FINE-GRAIN IN-MEMORY DEDUPLICATION FOR LARGE-SCALE WORKLOADS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

John Peter Stevenson

December 2013

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Cheriton

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Patrick Hanrahan

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

Abstract

Memory is a large component of computer system cost and current trends indicate this cost is increasing as a fraction of the total. Emerging applications such as in-memory databases, virtual machines, and big-data key-value stores demand more memory relative to compute. Some of these high-memory applications incidentally store many duplicate values in memory: in some cases, duplicates account for over 75% of the total. Recent work on the HICAMP architecture provided a sophisticated hardware mechanism for memory deduplication to implement memory versioning, but without support for current software stacks. This thesis extends work on HICAMP by evaluating a deduplicated memory that is compatible with existing hardware and software. Memory content from actual workloads indicates that deduplicated memory effectively doubles capacity. After understanding the baseline cost-benefit tradeoff in terms of capacity, performance, and energy, this work proposes novel optimizations for machines with deduplicated memory. These optimizations reduce memory traffic and improve performance relative to both the baseline deduplicated memory and, in many cases, relative to the original machine. Energy consumption is reduced because memory devices are reduced with no penalty to performance. Further, deduplication reduces data transfer and improves performance for certain scientific applications. This thesis argues that in-memory deduplication is warranted by its own benefits, which are likely to grow in the future, and that it enables low-cost memory snapshots, as in the HICAMP architecture.

Acknowledgments

Dedicated to my wife, Siejen.

Most of all, this thesis is a product of the family that has supported me. Completing this work is a significant occasion along a longer trajectory – an arc of life that I have been privileged to enjoy, and that I hope to repay in kind. Necessarily, this trajectory begins with my mother and father, both of whom deserve first acknowledgement. Mom and Dad – thank you both, for raising me, putting up with me, and most of all, for inspiring me to go farther than I dared think possible.

Another significant turning point in my life was the day I met my future wife, Siejen – a day and a time that I will remember fondly, forever. Now, years later, I love her more and can say that she has made me a better man. My life has been richer and more fulfilling during the time I have spent with her. Siejen – thank you for loving me, and for supporting me through this program.

Mom and Dad – you let me wander far from home to seek my fortune. Siejen – you joined me on this journey, for better or for worse, and the journey has taken you far from home also. Many years later, I have completed this thesis, and I have also found a new home: namely, that is wherever my best friend and wife happens to be – I am at home, and safe. Personally, I have derived a great amount of satisfaction from completing this task, and I want you all to be likewise proud of it, both now, and as I continue this work – you gave me the freedom to walk down this path.

Siejen and I have benefitted, both, from the support of the rest of our respective families. To Debbie and Bill Edwards, my sister, and her husband, thank you for opening your house to us and for inspiring me, at an early age, toward my future alma mater, the U.S. Naval Academy. To Shong Yin, Ph.D., and Susing Yin, Siejen's brother and sister, thank you for providing good advice and friendship during the course of this work. And finally, to Robert and Nancy Yin, Siejen's parents, thank you for believing in me and taking me in as a member of your family.

I owe a debt of gratitude to the U.S. Naval Academy, to its staff and faculty, and to the U.S. Navy. At age eighteen, I knew the advantages and disadvantages of both active and passive SONAR – critical knowledge for my future career in submarines – but, I did not know *where* Stanford was or anything much about such opportunities. Tom Brennan – thank you for suggesting that I pursue graduate education, I doubt very much I would have thought of that myself. Larry Tankersley – thank you for teaching the most wonderful physics class I can imagine, and if someday I have the privilege of teaching, I hope to emulate your style. Carl Wick – thank you for being my advisor and best advocate: you helped me navigate one of my early successes.

To Stanford, thank you for filling in the gaps in my undergraduate knowledge. Tom Lee – I always knew that causality had to be a problem with the "ideal opamp" model: I thoroughly enjoyed your courses, and learned a great amount. Umran Inan – you laid to rest another engineering model that had always bugged me, for essentially the same reason: I finally grok that voltage does not appear instantaneously, thanks to you, and of course, James Clerk Maxwell. David Miller – thank you for passing me. I so desperately wanted to take a course in quantum mechanics, having no idea what it was, and being totally unprepared for the material. Of course, you

inspired my short foray into song writing, and to claim rightful authorship for eternity, I now, as befits such mathematically inspired work, using LaTeX, give the lyrics, sung to the music of *Jingle Bells:*[1]

Ohhh quantum well go to hell, I hate Erwin Schödinger

$\nabla^2\psi$ *makes me cry, divide* h *by* 2π

During this time, I have made many good friends, and among whom, I am humbled and honored to be included as *their* colleague. Omid Azizi – thank you for being my friend and officemate, and thank you for lending me your analytical mind and incredible intelligence.[2] Zain Asgar – thank you for being my friend, for being a fellow circuit optimization enthusiast, and for coming up with all the good ideas on how to divert my time toward better uses, such as camping trips and balloon festivals. Ofer Shacham – thank you for being so welcoming when I joined Mark's group; I know we have followed similar paths from very different starting points and I look forward to knowing your family and knowing of your successes in the future.

The work on this thesis has also been influenced by the ongoing work at a small company, HICAMP Systems. Small, though it may be, the combined intellectual quotient of its staff is formidable. Alex Solomatnikov and Amin Firoozshahian – thank you for blazing the trail at HICAMP and for teaching me the basics, or advanced basics, of computer architecture. Mahesh Maddury, Chandan Egbert, and Christophe Joly – thank you for bringing real experience to the table and for steering a safe course for our fledgling company.

Along this path I have learned more from one faculty in particular, and I hope one day to pass on some of his wisdom – both academic, and life. Mark Horowitz, thank you for taking me on as your student, and for inspiring me. You delivered

[1] The lyrics are somewhat hyperbolic, I actually love him, his equation, and his cat.

[2] i.e. Thank you for pre-screening all my ideas.

my first lecture at Stanford, and I will always remember it. It won't be so vivid in the retelling, but I admired the delivery which turned the complex into the simple and unveiled powerful ideas with large amounts of engineering leverage. It started by proposing to take an idea, a *big idea*, and pack it into a little tiny rectangle. On that rectangle, the contents of the idea would be drawn in basic primary colors, and particular significance would be given to certain combinations of such colors, such as red drawn over green. I had no idea, but it turns out that microchips are just drawings, printed in layers of metal, on top of silicon. And making microchips is as simple as making such drawings, where the colors distinguish the layers and red over green makes a transistor. I was hooked, and that was just a beginning – there was more to come, much more. Mark, you've been generous to me, and all of your students – thank you.

Finally, this work is really the byproduct of the formidable intellect and visionary creativity of David Cheriton. I chose to do a Ph.D. at Stanford not just because it is a premier institution, but also because they had given me a very generous offer of financial support, namely, the David R. Cheriton Stanford Graduate Fellowship. This was somewhat of a better offer of financial support than I had received elsewhere, and I already had it in my mind that Stanford offered a tighter link to what is actually happening in the electronics industry. Early in my Ph.D. career, Siejen and I happened to cross paths with David in the middle of downtown Palo Alto – in one of my better life moments, I failed to not notice that this was happening, so I decided to introduce myself and my wife. In this conversation David said he had a new project, known as HICAMP, and asked if I would be interested. At the time, I knew a little bit about computer architecture, enough to recognize HICAMP as being unorthodox, brilliant, and quite fascinating. And the last of those qualities was enough for me – my only criteria was to do interesting work. David Cheriton – thank you for generously providing my Stanford Graduate Fellowship which supported me at Stanford for five

years and thank you for bringing me on board to work on HICAMP. I have enjoyed it immensely and I find every aspect of it incredibly fascinating.

I hope you the reader will agree: what follows is both interesting, and somewhat surprising. In these acknowledgements, my meaning of inspired is twofold: academic, of course, but also figurative – for example: *the stars at night above the ship's swaying deck inspired me to think of mankind forging a path on new spacegoing vessels, through the galaxies*, (or some such); and of course, for now I will keep to academic writing. Mark Horowitz and David Cheriton, thank you for inspiring this work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer technology has advanced at an ever increasing rate, a trend referred to as "Moore's Law." This increasing rate of return has been so reliable that we have come to believe that it is inevitable, but, continued advances require solving ever more difficult problems. Technology trends indicate that demand for memory capacity is growing at an increased rate while growth in capacity – Moore's Law – is slowing. To address this issue, this thesis studies *deduplicated memory* – namely, a computer memory that eliminates duplicate values to provide increased memory capacity. Historically, deduplication has not been used for main memory and there is very little on this topic in the literature. In-memory deduplication was first proposed in a different context, namely, to provide architectural support for parallel programs. Prior work on the HICAMP architecture [24] had used deduplication to provide efficiently versioned instances of memory. This thesis extends this prior work and applies deduplication to current CPU memory systems. The results indicate that deduplication provides effectively twice the capacity for many large-scale applications and that it also saves power. Although a significant departure from mainstream memory architecture, current trends may compel its use to increase capacity – once implemented, deduplication also opens the door to many further innovations, such as HICAMP.

1.1 Cost of Memory

Memory has always been a significant cost in computer systems. Each new generation of memory technology requires an investment, but the size of that investment is now growing relative to the capacity increase delivered. At the same time, programmers have become accustomed to allocating large amounts of memory without worrying very much about its cost. The rise of internet technology, granting near ubiquitous access to communication, entertainment, and commerce, is fueling unprecedented demand for compute resources on many frontiers. This demand, in turn, has given rise to “cloud service providers” who specialize in providing the compute infrastructure and in hosting third-party applications. In the wake of this, a new sector, referred to as “big-data analytics” has emerged – these companies collect large amounts of seemingly trivial data and try to extract useful information. Collectively, these trends are driving an increased demand for memory.

In more detail, driven by emerging applications such as in-memory databases and key-value stores and by techniques such as server consolidation and virtualization, demand for memory is increasing relative to compute [47]. In-memory databases and key-value stores use terabytes of memory to minimize or completely avoid disk access thereby providing increased performance – in these applications, the ratio of memory to compute is very high. Further, memory is a significant cost in datacenters which run large-scale applications, such as the web services provided by Google or LinkedIn, and which provide hosting services, such as Amazon’s Elastic Compute Cloud. The datacenter business model requires efficient utilization of all resources. In this context, virtualization and opportunistic batch scheduling have delivered significant increases in efficiency by effectively sharing CPU time across many applications, but it remains difficult to share memory resources. Therefore, each new workload directly increases memory pressure, but only marginally increases the load on other resources.

As an example of the foregoing, the popular website Twitter runs memcached, a key-value store, on a cluster of 500 machines with 30 TB of DRAM [57]. Today, vendors are selling in-memory database appliances with 4 TB of DRAM total, in which each server board hosts 512 GB [1]. The memory analysis results in this work, given in Chapter 2, indicate that "big-data" workloads also require servers with large memories – for example, Shark,[1] an in-memory data mining engine [29], fully utilized 680 GB of DRAM on a cluster of 100 virtual machines hosted by Amazon Web Services. To quantify the cost of this memory, at the time of this writing, machines with 128 GB of DRAM are common [69]. At 128 GB of capacity, DRAM costs $1400[2] or close to 50% of the entire machine cost. Further, DRAM is 20% to 30% of machine energy and because each memory module has a high fixed cost in power, this cost grows as a fraction of the total when the datacenter is underutilized [10]. Therefore, DRAM is a significant fraction – about 1/3 – of datacenter total cost of ownership (TCO).

Furthermore, the issue of memory capacity is likely to become more significant in the future: in addition to increased application demand for memory, the trend is toward slower DRAM scaling and higher DRAM pricing. Forecasts indicate that DRAM prices will go up by 40% and that the total market will expand by 28% in 2013 [3, 4]. Tighter supply reflects the increasing challenge of DRAM scaling. Current 1T-1C DRAMs require 25 fF of capacitance per bit, and it is difficult to use smaller capacitors because of data refresh overheads [53]. While scaling continues for other silicon designs, it is increasingly difficult to translate this into DRAM scaling because maintaining the minimum required cell capacitance requires higher aspect ratios which put cell reliability at risk [40, 54, 30]. Demand for higher peak bandwidth further exacerbates the capacity issue: to improve channel electrical characteristics

[1] i.e. Hive (data warehouse software) on Spark (a distributed in-memory execution engine).
[2] 16 DIMMs, ECC DDR3-1600, ABMX & NewEgg accessed July 2013.

and hence improve bandwidth, the next generation DDR4 specification allows only one memory module per memory channel [60]. Because of chip pin-count limitations, the number of memory channels is unlikely to scale up significantly, therefore, at one-half the number of modules per channel, the cost of capacity for DDR4 is relatively higher than for DDR3: each module must provide twice the (normalized) capacity.

Given that the dominant memory technology, DRAM, is showing diminishing returns in terms of capacity, it makes sense to look for alternatives. Emerging non-volatile memory technologies may enable a new level in the memory hierarchy referred to as “storage-class memory” or “SCM.” An SCM solution would provide much more capacity than DRAM, with the additional benefit of retaining its data, even when powered down – this compares quite favorably to DRAM which needs millisecond scale refresh. Unfortunately, there is no viable SCM solution available today, and the most promising candidates for commercialization are still 5x to 10x slower than DRAM [9, 39]. Further, many of the non-volatile memory technologies face the challenge of limited write endurance and therefore require sophisticated wear leveling techniques, in effect, introducing a level of indirection. Even when SCM systems become available, it is unlikely that they will completely replace DRAM – rather, due to their relatively higher latencies, it is very likely that a large DRAM cache will be necessary to provide acceptable performance. Finally, given that deduplication can reduce the number of data writes into memory, the work in this thesis may be even more applicable to SCM, that is, to provide both increased capacity and increased write endurance.

This thesis answers to a critical need by addressing the issue of memory capacity. Emerging applications and the growing importance of datacenter computing are driving increased demand. Current price points make memory up to 50% of capital expenditure and 20% of operating expenditure, and worse, the memory continues to consume power, even when machines are idle. Empirically, DRAM scaling has already slowed and will very likely continue to slow. Finally, no current technology is likely to completely replace DRAM. Together, these trends and observations indicate that memory capacity will become more, not less, of a concern in the future.

1.2 Efficient Use of Memory

These trends, increased demand for memory capacity and slowing DRAM scaling, point to a need to make more efficient use of memory. Software techniques, both for compression and for deduplication, have been deployed to answer to this need, but these techniques find limited use due to significant costs and show little benefit relative to the opportunity. On the other hand, special purpose hardware can be used to the same end. Although such hardware requires a significant investment to design, verify, and deploy, these expenses appear warranted given that software solutions cannot fully exploit the opportunity without significantly harming user program performance.

As evidence of the need, three software techniques for increasing memory capacity have found wide-spread use. Virtual machine hosts use *transparent page sharing* to deduplicate memory at 4 KB page granularity [66]. Recently, the popular OS X operating system implemented page compression [5], and nearly every operating system provides virtual memory to user applications using demand paging. Although promising, these techniques all come with serious performance penalties.

In more detail, both compression and transparent page sharing steal CPU cycles from user applications. Therefore, when used, the operating system also throttles each of these techniques to a point where their performance impact is minimal. For example, in VMware ESX, transparent page sharing is limited to scan all of memory only once every ten minutes [32]. With such a slow scan rate, transparent sharing cannot deduplicate any pages that have been written to recently – a serious drawback that limits its potential upside. Moreover, to reduce page table lookup costs, virtual machine hosts are compelled to use much larger page sizes which further limits the utility of transparent page sharing. Turing to page compression, OS X compresses only the least recently used pages and only under memory pressure, again, to minimize performance impact. Succinctly, page compression is a last resort to avoid paging to disk. And although demand paging provides the illusion of near infinite memory capacity, paging to disk is generally avoided if at all possible.[3]

Because software techniques incur high costs, the question of implementing special purpose hardware appears viable. In particular, a hardware solution would not steal any CPU time from the running applications. Although such hardware does not directly steal CPU cycles, it does potentially increase the memory access latency which incurs an indirect performance cost. Therefore, this thesis evaluates a direct hardware implementation of fine-grain memory deduplication, both in terms of its benefit and performance. The remaining paragraphs in this introduction provide a brief overview of each of the chapters – the reader may skip these entirely, or refer to them as a guide to what each chapter has to offer.

1.3 Duplicates in Actual Workloads

While duplicates are known to be common for *virtualized* workloads, such as Amazon Elastic Compute Cloud (EC2), this thesis shows that they are also common in a broader class of *non-virtualized* datacenter workloads. Most computer architecture research questions are answered by running *benchmark* applications such as SPEC CPU, but these benchmarks do not contain representative data – rather, they use either random, constrained random, or null data. Therefore, to understand how common duplicate values are, this thesis turns to real workloads running in actual datacenters.

[3] Of course, the need to avoid paging had motivated the otherwise expensive page compression.

Results from counting duplicate values in these large-scale application memories, including results from well known companies such as Yelp and LinkedIn, indicate that duplicates account for nearly 2/3 of all memory content. If this could be exploited with no overhead, then 2/3 of memory is made redundant, or capacity is increased by 3x. To eliminate duplicate values, but still provide the abstraction of memory, the relationship between original content and unique content must be memorized, but in a compressed form – this is known as the *translation*. Including the cost of translation, the savings are reduced to 2.21x: conservatively, a factor of 2x increased capacity, or 50% cost savings on memory. Looking at datacenter total cost of ownership, or TCO, this translates into 15% savings, or $75 million over the lifetime of a datacenter.

1.4 Deduplicated Memory: Performance and Power

Motivated by the high potential gains, this thesis investigates the cost of deduplicated memory in terms of application performance and memory system power. In the worst case, the time to read a memory location, or *read latency*, is doubled because a deduplicated memory read requires first a translation fetch, and then a data fetch. Furthermore, a deduplicated memory write requires first duplicate search and then translation update, thus making what was originally a single memory operation into several. Therefore, deduplicated memory can increase both *read latency* and *memory bandwidth*. Prima facie, higher read latency degrades program performance and increasing bandwidth requires more power.

This thesis proposes novel optimizations that reduce the performance impact of deduplicated memory. Using the optimizations, this work shows that, in general, performance remains largely unchanged. More specifically, translation fetch is removed from the critical path using a direct translation buffer: a cache similar to the well known translation lookaside buffer. The cost of the direct translation buffer is low, relative to other caches, because the translation itself is a compressed representation of the original memory content. Further, the processor last level cache is extended to be duplicate aware, an optimization referred to as deduplicated cache or DDC. The DDC provides an effectively larger cache capacity, therefore reducing cache miss-rate, and therefore reducing memory bandwidth. To reduce memory accesses on write-back, the DDC also provides in-cache duplicate search. If a duplicate is not found by in-cache duplicate search, the DDC allocates an entry, and defers the costly global duplicate search until that entry is evicted and written back to main memory. A state-of-the art simulator is modified to model performance of systems with deduplicated memory. Simulation results indicate that a minority of applications suffer minimal performance degradation, and, on many applications, performance actually improves as a result of increased cache capacity.

In addition to providing increased capacity, deduplicated memory also saves power because the number of memory devices is reduced. With fewer memory devices, each individual device serves more requests and therefore power per device goes up. Although power per device increases, the savings on replacing two devices with one outweighs this increase. The deduplicated cache further improves power savings because it reduces the number of memory operations, in many cases to an amount lower than that required by the original non-deduplicated system.

1.5 Deduplicated Sparse Matrices

Using hierarchical deduplication, as used by HICAMP, this thesis shows that in-memory representations of large scale physical problems – or *sparse matrices* – can be significantly reduced in size. In the deduplicated memory introduced in Section 1.4, no duplicate values remain from the original content, but its compressed representation, the translation, may have duplicate entries. A hierarchically deduplicated memory also deduplicates the translation, and that translation's translation, until the original content is described by the connections between a set of unique memory blocks. Because certain sparse matrices contain many replicated sub-matrices, hierarchical deduplication can provide compounded benefit: a factor in excess of 4096:1 compression is achieved by a hierarchy of 2:1 compressions.[4] Using off-the-shelf hardware and software, this thesis shows that such memory savings improve performance in the context of scientific and industrial computing. HICAMP provides further speedup because it inherently recognizes the connections between unique memory blocks in hierarchical deduplication.

1.6 HICAMP Architecture

This thesis was inspired by previous work on the HICAMP architecture [24], which had been introduced to provide architectural support for parallel programs. Technology trends have pushed parallel computing into the mainstream: once the domain of supercomputers, today, even cellular phones have highly parallel processors. When a program is structured such that many machines can simultaneously work on the same problem, conflicts over the actual state of memory can arise. This happens when two or more machines, unbeknownst to each other, simultaneously attempt to update the same memory location – a problem referred to as a *memory race*. Memory races are a challenging problem: programmers who are aware of the issue can still introduce subtle, but fatal, program bugs known as *deadlocks*. A deadlock occurs when the programmer stops one portion of the program to prevent a race, but inadvertently introduces a cyclic chain of such dependencies eventually blocking the progress of the *entire* program. The specter of deadlock causes many programmers to sacrifice parallel program efficiency for program correctness.

[4] This is lossless compression, later referred to as "compaction."

HICAMP addresses these issues by providing a multi-version memory, made efficient by deduplication. In HICAMP, each concurrently executing portion of a program logically accesses and updates its own version of memory, including shared program state. As each concurrently executing task finishes, its updates are published, or *committed*, to the most up-to-date version of memory. Observing that most concurrent tasks do not produce true memory conflicts, HICAMP detects and resolves conflicting updates at the time of commit, thus allowing the most work to be accomplished in parallel.

1.7 Summary

As described above, and as noted in the literature, application trends are driving demand for memory capacity up relative to demand for compute performance. Further, the dominant memory technology is showing diminishing returns in terms of capacity. Therefore, this thesis addresses a critical problem, namely that of providing extra memory capacity.

Using techniques described by HICAMP, this thesis proposes fine-grain memory deduplication for current CPU architectures. By analyzing memory content in actual large-scale application memories, this thesis demonstrates that duplicates are common enough to be reliably exploited. It shows that the cost of duplicate suppression is very low, in terms of performance, and that memory system power is reduced.

Looking toward the future, continued performance gains must come from implementing parallel programs. As shown by the HICAMP Architecture, a multi-version main-memory is a natural fit for parallel programs. On the other hand, the cost of a multi-version memory is prohibitively high without duplicate suppression. Therefore, this thesis addresses the critical problem of providing extra memory capacity, and, by enabling multi-version memory, it also addresses the longer term problem of providing architectural support for parallel programs.

Chapter 2

Deduplicated Memory

The purpose of deduplicated memory is to provide extra capacity, but with the same physical resources – that is, to directly reduce the cost of memory capacity by never storing the same data twice. This chapter shows that deduplicated memory is feasible using current commodity memory devices and that it incurs only modest overhead. In specific, it shows that by reprogramming the CPU memory controller logic, deduplicated memory can be implemented using industry standard memory modules with no further changes. To prevent duplicate entries from being allocated, deduplicated memory must be able to check if some given data is already stored. Further, to maintain its function as a memory, deduplicated memory pays a cost, referred to as *translation*, to record the relationship between original memory content and the set of unique memory blocks that have been deduplicated. This chapter analyzes these costs. In specific, it shows that the proposed hash-based global content search is efficient and has few hash table overflows. Because of the cost of translation, deduplicated memory pays off only if enough duplicates exist. To motivate the remainder of the thesis, this chapter shows that duplicates are indeed common, that they occur with high reliability in many large-scale workloads, and that after accounting for the cost of translation, memory capacity can be doubled.

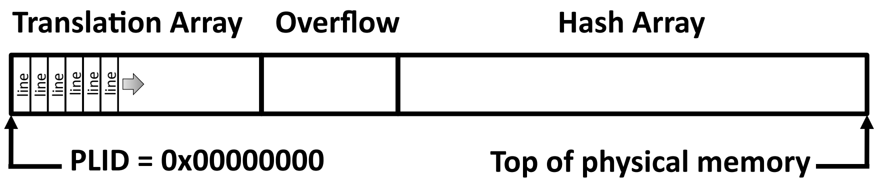


Figure 2.1: Deduplicated Memory Layout

2.1 Deduplicated Memory Organization

A deduplicated memory, or DDM, presents the abstraction of a standard computer memory, but application memory content is made unique at some granularity. To preserve application memory, the relationship between original content and unique content must be memorized, but in a compressed form – this is referred to as the *translation*. The translation has one entry per nominal memory location and each such entry specifies a certain block of unique content. If, for example, an application stores the same sequence of values to memory several times, then multiple entries in the translation refer to the same block of unique content. Logically, entries in the translation must be smaller than the unique blocks of memory. To maintain compatibility with existing hardware and software, deduplicated memory exposes the same interface as standard computer memory. To write to deduplicated memory, first, a duplicate search is performed, and second, the translation is updated.

In a deduplicated memory, the CPU does not have direct access to physical memory. Rather, the physical memory is managed by the memory controller as an array of 64 byte *memory lines*. Shown in Figure 2.1, the lines are identified by *physical line id*, or *PLID*, and are divided into three regions: the translation array, the hash array, and the overflow. When there is no ambiguity, a line is referred to by its purpose: as in *data line*, a line that stores unique content in the hash array, or *translation line*,

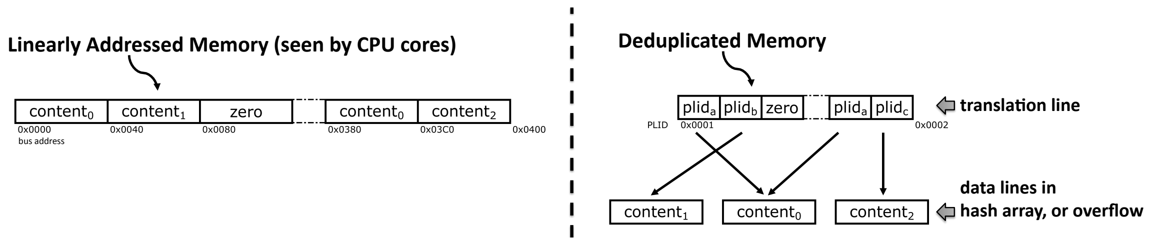


Figure 2.2: Address Translation and Content Deduplication

a line that stores several PLIDs and provides a mapping from processor bus address to deduplicated content.[1] As an optimization, a zero stored as a translation refers to a completely zero valued data line. Thus, the content referred to by the zero valued PLID never needs to be fetched from memory, and the PLID referring to zero valued content is known without the need for duplicate search.

To distinguish from locations in physical memory, this work refers to the nominal location of memory content, as seen by the CPU, as a *bus address*. Therefore, a bus address specifies a particular translation line, and also an entry in that translation line. That entry contains a PLID, which in turn specifies a particular data line. Figure 2.2 illustrates these ideas by showing the view of memory as seen by a CPU core and the actual memory content after deduplication. In the illustration, there are logically five lines stored in the system, but only three data lines and one translation are actually required.

2.2 Processor Reads and Writes

A processor write causes the deduplicated memory to search for a pre-existing instance of the content being written. This operation, referred to as *content lookup*, returns a PLID that points to a data line. On lookup, if no pre-existing identical content can

[1] A *reference count line* contains packed reference counts and is stored in the hash array, a *signature line* contains packed data line signatures and is stored in the hash array, an *overflow line* contains program data and is stored in the overflow region.

Read Sequence	Write Sequence
1. Read translation line	1. Hash Content
2. Extract PLID	2. Lookup content, using hash value from (1)
3. Read content	3. Read translation line
	4. Extract PLID from translation line from (3)
	5. Update translation line from (3) with PLID from (2)
	6. Write translation line from (5) back to memory
	7. Increment reference count of PLID from (2)
	8. Decrement reference count of PLID from (4)

Table 2.1: Read and Write Sequences

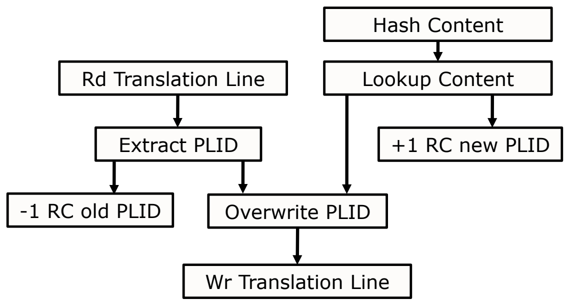


Figure 2.3: Write Sequence Dependencies

be found, a new data line is allocated. The write is recorded by storing the PLID in the translation array at an offset determined by the bus address. On read, the deduplicated memory returns a copy of either a data line or overflow line.

Table 2.1 shows the specific sequence of actions for a processor read or write. Without further optimization, deduplicated reads require two memory fetches, therefore potentially causing a CPU to stall while waiting on an additional memory read. On the other hand, the more complicated write sequence does not directly impact program performance because CPUs do not need to wait for writes to complete. Many steps in the write sequence can execute concurrently as illustrated in Figure 2.3. Logically, the write sequence requires only content lookup (Table 2.1 step 2), and translation update (Table 2.1 steps 3, 5, and 6): several other steps are listed for completeness; their purposes are explained in the following sections.

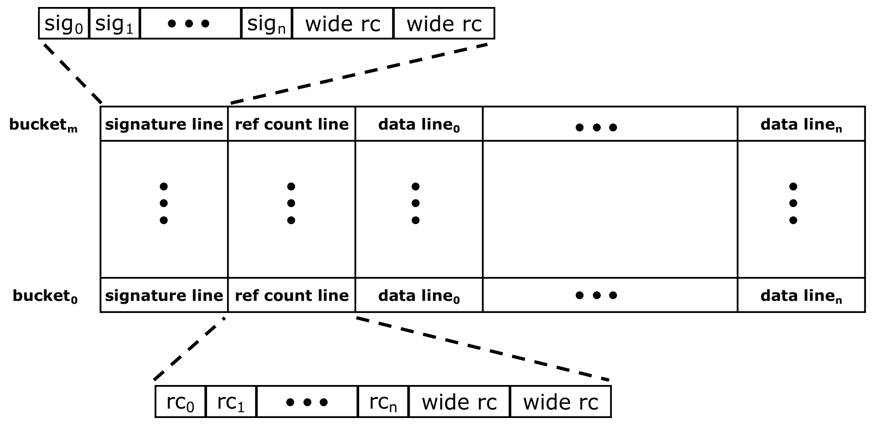


Figure 2.4: Hash Array Layout

2.3 Content Lookup

A deduplicated memory must function both as a linearly addressable memory and as a content addressable memory, or CAM. Compared to a linearly addressable memory, which uses an array index to access content, a content addressable memory uses content as an index into the memory. Typical implementations of CAM in hardware, often used in internet routers and switches for IP lookup, provide very low latency at the expense of high power and low capacity. On the other hand, a deduplicated memory needs high capacity, but CAM functionality is only required on write – because write latency usually does not impact CPU performance, the deduplicated memory can trade latency for capacity when implementing CAM.

To distinguish from typical CAM hardware, deduplicated memory implements an operation referred to as *content lookup* which indexes memory using content and returns the PLID which points to the data line containing that content. If no pre-existing instance of the content is found, space is allocated, the content is written, and the PLID is returned.

Content Lookup
1. Read signature line from hash bucket
2. Compare signature of content against entries in signature line
3. If no signature match, but a zero entry exists in signature line:
... 3.1. Allocate: insert signature of content into signature line from (1)
... 3.2. Write signature line to memory
... 3.3. Write content to memory
4. If no signature match, and no zero entry in signature:
... 4.1. Allocate in overflow: read next free entry from overflow free list
... 4.2. Write content to memory at location given by (4.1)
5. If signature match(es) exist:
... 5.1. For each signature match, read and compare content
... 5.2. If a content match is found: Return PLID for content
... 5.3. If no content match is found: Allocate content, as in (3) or (4)

Table 2.2: In-Memory Content Lookup

To avoid the high cost of special purpose CAM hardware, deduplicated memory uses an efficient in-memory hash table described by Cheriton, et al., in their work on HICAMP [24]. Shown in Figure 2.4, it contains a number of hash buckets, m, each with a number of data lines, n. Each bucket also contains two additional memory lines of metadata: the reference count line and the signature line. The signature line is used to optimize content lookup. For each of the n data lines in its bucket, it contains either a signature (a small, non-zero, secondary hash value) or zero to indicate a free line. Because the content of every line in a given bucket is represented in a compressed form in each signature, reading the signature line effectively searches the entire hash bucket in a single DRAM operation. For content lookup, two common cases exist: either no signature match exists and a free line is immediately allocated based on a zero entry in the signature line, or exactly one signature match exists and a subsequent data line read and content comparison verifies that a duplicate has been found. If there is more than one signature match, multiple data line reads and content comparisons are required. Table 2.2 describes the complete content lookup sequence.

To provide efficient deallocation when content is no longer in use, the deduplicated memory keeps a *reference count* for each data line in the hash array. To minimize

Symbol	Meaning
m	Number of hash buckets
n	Bucket capacity (ways)
u	Number of unique memory lines
l	Table load: $l = u \,/\, (m \cdot n)$
o	Number of memory lines that spill to overflow
p_b	Probability of hashing to a given bucket
k	Load of a given hash bucket
$P(k)$	Probability that a given hash bucket has load k
$N(k)$	Expected number of buckets with load k

Table 2.3: Hash Table Notation

reference count overhead, all reference counts for a given hash bucket are packed into a single memory line – the reference count line – which contains a narrow (1 byte) reference count for each data line in the bucket. If a given data line's reference count is greater than 250, a value between 251 and 254 indicates that its reference count is stored in one of the wide entries, also in the same reference count line. In the unlikely event that more wide reference count entries are needed than available,[2] the narrow reference count entry is assigned the value 255 to indicate "stuck at infinity," after which, it cannot be deallocated. The reference count lines are updated in accordance with the write sequence shown in Table 2.1.

2.4 Hash Bucket Overflows

Even using an ideal hash function, as content is added to memory, some hash buckets fill up before others – therefore, the deduplicated memory includes an overflow provision. This section first analyzes overflow behavior in the hash table from Section 2.3 and then explains allocating memory content in the overflow provision.

To minimize overflows, deduplicated memory uses a highly associative hash table, that is, a hash table with many *ways* or entries in each bucket. An ideal hash function

[2] Several highly referenced lines would need to map to the same bucket.

produces a uniform random distribution of hash values which, in turn, produces a binomial distribution in the hash buckets. Very strong hash functions with highly efficient hardware implementations already exist [14], therefore, deduplicated memory uses such a function. Using the notation shown in Table 2.3, the following analysis [18] provides hash table overflows as a function of table load and the number of ways per bucket. Assuming an ideal hash function, the following is by definition:

$$p_b = \frac{1}{m} \tag{2.1}$$

The probability of a bucket with a given load k, $P(k)$, and the number of buckets with that load, $N(k)$, follow from the binomial distribution:

$$P(k) = \binom{u}{k} p_b^k (1 - p_b)^{u-k} \tag{2.2}$$

$$N(k) = m \cdot P(k) \tag{2.3}$$

Using Equation 2.3, the total number of memory lines that spill to overflow can be calculated by summation:

$$o = m \cdot \sum_{k=n+1}^{\infty} (k - n) \cdot P(k) \tag{2.4}$$

When the hash table is under heavy load, the Poisson approximation can be used in place of the binomial distribution. Using the Poisson approximation, and normalizing the number of overflows, o, to the total number of entries in the table, $m \cdot n$, the percentage of lines that overflow is:

$$\textit{percent overflow} = \frac{1}{l \cdot n} \cdot \sum_{k=n+1}^{\infty} (k - n) \cdot \frac{(l \cdot n)^k}{k!} e^{-l \cdot n} \tag{2.5}$$

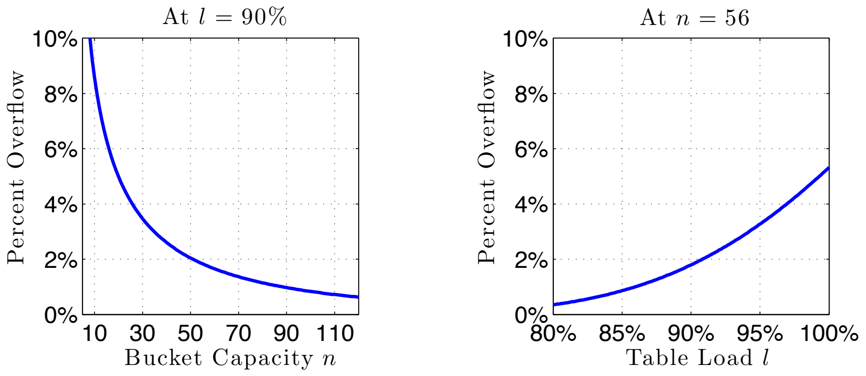


Figure 2.5: Minimal Overflows

Hash bucket overflows, therefore, occur with some probability based on hash table load, l, and bucket capacity (or ways), n.

Figure 2.5 plots the percentage overflow (Equation 2.5) both as a function of table load and as a function of bucket capacity. Two observations follow: the overflow percentage is a decreasing function of bucket capacity, and, with sufficiently high bucket capacity, there are few overflows, even at high table load.

For simplicity, and because few lines spill to overflow, overflow lines are not deduplicated. Using a well-known technique [64], a list of unused overflow lines, referred to as the *free list*, is kept in the unallocated overflow memory. If the content lookup operation (Table 2.2) cannot allocate in the hash bucket, it allocates an overflow line from the free list. Therefore, the number of memory operations for content lookup is bounded by the bucket capacity, n. Although $O(n)$ operations are required in the worst case, the cost of content lookup seldom approaches this limit because the probability of multiple false signature matches is very low. Bucket overflow is normally detected without the need for content fetch and compare. Because the free list entries are packed, overflow allocation normally requires no additional memory accesses, and at most, requires one extra memory access to fetch the next free list entry.

2.5 Deduplication and Overhead

After deduplication, only unique memory lines remain. Ignoring the cost of translation, the maximum upside to deduplication is the ratio of total memory lines to unique memory lines, referred to as the *total-to-unique ratio* or TTU:

$$\textit{total-to-unique ratio} = \frac{N_{\textit{lines-total}}}{N_{\textit{lines-unique}}} \tag{2.6}$$

A large total-to-unique ratio indicates that many duplicates exist. For example, with TTU = 100x, logically, only $^1/_{100}{}^{th}$ of memory resources are required, or capacity is increased by 100x when using those resources provided.

The practical benefit, referred to as *compaction*, is limited by the overhead of storing the translation, and, to a lesser extent, limited by the overhead of reference count and signature metadata. Fundamentally, the ratio of memory line size to PLID size sets the maximum compaction. Smaller lines increase deduplication but also cost more in terms of translation overhead. Larger PLIDs allow more unique data lines to be stored, but also increase the overhead of translation. To demonstrate the benefits of deduplicated memory, this work uses the following parameters in its evaluation: a memory line size of 64 bytes, a PLID size of 4 bytes, and 56 ways per hash bucket.

In this implementation, the line-size to PLID-size ratio is 16 – therefore, the maximum compaction is effectively 16x, even if the TTU is 100x. With 56 ways per bucket, there is room for two wide (4 byte) reference count entries per hash bucket. As shown in Figure 2.5, with 56 ways, even under high table load, overflows remain low. Therefore, signature and reference count metadata incur approximately 2 extra bytes per unique data line. Revising the total-to-unique ratio to include all overhead – translation, reference counts, and signatures – the deduplication benefit, hereafter referred to as *compaction*, is given by the following:

$$compaction = \frac{N_{\textit{lines-total}}}{{}^{1}\!/_{16} \cdot N_{\textit{lines-total}} + {}^{66}\!/_{64} \cdot N_{\textit{lines-unique}}} \tag{2.7}$$

The memory line size, 64 bytes, is chosen to match the cache line size used on many current CPU architectures. The PLID size, 4 bytes, is chosen to be evenly divisible into the memory line size, but also to be narrow and therefore limit the translation overhead. One limitation of using narrow 4 byte PLIDs is that only 2^{32} memory lines or 256 GB of DRAM can be addressed. This limit can be overcome by using wider PLIDs, which increases the overhead of translation, or by using multiple deduplication domains. For example, a two-socket server board may implement a separate deduplicated memory for each socket. In this two-socket system, 512 GB of DRAM can be addressed, but up to two instances of a given line of content may exist at any given time. Although not investigated in this work, this strategy of using multiple deduplication domains is expected to provide nearly the same compaction: as shown in the next section, deduplication is effective for different memory sizes from 7 GB to 1024 GB and a large amount of the benefit comes from data lines with high reference counts.

2.6 Duplicates In Real Workloads

Given that the translation is a direct overhead to deduplicated memory, a key question is how common are duplicate memory lines, and how much compaction is achieved after including that cost. It is well known that many coarse grain duplicates exist in virtual machine hypervisors [66], therefore, fine-grain duplicates also exist in

Company	Workload	Dataset	Memory	TTU	Compaction
Facebook	Tao	Social Graph	144 GB	1.07x	0.98x
Lightminer	Benchmark	TPCH	256 GB	1.76x	1.54x
SAP	SAP HANA-One	Private	1024 GB	1.94x	1.68x
Quantifind	Data Mining	Social web data	64 GB	2.87x	2.37x
LinkedIn	Profile Page	Professional Profiles	48 GB	3.09x	2.52x
Arista	Build Server	Private	128 GB	3.10x	2.53x
UC Berkeley	Shark	Conviva Server Logs	68 GB	3.18x	2.58x
NHN	Memcached	Private	48 GB	3.36x	2.71x
Yelp	Hadoop (EMR)	Private	7 GB	4.21x	3.25x
Ayasdi	Data Mining	Patient Health Records	96 GB	5.30x	3.89x
Geomean	-	-	-	2.68x	2.21x

Table 2.4: Duplicates in Real Workloads

these systems. On the other hand, there is no published work describing how many duplicate values exist in non-virtualized application memories. Furthermore, this question cannot be addressed using standard benchmark programs, such as SPEC CPU: such benchmarks typically use random, constrained random, or completely null data. To answer this question, this thesis provides data from actual large-scale deployed workloads.

A two step process is used to count duplicates. To avoid corrupting memory content, an image of physical memory is recorded to disk using the minimally invasive Linux Memory Extractor, or LiME [63]. After LiME, *zest*, a separate program, is used to count the number of unique memory lines. For these results, at least three images were recorded, and the reported values are for the largest number of unique lines found in each set. Little variation was observed in any given set of three, an observation consistent with prior results reported for memory compression (as opposed to deduplication) methods [28, 55].

Table 2.4 shows the results from *zest*, both as a total-to-unique ratio and as effective compaction. All of the workloads shown in Table 2.4 are non-virtualized. The results show that duplicate values are common in actual large-scale applications, and that in many cases, enough duplicates exist to provide an effective factor of 2x in

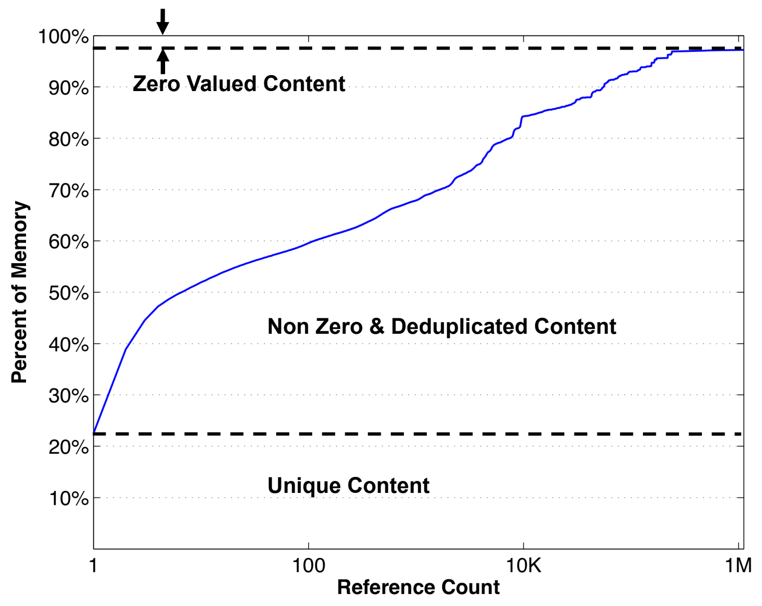


Figure 2.6: Cumulative Reference Count Histogram

memory capacity improvement. In aggregate, there are almost three times as many stored values as unique values.

Figure 2.6 shows that a significant amount of the savings comes from lines with a high reference count. In more detail, it shows the percentage of the original memory which had a reference count less than or equal to its corresponding coordinate on the abscissa.[3] About 22% of the original memory content was completely unique and therefore had a reference count of exactly one. On the other hand, 40% of the original memory had a reference count of 100 or more. The slope, measured at a given reference count, indicates the amount of memory that would have been consumed, i.e. without deduplication, by memory lines with that reference count.

Although much of the savings come from high reference count lines, the most common reference count is one, as indicated by its outsize contribution to the total:

[3] Data in Figure 2.6 is from Quantifind.

namely 22% versus just a few percent, or less, for all reference counts greater than one. Despite this, the accumulated effect of deduplication pushes the total-to-unique ratio almost to three and the resulting compaction is 2.37x.

As expected, not all workloads create many duplicate entries in memory. For example, the application Tao, running at Facebook, has a total-to-unique ratio of only 1.07x which is not enough to cover the overhead of translation: for Tao, compaction is 0.98x. Other workloads, not sampled in this study, may also have a very low total-to-unique ratio. For such workloads, deduplication does not pay off and only incurs the cost of extra memory operations for content lookup and capacity overhead for translation.

On the other hand, all results in Table 2.4, except Tao, have a high, or very high, total-to-unique ratio. Since Table 2.4 includes all data from all respondents, this suggests that duplicate values are common even in workloads not yet sampled. Based on this data, by geometric mean, average compaction is 2.21x. Thus the results indicate that duplicates are indeed common in deployed large-scale applications. And, for many applications, deduplicated memory reliably provides 2x compaction.

2.7 Operating System Support

Although deduplicated memory, as described to this point, is transparent to the operating system, some modification to operating system memory management is prudent. As shown in Section 2.6, many applications have a very high steady-state total-to-unique ratio – on average, 2.68x – but, if a new application is started, or if application behavior changes, this total-to-unique ratio may drop. If the operating system is unaware of the current degree of compaction, there is a risk that a write to the DDM will fail: this happens if the write maps to a full hash bucket and there are no free overflow lines. Such a write failure can, in turn, cause an unrecoverable

CPU error – an unacceptable situation. To prevent such unrecoverable errors, the operating system memory manager needs to know both the apparent size of memory and the actual utilization of memory resources.

To supply this information, namely the actual utilization of memory resources, deduplicated memory exposes an additional interface referred to as the *statistics interface*. This interface can be accessed with current CPU instruction sets by extending pre-existing opcodes.[4] The statistics interface provides both the total number and the number of allocated lines in both the hash array and overflow region. It also provides reference count sampling based on ranges of CPU visible bus addresses. In particular, it provides a count of lines that would be deallocated – i.e. data lines whose reference count is one and overflow lines – if a page was completely overwritten. The deduplicated memory maintains internal registers which track both hash array and overflow utilization and it issues an interrupt if the overflow region is dangerously close to exhaustion.

As shown by transparent page sharing [66], implemented in VMware ESX, operating system techniques for overcommit of physical memory are already well understood and tested. To prevent out-of-memory conditions, VMware ESX uses two techniques: *ballooning* and *demand paging*. Ballooning is used when apparent memory consumption is low, but actual utilization is high: the operating system allocates pages and writes them with zeros which drives up apparent memory consumption without using any extra physical resources. Demand paging frees memory by saving, to disk, the least recently used pages that also have a low reference count. Because of its significant performance penalty, most infrastructures are designed to avoid paging. Further, because the process of writing to disk is slow, there is a risk that paging cannot keep up with demand. As a last resort, the operating system pauses all user processes, stops allocating new memory, and continues paging.

[4] For example, in x86-64, opcodes `rdmsr` and `wrmsr`.

Given that the operating system tracks the true state of memory utilization through the statistics interface, and that it can do so in conjunction with requests for new memory from user processes, the deduplicated memory should not normally approach the threshold at which it issues an interrupt for low memory. Therefore, such an interrupt is also a last resort. The threshold for this interrupt depends on two parameters: the maximum rate at which overflow lines can be generated (essentially the LLC write bandwidth to DRAM) and the worst case time for the operating system to receive the interrupt and pause non-critical processes. To illustrate, consider a system with four DDR3 1333 MHz memory channels (42.6 GB/sec), with 64 GB of physical memory, and with 128 GB apparent capacity made visible to the operating system. Based on Section 2.4, conservatively, 10%, or 6.4 GB, is provisioned for overflow. Conservatively assuming that 5 milliseconds are required for the operating system to receive an interrupt, then the threshold is 220 MB of overflow remaining: only 3% of overflow space or 0.15% of apparent capacity. Therefore, memory can be nearly fully utilized before it is necessary to issue an interrupt for demand paging.

These techniques cover both steady state and transient memory demands. During steady state, ballooning prevents further overcommit because all apparent capacity is already marked as in use. Demand paging is invoked either when the operating system detects a near out-of-memory condition, or by an operating system interrupt, issued by deduplicated memory. The successful use of transparent page sharing in VMware ESX provides evidence that these techniques enable both correct and reliable deduplicated memory management.

2.8 Summary

This chapter shows that deduplicated memory is feasible using only commodity memory devices, that the cost of deduplication is low, in terms of translation overhead and hash table overflows, and that many duplicates exist in real application memories. In more detail, deduplicated memory is implemented by modifying the memory controller logic to perform the read and write sequences shown in Table 2.1. Deduplicated memory uses the highly efficient in-memory hash table described by HICAMP [24], which, in the common case, achieves content lookup in only two DRAM operations.

Translation overhead is minimized by using a line-size to PLID-size ratio of 16. Although this limits maximum compaction to 16x, this sufficiently exceeds typical total-to-unique ratios, and therefore, this cost is minimal. Further, this chapter proposes a highly associative in-memory hash table and shows that such highly associative hash tables minimize bucket overflows.

Importantly, duplicates are common in deployed, large-scale, application memories. While some applications do not contain many duplicates, many more do. Across all applications sampled, the average compaction is 2.21x. Therefore, this chapter shows that deduplicated memory reliably provides a factor of 2x extra memory capacity for a wide variety of high-memory, large-scale, workloads.

Chapter 3

Performance Optimizations

Based on results from actual large-scale applications (Table 2.4), deduplicated memory increases capacity by 2x, but it requires extra memory accesses for content lookup and, in the worst case, doubles memory read latency – this chapter describes simple and low cost hardware caches that can be used to mitigate these costs. As shown in Table 2.1, without further optimization, both deduplicated reads and writes require more memory accesses and therefore take longer to complete – that is, deduplicated reads and writes have *higher latency*. Although most reads are served from the CPU data cache, on cache miss, the CPU can stall while waiting on a value to be read from memory. Therefore, the additional latency incurred by translation fetch can degrade program performance. Although deduplicated writes have even higher latency, application performance is not very sensitive to this because, logically, the CPU does not have to wait for writes to complete. Although writes do not directly harm performance, their extra memory operations can consume hardware resources and starve reads from accessing memory.

This chapter describes two types of caches that optimize performance for deduplicated memory. The direct translation buffer (DTB) optimizes reads by removing translation (Table 2.1, read sequence) from the critical path. The deduplicated

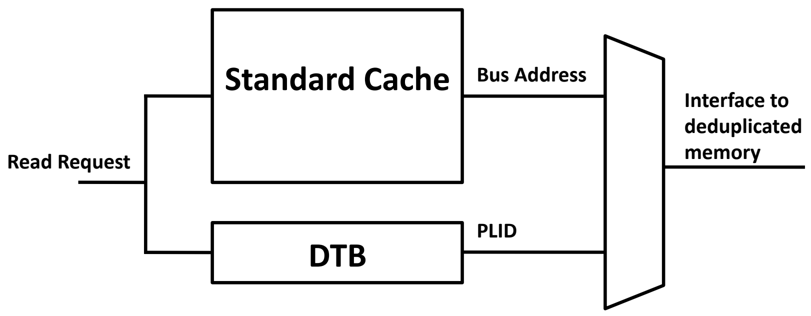


Figure 3.1: Direct Translation Buffer

cache (DDC) attempts to serve all deduplicated memory operations – including content lookups and reference count updates – and therefore further reduces memory accesses. Because of deduplication, the DDC is effectively larger and therefore also improves program performance.

3.1 Direct Translation Buffer

Shown in Figure 3.1, the direct translation buffer, or DTB, removes translation fetch from the critical path. The DTB caches translation lines and is searched in parallel with the last non-deduplicated cache. In the event of a cache miss, but DTB hit, the data line can be accessed directly because the translation has already been obtained: in this case, the read latency is exactly the same as with a non-deduplicated system. Because each cache line contains many translations, the DTB does not need to be very large to provide a significant hit rate. For example, if a program makes sequential data accesses, only one out of every 16 cache misses requires an extra translation fetch: the remaining cache misses all hit in the DTB. Because application performance is much more sensitive to read latency, as opposed to write latency, with sufficient hit rate, the DTB enables a deduplicated memory with little to no performance degradation.

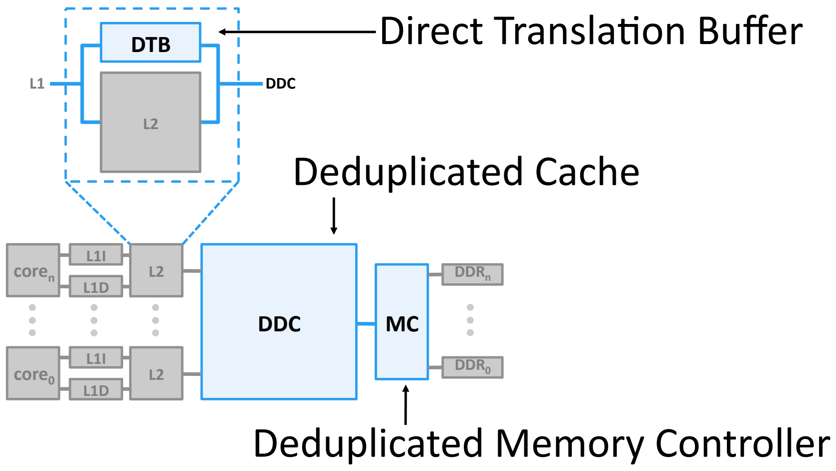


Figure 3.2: CPU With DDC and DTB

3.2 Deduplicated Cache

The deduplicated cache (DDC) implements a cached version of deduplicated memory. It exposes the same interface as a normal last level cache (LLC), but explicitly caches translation lines, data lines, and reference counts. The main purpose of the DDC is to reduce the number of DRAM accesses required by deduplicated memory, but, because of deduplication, it also provides an effectively larger cache capacity. Therefore, programs that are sensitive to LLC size also experience an actual performance gain. Figure 3.2 illustrates a multi-core CPU system with both DTB and DDC.

In specific, the DDC executes the read and write sequences (Table 2.1). For the read sequence, either (or both) translation read and data read can hit in the cache. For the write sequence, because the cache is maintained inclusive,[1] the translation line read and update always hit in the cache. For writes, the DDC also provides in-cache content lookup. If an identical data line to the content being written is currently held in the DDC, then the DDC serves the content lookup operation without accessing

[1] Inclusive LLC is common in current CPUs.

DRAM. If such a data line is not cached, there are two options: immediately perform content lookup in DRAM, or allocate in-cache without deduplication. The following subsections elaborate on the details of the DDC, starting with cache-coherency.

3.2.1 Coherency

As the last level cache, the DDC is responsible for implementing the cache coherence protocol. The cache coherence protocol guarantees that any read obtains the most up-to-date version of a cache line: if $core_0$ updates a line, then $core_1$ updates the same line, and *then*, $core_0$ reads that line again, $core_0$ will receive the value written by $core_1$ – as you would expect. But, this *anticipated* behavior is non-trivial to guarantee: the value cached in the LLC, and indeed the value written to DRAM, may be stale – therefore, in such cases, the read must be served from the L1 or L2 cache that belongs to the CPU core having executed the most recent write.

This work investigates the performance of an inclusive DDC that implements a modified-exclusive-shared-invalid (MESI) protocol using directory information stored in, or with, the DDC.[2] In the MESI protocol, a write causes the L1 cache to request a line in the exclusive state – after which, it can locally transition the line to modified. A request for exclusive invalidates any copies of this line held in other L1 and L2 caches (see Figure 3.2 for an illustration of a typical system). Similarly, a read from one core can downgrade a line held in a separate L1 or L2 cache, from exclusive or modified, to shared. Most often, no such downgrades and invalidations are required. To avoid sending unnecessary downgrade and invalidation commands, the DDC tracks which child caches have what data. To do so, it stores a sharing vector per translation line. The sharing vector requires one bit per translation entry per CPU, or $16 \cdot N_{CPU}$ bits.

Because a translation line contains references to several data lines, evicting one translation line from the DDC can cause multiple invalidations in any single child

[2] This is similar to many current x86-64 CPU designs.

cache. In the worst case, all 16 cache lines referred to by a given translation are invalidated in several children. This scenario is rare because content referred to by cold translation lines is typically already evicted from the caches closer to the cores.

Data lines are immutable over their lifetime. Therefore, they cannot be inconsistent and thus never require any coherence protocol actions. Although inclusion is enforced for translation lines, without a back-map from data to translation, it is not possible to guarantee inclusion for data lines. In other words, inclusion guarantees that if a child cache holds a particular cache line, then the translation for that line is in the DDC. Because translations are packed, 16:1 in this work, this actually reduces the cost of inclusion – cold data lines are evicted regardless of whether a child cache has a copy, and refills follow demand. Although immutable, a data line is considered dirty if its cached reference count changes. Evicting a "dirty" data line is actually reference count write back, but can cause a data line deallocation in the DDM.

The coherence directory sharing vector can be used to further optimize certain cache accesses. Recalling that inclusion cannot be maintained for data lines, for a read that has a translation line hit, but data line miss, it is possible that a child cache has a copy of the data. This can be detected and exploited using the sharing vector – although a data line miss normally requires DRAM access, this data line miss can be served out of the child cache. Such accesses often require an invalidation or a downgrade from exclusive to shared, therefore serving the content out of a child cache is a built-in cost, and in any event, lower latency than accessing DRAM. In a related scenario, although DDC reads normally require two cache accesses, a read for exclusive access only requires one access if it is actually an upgrade request, that is if a child cache requests a shared to exclusive transition.

RC Status Bits	Meaning
00	Unmodified absolute value
01	Modified (dirty) absolute value
10	Reference count delta (RCΔ)
11	RCΔ, was absolute one

Table 3.1: Cached Reference Count States

3.2.2 Physical Resources

The DDC uses the same underlying physical resources as a standard set associative LLC, but a small amount of additional tag data is required. The tag overhead depends on certain implementation choices: for example, if the sharing vector is stored in the tag, then each translation line requires $16 \cdot N_{CPU}$ bits of additional tag storage. This work evaluates two implementations – *partitioned* and *merged* – and compares them in terms of tag overhead and, in Chapter 4, in terms of performance.

Partitioned DDC

The partitioned DDC reserves a certain number of cache ways to contain only data lines while the remaining ways contain translation lines. Because a small fraction of the total is translation, a large translation tag does not significantly impact the total resources required. Therefore, in the partitioned DDC, the translation line tag includes the sharing vector in addition to the MESI bits. For data lines, the MESI bits are repurposed to indicate the reference count status, as shown in Table 3.1. Therefore, the data line tag identifies its PLID, and tracks the reference count and its status. As an implementation choice, this work uses 16 bits for the cached reference count – if these bits overflow, or underflow, the reference count is written back to memory and its status is reset to RC$\Delta = 0$.

Common System Parameters			
CPU address space	*addr-bits*	48	bits
Line size	*line-size*	64	bytes
Number of CPUS	N_{CPUs}	8	cores
LLC size	LLC_{size}	20	MB
Total LLC cache lines	N_{lines}	327,680	lines
Number of LLC banks	LLC_{banks}	10	banks
LLC bank associativity	LLC_{assoc}	16	ways
LLC sets per bank	N_{sets}	2048	sets
L2 size (per core)	$L2_{size}$	256	kB
PLIDs per translation line	N_{PLIDs}	16	PLIDs

Conventional LLC			
Line address tag	42 - $\log_2(N_{sets})$	31	bits
Tag + metadata	31 + 2 MESI + 8 sharing	41	bits
Tag store	41 bits $\cdot N_{lines}$	1,640	kB
Data store	64 bytes $\cdot N_{lines}$	20,480	kB
Total	tag store + data store	22,120	kB

Partitioned DDC			
Translation line tag	42 - $\log_2(N_{sets})$ - $\log_2(N_{PLIDs})$	27	bits
Data line tag	32 - $\log_2(N_{sets})$	21	bits
Translation tag + metadata	27 + 2 MESI + 16·8 sharing	157	bits
Data tag + metadata	21 + 2 RC status + 16 RC	39	bits
Translation size	$2/16 \cdot N_{lines}$	40,960	lines
Data size	$14/16 \cdot N_{lines}$	286,720	lines
Translation tag store	157 bits $\cdot N_{tr\text{-}lines}$	785	kB
Data tag store	39 bits $\cdot N_{data\text{-}lines}$	1,365	kB
Total	tag store + data store	22,630	kB

Merged DDC			
Translation line tag	42 - $\log_2(N_{sets})$ - $\log_2(N_{PLIDs})$	27	bits
Data line tag	32 - $\log_2(N_{sets})$	21	bits
Translation tag + metadata	27 + 2 MESI + 16 sharing	45	bits
Data tag + metadata	21 + 2 RC status + 16 RC	39	bits
Tag + metadata	max(translation tag, data tag)	45	bits
Directory cache entries	$N_{CPUs} \cdot L2_{size} / \textit{line-size}$	32,768	entries
Directory cache entry size	N_{CPUs}	8	bits
Directory cache tag size	42 - $\log_2(N_{sets})$	31	bits
Directory tag + metadata	31 + 1 valid	32	bits
Directory cache size	8 bits $\cdot N_{entries}$	160	kB
Tag store	45 bits $\cdot N_{lines}$	1,800	kB
Total	data store + tag store + directory cache	22,440	kB

Table 3.2: DDC Physical Resources

Merged DDC

The partitioned DDC is a simple design, but merging translation and data – allowing translation and data lines to compete for space in the cache – improves overall system efficiency. In the merged DDC, the degree of content locality in the underlying data access pattern determines the ratio between translation and data lines in cache. If the cores access sequential addresses, then the cache refills a translation only once every 16 accesses and the number of translation lines floats down to 1/16th of the total capacity. If the cores access random addresses referencing random values, then the cache refills a translation and data line on every access: in this case, the cache is filled with 50% translation and 50% data lines.

For the merged DDC, providing a complete sharing vector of $16 \cdot N_{CPU}$ bits for every cache line is wasteful over-provision: the physical resources required for a 20 MB DDC grow to 26,760 kB, an increase of 20% relative to a 20 MB non-deduplicated LLC. Therefore, the merged DDC uses a separate sharing vector cache: the number of entries in this cache depends on the total L2 capacity, and not on the number of DDC cache lines. Because the sharing vector cache is not guaranteed to contain an entry corresponding to a certain bus address, the translation line tag contains a fixed-size sharing vector with one bit per translation entry. If the sharing information is not found in the directory cache, but this bit is set, it implies that the specified bus address is cached in either one, or several, child caches – therefore the coherence protocol must conservatively send invalidations or downgrades to all cores. Finally, in the merged DDC, each tag must be large enough to serve the needs of either data or translation.

Table 3.2 compares the physical resources required for a conventional LLC, a partitioned DDC, and a merged DDC. The calculation assumes a CPU with 8 cores that has access to a 48 bit address space; both the conventional LLC and DDC use 64 byte cache lines. All caches, conventional and DDC, track which children caches are

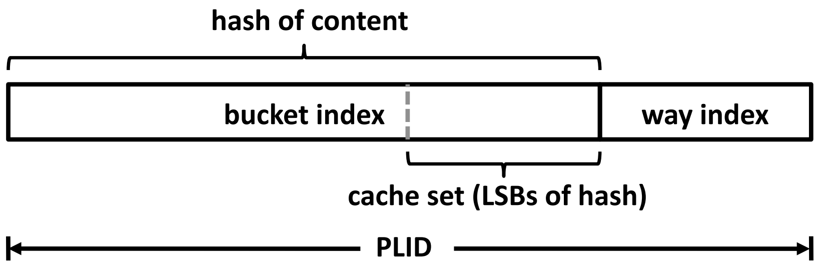


Figure 3.3: PLID Composition: hash bits and way index

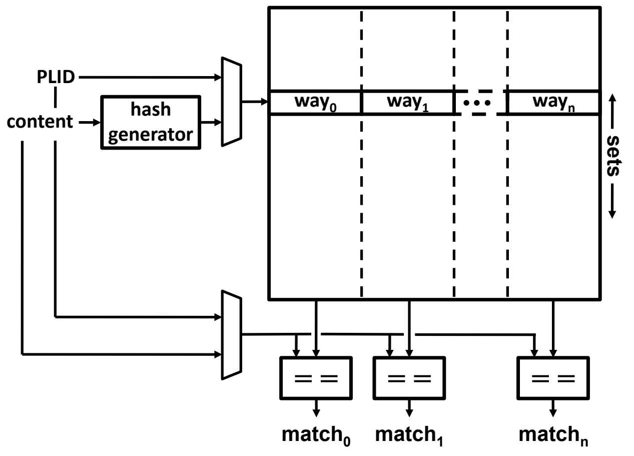


Figure 3.4: Conceptual Illustration of a Deduplicated Cache: hardware to provide cache search by address and in-cache content lookup

sharing a cache line on a per CPU basis. For the partitioned design, total resources grow by only 2% because the large translation tag impacts only a small fraction of the cache. By storing the sharing information in a separate cache, sized to achieve high utilization, the merged DDC actually incurs slightly less overhead than the partitioned DDC. Its overhead is 1.4%: significantly lower than the 20% overhead required to reserve space for a sharing vector for every possible cached translation.

3.2.3 Content Lookup

As proposed by HICAMP [24], the DDC provides in-cache content lookup. Normal caches only provide search by address, which returns data content. In addition to search by address, the DDC provides search by content, namely, the content lookup operation. To enable this, a data line PLID, as shown in Figure 3.3, contains both the hash of its content and the index to its entry (way index) in the DDM hash bucket. Therefore, a partial PLID can be generated by hashing the content: after obtaining this partial PLID, the correct cache set is selected and each data line is compared to see if it is a duplicate to that being searched on. Because this comparison can be performed in parallel across all lines in the set, the in-cache content lookup is accomplished in a single cache access. Figure 3.4 shows a conceptual illustration of a cache that supports content lookup.

This design enables the DDC to complete the entire write sequence (Table 2.1) without accessing DRAM. For content lookups that do not find a duplicate in cache there are two options: eager or lazy deduplication. With eager deduplication, a content lookup that misses in the DDC is immediately served by content lookup in DRAM. The next section describes lazy deduplication, referred to as in-cache allocation.

3.2.4 In-Cache Allocation

With in-cache allocation (ICA), a write-back to the DDC does not require content lookup to succeed, rather, if in-cache content lookup fails, then the content is directly written into the cache, possibly creating a duplicate value. Therefore, the write-back itself requires no DRAM access. On the first such write-back to a given bus address, the DDC must allocate an entry in which to store the content which typically requires that another line be evicted – this eviction may require DRAM access, i.e.

Translation Line Eviction	Data Line Eviction
1. Buffer dirty translation line 2. In parallel, for each TLID: ... 2.1. Lookup content in DDM ... 2.2. Update translation with PLID ... 2.3. Invalidate ICA content in DDC 3. Write translation line to DDM	1. Lookup ICA content in DDM 2. Update translation line using back-pointer 3. Invalidate ICA content in DDC

Table 3.3: Translation and Data Line Eviction for In-Cache Allocation

if the victim is a dirty translation line, a data line with non-zero reference count delta, or non-deduplicated (temporary) data. Rather than identify in-cache allocated content by PLID, because such content is not stored in DRAM, it is identified by *temporary line id* or *TLID*. A TLID is distinguished from a PLID either by adding one bit of metadata to the cache tag, or by stealing one bit from the PLID itself. Because this temporary content is not yet stored in DRAM, if its respective translation line is evicted, then memory is corrupted. To implement in-cache allocation, both translation and data eviction must be handled correctly.

Translation Line Eviction

When the DDC evicts a translation line that refers to some ICA content, it must first deduplicate that content and then update the translation. To do so, it buffers the evicted translation line and then issues content lookup commands to the DDM on the content referred to by the TLID(s). After a PLID is obtained for each ICA data line referred to by the TLID(s), the updated translation is written back to the DDM. The non-deduplicated ICA content is invalidated and the translation line eviction is complete.

Data Line Eviction

To evict a non-deduplicated ICA data line, the DDC first issues a content lookup operation to the DDM, but a back-pointer is required to update the translation. Although a general back-map is difficult to implement because the relationship between deduplicated content and translation is one-to-many, this back-map is feasible because ICA data is not deduplicated and therefore its relationship to translation is one-to-one. This back pointer is generated when the non-deduplicated content is allocated in the cache and used at time of eviction. Logically, the translation line must still be in the DDC because a translation line eviction has not yet caused this ICA content to be deduplicated. Therefore, as an implementation choice, this work uses a back-pointer large enough to span only the DDC, not the entire physical memory: the otherwise unused 16 bit reference count field (Table 3.2) is sufficient. Table 3.3 details the sequence for both translation and data line eviction.

3.2.5 Zombie Suppression

A zombie is a live data line whose absolute reference count is zero. Zombies occur because the DDC normally treats the reference count as a difference, rather than absolute value.[3] A negative RCΔ can cause deallocation on RC write-back: if so, then that data line *was* a zombie. Such a line cannot be referenced by a program, and keeping it in the cache reduces efficiency. Therefore, it is desirable to detect and evict potential zombies.

If the DDC does not know the reference count absolute value, then a heuristic method is needed for zombie detection. As shown in Figure 2.6, the most probable reference count is 1. Therefore, RCΔ = -1 is a good indicator of zombie status.

[3] RCΔ is used because it requires fewer DRAM read operations.

In-cache allocation allows immediate zombie suppression because it offers an extra degree of freedom in choosing a replacement candidate. In specific, the non-deduplicated ICA content can be placed in any cache set because it can be found directly, that is without search, by the TLID held in the translation line. Rather than picking an eviction candidate in the “correct” cache set, a DDC with ICA can overwrite data content in place. To suppress zombies, the DDC uses a simple heuristic to choose whether or not to use update in-place: if the reference count delta is -1, then update in-place is chosen because it is statistically likely that the line in question will be deallocated when its -1 reference count delta is written back to the DDM.

3.2.6 Prefetching

Two kinds of prefetching are feasible with deduplicated memory: translation prefetch, and data prefetch. Hardware prefetch units normally work by finding sequential or strided access patterns, or even more simply by always fetching the next cache line in sequence, regardless of access pattern. In the DDC, the same patterns can be observed by monitoring the bus addresses requested by the L2 caches.[4]

Therefore, by observing bus address access patterns, prefetch can be implemented with the DDC, but new prefetch policies are possible. For example, to speculatively read the translation for an entire 4 kB page requires only three extra cache lines after the initial page touch. And, because each translation line contains multiple translations, translation line prefetch is statistically more likely to pay off. For sequential access, data line prefetch is implemented using the current translation line to speculatively fetch the next, or next several, data line(s) in the sequence.

[4] Alternately, a prefetch unit can be implemented in each L2.

3.2.7 Multi-Bank Implementation

In current CPU designs, each request to the LLC is mapped to exactly one cache bank, but in a DDC, a translation line, and the data lines it points to, may map to different cache banks. As described to this point, data lines are unique across all DDC cache banks. In this scheme, referred to as *cooperative banking*, a request to the DDC is mapped to a particular cache bank based on the bus address, but the data line may need to be served from a separate bank. To simplify the design and to reduce read latency, the DDC can be designed such that both translation lines, and the data referred to thereby, all reside in the same cache bank. This scheme, referred to as *co-located*, trades DRAM bandwidth and effective cache capacity for read latency and design simplicity.

In more detail, co-location simplifies protocol design because a given cache bank cannot cause an eviction in any other cache bank. By contrast, with cooperative banking, serving a read in one bank may require a data line refill in a different bank. If data and translation share the same resources (merged design), this refill may evict a translation line, which would require a different bank from the one serving the read to take action according to the coherence protocol. In a partitioned design, this same read scenario is less complicated because the data line refill can only cause a data line eviction. In either case, with cooperative banking, this read is slower because it incurs the latency of two messages sent across the bank-to-bank interconnect. Therefore, co-location provides lower read latency and it significantly reduces complexity in the cache protocol.

On the other hand, with co-location, the mapping from unique content to cache bank is no longer one-to-one. This reduces the probability that a given in-cache content lookup succeeds which increases DRAM bandwidth relative to cooperative banking. Further, because duplicate data lines may exist in different cache banks, effective capacity is lower relative to cooperative banking. For simplicity, the DDC performance evaluation, presented in Chapter 4, does not explicitly consider the issue of multiple cache banks. Rather, it treats the DDC as one bank, and conservatively charges two full access latencies (i.e. both including the latency of interconnect) for translation and data access.

Although co-location logically does reduce effective capacity and increase bandwidth, neither of these effects may be significant. As shown in Section 2.6, much of the deduplication benefit is conferred by lines with a high reference count. As with multiple deduplication domains (Section 2.5), this suggests each respective cache bank will experience the same increase in effective capacity and the same number of successful in-cache content lookups. Therefore, evaluating the trade-off between co-location and cooperative banking is an interesting topic for future work.

3.3 Summary

This chapter describes cache techniques that can be used to mitigate the cost of deduplicated memory. In more detail, it describes a direct translation buffer (DTB) that removes translation from the critical path on DTB cache hit. The purpose and operation of the DTB are similar to the well known translation lookaside buffer (TLB) used to hide the cost of virtual to bus address translation. Because many PLIDs are packed into each translation line, a small DTB covers a wide range of bus addresses – therefore, DTB overhead is low relative to its benefit.

This chapter describes how the CPU last level cache (LLC) can be modified to implement a deduplicated cache (DDC). Through deduplication, the DDC provides extra cache capacity and therefore decreases the number of DRAM accesses for CPU reads. The DDC further reduces the number of DRAM memory operations because it implements in-cache content lookup which decreases the number of DRAM accesses on write-back. In more detail, this chapter shows that the DDC uses the same underlying physical resources as a standard LLC, and requires only a modest increase in tag storage overhead. The next chapter, Performance and Power, evaluates the performance impact of the deduplicated cache and its extensions, such as lazy deduplication.

Chapter 4

Performance and Power

Chapter 2 showed that deduplicated memory is feasible, and that it provides a factor of 2x increased capacity – using the direct translation buffer (DTB) and deduplicated cache (DDC) optimizations described in Chapter 3, this chapter shows that systems with deduplicated memory provide application performance at parity with their non-deduplicated counterparts and, at the same time, save power. A state-of-the-art cycle-accurate x86 CPU simulator is modified to model systems with deduplicated memory. To understand performance in various corners, several types of workload are simulated. A synthetic benchmark is used to stress the memory system and verify that the simulator produces reasonable performance results. Two different benchmark suites, SPEC CPU 2006 and PARSEC, are simulated to show general purpose and worst-case performance. Finally, three workloads from the targeted datacenter environment are simulated: PostgreSQL, a widely used database, memcached, a widely used webserver frontend, and Hadoop, a widely used distributed task engine.

Averaged across SPEC and the datacenter workloads, the results show an application performance improvement of 1.06x – indicating that deduplicated memory provides at least the same, if not better, performance as conventional systems. As expected, applications in the PARSEC benchmark have few duplicates and have random memory access patterns. In spite of this, PARSEC application performance remains at 0.98x – a far smaller degradation than expected for what is essentially the practical worst case.

Although deduplicated memory nominally requires more memory accesses to accomplish the same reads and writes to memory, the DTB and DDC are effective in filtering those accesses. Deduplicated memory with deduplicated cache and in-cache allocation, described in Chapter 3, Section 3.2.4, requires only marginally more bandwidth in the worst case: 1.38x for PARSEC and 1.36x for Hadoop. For other workloads, bandwidth is reduced to 0.79x relative to a non-deduplicated machine.

Because deduplicated memory reduces the number of memory devices, it also saves power. Assuming a 2x increase in memory capacity, deduplicated memory saves 40% of memory system power. This savings translates to 4% to 8% of total datacenter power given that memory is 10% to 20% of that total [10].[1] This savings is significant given that current datacenters are power constrained. Furthermore, in the worst case where deduplicated memory requires extra bandwidth, this bandwidth is already available because datacenter workloads use only a small fraction – about 5% – of available bandwidth.

In the following sections, this chapter elaborates on the above. It starts by describing the performance simulation methodology. After which, it provides a detailed analysis of the simulation results. Finally, using the simulation results, it shows power savings due to reduced device count, even in the worst case of increased bandwidth use.

[1] Barroso states memory is 30% of server power, therefore these numbers are derated for PUE.

4.1 Evaluation Methodology

To evaluate performance, this work uses *zsim* [61], a state-of-the-art cycle-accurate architectural simulator. Using dynamic binary translation [48], zsim intercepts all opcodes of the specified processes on a Linux/x86-64 machine – thus, zsim simulates a computer architecture based on an instruction trace from a process running on actual hardware. To simulate systems with deduplicated memory, zsim is modified to read memory content from the actual running process on every simulated LLC miss. In response to LLC misses, the zsim memory controller code is modified to perform the read and write sequences given in Table 2.1. For deduplicated cache, the zsim set-associative cache and coherence protocol objects are subclassed and modified to implement in-cache translation and in-cache content lookup. When simulating a system with deduplicated cache, application memory content is sampled as required by DDC misses.

4.1.1 Simulated Machine Configurations

This work compares performance amongst three categories of machine: a server-grade cache-coherent multicore (CC), the same machine with deduplicated memory (DDM), and with deduplicated memory and deduplicated cache (DDC machines). The baseline simulated machine is specified in Table 4.1. Its parameters are based on the cache sizes, memory channels, and latency measurements from an Intel Xeon E5-2670 CPU.

Table 4.2 describes the individual machine configurations: for DDC, three separate variants are simulated – using optimizations described in Chapter 3, each is an improvement on the prior. The DDC-P(artitioned) machine is the baseline DDC machine. The DDC-M(erged) machine improves on the DDC-P by allowing translation and data lines to compete for space in the cache. The DDC-ICA machine also uses a

Parameter	Value	Latency	Instances
Core Frequency	2.6 GHz		8
Private L1I	32 KB, 4-way	1 cycle	8
Private L1D	32 KB, 4-way	1 cycle	8
Private L2	256 KB, 4-way	18 cycles	8
Shared L3	20 MB, 16-way	22 cycles	1
Memory Channels	DDR3 1333 MHz	215 cycles	4

Table 4.1: Common Machine Parameters

Machine Name	Memory Type	LLC Type
CC	Standard	Standard
DDM	Deduplicated	Standard
DDC-P	Deduplicated	Deduplicated (partitioned)
DDC-M	Deduplicated	Deduplicated (merged)
DDC-ICA	Deduplicated	Deduplicated (merged) + ICA

Table 4.2: Individual Machine Configurations

merged DDC and implements in-cache allocation to further reduce DRAM accesses. Each machine with deduplicated memory has a direct translation buffer (DTB). The DDM machine has a 16 line (1024 byte) DTB, and the DDC machines have a 4 line (256 byte) DTB included with each private L2 cache. Because the baseline machine (CC), unmodified from zsim, does not have prefetch, prefetch is not implemented for any machine.

4.1.2 Performance Model

In zsim, the baseline machine is a modern 4-issue out-of-order architecture with a tiered and coherent cache system. As shown by Sanchez, et al. [61], zsim accurately models fine grain performance details as well as overall performance based on comparisons to actual systems. To further improve accuracy, that is, to remove any common mode simulation errors, results for machines with deduplicated memory are normalized to the simulated baseline system.

In zsim, instructions requiring memory access, and their dependents, are retired according to a latency calculated by the simulated cache and memory system. Simulated latency accumulates according to the fetches required by each level of the memory hierarchy and according to any invalidations required by the coherence protocol. For the DDM and DDC systems, the rules for accumulating latency are based on the read and write sequences given in Table 2.1. For example, in the DDC system, a read that requires both data and translation refill accumulates the latency of two cache accesses and two DRAM accesses: a total of 493 CPU cycles (Table 4.1).

Because the CPU does not wait for writes to complete, write latency has only an indirect impact on CPU performance. The simulator calculates write latency and accounts for memory system resource contention: in certain cases, the higher write latency causes the memory system to starve reads – succinctly, the effect of write latency is captured in the simulation. For DDM and DDC, the performance model includes all DRAM operations for content lookups, signature line updates, and reference count updates – in short, it includes all operations shown in Tables 2.1 and 2.2.

4.1.3 Performance Metrics

The following key performance metrics are measured: speedup, compaction, and bandwidth. In the following, the subscript *dd* is used for machines with deduplicated memory and *cc* for the baseline (cache-coherent) machine. Compaction, calculated by Equation 2.7, uses the total number of conventional line addresses accessed and the highest number of unique memory lines across the entire simulation. For deduplicated machines, the terms $bytes_{rd}$ and $bytes_{wr}$ include all DRAM operations for content lookup in DRAM, for signature lines reads and writes, and for reference count updates.

$$speedup = \frac{t_{exe\_cc}}{t_{exe\_dd}} \tag{4.1}$$

$$BW = \frac{bytes_{rd} + bytes_{wr}}{t_{exe}} \tag{4.2}$$

$$Additional\ BW\ Required = \frac{BW_{dd}}{BW_{cc}} \tag{4.3}$$

4.1.4 Simulated Workloads

To understand performance in different corners, several different benchmarks and applications are simulated. In particular, three different benchmarks are simulated: a synthetic benchmark that has both random and highly duplicated data, SPEC CPU 2006, a general purpose benchmark, and PARSEC, a highly parallel scientific and high performance computing benchmark. Additionally, three different representative datacenter workloads are simulated: PostgreSQL, Hadoop, and memcached. To more accurately capture the impact of deduplicated memory, for the latter two of these, an attempt is made to use representative input data. PostgreSQL uses constrained random input data as specified by the TPC-E benchmark.

Synthetic Benchmark

The synthetic benchmark is multi-threaded and each thread executes a small code kernel whose only purpose is to stress memory bandwidth. It is essentially read only and it has two data access patterns: random, and sequential, and two underlying datasets: random, and highly duplicated.

Standard CPU Benchmarks

SPEC CPU 2006 is used to understand general purpose performance and PARSEC is used to understand performance for applications with little to no deduplication benefit. For SPEC CPU 2006, each benchmark application is run single-threaded for 20B instructions. For PARSEC, each benchmark is run with eight threads for 80B instructions, approximately 10B instructions per-thread. Both benchmarks are run with the full-size input data sets, but the benchmark input datasets themselves do not necessarily contain representative data. Many benchmarks use random, or constrained-random, input data, but it is difficult to gage the impact of this. In PARSEC, one benchmark was clearly skewed to favor deduplicated memory. This benchmark, *blackscholes*, uses the same option price data for many "different" option price calculations. Therefore, results for *blacksholes* are discarded because its compaction is unrealistically high: near to 16x. In general, duplicates are not expected in the PARSEC applications, regardless of whether the underlying dataset is random or actually representative data.

Datacenter Workloads

To understand performance in a context where deduplicated memory provides a great advantage, three different representative datacenter workloads are simulated. In particular, PostgreSQL is simulated running DBT-5, an implementation of TPC-E, memcached is simulated serving tweets scraped from the Twitter public API, and Hadoop is simulated creating an inverted index over Wikipedia data from the PUMA MapReduce benchmark [6]. Memcached is configured to use 8 threads and, to simulate the relative popularity of individual Tweets, the load generator (not running in zsim) requests tweets according to a Zipfian distribution. PostgreSQL is configured for 8 CPUs with 16 GB DRAM, the practical limit of zsim. Because PostgreSQL uses

	Compaction		Speedup		Additional BW	
	Line	Page	DDM	DDC	DDM	DDC
SPEC CPU 2006	2.62x	1.29x	1.01x	1.09x	2.08x	0.93x
PARSEC	1.28x	1.04x	0.96x	0.98x	1.99x	1.38x
Memcached	1.50x	1.01x	0.87x	1.05x	1.74x	0.71x
PostgreSQL	1.40x	1.01x	0.99x	1.04x	1.86x	0.76x
Hadoop	2.10x	1.04x	0.98x	1.05x	3.29x	1.36x

Table 4.3: Overall Results: DDM and DDC-ICA

sys-v shared memory, *zsim* is further modified to map all shared memory virtual addresses to the same physical addresses. Finally, Hadoop is similarly configured to take advantage of a machine with 16 GB DRAM and also all 8 CPUs.

The simulation results report application speedup, but with the same amount of physical memory revealed to the application. On the other hand, both PostgreSQL and memcached are sensitive to memory capacity. For such applications, adding memory tends to increase application performance as measured by queries-per-second, or similar metrics. In particular, for PostgreSQL, the operating system disk cache is actively exploited by the program, but it is not visible in user space and therefore not visible to zsim. Because it is difficult to measure, the additional application level benefit of adding memory is not shown in the results.

4.2 Performance

Table 4.3 shows the overall simulated performance results. Unless otherwise specified, the deduplicated cache is merged (Section 3.2.2) with in-cache allocation (Section 3.2.4). In general, the results indicate no penalty to application performance and that bandwidth is reduced. In the worst case, there is little penalty to performance and marginally increased bandwidth. For example, taking the geometric mean across the three datacenter workloads, the DDC machine achieves a speedup of 1.05x and uses only 0.90x of the bandwidth.

Access	Data	Compaction	Speedup		Additional BW	
			DDM	DDC	DDM	DDC
Sequential	Duplicated	15.9x	0.94x	4.08x	1.00x	0.51x
Random	Duplicated	15.9x	0.50x	0.96x	1.00x	1.00x
Sequential	Random	0.9x	0.94x	0.94x	1.00x	1.00x
Random	Random	0.9x	0.50x	0.50x	1.00x	1.00x

Table 4.4: Synthetic Benchmark Results: DDM and DDC-ICA

The following subsections describe these results in more detail. Using various corners, such as random access to random data, the synthetic benchmark demonstrates that the simulation provides reasonable performance estimates. All five simulated machines are compared on the three categories of workload: general purpose (SPEC), worst-case (PARSEC), and datacenter. Finally, the effect of various DDC optimizations is investigated.

4.2.1 Synthetic

The synthetic benchmark results (Table 4.4) show that the simulation produces reasonable performance estimates and correctly models limited memory system bandwidth. The synthetic benchmark achieves maximum sustainable bandwidth on the simulated machine. Thus, no result shows any bandwidth increase, rather, if more memory accesses are required, performance degrades.

The deduplicated machines (DDM and DDC) perform at near parity for streaming sequential data access patterns, even though no prefetch is used (Chapter 3, Section 3.2.6). Fundamentally, this is because the memory access overhead is low: only 1 out of every 16 cache accesses requires translation refill. At a more nuanced level, this shows that the DTB (Chapter 3, Section 3.1) is effective for both the DDM and DDC machine. Without the DTB, both the DDM and DDC show further slowdown in the {sequential, random} corner. In the {sequential, duplicated} corner, the DDC machine provides a 4.08x speedup, limited only by the rate of instruction fetch.

	DDM	DDC-P	DDC-M	DDC-ICA
SPEC CPU 2006	1.01x	1.07x	1.10x	1.09x
PARSEC	0.96x	0.97x	0.98x	0.98x
Memcached	0.87x	0.98x	0.98x	1.05x
PostgreSQL	0.99x	1.03x	1.03x	1.04x
Hadoop	0.98x	1.02x	1.00x	1.05x

Table 4.5: Average speedup for all workloads and machines

	DDM	DDC-P	DDC-M	DDC-ICA
SPEC CPU 2006	2.08x	1.39x	1.27x	0.93x
PARSEC	1.99x	2.04x	1.93x	1.38x
Memcached	1.74x	0.91x	0.91x	0.71x
PostgreSQL	1.86x	3.43x	3.43x	0.76x
Hadoop	3.29x	6.14x	6.00x	1.36x

Table 4.6: Average bandwidth for all workloads and machines

4.2.2 Standard CPU Benchmarks

Compaction, performance, and bandwidth results for SPEC CPU 2006 and PARSEC are shown in Figures 4.1, 4.2, and 4.3 and in Tables 4.5 and 4.6. For datasets that were compacted to fit in-cache, the DDC machine achieves orders of magnitude bandwidth reduction – conversely, for applications with very small uncompacted working sets, the DDC requires a disproportionate bandwidth increase. Therefore, to make a fair comparison, bandwidth results include only those benchmarks whose working set exceeds the cache capacity either uncompacted (traditional machine and DDM machine), or deduplicated (DDC machines).

The SPEC CPU 2006 results indicate that machines with deduplicated memory perform well for general purpose workloads. For DDC-ICA, the speedup is 1.09x and bandwidth is reduced by a factor of 0.93x. As expected, the compaction in PARSEC is low: on average, only 1.28x (excluding *blackscholes* due to its replicated dataset). Even with low compaction, performance is near parity for PARSEC (Figure 4.2). Fundamentally, this is because translation line caching is effective at minimizing additional performance critical DRAM reads.

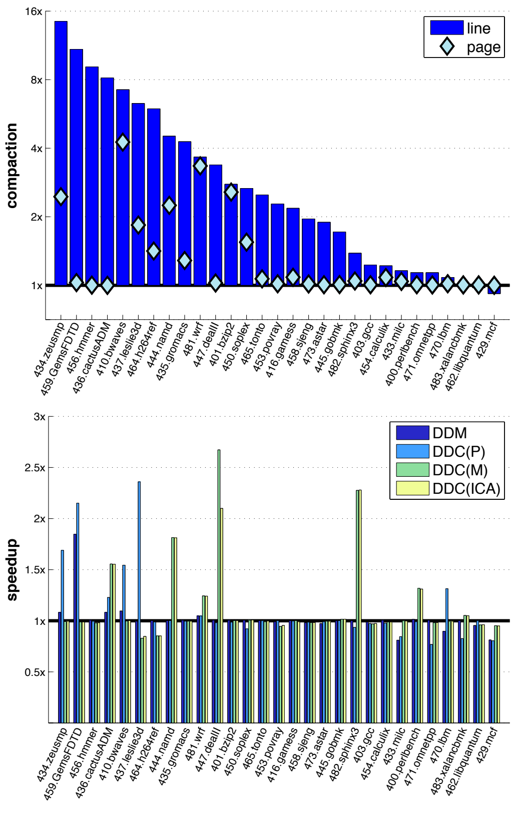


Figure 4.1: SPEC CPU 2006: compaction at line and page granularity and speedup for machines with deduplicated memory

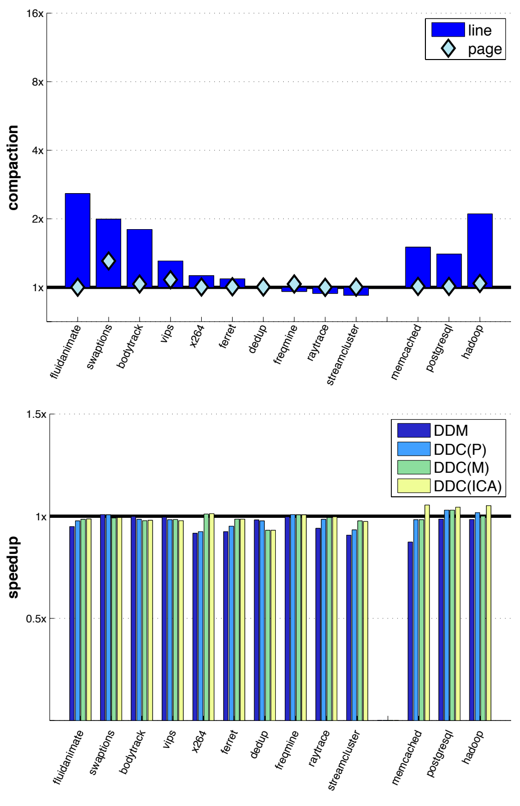


Figure 4.2: PARSEC and Datacenter Applications: compaction at line and page granularity and speedup for machines with deduplicated memory

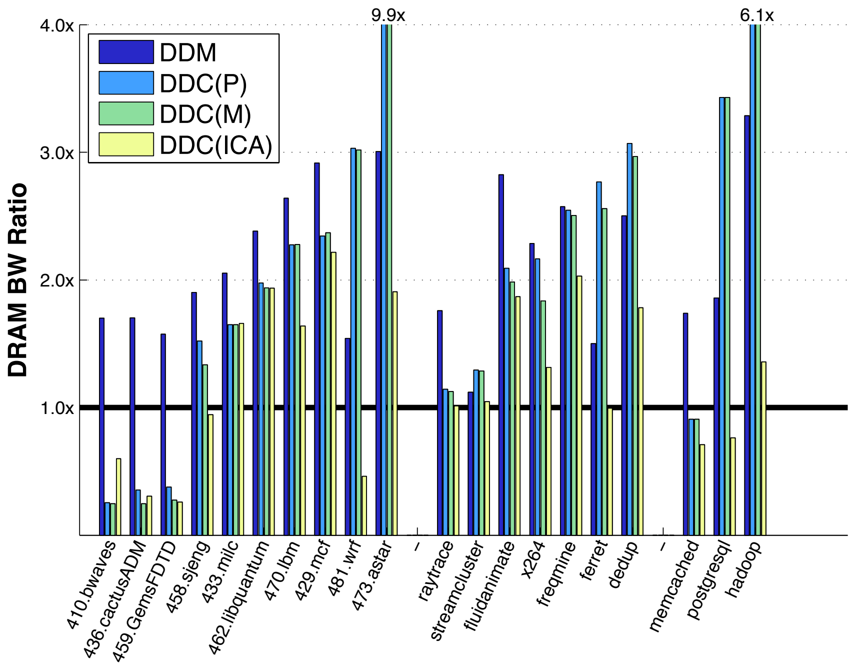


Figure 4.3: Bandwidth Results: benchmarks with working set size exceeding both LLC and DDC capacity

4.2.3 Datacenter Workloads

On the targeted datacenter workloads, the DDC-ICA machine provides an average speedup of 1.05x and reduces bandwidth by a factor of 0.90x. The average compaction, 1.64x, is slightly lower than indicated by counting duplicates in real workloads (Chapter 2, Table 2.4). This is because zsim only captures duplicates in the simulated process, and cannot find duplicates across the entire system: in specific, it does not include those duplicates that arise from operating system IO buffers and disk cache. Furthermore, although care was taken to mimic representative data for memcached (Tweets) and for Hadoop (Wikipedia), the underlying data for TPC-E is effectively random.

4.2.4 DDC Performance

The main purpose of the DDC, as opposed to DDM with non-deduplicated LLC, is to reduce the number of memory accesses required for deduplication. Because it is deduplicated, the DDC also provides increased cache capacity. To evaluate how well the DDC meets these goals, three different DDC configurations are simulated:

1. DDC-P: partitioned, 2 ways for translation, 14 ways for data
2. DDC-M: merged, translation and data compete for space
3. DDC-ICA: merged, with in-cache allocation

Looking only at the results for SPEC CPU 2006 (Tables 4.5 and 4.6), the DDC-P appears effective in both regards: it reduces bandwidth for deduplicated memory (from 2.08x to 1.39x) and improves performance (from 1.01x to 1.07x). Relative to DDM, DDC-P improves performance on all simulated workloads. On the other hand, rather than reduce memory operations for all workloads, for some workloads – PostgreSQL and Hadoop, in particular – relative to DDM, DDC-P actually increases bandwidth. The following paragraphs further elaborate on the effect of DDC optimizations.

Merged vs. Partitioned

In the DDC-M, the ratio between translation and data lines depends mainly on their respective miss rates: if there are more translation line misses – for example, a program accessing random, rather than sequential, addresses – then more translation refills are required and the percentage of cache lines holding translations increases.

As shown in Tables 4.5 and 4.6, for SPEC CPU 2006, DDC-M provides improved performance and reduced bandwidth relative to DDC-P. For other benchmarks, the improvement is less significant. For Hadoop, despite slightly lower performance, it still reduces bandwidth. Therefore, for deduplicated caches, the distinction between merged and partitioned is not particularly significant.

Eager vs. Lazy Deduplication

Lazy deduplication significantly reduces DRAM bandwidth relative to eager because it avoids unnecessary content lookups in DRAM – specifically, it avoids repeated lookups for content generated by writes to the same bus address, an effect referred to as *deduplication induced write-through*. With the DDC-P and DDC-M machines, the DDC must issue a content lookup to the DDM if a write-back from L2 cannot be served by in-cache content lookup. For many applications this is not a problem: frequently updated memory locations normally do not require content lookup for every write because the modified cache line remains in the L1 or L2 cache. However, some multi-threaded programs make frequent updates to values shared between cores – with eager deduplication, this increases bandwidth with no corresponding deduplication benefit. In-cache allocation (Section 3.2.4), simulated by the DDC-ICA machine, defers deduplication until eviction from DDC and write-back to DRAM.

The DDC-ICA machine effectively eliminates deduplication induced write-through. Again, looking at Hadoop, the DDC-P machine increases bandwidth to almost twice that of the DDM machine, or to 6.14x relative to the baseline machine. Although relatively high, in absolute numbers, this is only 20% of peak theoretical bandwidth (8.6 GB/sec out of 42.6 GB/sec). For Hadoop, in-cache allocation reduces bandwidth by 4.4x relative to eager deduplication, and requires only 1.36x bandwidth relative to the baseline machine. Therefore, in-cache allocation, or lazy deduplication, is a significant improvement over eager deduplication.

Zombie Suppression

As described in Chapter 3, Section 3.2.5, zombie lines cannot be referenced by a program, and therefore consume resources that could otherwise be put to use. As shown in Table 4.7, by using a zombie suppression heuristic, the DDC-ICA machine

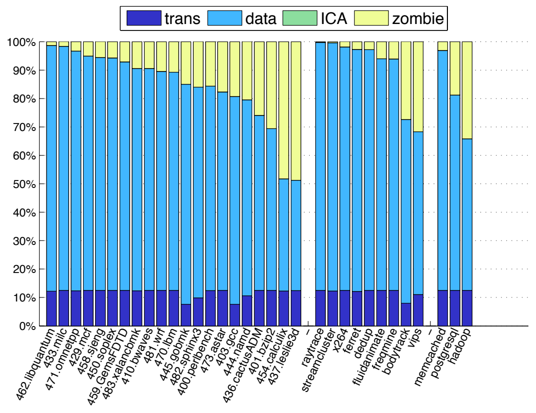


Figure 4.4: DDC-P: Line Statistics

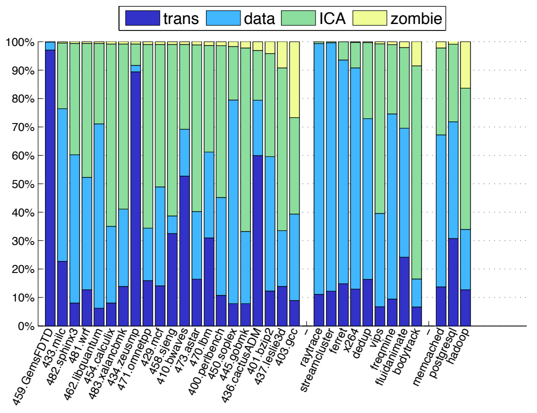


Figure 4.5: DDC-ICA: Line Statistics

	DDC-P	DDC-ICA
Translation	12.5%	15.9%
In-Cache Allocated	0.0%	20.7%
Zombies	7.9%	0.9%

Table 4.7: Line Statistics Comparison

significantly reduces zombies: from 7.9% to 0.9%. Figures 4.4 and 4.5 show the relative distribution of line types in the DDC-P and DDC-ICA machines, respectively.

4.3 Power Savings

Deduplicated memory, with, or without deduplicated cache, reduces memory system power. The main cost of deduplicated memory is the additional DRAM memory operations required for content lookup and for updating hash table metadata. On the other hand, deduplicated memory cuts the most significant power cost in modern

Parameter	Units	Baseline	DDM	DDC-ICA
Utilized Bandwidth	GB/sec	3.00	6.60	2.71
Number of DIMMs	-	8	4	4
BW/DIMM	GB/sec	0.38	1.65	0.68
Power/DIMM	Watts	2.65	3.16	2.77
Memory Power	Watts	21.2	12.6	11.1

Table 4.8: Power Savings from Deduplicated Memory

DRAM systems, namely, the fixed cost of powering up a DIMM. Memory power is nearly linear with bandwidth, therefore, the following, extracted from the Micron DDR3 power calculation data-sheet [2], provides a good estimate of memory power:

$$P_{DIMM} = 2.5 + 0.4 \cdot BW \tag{4.4}$$

Table 4.8 compares memory system power for the baseline system against two systems with deduplicated memory: DDM and DDC-ICA. The bandwidth cost for DDM and DDC-ICA is the geometric mean across datacenter applications (Table 4.6), and the calculation assumes that deduplicated memory provides a factor of 2x increased memory capacity (Table 2.4). Although the power per DIMM increases, the total memory system power decreases because static power is reduced by fewer DIMMs. The power savings are the same if one uses deduplicated memory to provide extra capacity with the same number of DIMMs, and the savings are again the same when translated to energy per bit.

Even with a significantly lower compaction factor, power is saved. For the DDM machine (2.20x bandwidth) power is saved with compaction factors as low as 1.2x and, for the DDC machine, power is saved down to a compaction of 1.1x.[2] With a capacity expansion of 2x, deduplicated memory saves 40%, or more, memory system power.

Further, assuming that extra bandwidth is required, it is already available: datacenter memory bandwidth is highly over-provisioned. Kozyrakis, et al. [44], show that Microsoft Bing uses only 2% of provisioned bandwidth, and that Microsoft Cosmos, a distributed task execution engine (similar to Hadoop), uses only 1% of provisioned bandwidth. Ferdman, et al. [31], find 5% average bandwidth utilization across a suite of representative datacenter applications – the authors specifically state "Off-chip bandwidth exceeds needs by an order of magnitude." Table 4.8 uses 3 GB/sec as

[2] Calculation assumes amortized cost across many machines.

the baseline because results in this work indicate 6.3% percent average bandwidth utilization across simulated datacenter applications.

4.4 Summary

Using a state-of-the-art, cycle-accurate, x86 CPU simulator, this chapter shows that, in general, systems with deduplicated memory attain application performance at parity with their non-deduplicated counterparts, and that in the worst case, the performance penalty is negligible. In fact, for all workloads except for PARSEC, performance improves slightly – 1.06x average speedup – because of effectively increased cache capacity. For PARSEC, which is not expected to have duplicate values, the performance impact is negligible: 0.98x.

This chapter shows that deduplicated cache effectively reduces the number of memory accesses required for deduplication: DDC-P reduces bandwidth from 2.08x to 1.39x relative to DDM. For some multi-core workloads, frequently updated values are live in the last-level cache and cause deduplication induced write-through. In-cache allocation eliminates this, and in some cases, reduces bandwidth below that required by a non-deduplicated system. Further, DDC-ICA immediately evicts potential zombies, thereby reducing zombies from 7.9% to 0.9%. Although technically more efficient, DDC-M provides only marginal improvement over DDC-P.

Finally, because deduplicated memory requires fewer devices, it saves power. The savings depend both on deduplication and bandwidth, but, because each DIMM incurs a high fixed cost, even a small amount of deduplication saves power.

Chapter 5

Sparse Matrix-Vector Multiply

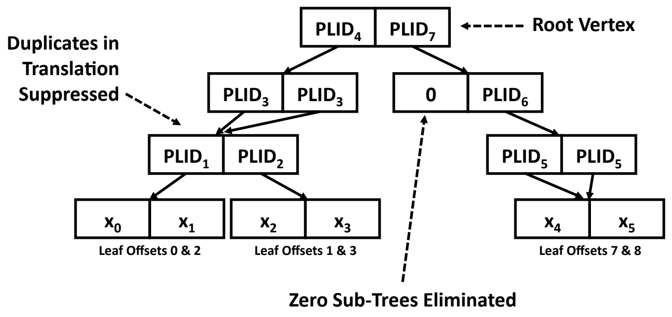
This chapter shows how to use deduplicated memory to improve program performance in the domain of scientific and high performance computing. Applications in these domains often model large physical systems using sets of related equations known as linear systems. For practical models, most variables only strongly influence what can be thought of as their local neighbors. Because of this, many of the equations in large-scale linear systems each contain only a few variables out of thousands, or tens of thousands. In computer systems, such models are represented as a matrix that has one row per equation and one column per variable. Because most variables are logically omitted from each equation (matrix row), most matrix entries are zero valued. Such a matrix, that is mostly filled with zeros, is known as a *sparse matrix*. Machines with deduplicated memory offer new and potentially much more efficient methods for handling sparse matrices, as described in the following.

Hierarchical deduplication, as opposed to deduplicated memory as described in Chapter 2, provides the ability to efficiently store and manipulate a sparse matrix as if it were a dense matrix – a significant advantage that eliminates the need to handle sparse and dense as two separate cases – and, it also significantly reduces the storage, and therefore bandwidth, required when accessing certain large sparse matrices. Hierarchical deduplication recursively deduplicates each translation, until all memory blocks are made unique, and the original content is represented by the connections between a set of unique and deduplicated blocks.

Sparse matrix-vector multiply, or SpMV, is a performance critical computational kernel in many high performance computing and scientific applications, but its performance is limited by main memory bandwidth. Because a hierarchically compacted sparse matrix requires less memory, it can be read from memory with fewer memory accesses, and therefore SpMV completes in less time. Using a hierarchically deduplicated SpMV algorithm implemented in software, this work demonstrates SpMV speedup on current off-the-shelf hardware. The algorithm requires extra CPU opcodes to interpret the hierarchically deduplicated data structure which suggests that hardware support for hierarchical deduplication would provide further speedup. Prior work on the HICAMP architecture [24] introduced such hardware, known as the *iterator register*. Using the iterator register, this work shows that hierarchical deduplication provides sparsity and cache oblivious linear algebra computations.

5.1 Hierarchical Deduplication

In hierarchical deduplication, the translation is deduplicated – recursively – until each memory line, whether translation or data, is unique. Figure 5.1 illustrates hierarchical deduplication using memory lines that contain either two PLIDs, or two data values, each. In Figure 5.1, notice that vector v contains the sequence x_0, x_1, x_2, x_3 twice. In a non-hierarchical deduplicated memory, as described in Chapter 2, the memory line referring to this sequence, namely that line containing $PLID_1$ and $PLID_2$ would appear twice in the translation. Here, because of hierarchical deduplication, that line is also made unique: rather than being *stored twice*, it is *referred to twice*, by the line that contains $PLID_3$, and $PLID_3$.



$$v = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_0 & x_1 & x_2 & x_3 & 0 & 0 & 0 & 0 & x_4 & x_5 & x_4 & x_5 \end{bmatrix}$$

Figure 5.1: Hierarchically Deduplicated Vector: vector v, recursively deduplicated as a DAG

In hierarchical deduplication, the arrangement of memory content, as originally stored by linear index, is memorized by the connections between a set of unique memory lines – or, using some additional terminology, each memory line is a unique *vertex* in a *directed acyclic graph* or *DAG*. The program data is stored in the leaf vertices, or leaves, of the graph. If the data were completely unique and non-zero, the graph would become a fully balanced binary tree (assuming a 2:1 line to PLID size ratio).

Because it is recursive, hierarchical deduplication can provide compounded benefit. For non-hierarchical deduplication, the maximum compaction is fixed by the number of PLIDs that fit into one memory line; in hierarchical deduplication, this quantity is referred to as the *branching factor*, denoted by F. With $F = 2$, the maximum non-hierarchical deduplication is 2x. But, using hierarchical deduplication, this factor of 2x is compounded: in the best case, each level in the DAG multiplies the benefit such that the compaction is 2^N for a DAG with N levels from root to leaf vertex. The maximum hierarchical deduplication possible is given by the following:

$$N = \log_F(\,\textit{number of leaf vertices}\,) \tag{5.1}$$

$$\textit{maximum hierarchical compaction} = F^N \tag{5.2}$$

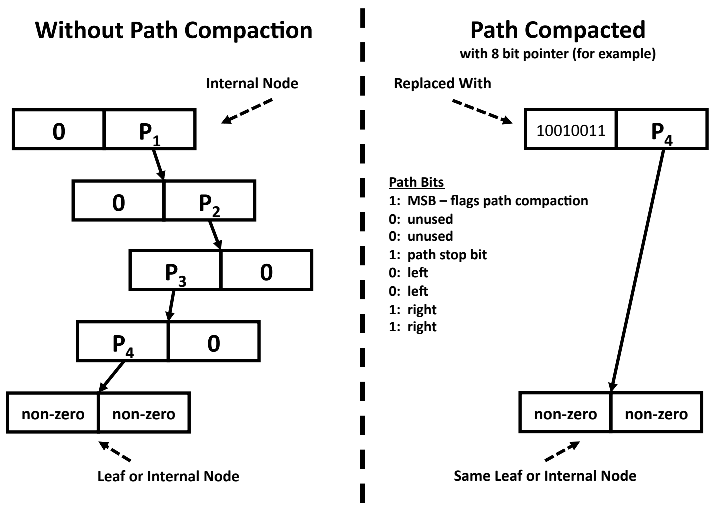


Figure 5.2: Path Compaction

Hierarchical deduplication can also use a technique known as *path compaction* to reduce the number of DAG vertices when storing sparse data. Although the DAG efficiently represents sparse data by referring to large zero valued subtrees with a zero PLID stored near the root vertex (Figure 5.1), it still takes several internal vertices to describe the *path* to the leaf. As shown in Figure 5.2, rather than store all the internal vertices leading to a filled in portion of sparse data, a path compacted vertex points directly to the filled in portion and encodes the path in the otherwise unused portion of the memory line.

In this chapter, several smaller, or finer granularity, memory line sizes are investigated. The deduplicated memory described in Chapter 2 used 64 byte memory lines and 4 byte PLIDs and therefore had a branching factor of 16. This chapter also uses 4 byte PLIDs, but reduces memory line size to 8, 16, or 32 bytes – thus, the branching factor is 2, 4, or 8, respectively. As a convenience, the following informal notation is adopted: a DAG with a branching factor of 2 is referred to as a binary-tree (*b-tree*), a DAG with a branching factor of 4 is referred to as a quad-tree (*q-tree*), and 8 is referred to as an oct-tree (*o-tree*).

5.2 Deduplicated Sparse Matrix Storage

Because hierarchical deduplication removes zero valued subtrees, it provides inherently efficient sparse matrix storage, as compared to non-deduplicated systems which must use specialized sparse matrix storage formats. Therefore hierarchical deduplication provides *sparsity oblivious* storage. Alternately, hierarchical deduplication can be directly applied to the specialized data structures used by non-deduplicated systems.

In the first strategy, referred to as *logically dense*, the row and column indices for each matrix value are implicit in that value's location in the data structure. In the second strategy, referred to as *compressed*, the row and column indices are explicitly stored with the non-zero values and zero, the most common value, is assumed for every entry not explicitly stored. Finally, a third strategy, referred to as *hybrid*, uses a logically dense matrix to store some portion of the matrix data and a compressed list for the remainder. This section describes hierarchically deduplicated matrix formats using these three techniques.

The canonical non-deduplicated *compressed sparse row* or *CSR* format is used as the baseline for comparison. While there are many efficient storage formats for non-deduplicated systems, CSR is simple and highly efficient: it is, essentially, the default sparse matrix format. In CSR, each matrix non-zero value is stored, packed together, in a list. A separate list stores the column index for each respective non-zero value. And, a third list indicates where in each of the previous two lists each new row begins. For an m by n matrix with nnz non-zero values, the first two lists have size nnz and the third list has size m. Assuming double precision floating point values, and 4 byte matrix indices, CSR storage size is given by the following:

$$bytes_{CSR} = 12 \cdot nnz + 4 \cdot m + 4 \tag{5.3}$$

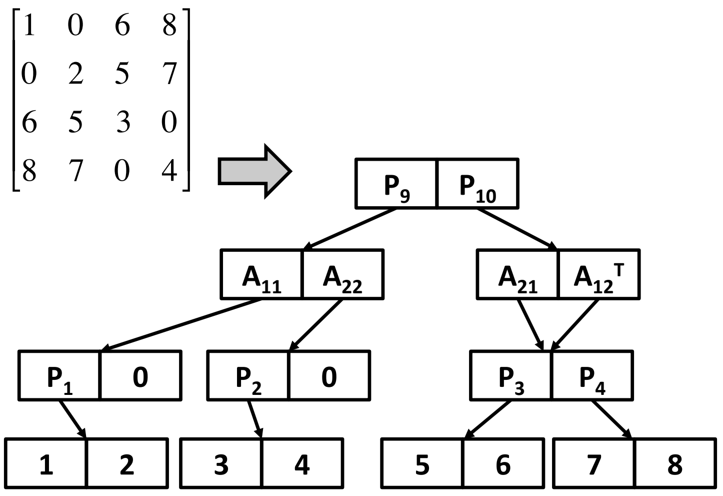
Because nnz is typically larger than either matrix dimension, m or n, the cost of storing the list of non-zero values and their column indices, $12 \cdot nnz$, dominates the storage cost for CSR. Therefore, CSR storage is considered order nnz, or $O(nnz)$.

5.2.1 Row Major Array

The simplest matrix format, in general, is row major – although it is not used for sparse matrices on non-deduplicated systems, it is efficient for sparse matrices when hierarchically deduplicated. Given an m by n matrix A, every value in A is stored in a list. The row index is given by the list index, divided by the matrix column dimension, n, and the column index is given by the list index, modulo the matrix column dimension. In other words, each matrix row is stored together in memory. When hierarchically deduplicated, this is referred to as *row major array* or *RMA*. Although simple, this method achieves storage efficiency at parity with the canonical CSR format (see Table 5.1).

5.2.2 Quad-Tree Symmetric

A second logically dense format, *quad-tree symmetric* (*QTS*) hierarchically exploits *any amount* of matrix symmetry. A given matrix, A, is symmetric if $A(r, c) = A(c, r)$ where r and c, respectively, denote a specific row and column in the matrix. Therefore, there are many duplicates in symmetric matrices, but importantly, there are also entirely replicated sub-matrices. Non-deduplicated systems typically store only the matrix upper (or lower) half because the remaining entries can always be inferred by *transposing* the row and column indices in the half that is explicitly stored. This strategy is efficient for symmetric matrices, but some non-symmetric matrices still have regions with symmetry: the QTS format inherently deduplicates symmetric regions, oblivious to how much, or how little, symmetry exists.



$\begin{bmatrix} 1 & 0 & 6 & 8 \\ 0 & 2 & 5 & 7 \\ 6 & 5 & 3 & 0 \\ 8 & 7 & 0 & 4 \end{bmatrix}$
P_9
P_{10}
A_{11}
A_{22}
A_{21}
A_{12}^T
P_1
0
P_2
0
P_3
P_4
1
2
3
4
5
6
7
8

Figure 5.3: Quad-Tree Symmetric Matrix

To exploit symmetry, quad-tree symmetric divides the matrix into four sub-matrices (A_{11}, A_{12}, A_{21}, and A_{22}) and stores each in a separate sub-tree in the DAG. This pattern is repeated, recursively, until the sub-matrix has dimension 1 *by* 1, that is, a single non-zero value. When a given sub-matrix is on the main diagonal, the storage order is A_{11}, A_{22}, A_{21}, then A_{12}^T. If the matrix is symmetric, then $A_{21} = A_{12}^T$. Because these two matrices are in explicitly separate sub-graphs in the DAG, their root vertex is identical and *all* of their values *and* internal vertices are stored exactly once in the hierarchically deduplicated memory. In this way, the QTS format automatically exploits the duplicate content found in symmetric matrices. Even if the entire matrix is not symmetric, but only a portion, the QTS format completely deduplicates that region. Figure 5.3 shows an example, using a small symmetric sparse matrix mapped to a QTS encoded DAG.

5.2.3 Hierarchical Compressed Sparse Row

The *hierarchical compressed sparse row* or *HCSR* format uses the same underlying data structures as non-deduplicated CSR, but stores them in hierarchically deduplicated DAGs. To enhance deduplication, the column indices and row breaks are delta encoded. Delta encoding exposes more duplicate values where regular patterns exist. For example, the sequence $10, 11, 12, 13, 15$ becomes $10, 1, 1, 1, 2$ when encoded as consecutive differences, or delta encoded; note that an additional value, the seed value, is stored first. The same sequence can be delta encoded as $10, 0, 0, 0, 1$ if one assumes that each value is incremented by one. Because column indices are often clustered in sequential order, in HCSR, they are delta encoded, with an assumed increment of one, to expose extra zeros. The row pointers are similarly delta encoded.

5.2.4 Hierarchical Coordinate Format

The *hierarchical coordinate* format, or *HCOO*, hierarchically deduplicates a common, but less efficient, non-deduplicated sparse matrix format known as coordinate, or COO. On non-deduplicated systems, the CSR format requires less storage than COO. On the other hand, COO simplifies the insertion of new non-zero values by allowing non-zero elements to be stored in any order, rather than grouped as rows. In the coordinate format, each matrix element is stored along with explicit row *and* column indices. This can be implemented by storing all elements in a single list of 3-tuples (row index, column index, non-zero value), or as three separate lists: row indices, column indices, and non-zero values. To expose more duplicate values, HCOO uses delta encoding on the list of column indices (with an assumed increment of one) and on the list of row indices. The delta-encoded row and column indices are interleaved into a list of 2-tuples (row index, column index). Because the row and column indices are 4 bytes each, there are logically as many bytes stored in the list of 2-tuples as in

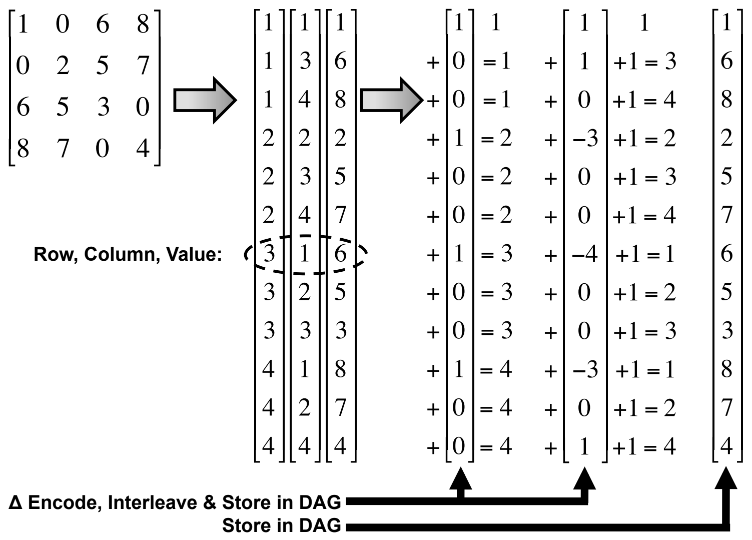


Figure 5.4: Hierarchical Coordinate Encoding

the list of 8 byte double precision (non-zero) values. The two lists, row and column indices *and* non-zero values, are hierarchically deduplicated, which results in two DAGs. Because each list is logically the same length, the two DAGs have the same height – by itself, of no importance, but a fact which is later exploited in the software implementation of sparse matrix-vector multiply. Figure 5.4 shows the delta encoding and interleaving for the HCOO format.

5.2.5 Non-Zeros Dense Format

The hybrid format *non-zeros dense*, or *NZD*, stores a compressed list of non-zero values and uses a pattern matrix stored in QTS format to indicate their row and column indices. The pattern matrix leaf elements are logically only one bit in size, therefore, 64 pattern matrix entries can fit in the same space as one double precision floating point value. Figure 5.5 shows an example of the NZD format.

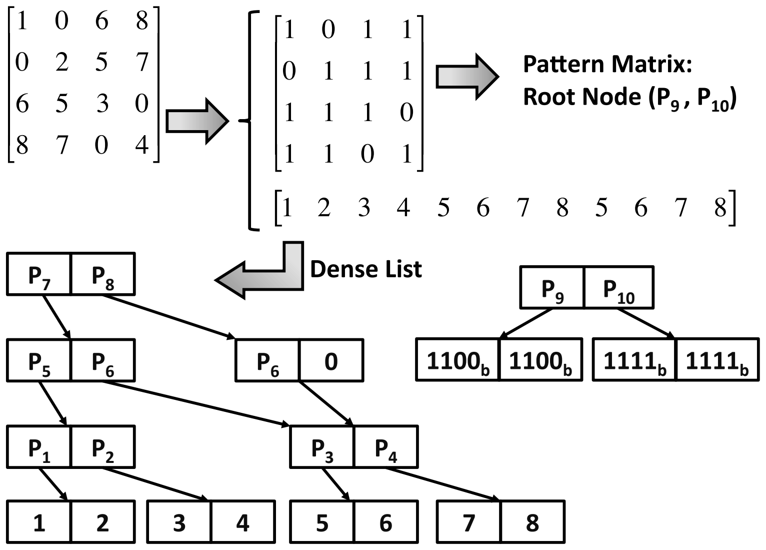


Figure 5.5: Non-Zeros Dense Format

5.3 Sparse Matrix Compaction

Matrix compaction is the ratio of the storage required by the non-deduplicated CSR format (Equation 5.3) to the storage required by deduplicated DAG:

$$\textit{matrix compaction} = \frac{bytes_{CSR}}{bytes_{DAG}} \tag{5.4}$$

Three different memory line sizes are used for the DAG compacted matrices: 8, 16, and 32 bytes. The DAG storage requirement is the number of unique memory lines, or DAG vertices, multiplied by bytes per vertex, and the number of bytes per vertex is set by the memory line size:

$$bytes_{DAG} = \frac{bytes}{vertex} \cdot N_{unique\text{-}vertices} \tag{5.5}$$

$$\frac{bytes}{vertex} = \begin{cases} 8 & b\text{-}tree \\ 16 & q\text{-}tree \\ 32 & o\text{-}tree \end{cases} \tag{5.6}$$

Format	Degree	Geomean	Best	Worst	Bound
RMA	b-tree	1.1x	18x	0.45x	0.06x
	q-tree	0.8x	11x	0.28x	0.06x
	o-tree	0.6x	8x	0.16x	0.05x
QTS	b-tree	2.5x	5719x	0.46x	0.06x
	q-tree	2.3x	5244x	0.29x	0.06x
	o-tree	1.8x	4028x	0.18x	0.05x
NZD	b-tree	2.5x	1182x	0.60x	0.06x
	q-tree	2.8x	1160x	0.79x	0.06x
	o-tree	2.6x	918x	0.79x	0.05x
HCSR	b-tree	1.9x	1289x	0.54x	0.50x
	q-tree	2.2x	1289x	0.76x	0.75x
	o-tree	2.2x	979x	0.88x	0.88x
HCOO	b-tree	1.8x	1513x	0.54x	0.38x
	q-tree	2.2x	3185x	0.65x	0.56x
	o-tree	2.0x	1141x	0.67x	0.66x

Table 5.1: Comparison of Deduplicated Matrix Formats

5.3.1 Compaction Results

Table 5.1 shows compaction for deduplicated sparse matrix formats over a set of 74 non-symmetric matrices from the University of Florida Sparse Matrix Repository [26]. Figure 5.6 shows best case compaction (Equation 5.4) for each of the 74 non-symmetric matrices versus non-deduplicated size (Equation 5.3). As shown in Figure 5.6, QTS and NZD tend to provide the best matrix compaction. This is because QTS provides compounded benefit when entirely replicated sub-matrices exist, and NZD provides compounded benefit when the *pattern* of non-zero values is replicated.

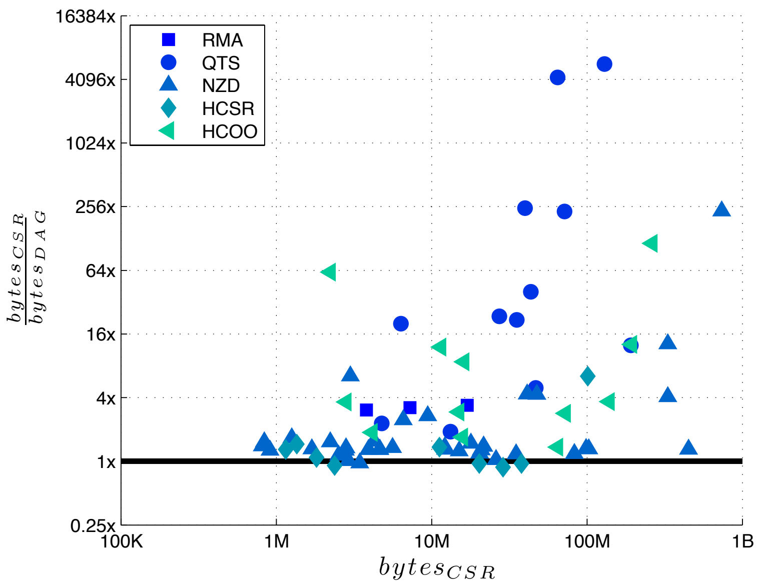


Figure 5.6: Sparse Matrix Compaction Results

5.3.2 Storage Bounds

As shown in Section 5.3, hierarchical deduplication provides very high compaction in the best case – this section shows that it also provides reasonable bounds in the worst case. Equation 5.7 shows the total storage requirement for a DAG with completely unique leaf vertices as calculated by geometric series. For a matrix with unique non-zero values, and pessimistically assuming that no duplicates are found in the row or column indices, Equation 5.7 precisely computes storage for the compressed formats HCOO and HCSR.

$$N_{total\text{-}vertices} = \begin{cases} 2 \cdot N_{leaf\text{-}vertices} & b\text{-}tree \\ 4/3 \cdot N_{leaf\text{-}vertices} & q\text{-}tree \\ 8/7 \cdot N_{leaf\text{-}vertices} & o\text{-}tree \end{cases} \tag{5.7}$$

For HCOO and HCSR, the number of leaf vertices is $O(nnz)$ and thus the storage is bounded by $O(nnz)$. For the logically dense formats, the number of leaf vertices is nnz, in the worst case, and each of these costs up to $\log(m \cdot n)$ parent vertices. Therefore, for the remaining hierarchically deduplicated formats, the storage is bounded by

$O(nnz \cdot \log(m \cdot n))$. To tighten the bound, the calculation for RMA, QTS, and NZD shown in Table 5.1, assumes that each logically dense matrix is filled in dense at the top of each DAG and that each leaf vertex costs $\log(n)$ parent vertices, rather than $\log(m \cdot n)$. Given the empirical results shown in Table 5.1, even this bound is very loose. This looseness results from both path compaction, which reduces the cost of each leaf vertex, and the fact that non-zero values are often clustered, which makes storage locally more efficient.

5.4 Deduplicated Sparse Matrix-Vector Multiply

Sparse matrix-vector multiply (SpMV) is a critical task in the inner loop of modern iterative linear system solvers. In such programs, the linear system is defined by an m by n sparse matrix, A, with nnz non-zero values. The goal is to find a vector, x, that satisfies the equation $y = Ax$. Rather than directly find a matrix inverse, A^{-1} such that $A^{-1}y = x$, it is often much more efficient to iteratively search for the solution. Using this strategy, a sequence of trial solutions is computed, and refined. To test and refine each trial solution, x_i, requires one sparse matrix-vector multiply: Ax_i. The remaining steps to generate x_{i+1}, for the next trial, are relatively inexpensive compared to the SpMV operation. Therefore, any speedup to SpMV also speeds up the entire equation solver.

The cost of SpMV is dominated by memory access latency because the CPU performs very few operations per byte read from memory. Therefore, for non-deduplicated matrices, every value must be read from DRAM memory on every iteration. As an algorithm, SpMV is also easily parallelized, therefore these accesses occur at the maximum sustainable rate, that is, at maximum memory bandwidth.

Sockets	2
Cores	12
Threads	24
LLC	12 MB
f_{clk}	2.93 GHz
Mem Channels	3
f_{mem}	1066 MHZ
Mem BW	25.6 GB/sec

Table 5.2: SpMV Machine Specifications

Because of compaction, deduplicated sparse matrix-vector multiply requires less data transfer and therefore takes less time. Although no matrix entry is reused algorithmically, with deduplication, common values can be found in cache – for deduplicated SpMV, *not* every matrix value need be read from memory. Therefore, in the best case, deduplicated SpMV is limited by CPU speed, rather than main memory bandwidth – and, on current CPUs, this confers a significant speedup. Furthermore, this speedup will grow as the combined effect of Moore's Law and limited IO pin density drives up the compute to memory bandwidth ratio.

5.4.1 SpMV Evaluation Methodology

To evaluate deduplicated SpMV, this section compares a multi-threaded DAG traversing SpMV kernel to a state-of-the-art multi-threaded CSR SpMV kernel. The pseudocode in Figure 5.7 provides a simplistic illustration of DAG traversal software using a branching factor of two. This evaluation reports speedup and other related metrics measured on actual hardware: an off-the-shelf x86-64 Linux machine with specifications shown in Table 5.2.

```
procedure TraverseDAG( vtxs, height ):
  sp = 1                # initialize stack pointer
  ptrStack[0] = 0       # vertex pointer stack
  offStack[0] = 0       # tracks the leaf index
  lvlStack[0] = height  # DAG level stack
  while( sp ):
    sp -= 1
    level  = lvlStack[ sp ]
    offset = offStack[ sp ]
    ptr    = ptrStack[ sp ]
    vtx    = vtxs[ ptr ]
    if level == 0:
      print 'offset =', offset, ', leaf =', vtx
    else:
      if vtx.left & pathbit:
        # decode path & push new offset, level, & vertex pointer
      else:
        if vtx.right:
          ptrStack[ sp ] = vtx.right
          lvlStack[ sp ] = level
          offStack[ sp ] = offset + 1 << level
          sp += 1
        if vtx.left:
          ptrStack[ sp ] = vtx.left
          lvlStack[ sp ] = level
          offStack[ sp ] = offset
          sp += 1
```

Figure 5.7: DAG Traversal Code

The DAG kernel implements SpMV using the HCOO format with quad-tree DAGs. HCOO is used because it provides the best tradeoff in terms of SpMV kernel implementation complexity and matrix compaction. Although QTS and NZD provide better compaction, their code is complicated by the requirement that they track the row and column indices based on the path traversed to the leaf. Because HCOO has two DAGs of the exact same logical size, its code is simplified relative to the implementation for HCSR. Quad-tree, or branching factor of 4, DAGs are used because they provide the best compaction for HCOO. Because of deduplication, the memory access pattern is random, rather than sequential as in the non-deduplicated CSR SpMV kernel. To mitigate the effect of random memory access, the HCOO SpMV kernel works on two regions of the matrix – therefore, 4 DAGs – simultaneously. This helps hide latency because 4 DAG vertices are dereferenced on every pass through the loop.

Execution time for matrices that exceed the LLC capacity is measured, and speedup, defined by Equation 5.8, is reported.

$$speedup = \frac{t_{SpMV\text{-}CSR}}{t_{SpMV\text{-}DAG}} \tag{5.8}$$

Although DAG storage significantly compacts some of the matrices, the two vector operands are not deduplicated, and their storage cost is the same for both methods. For each matrix, the cost of storing the two vectors is an additional $8 \cdot m + 8 \cdot n$ bytes. The working set size, including the cost of vector storage, is given by Equation 5.9.

$$working\ set\ size = matrix\ size + 8 \cdot m + 8 \cdot n \tag{5.9}$$

Therefore, when memory bandwidth is the active limit, the speedup is correlated to working set compaction, as defined in Equation 5.10.

$$working\ set\ compaction = \frac{working\ set\ size_{CSR}}{working\ set\ size_{DAG}} \tag{5.10}$$

In some cases, the working set compaction is so large that the active limit is no longer memory bandwidth, rather it is compute. To evaluate which limit is active, several metrics are measured during SpMV execution and then used to compute empirical performance limits. The bandwidth limit, defined in Equation 5.11, is based on measuring the total amount of DRAM traffic during one iteration of SpMV.

$$bandwidth\ limit = \frac{memory\ traffic_{CSR}}{memory\ traffic_{DAG}} \tag{5.11}$$

While the working set compaction shows the reduction in memory traffic that is *possible*, the bandwidth limit shows the reduction in memory traffic that is achieved. Therefore, the bandwidth limit shows the speedup that is possible, using the same cache system, if instructions could be retired arbitrarily fast. The compute limit, defined in Equation 5.12, is based on the number of CPU instructions required for one iteration of SpMV and the maximum sustainable number of instructions retired per clock cycle, or IPC.

$$\textit{compute limit} = \frac{N_{\textit{instructions-CSR}}/IPC_{CSR}}{N_{\textit{instructions-DAG}}/IPC_{max}} \tag{5.12}$$

To show the speedup that is possible, regardless of the amount of memory traffic required, the compute limit uses the maximum observed IPC to calculate clock cycles required for DAG SpMV, and for other terms, the empirical measurements are used.

5.4.2 SpMV Results

Table 5.3 shows the speedup and working set compaction for each of the 28 matrices that exceed the LLC capacity. The best speedup is 3.7x for matrix *atmosmodl*, and the average speedup, by geometric mean, is 1.5x. As expected, SpMV speedup is correlated to the total working set compaction and bounded by the compute limit, but, the realized speedup is sometimes noticeably less.

To understand this discrepancy, Figure 5.8 plots speedup versus both the memory traffic limit and the compute limit (Equations 5.11 and 5.12). For highly compacted matrices, the active limit is compute, rather than bandwidth. In more detail, for HCOO SpMV, the maximum sustainable IPC was 2.1 and for CSR SpMV, the average IPC is 0.4. Therefore, the speedup *should* be approximately 4x. On the other hand, HCOO SpMV requires an average of 31 CPU instructions per non-zero while CSR SpMV requires only 14. After derating for instruction count, the expected speedup drops from 4x to 2.4x, as indicated by the compute limit plotted in Figure 5.8.[1]

[1] Figure 5.8 uses actual instruction count for each matrix, therefore it varies around 2.4x.

Matrix	Uncompacted		Compaction		Speedup
	Matrix	Working Set	Matrix	Working Set	
barrier2-1	25 MB	27 MB	0.9x	0.9x	0.9x
poisson3Db	28 MB	29 MB	0.6x	0.7x	0.4x
mc2depi	26 MB	34 MB	1.9x	1.6x	1.6x
TSOPF_RS_b300_c2	34 MB	34 MB	13.4x	11.6x	1.9x
thermomech_dK	33 MB	36 MB	0.9x	0.9x	0.7x
sme3Dc	36 MB	37 MB	0.9x	0.9x	0.7x
stat96v3	38 MB	47 MB	12.5x	4.0x	1.7x
xenon2	45 MB	47 MB	4.4x	3.7x	1.5x
webbase-1M	39 MB	55 MB	1.7x	1.4x	0.9x
rajat29	45 MB	55 MB	1.6x	1.4x	1.1x
stormG2_1000	42 MB	56 MB	6.5x	2.7x	1.6x
Chebyshev4	62 MB	63 MB	1.3x	1.3x	1.1x
largebasis	62 MB	68 MB	3.3x	2.7x	2.3x
pre2	69 MB	79 MB	2.8x	2.3x	1.3x
ohne2	79 MB	82 MB	1.0x	1.0x	0.9x
Hamrle3	69 MB	91 MB	85.9x	4.0x	2.2x
PR02R	94 MB	97 MB	1.1x	1.1x	1.1x
torso1	98 MB	100 MB	1.1x	1.1x	1.2x
Rucci1	97 MB	113 MB	5.7x	3.4x	2.8x
tp-6	133 MB	141 MB	2.3x	2.2x	1.5x
atmosmodl	124 MB	147 MB	3185.0x	6.4x	3.7x
TSOPF_RS_b2383	185 MB	186 MB	12.8x	12.3x	1.8x
circuit5M_dc	184 MB	237 MB	1.7x	1.5x	1.2x
rajat31	250 MB	322 MB	115.1x	4.4x	2.2x
cage14	316 MB	339 MB	2.0x	1.9x	1.0x
FullChip	316 MB	362 MB	1.7x	1.5x	0.9x
RM07R	430 MB	436 MB	1.2x	1.2x	1.1x
circuit5M	702 MB	787 MB	3.0x	2.5x	1.8x
geomean	n/a	n/a	3.8x	2.1x	1.5x

Table 5.3: HCOO Compaction and SpMV Speedup

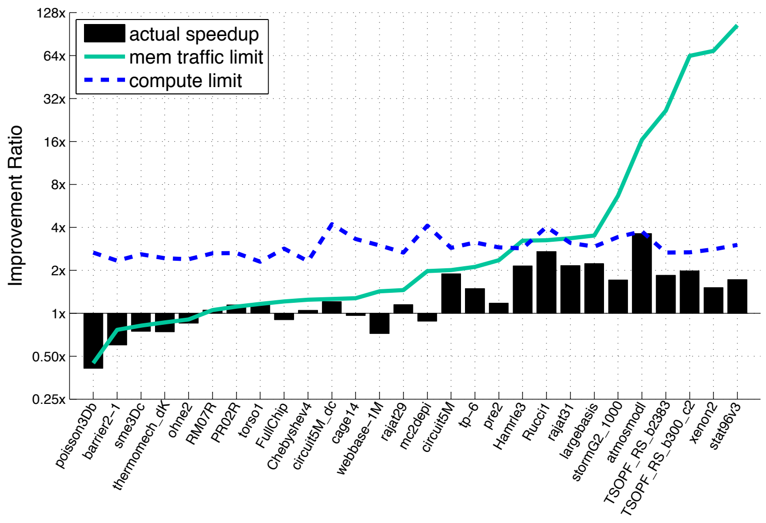


Figure 5.8: SpMV Speedup and Performance Limits

Therefore, HCOO SpMV is compute limited for highly compacted matrices. It is compute limited because less data transfer is required since deduplication provides data reuse where there was none before. This is in direct contrast to current state-of-the-art SpMV implementations which are always bandwidth limited. All current trends indicate that compute resources are increasingly relatively faster than main memory bandwidth [11]. Therefore, HCOO SpMV performance will improve faster than CSR SpMV performance.

5.5 Iterator Register Hardware

In addition to being compute limited because less data is transferred, HCOO SpMV is compute limited because extra CPU opcodes are required to interpret the DAG data structure. This suggests that dedicated hardware for DAG traversal would further improve HCOO SpMV performance. The HICAMP architecture [24] describes such special purpose hardware, known as an *iterator register*. The iterator register

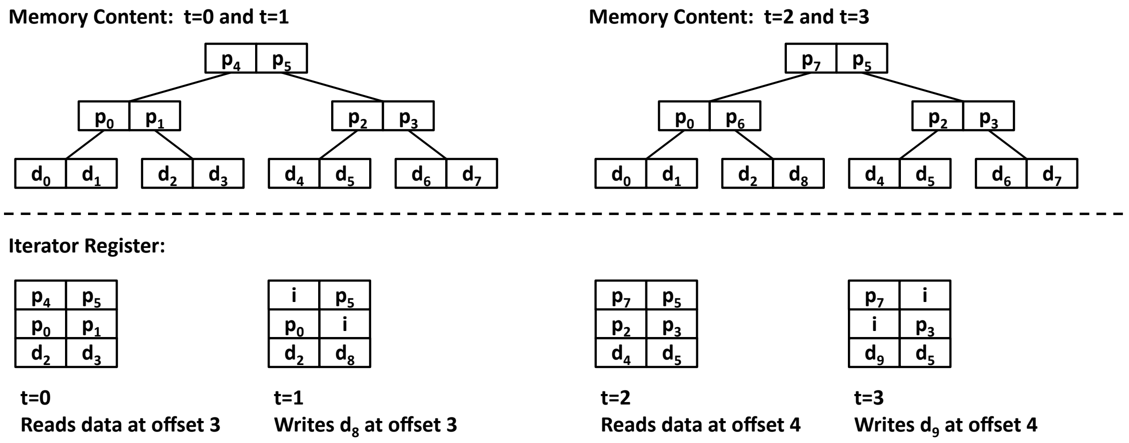

Memory Content: t=0 and t=1
p_4 p_5
p_0 p_1
p_2 p_3
d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7
Memory Content: t=2 and t=3
p_7 p_5
p_0 p_6
p_2 p_3
d_0 d_1 d_2 d_8 d_4 d_5 d_6 d_7
Iterator Register:
p_4 p_5 p_0 p_1 d_2 d_3
t=0
Reads data at offset 3
i p_5 p_0 i d_2 d_8
t=1
Writes d_8 at offset 3
p_7 p_5 p_2 p_3 d_4 d_5
t=2
Reads data at offset 4
p_7 i i p_3 d_9 d_5
t=3
Writes d_9 at offset 4

Figure 5.9: Iterator Register Overwriting Portions of a DAG

provides a programming interface to a hierarchically deduplicated memory. For hierarchical deduplication, a deduplicated memory, as described in Chapter 2, is modified to expose additional operations. In particular, the iterator register can directly issue content lookup operations, directly dereference PLIDs, and issue reference count increments and decrements.

The iterator register provides two fundamental operations: read value by logical offset and write value by logical offset. The fundamental data size can be much smaller than the memory lines size, and the logical offset is relative to the left most entry in the left most leaf vertex. In response to a read, the iterator register automatically traverses from DAG root vertex to the appropriate leaf vertex. It selects the appropriate entry from the leaf vertex, and returns that to the CPU general purpose register.

On write, the iterator register provides the abstraction that the DAG is an array of unbounded size. To write a value, the iterator register issues lookup commands to the deduplicated memory and builds the path from leaf vertex to the root. Therefore, the iterator register supports the creation of new leaf vertices, and automatically grows the DAG if necessary. When overwriting a pre-existing leaf vertex, it modifies the appropriate pre-existing internal vertices.

For efficiency, the iterator register caches the DAG internal vertices that it encounters while traversing from root to leaf. Therefore, subsequent reads near to the current leaf do not require fetch of the entire path from DAG root to leaf: to read the next data item, in the common case, only one memory operation is required because the shared parent vertex is already cached.

Figure 5.9 illustrates both the read and write process using DAGs with $F = 2$ and a fundamental data size that is 1/2 of the memory line size. In Figure 5.9, at t_0, the iterator register is instructed to read the data at offset 3. To do so, it loads the DAG root vertex, and the internal nodes along the path to appropriate leaf vertex. At t_1, the CPU writes a new value, d_8, at offset 3. In response, the iterator register marks the PLIDs along the current path as invalid – content lookup commands are deferred until a vertex is evicted from the iterator register. At t_2, the CPU loads data at offset 4. In response, the iterator register loads one new internal vertex and the appropriate leaf vertex. Lookup commands are issued to generate PLIDs p_6 and p_7, and reference count decrements are issued to each line which is no longer included in the DAG.

Therefore, the iterator register provides an efficient programming interface to a hierarchically deduplicated memory. It minimizes read operations by caching DAG internal vertices, and it also minimizes content lookup operations.

5.6 Iterator Register SpMV

By implementing DAG traversal, the iterator register removes instruction count overhead from DAG based SpMV. Figure 5.10 shows pseudo-code for SpMV using an iterator register and the RMA matrix format described in Section 5.2.1. Compared to the pseudo-code in Figure 5.7, the code shown in Figure 5.10 eliminates the CPU

```
procedure RMA_SpMV( dagID, m, n, y, x ):
  it = IterReg( dagID )
  it.SetSkipZeroSubTrees()
  while( it.Pos() != it.End() ):
    v = it.Val()
    p = it.Pos()
    r = p / n
    c = p % n
    y[r] += v * x[c]
    it.PosInc()
```

Figure 5.10: SpMV Using an Iterator Register

opcodes required for DAG traversal. Because adding more iterator registers is relatively less expensive than providing additional memory bandwidth – by adding memory channels, for example – given enough iterator registers, DAG SpMV can work at the bandwidth limit, rather than the compute limit. For some matrices, the potential speedup is very large. As shown in Figure 5.8, for matrix *stat96v3*, data transfer is reduced by 90x. Although a 90x speedup is unlikely, the iterator register enables speedup beyond the best case of 3.7x demonstrated in this work.

Furthermore, the iterator register enables highly efficient concurrent symmetric SpMV by using the quad-tree symmetric format (QTS), described in Section 5.2.2. On current CPU architectures, concurrent symmetric SpMV is challenging. Symmetric storage promises a factor of 2x advantage in reduced data transfer, but it also inhibits parallelism because it is more difficult to partition the work. For non-symmetric concurrent SpMV, threads are normally assigned distinct matrix rows. This works well because it guarantees that each thread has a disjoint write set. Unfortunately, this partitioning inhibits performance for concurrent symmetric SpMV: because each thread now owns both one row and one column of the matrix, each thread can write to any value in the output vector. Therefore, current approaches to concurrent symmetric SpMV incur extra overhead for matrix permutation and write synchronization [19, 45].

The QTS format logically stores a symmetric matrix as if it were non-symmetric, but also achieves a factor of 2x reduction for storage and data transfer. Using QTS, the same SpMV algorithm can be used for both symmetric and non-symmetric matrices. If the matrix is symmetric, the QTS format inherently reduces data transfer by deduplicating identical transpose sub-matrices. Although QTS SpMV incurs more CPU instruction overhead than HCOO SpMV, QTS SpMV is made efficient by iterator register DAG traversal. Therefore, the QTS format provides efficient symmetry oblivious sparse matrix-vector multiply.

5.7 Sparsity Oblivious Algorithms

This section describes a class of sparsity oblivious algorithms, namely algorithms which are work efficient for both sparse and dense input data. Such algorithms, when implemented on a conventional architecture, incur a large performance penalty because many extra CPU opcodes are required to traverse the DAG based data structures. The iterator register, as described to this point, removes some overhead in these algorithms by automatically fetching DAG leaf nodes for use by the arithmetic execution units. On the other hand, an algorithm written for a dense input would perform arithmetic for every zero valued leaf – even if those leaves are not actually stored in memory, i.e. because of deduplication. Building on the bounds for logically dense sparse matrices given in Section 5.3.2, this section shows that sparsity oblivious matrix-matrix multiply performs exactly the same number of floating point arithmetic operations and makes efficient memory accesses to a data structure whose size is at most $O(nnz \cdot \log(m \cdot n))$ bytes.

To enable sparsity oblivious algorithms, the iterator register programming interface is extended to support two new operations: read PLID by DAG level and linear index and insert PLID by DAG level and linear index. The first of these operations

```
procedure DAGRecursiveSparseVectorAdd( c, a, b ):
    # computes c = a + b
    leaf = a.IsLeaf() and b.IsLeaf() and c.IsLeaf()
    if leaf:
        # operands a & b are both non-zero
        c = a + b
    else:
        for i = 1 to a.NumSubGraphs():
            ai = a.SubGraph( i )
            bi = b.SubGraph( i )
            if ai.NonZero() and bi.NonZero():
                ci = c.SubGraph( i )
                DAGRecursiveSparseVectorAdd( ci, ai, bi ):
            else if ai.NonZero():
                # copy from a
                c.InsertSubGraph( i, ai )
            else if bi.NonZero():
                # copy from b
                c.InsertSubGraph( i, bi )
```

Figure 5.11: Sparsity Oblivious Vector Addition

returns a PLID that refers to a root vertex for a sub-graph within the DAG, and the second stores an entire sub-graph into the DAG. For the logically dense format QTS, this sub-graph is actually a sub-matrix. High performance dense matrix-matrix multiply algorithms use operations over sub-matrices to achieve high cache hit-rates and thus maximize performance. Because QTS allows both operations on sub-matrices and efficient detection of zero-valued sub-matrices, it combines naturally with tree-recursive cache-oblivious algorithms which have been proposed as a middle ground between performance tuning [68] and naïve algorithms [16, 35].

5.7.1 Sparse Vector Add

Using sparse vector addition as a simple example, Figure 5.11 illustrates the key insight that zero-valued sub-graphs are handled efficiently. Three iterator registers are required: two for the operands, and one for the result. Rather than iterate over individual vector elements (leaves), sub-vectors are added together by iterating over internal vertices. If a zero valued sub-graph is encountered, it is much more efficient to copy the non-zero sub-graph to the destination, rather than copy each individual leaf. The copy operation maps to a single iterator register command, PLID insertion, as described above. The PLID insertion command inserts the PLID, which refers to a non-zero sub-graph from the non-zero operand vector, into an internal vertex of the destination vector and it increments the reference count for that sub-graph.

In terms of floating point arithmetic, this algorithm is work efficient – it performs exactly the required number of floating operations and no more. It is sparsity oblivious, but it pays for that by using a modest additional amount of storage given by Equation 5.7. In certain cases, this algorithm requires far fewer instructions because large copies are made efficient by copying the root PLID of a DAG sub-graph. The total number of memory accesses required depends on the amount of deduplication in the source vectors, and the number of content lookups that escape to DRAM as the destination DAG is built. If fewer CPU instructions are required, or if many duplicates are found, then it is likely that fewer memory accesses will be required. This simple example sets the pattern, namely tree-recursive and zero sub-graph detection, used, in the following sections, to construct an efficient sparsity oblivious matrix-matrix multiply.

5.7.2 General Matrix-Matrix Multiply

Dense matrix-matrix multiply, or general matrix-matrix multiply (GEMM), is an important general purpose linear algebra algorithm. For large matrices, direct implementation of a dot product on each operand row and column is inefficient because the CPU caches cannot hold the working set. High performance GEMM implementations operate on matrix sub-blocks (also known as tiles), rather than individual rows and columns. The sub-blocks are sized so that the maximum number of arithmetic operations can be performed over some working set that fits entirely in the CPU L1 cache. On current CPUs, for peak performance, the algorithm must be implemented with several tiers of sub-blocks, each sized appropriately according to the

```
procedure DAGRecursiveMtxMtxMult( C, A, B ):
    # computes C = A*B
    leaf = A.IsLeaf() and B.IsLeaf() and C.IsLeaf()
    if leaf:
        # can cause insertion if C is zero valued
        C = C + A*B
    else:
        for i = 1 to A.NumSubMatricesInCol():
            # iterate, as sub-matrices, over rows in A
            for j = 1 to B.NumSubMatricesInRow():
                # iterate, as sub-matrices, over columns in B
                # (and) grab the destination sub-matrix from C
                Cij = C.SubMtx( i, j )
                for k = 1 to A.NumSubMatricesInRow():
                    Aik = A.SubMtx( i, k )
                    Bkj = B.SubMtx( k, j )
                    if Aik.NonZero() and Bkj.NonZero():
                        # never get here for zero valued sub-DAGs in A or B
                        DAGRecursiveMtxMtxMult( Cij, Aik, Bkj )
```

Figure 5.12: Sparsity and Cache Oblivious Tree-Recursive Matrix-Matrix Multiply

multi-level cache hierarchy. Therefore, there is no generic GEMM implementation that consistently achieves peak performance across all available CPUs. Rather, high performance GEMM implementations are typically tuned to a specific CPU cache hierarchy by setting the tile sizes as a parameter [68].

Tree-recursive algorithms have been proposed to resolve the difficulty of tuning algorithms for different multi-core CPUs [16, 35]. Because such algorithms recurse into smaller and smaller tiles, they naturally create working sets that fit into each level of the CPU cache hierarchy. Therefore, tree-recursive algorithms are said to be cache-oblivious.

5.7.3 Sparsity and Cache Oblivious Tree-Recursive Matrix-Matrix Multiply

Although less widely used than dense matrix-vector multiply, sparse matrix-matrix multiply, or SPGEMM, is important in the context graph algorithms and is also used as a subroutine in other linear algebra algorithms [21]. The most common sparse matrix-matrix multiply algorithm, introduced by Gustavson [33], works column by column using the compressed sparse column, or CSC format.[2] When computing the matrix product $C = A \cdot B$, matrix C is computed one column at a time by reading from matrix B one column at a time. Therefore, the values from source matrix B need be read from memory only once, but the values from the other source matrix A are brought in from memory, on demand, as non-zero intersections are discovered. This guarantees that values from B are reused from processor registers, or L1 cache, but values from A, when reused, are likely to incur the full latency of DRAM access. In effect, for each non-zero value, $B(k, j)$, matrix A is searched for corresponding non-zero entries $A(i, k)$. Because new entries in the current column of C can be discovered as the algorithm works on a particular column in B, an associative array is used to store the current output column of C.

A sparsity and cache oblivious tree-recursive sparse matrix-matrix multiply algorithm is shown in Figure 5.12. As with highly tuned dense matrix-matrix multiply algorithms and tree-recursive cache oblivious algorithms, it iterates over sub-matrices. The sub-matrices are extracted from a logically dense matrix format, such as QTS, using the DAG internal vertices. For QTS, each sub-matrix has dimension $m/2$ by $n/2$, relative to its parent matrix. The sub-matrix accessor, denoted as `M.SubMtx( i, j )` in Figure 5.12, maps to a single iterator register operation, namely read PLID by DAG level and linear index. The number of levels beneath the root vertex is, trivially, the current level plus one. The linear index is a function of the row and column indices `i` and `j` and it is calculated based on the underlying matrix format, such as QTS.

As with sparse vector add, this algorithm is work efficient because only the non-zero pairs $A(i, k)$ and $B(k, j)$ are visited. Storage is $O(nnz \cdot \log(m \cdot n))$ in the worst case, but this is empirically quite pessimistic as shown in Table 5.1. By extension, the number of operand fetches is a factor of $\log(m \cdot n)$ worse than the minimum possible number of operand fetches. This worst case estimate is likely to be very pessimistic

[2] CSC is essentially the transpose of CSR.

for three reasons: the bound is empirically very loose, the iterator register itself maximally reuses internal DAG vertices, which further tightens the bound, and the tree-recursive algorithm naturally utilizes the CPU cache hierarchy. Although there is a cost to sparsity oblivious, it is very efficient compared to the cost of naïve which is $m \cdot n$. Furthermore, there is an intuitive argument that it may be competitive in practice: because it uses matrix tiles, it is arguably more effective at using the CPU cache than competing sparse matrix-matrix multiply algorithms.

5.8 Summary

This chapter shows that hierarchical deduplication provides efficient sparse matrix storage. In current computer systems, sparse matrix formats are special cased to avoid the overhead of storing many zeros. In contrast to current software, hierarchical deduplication provides sparsity oblivious, yet efficient, sparse matrix storage. In fact, hierarchical deduplication improves storage efficiency for sparse matrices with many replicated non-zero values and replicated sub-matrices. The quad-tree symmetric (QTS) format, designed to take advantage of replicated sub-matrices, provides a best case compaction of 5700x relative to the canonical CSR format, which already eliminates the overhead of storing zero valued entries.

In addition, this chapter shows that hierarchical deduplication speeds up sparse matrix-vector multiply (SpMV), a critical task in the inner loop of modern iterative linear system solvers. Using the hierarchical coordinate (HCOO) format, this chapter demonstrates an average speedup of 1.5x. Although there is very little algorithmic data re-use in SpMV, the speedup is a result of finding cache hits because of duplicate values.

Importantly, speedup is demonstrated using current off-the-shelf hardware. Because this hardware has no support for deduplicated memory or for hierarchical deduplication, extra CPU opcodes are required to traverse the DAG. These extra opcodes reduce the efficiency of hierarchically deduplicated SpMV. With iterator registers, projected speedup improves from 1.5x to 4x for compacted matrices that are not bandwidth limited. The speedup of 4x is based on current CPU architectures, but compute resources are increasing faster than bandwidth. Therefore, hierarchically deduplicated SpMV is expected to provide further speedup given current technology trends.

Finally, this chapter describes a new class of sparsity and cache oblivious linear algebra algorithms that are enabled by the iterator register programming model. A key insight is that the iterator register provides an efficient programming interface for tree-recursive algorithms. In addition, the iterator register provides insertion at arbitrary locations in arrays of unbounded size. If implemented using actual iterator register hardware, such algorithms would reduce the need to special case linear algebra subroutines based on particular matrix properties such as sparsity.

Chapter 6

HICAMP and Related Work

This thesis was influenced by many prior research efforts, and, in particular, was inspired by prior work on the HICAMP architecture [24]. The HICAMP architecture provides improved CPU support for parallel programming by implementing a multi-versioned main memory. To reduce the cost of keeping many versions of memory content, and to isolate concurrent workers from corrupting their counterparts' work, it introduced fine-grain hierarchical deduplication. This thesis extended work on HICAMP. In particular, it provided a deduplicated memory compatible with existing hardware and software, and showed that such a memory saves 2x capacity and 40% memory system energy. By extending the deduplicated memory to provide hierarchical deduplication, it also showed significant compaction for certain large-scale sparse matrices and improved performance for sparse matrix-vector multiply. This chapter places this work in the context of HICAMP and other related work. It first describes the HICAMP architecture and how it uses deduplicated memory to provide improved architectural support for parallel programming. Next, it reviews other related work, such as prior work on in-memory compression.

6.1 The HICAMP Architecture

Today, multi-core CPUs have become the norm because single core performance no longer improves at the rate of technology scaling [25]. Despite this, the individual processing cores expose nearly the same instruction set and memory interface as were in use twenty years ago. To support parallel programs, the HICAMP architecture provides a fundamentally new way for program logic to interface with machine memory.

6.1.1 The Need for Parallel Machines

Originally, Moore's Law was an observation, made in 1965 by Gordon Moore, founder of Intel, that the transistor density on silicon had been doubling every year, and he predicted that this would continue into the near future [51]. Since computer architects use these transistors to improve performance, it also became a self-fulfilling prophecy that computer performance will continue to grow exponentially.

Because smaller transistors are both faster and more power efficient, scaling down transistor size increases the performance of a given design, but during the 1990s, designers scaled machine performance even faster than transistor scaling. This additional performance scaling, through circuits and architecture, required higher energy per instruction executed [25]. Historically, this growing energy was not a concern because the early designs, constrained by area, were relatively simple and not power limited. But, in the early 2000s, designs became power limited. Since power is the product of energy per instruction and instructions per second, in a power limited design, the only way to improve performance (instructions per second) is to decrease energy per instruction. Thus the industry opted to reduce the performance of each processor – decreasing energy per instruction – and replicate multiple processing cores on the same silicon die, to increase the total number of instructions per second. This began what is now referred to as the "multi-core" era.

Although it is now common to have 8 cores in a multi-core CPU, it is difficult to write a program that can perfectly exploit this and therefore complete 8 times faster. As pointed out in 1967 by Gene Amdahl, there are practical difficulties in writing *parallel programs* [7]. Every program has instructions that inevitably depend on prior program results, that is, a *sequential dependency*. Fortunately, many programs of practical interest have large portions that can be factored into independent tasks, and therefore can benefit from parallel machines. Ultimately, because these separate tasks are all part of the same program, they all must access, and update, *shared program state*. Difficulties arise when more than one concurrent task attempts to read from and write to the same shared memory location. Specifically, it is possible that one task reads a value – ostensibly the most up-to-date value – at the same time as another task makes an update to *that* value. In this scenario, known as a *memory race*, the task reading the value has stale data and therefore, the program may also produce incorrect results. It is difficult to detect, let alone correct, such subtle program logic errors, therefore, the task of dealing with memory races in parallel programs has its own history of research and literature [46, 62].

Such problems are exacerbated by the lack of hardware support for parallel programs. Today, most programmers avoid memory races by explicitly synchronizing access to shared program state. Current mainstream CPU hardware provides *atomic* – that is race free – update over very small regions of memory. To synchronize access, such a region is used as a *lock* – that is, a value in memory that signals whether or not a concurrent task is granted access to a larger region of shared program state. Locks tend to reduce performance, because a given concurrent task will stall if it is locked out of a region. To avoid this behavior – that is, to avoid *blocking* on synchronization – programmers use separate, or fine-grain, locks to protect logically distinct regions of shared state. In all but the simplest cases, fine-grain locks are difficult to reason about and eventually lead to even more intractable problems such as deadlock: for example, two separate tasks, each waiting on a lock that the other task holds, stall and therefore the program itself makes no progress. These problems motivate both different programming models and thus hardware support for them.

6.1.2 Transactional Memory

First described by Herlihy in 1993 [37], transactional memory provides hardware support for *non-blocking synchronization*. Now, 20 years later, transactional memory has gained limited hardware support in commercial and supercomputing machines released by both Intel and IBM [67, 22, 38]. Interest in transactional memory increased in the past 10 years as it became clear that multi-core CPUs would become common. This resulted in many research papers proposing improvements to transactional memory [34, 59, 8, 52, 50, 23, 73, 17], and the proposals made in some of those have now been implemented, such as speculative lock elision [58]. The essential idea running through these efforts is to have concurrent tasks optimistically proceed without stalling others by keeping a private record of their updates to shared program state. Each concurrent task tracks a read-set and a write-set, or specifically, the set of shared memory locations it reads from and the set of shared memory locations it writes to. A given concurrent task is forced to abort if a separate task completes and its write-set intersects with the read *or* write set of the task in question. If the intersection is in the write-set, then it is termed a *write-write* conflict. Because each task reads from the non-isolated currently committed version of shared program state, it can read inconsistent state. Such a conflict is known as a *read-write* conflict. If no conflict is detected, when the task completes, all of its updates are made visible to all other tasks at a given instant, an operation referred to as *atomic commit*.

6.1.3 Multi-Versioned Memory

By implementing a multi-versioned memory, HICAMP provides an alternate model for non-blocking synchronization. Distinct from transactional memory, HICAMP provides each concurrent task with an isolated copy of shared program state, known as a *snapshot*. A snapshot is a copy of some portion of shared program state at a given instant in time, typically the time at which a given concurrent task began its execution. Because of snapshots, no task can read data from logically distinct points in time, that is, no task can read inconsistent data. Therefore, HICAMP has no read-write conflicts, and only write-write conflicts need be detected and correctly managed. Because read-write conflicts are far more common than write-write conflicts HICAMP allows more concurrent tasks to complete, thus providing higher program performance.

In HICAMP, as in transactional memory, when a given task completes, it publishes its updates by atomic commit, but unlike transactional memory, this update is accomplished by swapping entire memory snapshots by a *single* compare-and-swap operation on the root vertex of a hierarchically deduplicated memory region. If, in the time between snapshot creation and commit, some other task updated a value in that memory region, a write-write conflict is detected by root vertex comparison. In addition, HICAMP provides support for merge-updates which can resolve write-write conflicts according to programmer supplied semantics. The following sections describe how HICAMP implements a multi-versioned memory using hierarchically deduplicated memory.

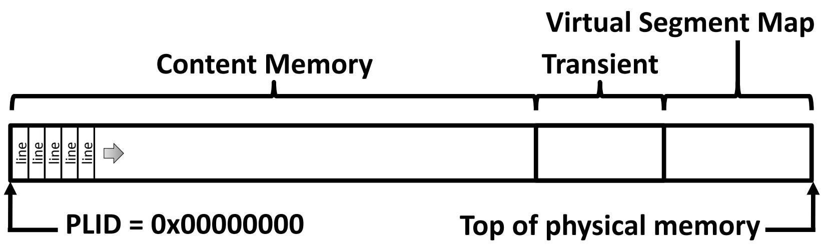


Figure 6.1: HICAMP Architecture Memory Layout

6.1.4 HICAMP Memory Model

HICAMP efficiently implements the abstraction of a multi-versioned memory using the following primitives:

1. Memory lines: unique and fixed size (64 bytes, e.g.) regions of memory, as described in Chapter 2.

2. Memory segments: sets of unique memory lines, each set organized as a directed acyclic graph, or DAG, and accessed by iterator register, as described in Chapter 5. The memory segments contain hierarchically deduplicated program data.

3. Virtual segment map: a list of software visible *virtual segment ids*, or *VSIDs*, and their respective memory segments, each identified by a PLID that refers to a particular DAG root vertex.

As with deduplicated memory, introduced in Chapter 2, in HICAMP, the CPU cores do not directly access physical memory locations. In HICAMP, hardware manages all physical memory locations, and provides the abstraction of memory segments of unbounded size. To do so, memory is split into three regions, as shown in Figure 6.1: the content memory, containing deduplicated memory lines, the transient memory, containing non-deduplicated content and hash-table overflows, and the virtual segment map.

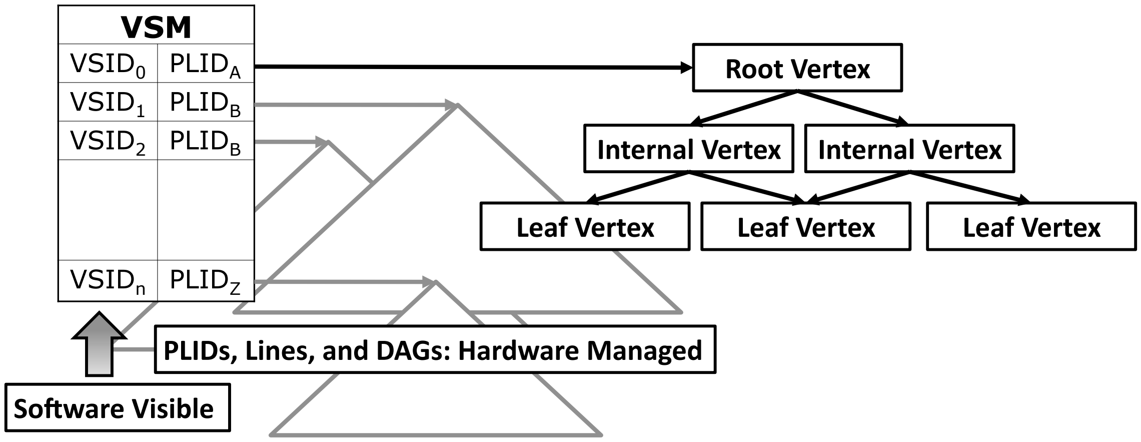


Figure 6.2: HICAMP Virtual Segment Map (VSM)

The *virtual segment map*, or *VSM*, provides the programming interface for software. To access memory, software specifies a VSID[1] and a linear offset into the segment. The linear offset effectively selects a particular leaf node in a DAG, as shown in Chapter 5, Figure 5.1. The virtual segment map, illustrated in Figure 6.2, connects the software visible VSIDs to root PLIDs, each of which identifies a particular DAG comprised of unique memory lines.

HICAMP provides isolated snapshot copies by giving a reference to a segment. Specifically, to create a snapshot, HICAMP reads the PLID associated with a VSID from the virtual segment map and increments the reference count of the root vertex (a memory line) that is referred to by that PLID. By logical extension from the fact that the root vertex also references its children, and therefore its grandchildren, this effectively maintains a reference to the entirety of the DAG content. The snapshot and the globally shared program state exist as two separate logical entities, but because of hierarchical deduplication, they share exactly the same physical memory resources. Therefore, HICAMP provides a snapshot copy in exactly one operation – reference count increment – and, at no additional cost to memory capacity. The snapshot is

[1]It is possible to request a new VSID, or release a VSID.

isolated, that is guaranteed to remain immutable and in memory, as long as the root vertex is referenced.

The iterator register, described in Chapter 5, provides the programming interface for the HICAMP architecture. It supports simple load and store commands by providing DAG read and write by data and offset. As its name suggests, it also supports iteration over the data elements stored in a DAG, both including and skipping zero valued elements. Finally, it provides the programming interface for atomic segment update, as described in the next section.

6.1.5 Atomic Segment Update

To provide atomic commit of entire memory segments, HICAMP implements atomic compare-and-swap (CAS) for PLIDs in the virtual segment map (VSM). Logically, if each vertex in the DAG is unique, so too is the root vertex. Therefore, two "separate" DAGs, or segments, can be compared for equality by comparing only the memory line that is their root vertex. By extension, atomic CAS on the VSM is atomic CAS for an entire memory segment.

As each concurrent task executes, it makes local updates creating a new private memory segment in addition to its snapshot. Therefore, when the task completes, it holds a reference to two separate DAGs: the snapshot, and the new DAG representing its updated version of memory. Using a VSID, the commit, or end-of-transaction, opcode issues a CAS on the segment root PLID in the VSM. The root PLID in the VSM is compared to the snapshot root PLID. If they match, then the VSM is updated with the PLID that refers to the updated version of memory, and the snapshot is discarded. Therefore, using virtual segment map compare-and-swap, HICAMP provides atomic update to regions of unbounded size in a *single* memory operation.

If the PLID in the VSM does not match the PLID for the original snapshot, then the CAS operation fails, and one of two options is invoked: according to behavior specified by the programmer, either the task re-executes using a new snapshot, or a merge-update is performed. Even if the memory segment CAS fails, there may be no write-write conflicts at the leaf level. If program semantics allow, HICAMP can merge the two updated memory segments by CAS on internal DAG vertices – effectively performing CAS on DAG sub-graphs. Furthermore, if program semantics allow, writes that conflict at the same offset into the segment can be merged if the correct final value can be inferred from the snapshot, as in the following: $\textit{final shared value} = \textit{current shared value} + \textit{updated value} - \textit{original value}$. This rule is correct if the two disparate tasks are applying simple arithmetic increments to shared program state.

6.1.6 Advantages of HICAMP

HICAMP offers several advantages over current architectures and over proposed transactional memory implementations. HICAMP provides a practical implementation of Herlihy's theoretical construction of non-blocking synchronization by compare-and-swap [36]. By logically eliminating read-write conflicts, it allows more transactions to succeed and therefore provides a performance advantage. It allows long running read-only transactions to succeed without any additional explicit sequencing. Merge update allows transactions with apparent write-write conflicts to commit where program semantics allow. Many of the proposed implementations for transactional memory rely on small buffers, typically the L1 cache, to store logically private copies of shared state. HICAMP semantically avoids this restriction and allows transactions to access and modify memory regions of unbounded size. It implements fine-grain hierarchical deduplication which allows maximum sharing between logically isolated snapshot copies, and maximizes memory resource sharing across separate processes and virtual machines. Therefore, HICAMP is an attractive option relative to current state-of-the-art support for parallel programming and it captures the benefit of deduplicated memory.

6.2 Related Work

Other prior research has addressed increasing memory capacity by compression and improving sparse matrix-vector multiply performance. This section places this thesis in the context of these related efforts.

6.2.1 Page Deduplication

It is well known that duplicate values exist in physical memory in "cloud" virtualized systems such as Amazon EC2, or privately managed virtual machine infrastructures. Waldspurger [66] shows that virtual machine hypervisors can exploit duplicate pages across virtual machines by actively scanning main memory, a technique known as *transparent page sharing*. Prior work on HICAMP [24] also shows that there are almost twice as many fine-grain duplicates as page duplicates in virtual machine systems.

Relative to transparent page sharing, deduplicated memory is very fine-grain (64 vs. 4096 bytes), operates at a much finer time scale (nanoseconds versus minutes), and incurs no additional processing overhead [32]. Furthermore, deduplicated memory decouples sharing granularity from page granularity. Virtualized TLB misses incur expensive nested page-table walks, therefore, the trend is toward large (2 MB) pages. Because there are so few 2 MB page duplicates and because of the non-trivial CPU cost to scan memory, VMware ESX does not enable transparent page sharing with large pages [32]. Deduplicated memory provides fine-grain sharing regardless of page size.

Based on results from both *zest* and *zsim*, and prior work on HICAMP, deduplicated memory outperforms page sharing in terms of memory capacity. This work calculates page sharing compaction using just the total-to-unique ratio because the translation overhead of page sharing is inherent to the page table. As shown in Figures 4.1 and 4.2, for SPEC and PARSEC, there are far more fine-grain duplicates than page duplicates. For the applications running in actual datacenters, Chapter 2 Section 2.6, zest calculates page compaction to be 1.47x. Transparent page sharing cannot share pages that are being modified by programs, but zest does not know which pages are being modified, and which are static and read-only. Therefore, deduplicated memory provides a factor of 2x savings, in excess of the highly optimistic estimated 1.47x page sharing indicated by zest.

6.2.2 Exploiting Zero Values

Dusser and Seznec [27] exploit zero values to provide increased memory capacity. Results in this work indicate that zeros are not common enough to provide a significant benefit. On real systems, all memory is always in use either by applications, or by the operating system for IO buffers and disk cache – while zeros remain common, logically, they are not useful to either application memory or to the operating system, and therefore their contribution to the total remains low. Results from *zest*, shown in Figure 2.6, indicate that zeros can be as low as 3% of the total while fine-grain deduplication still provides in excess of 2x compaction.

6.2.3 In Memory Compression

Several efforts have been made to provide increased memory capacity through compression. Circa 2000, IBM went to market with their memory expansion technology (MXT) [64] which used a sophisticated lossless compression algorithm applied to 1 KB chunks of data. More recently, GPUs have employed in memory compression to reduce bandwidth for highly compressible texture data [72]. Ekman and Stenstrom [28] describe a line based memory compression scheme using a set of frequent patterns and cite decompression latency as an advantage over MXT. And, most recently, Pekhimenko, et al. [55], describe a similar line based memory compression scheme, but instead use base-delta-immediate [56] compression which has single cycle decompression latency.

Deduplication is actually simpler and higher performance than compression. For both compression and deduplication, allocation occurs on write, and therefore does not stall the CPU. Both introduce an extra level of indirection. Compression schemes (as distinct from deduplication) also require decompression – this adds additional latency, no matter how efficient. Additionally, compression schemes must reallocate and copy if a block grows or shrinks. MXT reallocation is handled in hardware based on its free list and sector translation table, similar to the translation array described in this work. On the other hand, the line based compression schemes [28, 55] are not transparent to the operating system. Both of these methods store their translation metadata in the page table which requires operating system modification and introduces the complicated issue of TLB consistency. As a result, these methods incur a costly operating system trap when blocks grow beyond their compressed boundaries. Both line compression schemes also introduce the problem of fragmentation, namely regions of unused memory that are too small to be useful. To handle fragmentation, they quantize the compressed page sizes. Again, requiring extensive operating system modification, they require that the operating system maintain a pool of free pages for each of the compressed sizes. Although this guarantees contiguous memory use within each pool, the challenge of one particular pool running out of free pages is not addressed. In contrast, with deduplicated memory, every write is the same, it is transparent to the operating system, and there is no fragmentation.

MXT [64] reports compression between 2x and 3x: similar to results from zest. On the other hand, to achieve high compression, MXT uses large 1 KB memory blocks which risks significant over-fetch for many applications. Ekman and Stenstrom [28] report 1.4x increase in capacity and Pekhimenko, et al. [55], report 1.7x increase in capacity. Although in-memory texture compression is used by some current GPUs [72], it is difficult to compare to because the underlying data is fundamentally different. It is well known that compression faces a fundamental tradeoff between block size and compression ratio. Comparing the results from MXT [64], which compresses 1 KB blocks, to the line based methods [28, 55], illustrates this tradeoff. The line based compression methods were motivated by the fact that large blocks, such as 1 KB as used by MXT, cost too much in terms of over-fetch, but they do not provide nearly the same capacity increase as fine-grain (i.e. line based) deduplication.

6.2.4 Sparse Matrix-Vector Multiply

The literature on optimizing sparse matrix-vector multiply (SpMV) is vast: a testament to its importance as a scientific and high performance computing subroutine. Historically, it was difficult to achieve peak performance due to the very tight inner loop of SpMV coupled with the indirect and random access pattern into the source vector. This motivated research into methods for adding extra work into the inner loop and regularizing data access patterns. Both can be achieved by finding dense matrix sub-blocks or through domain specific knowledge that allows a highly customized SpMV kernel to be invoked. Because the bandwidth limitation has long been recognized, research has also focused on methods for reducing memory traffic. Thus, most research into optimizing SpMV can be categorized as either reducing memory traffic, adding work to the inner loop, regularizing data access patterns, or some combination of these.

Autotuning

Vuduc and Williams [65, 71], et al., describe *OSKI*, a framework for autotuning block compressed sparse row SpMV kernels. Using a CSR index structure to point to dense submatrices (or blocks), the block compressed sparse row (BCSR) format captures all three SpMV optimization methods: more work per loop iteration, regularized data access, and reduced data transfer. Because the blocks are fixed size and dense, they exhibit regular data access patterns and allow for optimal loop unrolling. Further, matrices with high block fill ratios require less storage because of reduced index data overhead.

Regularized Data Access

With the advent of programmable GPUs, researchers have also attempted to utilize the high GPU to video-memory bandwidth to improve SpMV performance. Bell and Garland [13] describe methods to approach maximum sustainable bandwidth for SpMV on GPU through several techniques directed at regularizing data access. Because power constraints now force GPUs to include large on-die caches, the regular control flow of the HCOO format makes it an interesting candidate for GPU implementation.

Reduced Data Transfer

In the reduced bandwidth compressed sparse block format, Buluç, et al. [19], use bit-masked register blocks, similar to the NZD pattern matrices (Chapter 5, Section 5.2.5) to reduce the storage requirements of the compressed sparse block format [20]. Kourtis, et al. [41, 42], use delta-encoding of index values and non-zero value deduplication to reduce data transfer. In the compressed sparse extended method [43], Kourtis uses offline analysis to customize the delta encoding scheme to the specific matrix which results in a custom SpMV kernel on a per matrix basis. Willcock and Lumsdaine [70] propose delta-coded sparse rows, similar to the work of Kourtis. In the row pattern compressed sparse row format, Willcock and Lumsdaine [70] exploit patterns in each matrix row. Finally, Belgin, et al. [12], exploit distinct sub-matrix patterns.

Recursive Formats

Recursive matrix layouts, similar to QTS (Chapter 5 Section 5.2.2), have inspired recent papers [15, 49]. In the recursive sparse blocks format, Martone, et al. [49], use a quad-tree with Z-Morton ordering to store sparse sub-matrices that are sized to balance parallel work partitions. Recognizing that the path through the tree structure implies an offset, to save storage and data transfer, Martone, et al., use smaller 16 bit indices in their leaf elements.

Concurrent Symmetric SpMV

Several recent papers have also explored techniques for enabling concurrent symmetric SpMV. Buluç, et al. [19], permute a sparse blocked matrix so that most non-null blocks are close to the diagonal. Blocks that are one and two positions off diagonal are then assigned to separate rounds, coordinated to avoid write-write conflicts. Blocks further off diagonal, hopefully few in number, require atomic updates to the destination vector. Krotkiewski and Dabrowski [45], also permute the matrix to cluster non-zeros on the diagonal and use local result buffers to allow separate workers to proceed without conflict. Afterwards, the result is accumulated and the method attempts to overlap communication and computation across rounds. Such methods illustrate the challenge of concurrent symmetric SpMV. They provide the benefit of parallelism, but require pre and post-processing steps, and still require explicit synchronization.

Tangwongsan, et al. [15], also provide lock and synchronization free concurrent symmetric SpMV using a hierarchical storage format similar to QTS. For symmetric matrices, their method achieves an average of 1.8x less memory traffic, but does not inherently exploit symmetry in non-symmetric matrices.

Method	Best Compaction	Best Speedup	Index Compaction	Value Compaction	Data Reuse	Kernel Code
DCSR [70]	1.4x	1.5x	Yes	No	No	Generic
PBR [12]	1.5x	1.5x	Yes	No	Via Code[1]	Per Matrix
RPCSR [70]	1.5x	1.5x	Yes	Yes	Via Code[1]	Per Matrix
CSR-DU [41, 42]	1.3x	1.8x	Yes	No	No	Generic
CSX [43]		1.9x	Yes	No	Via Code[1]	Per Matrix
RSB [49]		2.0x	Yes	No	No	Generic
CSR-VI [41, 42]	2.4x	2.5x	Yes	Yes	Yes	Generic
RBCSB [19]		3.5x	Yes	No	No	Generic
DAG (this work)	5700x	3.7x	Yes	Yes	Yes	Generic
(1) Matrix structure embedded in custom generated code						

Table 6.1: Comparison of Non-Symmetric SpMV Compaction Techniques

SpMV Related Work Comparison

Table 6.1 compares the techniques using explicit compaction and shows the best-case speedup for each method. Data in Table 6.1 are based on a comparison to multi-threaded CSR when available, and otherwise the best proxy available in the respective paper.

6.3 Summary

This thesis extends prior work on the HICAMP architecture [24]. Using a multi-version memory, HICAMP improves on the programming semantics offered by traditional transactional memory systems by providing isolated and consistent read. Therefore, HICAMP eliminates read-write conflicts and improves parallel program performance. While HICAMP provides improved architectural support for parallel programs, this work focuses on providing memory capacity and compatibility with existing hardware and software.

Past efforts at increasing memory capacity have focused on either software managed page deduplication or hardware optimized memory compression. Results in this work indicate that fine-grain sharing outperforms page sharing in terms of capacity, even though the page sharing estimate is optimistic. In-memory compression incurs extra read latency for decompression that cannot be hidden by optimizations such as translation caching in the direct translation buffer. Further, prior efforts at memory compression did not achieve a factor of 2x capacity increase, except for IBM MXT which used large 1 KB allocations. Finally, the trend toward virtualization suggests deduplication should be used, in any case, regardless of whether compression is applied afterward.

Prior work on SpMV has used various kinds of compression and deduplication. This work shows that hierarchical deduplication provides compaction far in excess of previous results, and also a best case speedup in excess of previous results. The extra CPU instructions required for DAG traversal are eliminated when using the iterator register in the HICAMP architecture. Therefore, HICAMP enables even higher speedup than demonstrated in this work, and also enables sparsity oblivious algorithms.

Relative to prior work, fine-grain deduplication provides higher capacity and hardware based deduplication provides higher performance. Deduplication, itself, is global relative to compression, which is local. In summary, many prior research efforts have been improved upon in this thesis, while acknowledging this debt, the results in this thesis indicate that hardware based fine-grain in-memory deduplication is preferable.

Chapter 7

Conclusion

In conclusion, deduplicated memory provides significant additional memory capacity at low cost. Alternate approaches to expanding memory capacity, such as page sharing or memory compression, have higher costs and less benefit. Emerging non-volatile storage technologies show promise at delivering "storage-class memory," but none of these are viable at present and their increased latency necessitates large DRAM caches. For over one decade, virtual machine hypervisors have used software to suppress duplicate pages, but no one had previously investigated deduplication in non-virtualized systems. Results from the *zest* memory content analysis indicate that many fine-grain duplicates exist in large-scale high-memory non-virtual workloads. In these workloads, on average, the total number of values stored in memory is three times higher than the number of unique entries.

Deduplicated memory is feasible and requires very few changes to existing CPU hardware and operating systems. It is implemented by modifying the CPU memory controller to automatically read from and insert to an in-memory hash table. These modifications, and the deduplicated memory layout, described in Chapter 2, incur some overhead in terms of storage and memory access latency. The translation, used to memorize the relationship between bus address and unique content, costs 1/16th the apparent size of memory. Including the cost of translation, deduplication increases effective memory capacity by over 2x across a broad range of applications.

Deduplicated memory requires several memory operations for each logical memory read or write – this raises the concern of reduced performance due to read latency and bandwidth. Simulated performance results, for deduplicated memory without further optimization, indicate that the performance impact is typically very small. This is because the CPU cache is effective at hiding memory access latency and because bandwidth use is usually not a performance bottleneck.

Even if the performance impact of memory deduplication is low, prima facie, it provides only the same performance, in the best case. Deeper integration into the CPU caches further mitigates this impact, and in certain cases improves performance. The direct translation buffer (DTB) and deduplicated cache (DDC), described in Chapter 3, reduce read latency and bandwidth use. Performance results, in Chapter 4, indicate that with the DTB and DDC with lazy deduplication, performance improves in certain cases, and memory bandwidth use is, on average, reduced. Both effects, reduced bandwidth, and increased performance, are fundamentally a result of increased effective cache capacity. Therefore, deduplicated memory increases performance when the size of memory, or cache, is effectively increased and application performance is sensitive to memory or cache size.

Without the DDC, bandwidth use does increase – raising the concern of increased dynamic power use. Current and emerging DRAM memory technology has very high static power use relative to dynamic power. In this context, deduplicated memory saves power because fewer memory devices are needed and the static power savings outweigh the dynamic power cost. Therefore, the extra bandwidth use pays for itself both in terms of capacity and power. Because the DDC reduces bandwidth, on average, both static and dynamic power are reduced – including the DDC, memory power savings improve from 40% to 48%.

Current technology trends make the case for deduplicated memory even more urgent. Emerging application domains, such as server virtualization, big-data analytics, and in-memory key value stores all demand machines with large memories. The underlying data in such applications contains many duplicates, as demonstrated by the results from *zest*. Furthermore, it is becoming difficult to continue scaling DRAM memory because of the capacitor aspect ratio. The increased cost of advanced technology nodes has caused some memory manufacturers to leave the business and it is unlikely that investors will take the risk of starting a new company to enter that market. Together, these trends indicate that both hardware and software engineers will face increasing pressure to use memory efficiently. Because deduplicated memory provides this increased efficiency at a reasonable cost, the industry may be compelled to use such technology in future CPU designs.

In this context, it makes sense to explore additional benefits of deduplicated memory that can be enabled after its integration. For deduplicated memory, as described in Chapter 2, compaction is bounded by the cost of translation. Hierarchical deduplication, described in Chapter 5, provides compounded benefit, but incurs the cost of additional indirection. Although deduplicated memory appears most applicable to large-scale web services, hierarchical deduplication offers the possibility of sparsity oblivious algorithms in the context of high-performance and scientific computing. For certain sparse matrices, hierarchical deduplication provides very high compaction, but importantly, sparsity oblivious storage is efficient relative to compact sparse formats on current machines. Unfortunately, the sparsity oblivious formats require many additional CPU opcodes to traverse the graph based data structure. This overhead is significantly reduced when using the iterator register, described by HICAMP [24]. If a basic deduplicated memory is integrated into future CPUs, then the iterator register is a possible extension that enables both hierarchical deduplication and efficient manipulation of sparse graph based data structures.

Memory capacity is not the only problem that can be addressed by deduplication. As described in Chapter 6, power limits have driven the emergence of multi-core CPUs. Unfortunately, the current mutual-exclusion based programming interface to these parallel machines is severely lacking. The HICAMP architecture improves on emerging update-in-place transactional memories by providing thread isolation and by eliminating all read-write conflicts. The HICAMP programming model is both elegant and efficient, but its barriers to adoption are very high. It requires new hardware, such as iterator registers, and it requires significant changes to the ISA. On the other hand, because deduplicated memory is an integral part of HICAMP and is also valuable on its own merits, it could be the first step toward enabling such snapshot isolated memories. Therefore deduplicated memory is not only a compelling extension to current CPU architectures, it also enables further improvements that would be otherwise difficult to adopt.

Bibliography

[1] "The Cisco UCS with NetApp Storage for SAP HANA Solution Delivers Real-Time Decisions All Day, Every Day," http://www.cisco.com/en/US/solutions/collateral/ns340/ns517/ns224/ns944/sap_hana_scale_out_netapp_solution.pdf.

[2] "Calculating Memory System Power for DDR3," Micron, Technical Note TN-41-01, 2007.

[3] "Big Gains Forecast in Quarterly DRAM ASP," http://www.icinsights.com/data/articles/documents/570.pdf, July 2013.

[4] "Commodity DRAM Price Surges, Record Profits for Suppliers in 2Q13," http://www.dramexchange.com/WeeklyResearch/Post/2/3481.html, August 2013.

[5] "OS X Mavericks: Core Technologies Overview," https://www.apple.com/media/us/osx/2013/docs/OSX_Mavericks_Core_Technology_Overview.pdf, Apple, White Paper, October 2013.

[6] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing MapReduce on Heterogeneous Clusters," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, London, UK, March 2012, pp. 61–74.

[7] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the Spring Joint Computer Conference (AFIPS '67)*, Atlantic City, NJ, April 1967, pp. 483–485.

[8] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*, San Francisco, CA, February 2005, pp. 316–327.

[9] S. Baek, J. Choi, D. Lee, and S. H. Noh, "Energy-Efficient and High-Performance Software Architecture for Storage Class Memory," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, pp. 81:1–81:22, March 2013.

[10] L. Barroso and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines," *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–108, July 2009.

[11] A. Bechtolsheim, "Technologies for Data-Intensive Computing," presented at the 13th International Workshop on High Performance Transaction Systems (HPTS '09), Pacific Grove, CA, October 2009.

[12] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-Based Sparse Matrix Representation for Memory-Efficient SMVM Kernels," in *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*, Yorktown Heights, NY, June 2009, pp. 100–109.

[13] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors," in *Proceedings of the 2009 ACM/IEEE conference on Supercomputing (SC '09)*, Portland, OR, November 2009, pp. 18:1–18:11.

[14] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "The Keccak Reference," http://keccak.noekeon.org/, January 2011.

[15] G. Blelloch, I. Koutis, G. Miller, and K. Tangwongsan, "Hierarchical Diagonal Blocking and Precision Reduction Applied to Combinatorial Multigrid," in *Proceedings of the 2010 ACM/IEEE conference on Supercomputing (SC '10)*, New Orleans, LA, November 2010, pp. 1–12.

[16] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Low Depth Cache-Oblivious Algorithms," in *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*, Thira, Greece, June 2010, pp. 189–199.

[17] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood, "TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*, Beijing, China, June 2008, pp. 127–138.

[18] W. Buchholz, "File Organization and Addressing," *IBM Systems Journal*, vol. 2, no. 2, pp. 86–111, June 1963.

[19] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS '11)*, Anchorage, AK, May 2011, pp. 721–733.

[20] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks," in *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*, Calgary, AB, Canada, August 2009, pp. 233–244.

[21] A. Buluç and J. R. Gilbert, "Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[22] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust Architectural Support for Transactional Memory in the Power Architecture," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, Tel-Aviv, Israel, June 2013, pp. 225–236.

[23] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*, Boston, MA, June 2006, pp. 227–238.

[24] D. R. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi, "HICAMP: Architectural Support for Efficient Concurrency-Safe Shared Structured Data Access," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, London, UK, March 2012, pp. 287–300.

[25] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "CPU DB: Recording Microprocessor History," *Communications of the ACM*, vol. 55, no. 4, pp. 55–63, April 2012.

[26] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1:1–1:25, November 2011.

[27] 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), Heraklion, Greece, January 2011, pp. 77–86.

[28] M. Ekman and P. Stenstrom, "A Robust Main-Memory Compression Scheme," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, Madison, WI, June 2005, pp. 74–85.

[29] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Fast Data Analysis Using Coarse-grained Distributed Memory," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD/PODS '12)*, Scottsdale, AZ, May 2012, pp. 689–692.

[30] M. G. Ertosun, "Novel Capacitorless Single Transistor Dram Technologies," Ph.D. dissertation, Stanford University, May 2010.

[31] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: a Study of Emerging Scale-Out Workloads on Modern Hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, London, UK, March 2012, pp. 37–48.

[32] F. Guo, "Understanding Memory Resource Management in VMware vSphere 5.0," http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf, VMware, Technical White Paper, 2011.

[33] F. G. Gustavson, "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, September 1978.

[34] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, “Transactional Memory Coherence and Consistency,” in *Proceedings of the 31st Annual International Symposium On Computer Architecture (ISCA ’04)*, M&#252;nchen, Germany, June 2004, pp. 102–113.

[35] A. Heinecke and M. Bader, “Parallel Matrix Multiplication Based on Space-Filling Curves on Shared Memory Multicore Platforms,” in *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem? (CF ’08)*, Ischia, Italy, May 2008, pp. 385–392.

[36] M. Herlihy, “A Methodology for Implementing Highly Concurrent Data Structures,” in *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming (PPoPP ’90)*, Seattle, WA, March 1990, pp. 197–206.

[37] M. Herlihy and J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures,” in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA ’93)*, San Diego, CA, May 1993, pp. 289–300.

[38] M. Herlihy and N. Shavit, “Transactional Memory: Beyond the First Two Decades,” *SIGACT News*, vol. 43, no. 4, pp. 101–103, December 2012.

[39] L. Jiang, Y. Zhang, B. R. Childers, and J. Yang, “FPB: Fine-Grained Power Budgeting to Improve Write Throughput of Multi-level Cell Phase Change Memory,” in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’12)*, Vancouver, BC, Canada, December 2012, pp. 1–12.

[40] K. Kim, "Future Silicon Technology," in *Proceedings of the 42nd European Solid-State Device Research Conference (ESSDERC '12)*, Bordeaux, France, September 2012, pp. 1–6.

[41] K. Kourtis, G. Goumas, and N. Koziris, "Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression," in *Proceedings of the 37th International Conference on Parallel Processing (ICPP '08)*, Portland, OR, September 2008, pp. 511–519.

[42] ——, "Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression," in *Proceedings of the 5th Conference on Computing Frontiers (CF '08)*, May 2008, pp. 87–96.

[43] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: An Extended Compression Format for SpMV on Shared Memory Systems," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*, San Antonio, TX, February 2011, pp. 247–256.

[44] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server Engineering Insights for Large-Scale Online Services," *IEEE Micro*, vol. 30, no. 4, pp. 8–19, July/August 2010.

[45] M. Krotkiewski and M. Dabrowski, "Parallel Symmetric Sparse Matrix-Vector Product on Scalar Multi-Core CPUs," *Parallel Computing*, vol. 36, no. 4, pp. 181–198, April 2010.

[46] E. A. Lee, "The Problem with Threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[47] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated Memory for Expansion and Sharing in Blade Servers,"

in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, Austin, TX, June 2009, pp. 267–278.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, Chicago, IL, June 2005, pp. 190–200.

[49] M. Martone, S. Filippone, P. Gepner, M. Paprzycki, and S. Tucci, "Use of Hybrid Recursive CSR/COO Data Structures in Sparse Matrix-Vector Multiplication," in *Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT '10)*, Wisla, Poland, October 2010, pp. 327–335.

[50] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*, San Diego, CA, June 2007, pp. 69–80.

[51] G. Moore, "Cramming More Components onto Integrated Circuits," *Electronics Magazine*, vol. 38, no. 8, April 1965.

[52] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, "LogTM: Log-Based Transactional Memory," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA '06)*, Austin, TX, February 2006, pp. 254–265.

[53] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez, "Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM

Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, Tel-Aviv, Israel, June 2013, pp. 48–59.

[54] A. Pal, A. Nainani, S. Gupta, and K. Saraswat, "Performance Improvement of One-Transistor DRAM by Band Engineering," *IEEE Electron Device Letters*, vol. 33, no. 1, pp. 29–31, January 2012.

[55] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework," to appear in Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '13), Davis, CA, USA, December 2013.

[56] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-Delta-Immediate Compression: Practical Data Compression for On-chip Caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*, Minneapolis, MN, September 2012, pp. 377–388.

[57] M. Rajashekhar, "Caching at Twitter and Moving Towards a Persistent, In-Memory Key-Value Store," http://cloud.berkeley.edu/data/twittercache.pdf.

[58] R. Rajwar and J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," in *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture (MICRO '01)*, Austin, TX, December 2001, pp. 294–305.

[59] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, Madison, WI, June 2005, pp. 494–505.

[60] F. Rastgar and T. Rossi, "Tackling the Challenges of Transition to DDR4," *EE Times Asia*, February 2013.

[61] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, Tel-Aviv, Israel, June 2013, pp. 475–486.

[62] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, November 1997.

[63] J. Sylve, A. Case, L. Marziale, and G. G. Richard, "Acquisition and Analysis of Volatile Memory from Android Devices," *Digital Investigation*, vol. 8, no. 3-4, pp. 175–184, February 2012.

[64] R. Tremaine, P. Franaszek, J. Robinson, C. Schulz, T. Smith, M. Wazlowski, and P. Bland, "Ibm memory expansion technology (mxt)," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 271–285, March 2001.

[65] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A Library of Automatically Tuned Sparse Matrix Kernels," in *Proceedings of Scientific Discovery Through Advanced Computing (SciDAC '05)*, San Francisco, CA, June 2005, pp. 521–530.

[66] C. Waldspurger, "Memory Resource Management in VMware ESX Server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. Special Issue, pp. 181–194, 2002.

[67] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*, Minneapolis, MN, September 2012, pp. 127–136.

[68] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC '98)*, Orlando, FL, November 1998, pp. 1–27.

[69] A. Wiggins and J. Langston, "Enhancing the Scalability of Memcached," http://download-software.intel.com/sites/default/files/m/0/b/6/1/d/45675-memcached_05172012.pdf.

[70] J. Willcock and A. Lumsdaine, "Accelerating Sparse Matrix Computations via Data Compression," in *Proceedings of the 20th International Conference on Supercomputing (ICS '06)*, Cairns, Queensland, Australia, June 2006, pp. 307–316.

[71] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC '07)*, Reno, NV, November 2007, pp. 38:1–38:12.

[72] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, March/April 2011.

[73] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, Phoenix, AZ, February 2007, pp. 261–272.