When the contents of O2 are updated, the entry in the virtual segment map corresponding to S2 is updated to point to the new root PLID, and thus the other referencing objects (e.g. O1) do not have to change their references. VSIDs are protected by tags in memory, similar to PLIDs.

When the segment map itself is implemented as a HICAMP segment indexed by VSID, multiple segments can be updated by one atomic update/commit of the segment map. In particular, the revised segments are not visible to other threads until the commit of the revised segment map takes place. A segment map can be implemented in conventional memory, updating individual entries directly, when this property is not required.

There is no need for conventional address translation in HICAMP because inter-process isolation is achieved by the protected references. In particular, a process can only access data that it creates or to which it is passed a reference. Moreover, a reference (VSID) can be passed as read-only, restricting the process from updating the root PLID in the corresponding virtual segment map entry. Thus, a thread can efficiently share objects with other threads by simply passing the VSID, achieving the same protection as separate address spaces but without the IPC communication overheads forced by conventional virtual memory systems. Here, the segment mapping acts similar to a page table and virtual memory mechanism in a conventional architecture, but the "address translation" only occurs on the first access to the segment, not on every access to each data element.

3. Implementation

The HICAMP architecture is implementable using *conventional* DRAM, with reasonable hardware complexity and performance, avoiding any need for power-intensive hardware CAM memory, as is described in the following subsections.

3.1 Memory System

To implement *lookup by content*, the main memory is divided into hash buckets and each line is stored in a bucket corresponding to a hash of its contents (Figure 2). Each hash bucket is mapped to a single DRAM row. To further optimize the fetch by content, each hash bucket includes a secondary hash map of the line content to 8-bit *signatures* indicating the entries in the hash bucket, which could possibly contain the specified content. Signatures are stored in one line to reduce access latency (Figure 2). The lookup by content operation consists of the following steps:

1. calculate the hash and signature values
2. read the signature line in the corresponding hash bucket
3. compare the calculated and read signatures
4. if one or more signatures match, the corresponding data lines are read from memory and compared to the given content
5. if a data line matches, its PLID is returned
6. if there is no signature/data line match, the line is allocated and assigned a PLID, as follows:
   (a) an empty way in the hash bucket is found by checking the signature line for a zero signature
   (b) the PLID is assigned to be the *concatenation* of the way number and the hash bucket number (Figure 2)
   (c) the signature line is updated and written back to DRAM
   (d) the data content of the line is placed in the HICAMP cache

The common case of lookup by content incurs two line reads from DRAM when the line is present, i.e. read of the signature line and data line, because the probability of signature match for lines with different content is small[4]. When the line is not present, the common case of lookup by content requires a read of the signature line and a write of the signature byte. The data line itself is written

[4] For example, with twelve 16-byte data lines per hash bucket and 8-bit signatures the probability of false positives, i.e. a signature match without data line match, is less than 5%.

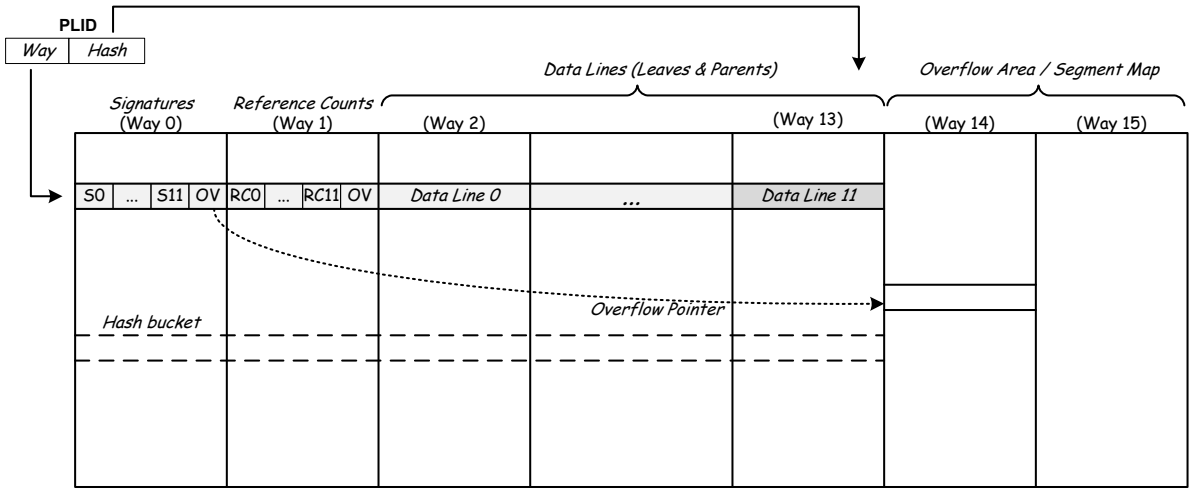


Figure 2: Main memory organization (16-byte lines)

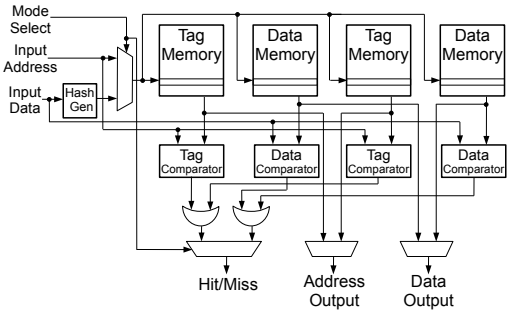


Figure 3: HICAMP cache

into the last level cache (LLC) and written into the main memory upon eviction from LLC if it is not de-allocated before then. Note that DRAM commands for performing the lookup operation access the same DRAM row, thereby minimizing DRAM command bandwidth, energy and latency. The line is written to a separate overflow area when the designated hash bucket is full.

Additionally, by recomputing the hash of the contents when a line is read from memory and comparing it to the hash bucket number where the line is read from, HICAMP can effectively increase the error-detection capabilities of the memory system.

The HICAMP cache, like the main HICAMP memory, supports two fundamental operations: read and lookup-by-content. Figure 3 shows the structure of the cache. The read operation accepts a PLID (i.e. the address) and returns a cache line of content. Thus, the read operation is essentially the same as in a conventional cache: the PLID is used to generate an index into the cache, tags of all ways within the indexed set are compared, and if a tag matches, the data is returned.

To support an efficient lookup-by-content operation, the HICAMP cache is designed such that each main memory hash bucket maps to only one cache set (lines from multiple hash buckets may map to a single cache set, but lines from a single hash bucket cannot map to multiple cache sets). This mapping is achieved simply by indexing the cache with a subset of the hash bits contained in the PLID. Because this indexing maps a hash bucket to exactly one cache set, the following key property is achieved: given the hash of a line content, only a single index in the cache needs to be searched. Thus, when a lookup operation is issued to the HICAMP cache, the hash of the line is first computed, and this provides an index into the cache set in which that data could be found. The data lines within that cache set are then compared with the provided content. If a match is found, the tag of the matching way is read and used to recompose the PLID, which is returned in response to the lookup-by-content operation.

For caches with high associativity, the lookup-by-content operation in the cache can be optimized by using tag bits to reduce the number of wide content comparisons required. Only a subset of the computed hash bits are used for indexing into the cache, thus the remaining hash bits are part of the cache tags. By comparing these tag bits to the calculated hash bits, the HICAMP cache can efficiently filter out ways that cannot contain the content.

Lines in HICAMP are reference counted by hardware, managed similar to a software implementation of reference counting. In particular, when a line reference count goes to zero, it is de-allocated by writing zeros to the corresponding signature. The line reference count is set to one when the line is created and is incremented by each lookup by content operation that matches to that line. The reference count of a line is decremented when a PLID reference to the line is overwritten, either in another line or in the virtual segment map. The most challenging aspect to implement in hardware is de-allocation of a line that recursively causes many other lines to be de-allocated; this is handled with a hardware state machine.

To reduce the number of DRAM operations, reference counts are also cached in the HICAMP cache. For example, when the line is allocated by lookup operation its reference count is written in the LLC and propagated to DRAM only when the line is evicted, if it was not de-allocated before that.

There is no traditional cache coherence problem for data lines because lines are immutable and can thus exist in multiple caches without conventional coherence protocol overheads. However, before an immutable data line can be de-allocated, it must be invalidated in all caches (e.g. using a protocol similar to invalidation-based broadcast or directory protocol). De-allocation is performed in the background and is not on the critical execution path; therefore, unlike cache coherent architectures, cache-line invalidation will not limit HICAMP's performance. However, the virtual segment map is mutable and will require a cache coherence protocol.

The question of scalability is a topic for future work. With no need for locks, semaphores, or cache coherence for data lines,

HICAMP accrues a significant scalability advantage over conventional shared memory architectures.

3.2 Path and Data Compaction

HICAMP applies both path and data compaction to each segment DAG to reduce the access cost (path length through the DAG) as well as the memory overhead of the segment DAG structure. *Path compaction* encodes the path to the line along with the PLID, storing it in the parent node, when a parent DAG node would otherwise have only one non-zero element (PLID). For example, in Figure 4a, the path to the leaf "Data" line is compacted into one entry in the root line, eliminating P2 and P3. The path is encoded using the unused bits of the non-zero PLID[5]. Note that the logical height of the DAG is stored in the corresponding virtual segment map entry and is not affected by such compaction.

HICAMP *data compaction* removes the most significant zero bytes from a word and inlines the resulting value directly into the parent line when possible. For example, in Figure 4b, S1 and S0 are values that can be represented as 32-bit values, and so are compacted into a 64-bit field in the parent line. More significantly, an array of small integers (less than 255) is compacted so that each element is stored in a single byte. Inserting a value that is larger only affects the compaction of that element and neighboring elements that can no longer be compacted.

Both forms of compaction can be applied multiple times in different levels and portions of the DAG, decreasing the number of memory reads to access data and reducing the DAG storage overhead. In particular, in a segment that contains a large number of zeroes, the interior nodes are compacted to provide an efficient sparse representation.

3.3 Iterator Registers

A HICAMP *iterator register* is an extended form of the conventional index or address register, extended to support efficient access to data in segments by storing the path through the DAG to its current line position (Figure 5). In HICAMP, each memory access is made through an iterator register. An ALU operation that specifies a source operand as an iterator register accesses the value of the current offset of the iterator register within the associated segment. A HICAMP processor ideally has a comparable number of iterator registers to the number of general-purpose registers in a conventional processor, e.g. 16 to 32 registers.

Upon initialization with a specific VSID and offset, an iterator register loads and caches the path to the element at the specified offset in the DAG. Upon increment of the iterator register, it moves to the next non-null element in the segment. In particular, the hardware detects if the next portion of the DAG refers to the zero line and moves the iterator register's position to the next non-zero memory line. The register only loads new lines on the path to the next element beyond those it already has cached. In the case of the next location being contained in the same line, no memory access is required to access the next element. Using the knowledge of the DAG structure, an iterator register can also automatically prefetch memory lines in response to sequential access to the segment.

Using an iterator register, sequential access to a segment representing a dense array is at most two times the number of lines of accessing the same segment stored in conventional memory system[6]. Moreover, in some cases, the overall cost of a HICAMP array can be less expensive because it does not need to be copied to be dynamically grown, as is typically the case in a conventional architecture. Also, with a sparse array such as a sparse matrix or associative array, the cache line references can be comparable or less, given the data and instruction overhead required for these software implementations and the deduplication of lines that HICAMP provides.

A store operation updates the current data element pointed to by the iterator register. To support efficient incremental updates, the iterator register caches an updated line as a *transient line*[7], namely a line in a pre-defined, per-processor area of the memory that operates outside of the normal duplicate-suppressed region. It also tags the update to indicate the requisite changes to the parent nodes on the path through the DAG. Then, after potentially multiple changes to a line (and potentially many lines), when the iterator register is "committed" by software, the transient lines are converted to permanent (duplicate-suppressed) lines using lookup by content operations, and a new root PLID for the modified segment is acquired, as described earlier. (An iterator register also supports an abort action, in which case the modifications are discarded, reverting the segment to the original state, i.e. old root PLID.)

Using the transient lines, the cost of allocating content-unique lines at the leaf and all the way up to the root of the DAG is deferred until the commit point, offering the ability to amortize this cost over many writes. In the extreme, a local temporary array can be created by a procedure using the transient lines, used and discarded and so never committed.

Overall, the iterator register provides a hardware construct that supports iterator concept in most modern programming languages as well as in SQL. Moreover, unlike conventional programming language constructs, it efficiently supports the *snapshot and atomic update semantics* of the SQL cursor.

3.4 Merge-update

HICAMP supports *merge-update* of segments to facilitate updates of high contention data structures such as maps, queues and counters. For example, if two non-conflicting entries are added concurrently to the same directory, represented as a segment, merge-update produces a segment that corresponds to the directory with these two entries added, rather than aborting one of the concurrent transactions. In more detail, if a segment is marked as merge-update, when a thread detects during a CAS that the root PLID has been modified by another thread (resulting in a new DAG for the segment), the system attempts to merge the thread's copy with the updated segment content by performing the following steps for each line offset:

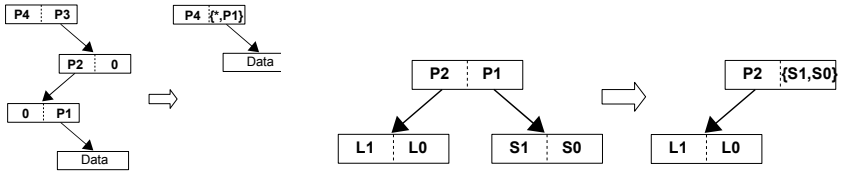
1. compute the difference between the "original" segment line and the "modified" segment line
2. apply this difference to the "current" line content of the segment
3. add this line to the "merged" segment at the current offset

If the segment contains a PLID field, as would be the case with a directory, the field in the "current" segment is required to be either the same as the "original" segment or the same as the "modified" segment. That is, the two updates cannot store distinct PLIDs in the same field; otherwise the merge-update fails. The uniqueness of segments and subsegments allows merge-update to efficiently identify sub-DAGs of the "current" and "modified" segment that are identical, and skip the line-by-line actions above for such a sub-DAG.

[5] Here, "unused" refers to high-order bits of the PLID not used because they are not needed to address the available DRAM. For example, with a 32-byte line, a 32-bit PLID is sufficient to access 128 gigabytes of DRAM.

[6] For example, with 16-byte lines and 64-bit PLIDs, the DAG overhead is at most 2x in terms of memory space and line reads for sequential access. With longer 32-byte or 64-byte lines this overhead is smaller.

[7] Transient memory lines require no cache coherency because the lines are allocated per core and converted to immutable lines before they are made visible to other cores.



(a) Path compaction (b) Data compaction

Figure 4: Compaction

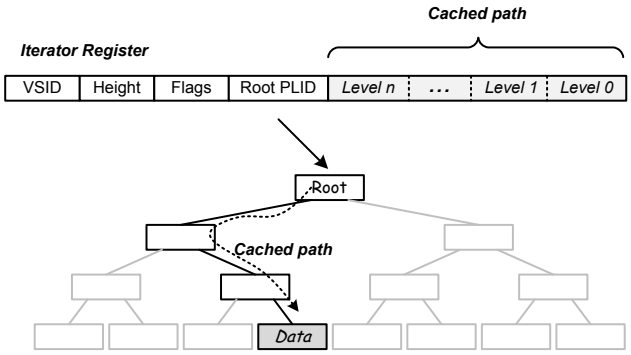


Figure 5: Iterator register

Merge-update can also apply to a segment of counters. In this case, the difference produces the increment applied by the thread, which is then added to the value in the "current" segment to produce the new count.

This capability can be exported to software level as an mCAS operation defined in pseudo-code as:

```
bool mCAS( old, curAddr, new ) {
   try {
     cur = *curAddr;
     while(!CAS(old,cur,new)) {
       new = merge(old,cur,new);
       old = cur;
     }
   } catch(...) {
      return false; /* merge failed */
   }
   return true;
}
```

Here, the merge operation performs the merge-update and returns true or else fails and returns false.

Merge-update addresses a recognized weakness of non-blocking synchronization (NBS) relative to lock-based synchronization, namely repeated aborts in the case of high contention structures. In particular, merge-update easily handles concurrent updates to maps, queues and counters, the most common high-contention data structures, only aborting when the updates are logically conflicting, which is expected to be rare.

4. HICAMP Programming Model

The HICAMP programming model is similar to a conventional *object oriented programming* (OOP) with minor changes for snapshotting/transactional behavior.

Each software object in HICAMP is mapped to a segment identified by a VSID and each pointer field in an object stores a VSID corresponding to the object to which it is pointing, or zero if a null reference. Each software iterator is implemented as an iterator register when the iterator is active.

4.1 Arrays and Maps

A software array is implemented as a HICAMP segment. Unlike an array in a conventional memory system, it can be extended in length dynamically without reallocation and copy; the DAG is simply extended with additional lines. This property also means that a *buffer overflow* bug or compromise cannot cause another object to be overwritten. If the array is sparse, it is automatically stored in a space-efficient form because of path and data compaction.

Thus, for example, an ordered collection of objects indexed by a 64-bit time stamp can be efficiently represented as a segment with the VSID of the object stored at the numeric index equal to its time stamp. In contrast, the same collection in a conventional memory system would require a red-black tree or similar data structure to handle the ordering, have a large index size and the sparsity of the data relative to the index, with attendant costs in memory accesses for performing lookup, rebalancing and locking for concurrency. An ordered collection indexed by a string value can be realized using two arrays, one mapping the root PLID of the string segment to the corresponding value and a second segment for storing the values in order for iteration. The memory deduplication minimizes the space overhead that this two-array solution would incur in a conventional memory.

A map is implemented as a sparse array, indexed by unique root PLID of the key, which can be an arbitrary byte string. Writing to the map entails storing through the iterator register. With HICAMP deduplication and path and data compaction, this structure incurs memory access overhead comparable to the various software tree data structures, which also require rebalancing. It entails more memory accesses than a map implemented as a contiguous array in a conventional architecture. However, it provides a far superior worst-case performance guarantee compared to a conventional implementation of a hash table. Moreover, it avoids the overhead incurred by the hash table for computing the hash on the key and further accessing the key to verify the match. Finally, iteration over the HICAMP map is significantly more efficient than over a conventional memory hash table that is sparsely populated (which is required to avoid collisions and chaining). In general, all the data structures we have considered have an efficient HICAMP implementation.

4.2 Iterators

The software concept of an iterator, an established design pattern for accessing data structures[8], maps directly onto the HICAMP iterator register. For example, iteration over a collection appears conventional:

```
for( it = obj.begin(); it != obj.end(); ++it ) {
  *it = newVal;
}
it->tryCommit();
```

[8] As provided in many programming languages and libraries such as C++ STL, Java and C#.

The ability to increment position pointed by the register to the next non-null element in the segment allows efficient iteration over sparse vectors and associative arrays. DAG prefetching and parent node caching in iterator register further support efficient access.

Moreover, the snapshot isolation provides superior *iterator semantics* compared to that feasible on a conventional architecture. Specifically, an iterator can iterate over the collection, being assured of visiting the collection exactly as it was when iterator register was loaded (snapshot was taken), independent of the actions of other threads. In contrast, conventional languages and systems resort to exceptions or undefined behavior when concurrent updates occur, a questionable approach for multi-core systems.

4.3 Using Merge-Update

A data structure for which merge-update semantics are acceptable can be updated using mCAS (Section 3.4) rather than CAS, thereby allowing many conflicts to be resolved more efficiently and eliminating an application level retry most of the time. For example, concurrent insert and delete operations on the key-value map that do not modify the same entry are resolved by merge-update rather than redoing the application-level operation (this is because an insert operation adds a value to a location which was zero and a delete operation sets a non-zero location to zero). Similarly, concurrent updates to counters can be resolved to produce the desired sum in a counter without application retry. High-contention queues can be implemented similarly, using the merge-update operation to store the queue data in a segment and merge-update of the counters to keep track of head and tail locations.

4.4 Example: Memcached Implementation

Memcached [15] is a practical example of how facilities provided by HICAMP can be effectively used to implement a real and performance-critical application. Memcached has a relatively simple API. The main commands are set key-value pair, get value corresponding to a key, and delete key-value pair. (There are also more complicated commands like add, replace, increment/decrement, and CAS but for brevity we limit our discussion to basic commands.) Its implementation on a conventional architecture is conceptually simple: a hash table is used for a key-value map.

However, in practice the implementation is complicated because the map is accessed as a separate process or processes over socket-based communication channels. Further complications arise from conventional memory management issues: the hash table needs resizing, which is performed by a separate thread. To avoid memory fragmentation and overheads of malloc() memcached pre-allocates user configured memory quota and uses a custom slab memory allocator. Reference counting is used to keep track of the allocated memory and to ensure that a chunk of data is not de-allocated before an I/O operation is completed. Additionally, a time-out mechanism is necessary to avoid memory leak that can arise from rare corner-case conditions of reference counting.

In contrast, a memcached implementation on HICAMP architecture is straightforward: the map of key-value pairs (KVP) is implemented as a HICAMP (sparse) array indexed by the root PLID of key strings. Each entry of the array stores the root PLID for the associated value or zero if no associated value is present in the map. HICAMP deduplication ensures the uniqueness of the root PLID for any given key.

At the beginning of processing a get command, a client thread (re)loads an iterator register with a read-only reference to the KVP map segment, at the offset corresponding to the desired key. Thus, for a memcached command that does not modify the map, access to the map does not require interprocess communication, locking or any other form of synchronization[9]. In a typical memcached workload such commands are 90-99% of all issued commands. Because of the HICAMP snapshot isolation, each client thread is completely isolated from any concurrent updates to the map. Moreover, because of the non-blocking synchronization, there is no interference between threads that can cause blocking or deadlock.

In the HICAMP implementation, a separate thread can perform updates, having a read-write reference to the map segment. Other threads queue update requests with this special thread. Alternatively, client threads can be trusted to update the map directly by providing them with a read-write reference to the map segment. In this case, an update is atomically committed when the new root PLID for the revised map is written to the corresponding segment map entry. Because this write is hardware-atomic, a client process cannot leave the map in an inconsistent state, even after being halted at an arbitrary point in its execution[10]. If the client fails before the root PLID is updated in the segment map, the reference to the revised DAG is eliminated and all the lines corresponding to the modifications are reclaimed, leaving the system in the original state. Thus, because of snapshot isolation and atomic update, HICAMP provides the hardware support equivalent to separate address spaces, while allowing direct access to shared data structures.

Memcached commands that update the key-value map can benefit from *merge-update* (Section 3.4), which eliminates re-execution of commands unless there is a true data conflict (Section 5.1.1).

Similar approaches can enhance access to shared state in applications beyond memcached, for example, an in-memory relational database. A client thread with a read-only reference to the database can access the state and process a query with its own private snapshot of the database state. It constructs a *view* as a new segment that specifies the result of the query, while referencing data directly in the database itself. Updates can be performed either by a designated updater thread or by the (trusted) client threads.

In summary, HICAMP provides substantial benefits to established software structures. Nevertheless, there is a long and sorry history of computer architectures that purport to do this, yet result in implementations that are slower than a careful software implementation on top of the conventional Von Neumann architecture, making these architectures unviable. Our evaluation in the next section addresses this legitimate concern.

5. Evaluation

Our preliminary evaluation of the HICAMP architecture uses both analytical techniques and software simulations with different data sets and traces, in various specific application domains. The three key measures that we consider are: off-chip memory traffic, reduction in memory footprint, and concurrency overhead. The conventional architecture was simulated using the PTLSim [33] cycle accurate processor simulator and DineroIV [1] memory hierarchy simulator, while the HICAMP memory system was simulated by a custom model. Both architectures used the same memory system parameters, namely: a 4-way 32 KB L1 data cache, a 16-way 4 MB L2 cache, and 16-byte cache lines. Some experiments use larger line sizes, as noted in the individual subsections.

[9] memcached does not have any ordering requirements for commands from different clients/network connections. Each client/connection is handled by a separate thread which processes commands in order and does not need to enforce any ordering with other connections (or threads that handle them).

[10] This implementation is appropriate when client software can be assumed to be *fail-stop*. If the client software is truly untrusted, i.e. can be malicious, a trusted updating process is required.

5.1 Memcached

The HICAMP implementation of memcached has significant performance advantages over the conventional "separate server" implementation because the socket communication and associated signaling, queuing, and context switching costs are eliminated. The following analysis shows how capabilities of the HICAMP architecture achieve *predictable* and efficient parallel performance.

5.1.1 Concurrent Performance

During execution of a set command, the write and commit of the new key/value pair and associated control state can be performed independent from other threads, because new segments are allocated for these data fields. However, update of the key/value map itself can cause conflict.

For conservative estimation we assume an 8-processor system that performs 200K commands per second [27] and a 10:1 get to set command ratio. One get command is executed every 5 microseconds and one set command is executed every 50 microseconds on average.

To update the key/value map, the software first reloads the iterator register with the root node of the DAG, fetching the path to the leaf node that is to be updated. The reload may well hit in the cache for several lines but assuming worst-case cache misses it would require $log_2(N)$ DRAM reads to access the leaf on average for N KVPs. The leaf line is then updated in the iterator register and the whole path from the leaf to root is regenerated. Because the leaf node is new content, all nodes up to the root of the map ($log_2(N)$ total for 16-byte lines) must be modified.

To perform a lookup for a line with new content, the memory controller reads the signature line and compares stored signatures with the signature of the new line (Section 3.1). If the new content is not present in the memory, the probability of a signature match is small (see footnote in Section 3.1). If there is no match, the memory controller assigns a new PLID for the line. Thus, signature read and compare are on the critical path of acquiring a PLID for new content, but other operations (updating signature line, etc.) are not and can be performed in parallel. Thus, the time to reload and update map segment is $2 \times log_2(N) \times (DRAM\ latency)$. Assuming that N is 10^6 and DRAM latency is 50ns, the total time to update the map is $2 \times 20 \times 50ns = 2\mu s$. The probability of a conflict can be estimated as $2\mu s/50\mu s = 0.04$. This result is not very sensitive to the number of KVPs in the map because of logarithmic dependence, e.g. for N equal to 10^9 the worst-case probability of conflict increases to 0.06. Moreover, for longer 32-byte or 64-byte lines the number of nodes that need to be updated and the probability of conflict decrease proportionally.

In case of a conflict, *merge-update* (Section 3.4) is used to merge changes to the map from the processor whose CAS operation failed with the new version of the map. In this case, the processor reloads the map and compares each level of the DAG with previous version (snapshotted by iterator register). Because two processors are unlikely to write into the same location in the map, new and previous versions have the same content at some level of the DAG. The merging processor reloads all nodes up to this level and regenerates the nodes above all the way to the root. Assuming that updates to the map are uniformly distributed, the probability of conflict one level below the root is 0.5, two levels below root is 0.25, etc. The average latency of merge-update therefore is the sum of a simple geometric series: $2 \times (DRAM\ latency) \times (1 + 0.5 + 0.25 + ...) \approx 4 \times (DRAM\ latency) = 200ns$, which is significantly smaller than the latency of original map update or application-level retry.

If contention on a map is high for merge-updates, the map can be split into an array of segments (i.e. a segment that points to the subsegments), indexed by several bits of the key PLID, while the rest of key PLID bits can be used as offset within the selected subsegment. Such a split would reduce probability of conflict and re-execution even further.

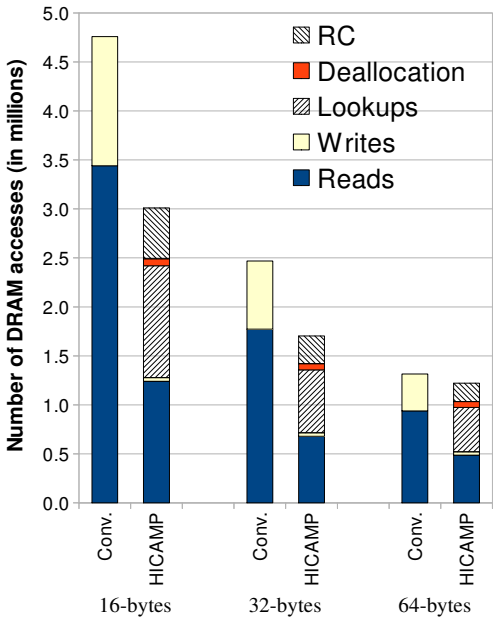


Figure 6: Memcached DRAM accesses

5.1.2 Off-Chip Memory Traffic

Figure 6 shows number of DRAM accesses for both HICAMP and a conventional memory system (using 16-, 32- and 64-byte line sizes) for memcached processing 15000 requests[11]. "Reads" is the number of DRAM reads due to cache misses. "Writes" is the number of DRAM writes due to cache writebacks. For HICAMP, Figure 6 also shows the number of DRAM operations necessary for content lookups[12], line de-allocation and reference counting. For the conventional memory system, we used trace option in VMware Workstation-7.0.0 (build 98145.x86_64) to generate the memory access trace, which includes more than 300 million loads and stores. It was processed by the DineroIV [1] cache simulator to generate DRAM access numbers. As shown in the figure, the number of off-chip DRAM accesses for HICAMP is comparable or smaller than for a conventional memory system.

Beyond these performance benefits, the HICAMP provides memory savings, as described next.

5.1.3 Memory Compaction

Table 1 shows HICAMP data compaction results for several types of data. For text data like HTML web pages and JavaScript scripts, compaction savings range from approximately 1.5x to more than 4x over conventional memory[13]. For binary data image files in compressed JPEG and GIF formats, which result in high-entropy data exhibiting few deduplication opportunities, HICAMP representation incurs a 10% overhead for short lines. Beyond these savings,

[11] These traces and statistics were generated after pre-loading 100000 items into memcached and processing another set of 15000 requests. Memcached items were generated from Facebook web-page dumps (downloaded from Stanford Infolab web-site) to emulate typical memcached dataset. The request trace was generated using a power-law distribution for item frequency and size which is typical for memcached workloads.

[12] Signature line reads/updates and data line reads/writes (Section 3.1).

[13] Compaction is defined as conventional memory requirement divided by the HICAMP memory requirement.

Dataset		Wikipedia webpages	Facebook					
			webpages		scripts		images	
		May'06	May'08	Sept'08	May'08	Sept'08	May'08	Sept'08
Number of items		99957	9741	65122	3106	1467	24827	31155
Total size, Mbytes		3249.58	179.38	3286.81	4.45	0.7	96.17	98.62
Compaction	LS=16	1.71	4.27	1.84	3.17	4.06	0.9	0.93
	LS=32	1.5	3.87	1.83	2.6	3.38	1.03	1.07
	LS=64	1.29	3.11	1.35	2.06	2.29	1.07	1.09

Table 1: Memcached data compaction (LS is line size)

HICAMP also eliminates duplication of data between processes and socket buffers, the size of which can be substantial [27].

We expect the HICAMP architecture to be compelling in other settings, including similar web and database-like applications. For example, some fault-tolerant embedded systems such as 10 Gbps Ethernet switches [5] incorporate a database and numerous client processes to achieve fault-tolerance and in-service software upgrade capability. These systems could use HICAMP to reduce communication overhead and memory duplication while retaining their robustness. Even in more restricted environments, such as handheld devices, HICAMP could provide protected sharing of data between untrusted applications while providing memory compaction to reduce power demands.

In the completely different domain of high-performance computation, HICAMP also demonstrates substantial benefits, as described next.

5.2 Sparse Matrix Computation

Sparse matrix computations, and, in particular sparse matrix dense vector multiplication (SpMV) [32], are often the critical inner loop of certain applications including general high performance computing (HPC), embedded control systems, linear programming, and convex optimization. For SpMV, performance is limited by memory bandwidth, yet peak bandwidth is not achieved because of the unpredictability of the vector access sequence. Performance as low as 10% of peak FLOP rate has been observed [30]. Thus, the programming language and architecture challenge is supporting efficient execution of sparse matrix computations, particularly in a highly concurrent processing environment [29].

In HICAMP, a general (dense) matrix representation can be used to efficiently represent a sparse matrix, as described earlier, with an iterator providing efficient access to the data. Alternatively, a representation similar to the conventional *compressed sparse row (CSR)* format can also be used. To optimize further, one can recognize that many applications use sparse matrices with a high degree of self-similarity – they are either symmetric or have a high degree of symmetry, or have repeating patterns of non-zero values. To exploit matrix self-similarity, we have used the *symmetric quadtree format* (QTS) that breaks a matrix into four regions, which have a dimension of an even power of two (and are zero padded if necessary). Sub-matrices A_{11} and A_{22} are stored in the left subtree of the DAG while sub-matrices with A_{12} and ${A_{21}}^T$ stored in the right subtree. If the sub-matrices are symmetric, A_{12} and ${A_{21}}^T$ are equal and hence have the same PLID as their root.

Some sparse matrices with little self-similarity in non-zero values are not compacted well by simple sparse array or quad tree representation, but still have a high degree of non-zero pattern similarity. In many of these cases, the underlying structure of this matrix can still be exploited using the *non-zero dense* (NZD) format where two HICAMP segments represent a matrix: one segment stores the non-zero pattern as a quad-tree and exploits pattern symmetry and self-similarity, while a separate segment is filled to be nearly dense with the original non-zero values.

Category	Matrices	Savings	StdDev
All	100	62.7%	36.5%
Non-symmetric	77	58.5%	33.9%
Symmetric	23	76.9%	41.8%
FEMs	29	70.7%	40.2%
LPs	15	43.0%	31.7%

Table 2: Sparse Matrix Compaction

The DAG structure lends itself to tree-recursive algorithms and many important operations in linear algebra can be naturally expressed in such form [18]. During tree traversal, zero and duplicate sub-matrices can be detected by PLID comparison. Such optimizations reduce number of memory accesses and increase the performance of the memory system.

5.2.1 Off-Chip Memory Traffic

The SpMV kernel was used to compare HICAMP to a conventional memory system in terms of number of accesses made to the off-chip memory. We compare a HICAMP SpMV algorithm with duplicate sub-matrix detection against a conventional CSR SpMV algorithm or against a symmetric CSR SpMV algorithm [24], as appropriate. In HICAMP, no distinction was made for symmetric matrices. Our test matrices were pulled from the University of Florida Sparse Matrix Collection [10] and represent a variety of application domains.

Considering only matrices that are larger than the L2 cache size, the results indicate that HICAMP reduces off-chip memory accesses on average by 20%.[14] Figure 7 plots the ratio of off-chip memory accesses in HICAMP to a conventional architecture using a $\log_2$ scale; matrix size is plotted on the abscissa. As shown in the figure, for most of the matrices HICAMP reduces main memory accesses by compacting the sparse matrix and reducing the storage requirements, as described next.

5.2.2 Memory Compaction

We compared the storage requirements for 100 large sparse matrices when stored in HICAMP versus the conventional memory, which uses CSR. In CSR, an m by n matrix with nnz non-zero entries requires $8 \times (1.5nnz + 0.5m)$ bytes (assuming 8 byte double precision floats and 4 byte integer indices). For a symmetric sparse matrix, the term nnz is replaced by $nnz_{\text{on-diagonal}} + 0.5nnz_{\text{off-diagonal}}$. We compare the best-known HICAMP format (QTS or NZD) against CSR, or symmetric CSR, as appropriate.

Table 2 lists the ratio of the matrix size in HICAMP versus conventional architecture, categorizing matrices in different classes and problem domains, while Figure 8 shows the same ratio for all matrices. Matrices are the same size or smaller in HICAMP except for a few having negligible increases due to the DAG overhead. The overall average is 62.7 bytes in HICAMP per 100 bytes on

[14] Excluding a matrix that was compacted by 4000x. Including this matrix the overall savings are 38%

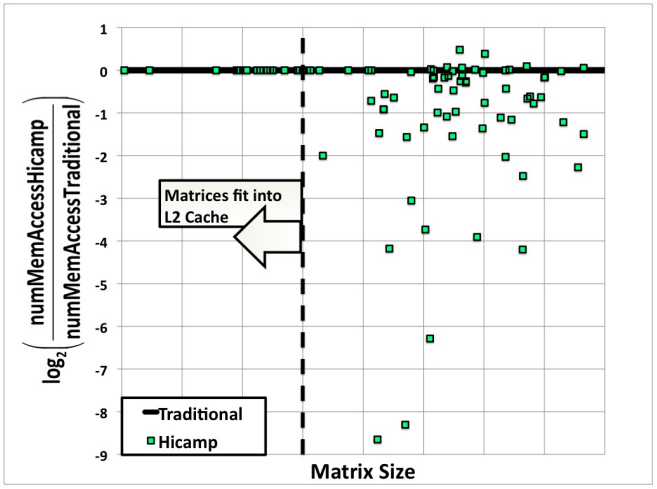


Figure 7: Main memory accesses for SpMV

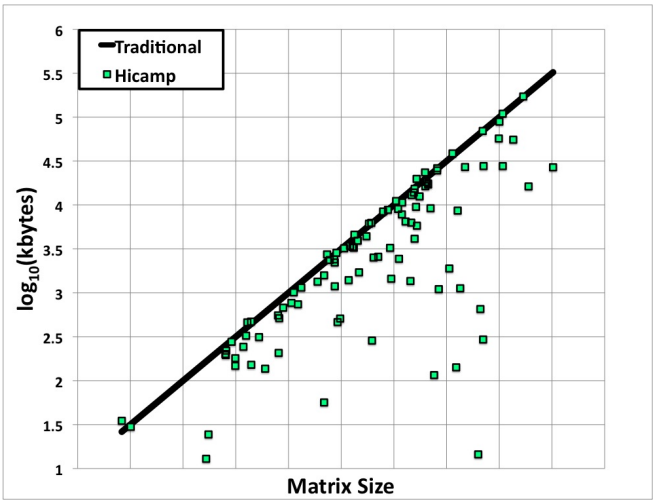


Figure 8: Sparse matrix memory footprint

a conventional architecture, indicating that HICAMP achieves significant data compaction over a broad range of matrices.

In addition to improving off-chip memory traffic and storage footprint, HICAMP also allows efficient, highly parallel execution of sparse matrix computations by partitioning the result matrix among K threads. Each thread computes from the previous matrix results, preserved by the snapshot isolation, generating the result partition it is assigned. At the end of a round, each partition is merged into the new result matrix, which is then made available as read-only data for the next round. Thus, the snapshot isolation, logical copy and merge-update provide architectural support for a simple but efficient programming model of concurrent matrix computation while ensuring protection between threads.

5.3 Virtual Machine Hosting

Virtual machine (VM) hosting is an important commercial application, which places significant demands on operating systems and hypervisors. It also provides a means for HICAMP to run legacy code, i.e. software designed for the conventional Von Neumann architecture. With an increasing number of cores and faster network connections allowing an increasing number of VMs per physical host, there can be a significant amount of duplication of code and data within the physical memory of the host, given duplicate copies of the same or similar operating systems and applications.

Some VM hypervisors such as VMware ESX server check for duplicate pages in software and replace multiple copies with a single shared copy in the machine memory [31]. Further research [17] has illustrated the benefits of sharing at finer granularity than a page. However, with the increasing size of physical memory, deduplication beyond just static code pages can incur significant software overhead. Moreover, with large (2 megabyte) pages, motivated by quadratic cost of nested page table lookup and reducing the size of page tables [7], the page-level sharing is significantly less effective.

HICAMP automatically provides fine-grain sharing at the line level without software overhead and reduces the memory footprint of hosted VMs. This compaction can potentially reduce the number of off-chip DRAM accesses, improving performance, reducing power and increasing overall efficiency.

To evaluate HICAMP for virtual machine hosting, we used the VMmark virtualization benchmark (version 1.1.1) [2] from VMware. VMmark organizes units of work in tiles, where each tile contains six virtual machines. There is a variety of 32-bit and 64-bit operating systems as well as various types of workloads in each tile (e.g. database, mail server, java server, etc.). We evaluate HICAMP's memory consumption when loading a number of virtual machines and tiles into the system.

We took snapshots of the memory of virtual machines while running benchmark workloads. These snapshots were then loaded into HICAMP's memory system simulator to compute the total number of memory lines required and were also analyzed to determine the number of duplicate pages. Figure 9 shows consumed memory when scaling number of VMs for each of the workloads in a VMmark tile and compares it to an ideal page sharing scheme where duplicate pages are detected and shared instantaneously. While not realistic, this provides an upper bound on how well a page sharing algorithm at the OS or hypervisor level can save memory. In practice, the benefits are less than this ideal scheme because scanning memory and detecting duplicate pages consumes processor cycles and occurs gradually over time.

Figure 10 shows the same comparison for VMmark tiles when loaded as a whole. HICAMP compacts memory footprint of VMs by a factor between 1.86x and 10.87x, while in ideal page sharing, the compaction ranges between 1.44x-5.21x. For tiles, HICAMP compacts the memory more than 3.55x while ideal page sharing compacts it only 1.8x.

Overall, HICAMP simplifies the OS/hypervisor, reduces processing overhead and improves the utilization of memory by providing transparent data sharing in hardware.

6. Related Work

Fresh Breeze [11] supports software structuring similar to HICAMP, providing protected "global" references and DAG structure support for sparse matrices. However, Fresh Breeze is focused primarily on supporting functional programming, not fault-tolerant thread isolation. It does not support a segment map or memory deduplication and compaction mechanisms of HICAMP. Fresh Breeze also does not appear to support an iterator register mechanism for efficient access to memory, instead relying exclusively on SMT to accommodate the additional latency to memory, and does not appear to support merge update.

Transactional Memory (TM) has been investigated by a variety of efforts, starting from the Herlihy and Moss paper [22]. However, TM does not provide isolation between threads. Moreover, TM uses the update-in-place approach to transactions so it suffers from read-write conflicts. For example, a transaction iterating over a large

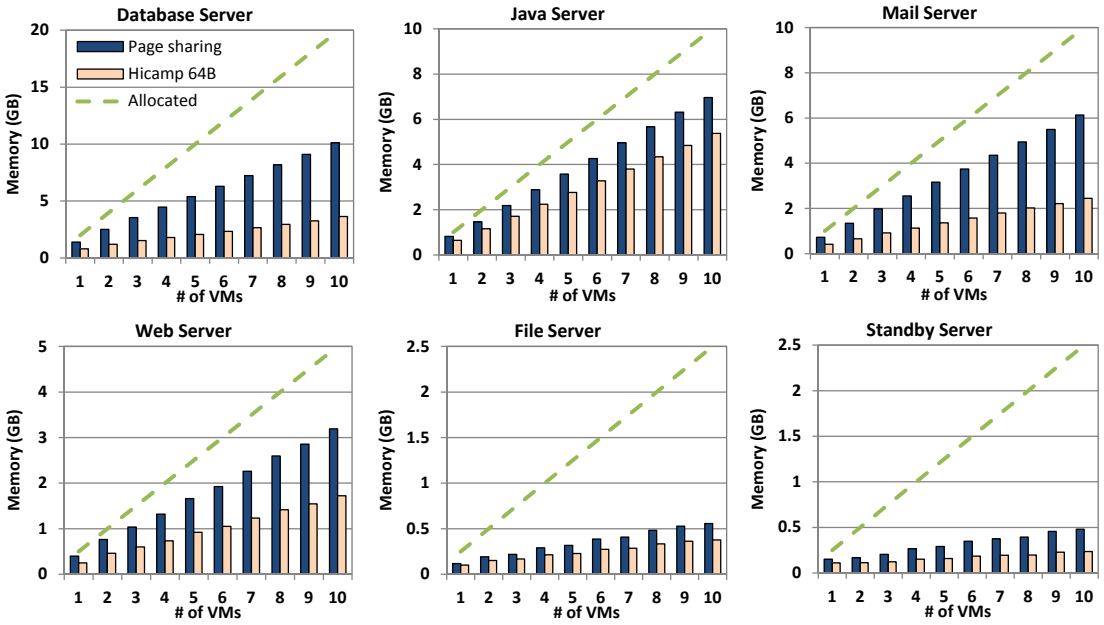


Figure 9: Memory consumption of individual VMs in a VMmark tile

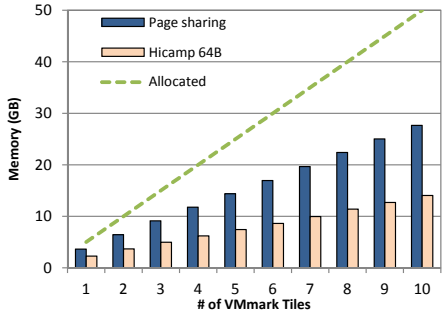


Figure 10: Memory consumption of VMmark tiles

collection can be repeatedly aborted by updates to the collection. In contrast, HICAMP implements an alternative approach based on snapshot isolation [6], so it only has write-write conflicts, which are far less frequent.

Hardware Snapshotting has been proposed separately [9]. However, the support in this case is coarse-grain, restricted in number of snapshots, and focused on actions such as checkpointing, not changing the application software execution semantics as in HICAMP. Snapshotting was also proposed as an extension of software transactional memory [26], however, it exhibits significant overheads of maintaining and tracking multiple versions of data in software.

Hardware support for *non-blocking synchronization* dates back many years to CAS on the early IBM computers and double-CAS on the Motorola 68040. It is known to have significant advantages over locking [16]. In HICAMP, a simple CAS suffices to update complex data structures and avoids the so-called ABA problem. Reference semantics, hardware support for data structures, and in particular the segment map mechanism allow atomic updates across multiple objects/data structures.

Memory Compression has been proposed for main memory [14], caches [4, 12] or both [13, 19]. A mergeable cache architecture was designed [8] to de-duplicate the content of second level cache. This approach only supports deduplication for cache lines that have the same virtual address, using page coloring to ensure that such lines are located in the same set. The HICAMP architecture supports general deduplication in main memory and caches, thereby reducing memory bandwidth and footprint.

VMware [31] provides transparent page sharing to reduce duplicate memory content at the page level. This process is configured by default to take place at a time scale of minutes in order to minimize processing overhead, so it only identifies static pages of long-running applications. The Difference Engine [17] project carries this direction further to detect intra-page differences, increasing software overhead and complexity. HICAMP provides finer-grain sharing in hardware without such overhead. However, we regard the key advantage HICAMP provides over software page sharing as providing line granularity in the presence of the trend to 2 megabyte pages [7].

There are some similarities between HICAMP and the Intel iAPX 432 project [3] including the protected segments and the support for higher-level programming constructs. However, the 432 was focused on providing protection with its segments and did not provide the functional and performance benefits HICAMP does of deduplication, snapshot semantics and atomic update. The 432 also suffered because of the limited gate count per chip that was available 30 years ago, forcing it to be implemented across multiple chips, with severe performance implications. In contrast, HICAMP appears to easily fit within current processor gate counts and tackles a key problem of modern applications and processors, namely the performance overhead/limitations arising from the memory system.

7. Conclusions

In this paper, we have presented HICAMP, a general-purpose computer architecture designed to address the key challenges facing programming languages and operating systems to support demanding complex, concurrent software on multi-core processors. It directly challenges conventional wisdom by providing hardware support for key programming language features, including variable-sized objects, iterators, “const” access, associative arrays and sparse matrices. It also supports non-blocking synchroniza-

tion and structured data prefetching. It further supports snapshot semantics and atomic update, building on the proven benefits of these properties in database systems [6]. Finally, it provides memory deduplication at cache line granularity, complementing the page-sharing mechanisms in many operating systems and virtual machine hypervisors.

The protected segment references that arise in this architecture together with the concurrency-safe data structure support allow multiple threads to safely share structured data without the space and IPC overhead of separate address spaces and without the loss of isolation associated with conventional shared memory threads.

Even with these significant benefits, HICAMP is practical to implement with conventional DRAM, with an acceptable level of hardware complexity, and can achieve competitive, if not superior, performance for appropriately structured applications. We also show that it can provide performance benefits for existing software applications running in a virtualization environment, based on its deduplication capability.

HICAMP performance is a key focus of our results in this paper. Our evaluation shows that HICAMP provides significantly better DRAM bandwidth utilization for applications structured to take advantage of its features compared to conventional architectures. In particular, our measurements show that the reduced memory footprint provided by deduplication, together with the iterator register support, including data and path compaction, largely counterbalances the extra memory accesses required by the DAG representation of segments and the extra cost of content lookup in memory, both in terms of number of DRAM accesses and DRAM bandwidth. Moreover, certain cases such as sparse matrix applications show substantial reduction in memory size and access.

Beyond these advantages, the HICAMP memory deduplication and compaction reduces memory requirements, reducing capital cost and improving energy efficiency. Finally, studies [23] show that typical server utilization of DRAM bandwidth is less than 10 percent in many cases, so HICAMP can be viewed as exploiting this under-utilized resource to improve application properties, including performance.

In this comparison, the efficiency of HICAMP should be compared to the inefficiency of the established database process approach, where up to 87 percent of the performance is lost on a conventional architecture [28] to overheads that HICAMP largely eliminates by implementing necessary support in hardware. At the same time, we recognize that certain computing actions are significantly more expensive in HICAMP than a conventional Von Neumann architecture, such as randomly accessing data in a large, dense array. Our claim is that applications do not generally contain such behavior in the performance-critical portions, these performance pitfalls of HICAMP can be characterized and understood by the programmer, and there are reasonable alternatives in the cases that we have investigated. In general, HICAMP tackles macroscale inefficiencies in applications such as interprocess communication overhead, lock synchronization overhead and memory copying, which we expect to become worse with increasing number of cores and have been somewhat neglected by the conventional focus on micro-benchmarks.

Overall, HICAMP offers a promising direction for future multi-core systems to properly support complex, concurrent, data-intensive software applications, simplifying operating systems and improving practical programming language semantics. It recognizes that access to shared structured data is a key problem in many demanding applications ranging from web servers to scientific computation to embedded systems, and demonstrates that memory system innovation can provide significant benefits to the software.

Acknowledgements

The authors would like to thank Min Xu, Rajat Goel, Steve Herrod and Vyacheslav (Slava) Malyugin of VMware for all their kind help and support regarding VMware Workstation product. The authors are also grateful for valuable comments on earlier versions of this paper from Al Davis and Stephen Richardson, and on the final version from Carl Waldspurger.

References

[1] `http://pages.cs.wisc.edu/~markhill/DineroIV/`.

[2] `http://http://www.vmmark.com/`.

[3] `http://wikipedia.org/wiki/Intel_iAPX_432`.

[4] A. R. Alameldeen and D. A. Wood. Adaptive Cache Compression for High-Performance Processors. *SIGARCH Comput. Archit. News*, 32(2):212, 2004.

[5] Arista Networks. EOS: An Extensible Operating System. `http://www.aristanetworks.com/en/EOSWhitepaper.pdf`.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.*, 24(2):1–10, 1995.

[7] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 26–35, New York, NY, USA, 2008. ACM.

[8] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-Execution: Multicore Caching for Data-Similar Executions. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 164–173, 2009.

[9] J. Chung, W. Baek, and C. Kozyrakis. Fast Memory Snapshot for Concurrent Programming without Synchronization. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, pages 117–125, 2009.

[10] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38:1:1–1:25, Nov. 2011.

[11] J. B. Dennis. Fresh Breeze: A Multiprocessor Chip Architecture Guided by Modular Programming Principles. *SIGARCH Comput. Archit. News*, 31(1):7–15, 2003.

[12] J. Dusser, T. Piquet, and A. Seznec. Zero-Content Augmented Caches. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, pages 46–55, 2009.

[13] J. Dusser and A. Seznec. Decoupled Zero-Compressed Memory. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 77–86. ACM, 2011.

[14] M. Ekman and P. Stenstrom. A Robust Main-Memory Compression Scheme. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 74–85, 2005.

[15] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, 2004.

[16] M. Greenwald and D. Cheriton. The Synergy Between Non-blocking Synchronization and Operating System Structure. In *OSDI '96: Proceedings of the second USENIX Symposium on Operating Systems Design and Implementation*, pages 123–136, 1996.

[17] D. Gupta, S. Lee, M. Vrable, S. Savage, A. Snoeren, G. Varghese, G. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 309–322, 2008.

[18] F. Gustavson, A. Henriksson, I. Jonsson, B. Kagstrom, and P. Ling. Recursive Blocked Data Formats and BLAS's for Dense Linear Algebra Algorithms. In *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science, pages 195–206. Springer Berline / Heidelberg, 1998.

[19] E. G. Hallnor and S. K. Reinhardt. A Unified Compressed Memory Hierarchy. In *HPCA '05: Proceedings of the 11th International*

Symposium on High-Performance Computer Architecture, pages 201–212, 2005.

[20] R. Hamming. Error Correcting and Error Detecting Codes. *Bell Sys. Tech. Journal*, 29(4):147–160, 1950.

[21] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 197–206, 1990.

[22] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[23] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server Engineering Insights for Large-Scale Online Services. *IEEE Micro*, 30:8–19, July 2010.

[24] B. C. Lee, R. W. Vuduc, J. W. Demmel, K. A. Yelick, M. de Lorimier, and L. Zhong. Performance Optimizations and Bounds for Sparse Symmetric Matrix-Multiple Vector Multiply. Technical Report UCB/CSD-03-1297, University of California, Berkeley, CA, November 2003.

[25] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[26] T. Riegel, C. Fetzer, and P. Felber. Snapshot Isolation for Software Transactional Memory. In *In Proceedings of the 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT 06)*, 2006.

[27] P. Saab. Scaling memcached at Facebook. `http://www.facebook.com/note.php?note_id=39391378919`, Dec 2008.

[28] M. Stonebraker and R. Cattell. 10 Rules for Scalable Performance in 'Simple Operation' Datastores. *Commun. ACM*, 54:72–80, June 2011.

[29] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, June 2005.

[30] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *Proceedings of Supercomputing*, 2002.

[31] C. A. Waldspurger. Memory Resource Management in the VMware ESX Server. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, 2002.

[32] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *Supercomputing*, November 2007.

[33] M. Yourst. PTLSim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of ISPASS*, January 2007.