TIWARI, DEVESH. Analyzing and Accelerating Runtime Systems on Multicore Architecture. (Under the direction of Yan Solihin.)

Technology scaling has made multicore architectures commercially prevalent. However, exploiting multicore parallelism for performance remains challenging for programmers, because of side-effects of parallel programming such as concurrency management, data-races, deadlocks etc. Therefore, there is a need for solutions that can exploit the computational power of multicore platform without burdening the programmers with concurrency management. In this dissertation, we use two specific runtime systems (dynamic memory management runtime system and shared memory MapReduce runtime system) as a vehicle to demonstrate that runtime systems, that can exploit the multicore parallelism transparently, can be part of the solution.

First, we show how to reduce the overhead of dynamic memory management runtime system by parallelizing it on a multicore architecture. However, traditionally dynamic memory management runtime systems execute sequentially and hence, cannot take advantage of multicore platform. Moreover, tasks such as malloc and free are often very small, and hence, executing them in parallel on a separate core may even degrade the performance due to high communication and synchronization cost. We use dynamic memory management runtime system as an example to show how to efficiently exploit the fine-grained parallelism in dynamic memory management. We also show the design of such a system that exploits the fine-grained parallelism in the runtime library while remaining transparent to the application and memory allocation library without modifying any of those.

Second, we focus on analyzing and optimizing shared memory MapReduce runtime system. Shared memory MapReduce runtime systems allow programmers to express parallelism at a higher level, and provide automatic management of concurrency. However, due to high level of abstraction programmers are often not aware of performance bottlenecks of such runtime systems. Hence, they may achieve only suboptimal performance gains. To address this, we build a new analytical model to analyze key performance factors of shared memory MapReduce runtime libraries and discover several previously unknown and non-intuitive performance trends. Findings and insights from our analytical model can help both programmers and system designers in understanding and explaining the performance bottlenecks of these runtime systems.

Finally, we optimize the shared memory MapReduce runtime system design for the cases where programs are often run multiple times with either identical or slightly-changed input to exploit the significant opportunity for computation reuse in such cases. We propose a novel technique for computation reuse in shared memory MapReduce runtime systems, which we refer to as MapReuse. MapReuse detects input similarity by comparing their signatures. It skips re-computing output from a repeated portion of the input, computes output from a new portion of input, and removes output that corresponds to a deleted portion of the input. We show that MapReuse achieves significant performance improvement in different scenarios, leaving the underlying shared memory MapReduce largely unmodified.
Analyzing and Accelerating Runtime Systems on Multicore Architecture

by
Devesh Tiwari

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering
Raleigh, North Carolina
2013

APPROVED BY:

James Tuck
Eric Rotenberg

Xiaosong Ma
Yan Solihin
Chair of Advisory Committee
DEDICATION

To my family,

especially to my brother, Abhishek Tiwari.
Biography

Devesh was born to Damodar and Malti Tiwari, and did his schooling in Allahabad, UP, India. He graduated with a bachelor's degree in Computer Science and Engineering from Indian Institute of Technology (IIT) Kanpur. Following that, he joined PhD program in Electrical and Computer Engineering Department at NC State University.

Devesh is broadly interested in computer systems performance modeling and analysis. He has published his research at various publication venues, including USENIX FAST, HotPower, IPDPS, HPCA and ISPASS. One of his papers was nominated for the best paper award at IPDPS 2012, and received best talk award at IBM's Student Workshop on Frontiers of Cloud Computing organized by IBM Thomas J. Watson research center. He has also spent brief research stints at Oak Ridge National Lab, National University of Singapore, and Seoul National University, South Korea. He is religiously passionate about teaching and developing technology for under-developed societies. Besides research and teaching, he likes to understand and discuss philosophical issues, e.g. justice, motivation, learning, policy-making, etc.

Next, Devesh will be joining Oak Ridge National Lab as a Research Scientist - Computer Systems Architect in Oak Ridge Leadership Computing Facility (OLCF) division. He will work on researching and designing the next generation of Supercomputers, to enable faster scientific discoveries.
Doctor: “Mrs. and Mr. Tiwari, as nerve system on the left part of his body is not functioning normally due to complications at birth. He will suffer from severe developmental delays. He is not likely to walk on his own and his intellectual abilities may be 5-10 years behind his age-group. It would be very hard for you both as parents to raise this physically challenged kid, in a society where physically disability is torturous both for the parents and the kid. You might want to seriously reconsider your intentions of keeping him with you.”

My parents: “When he was born, he was declared born-dead. It took a while for doctors to realize that he had life in him. He definitely seems to have something in destiny to survive against all odds. Now, that he has made this far (a little more than one year). We will make him proud and capable of whatever he has got.”

Doctor: “I am afraid he has not got enough, just the right side of the brain, hand and leg!”

My parents had the courage to act against doctor’s recommendation. But, definitely the act itself was never going to be easy. Just the thought that their third baby may never walk on his own and be 5-10 years mentally behind his age-group was frightening. As I am told, there were multiple nights when they would lock themselves up in the kitchen to cry over it. I give up imagining how much of trouble my elder sister and brother had to go through because of all this. But, my sister and brother made me feel that the whole world revolved around me, and were never insecure about their needs being overlooked. Every night, my family would gather after dinner, make me stand up and hope that I would walk on my own against all odds. They would spent countless number of hours on physical exercise, hoping one night things would change; while in the daytime “the cruel world” would tell them they had only two kids. One evening, the miracle indeed happened. My brother did what all of them were hoping for very long; I did take a few steps on my own. That day was so precious to them, that it has become more important than my birthday.

But that was only a start. As a “walking” child, I was far from being easy. I always had my own plans and ways about how things should be done, how much time should be devoted to my demands and so on. In hindsight, this personality trait proved to be beneficial as it has helped me immensely in my research, and I am glad that my parent encouraged and rightly-balanced this aptitude. But, definitely it was not easy on them. My siblings missed countless dinner/lunch because of my tantrums. Then, the fear of how would I do in school? My parents/siblings would have tears in their eyes as I received all the awards for academic excellence. I feel bad for my brother and sister that my academic achievements were celebrated so much that probably theirs got overlooked at times.

But, growing up was not easy for me outside my house. Thankfully, my family made sure how to keep my morale high. Thanks to them, I learned how to communicate effectively so that I could change others’ “traditional” perception quickly. How to focus on things that no one has traditionally discovered or thought of. Again, these qualities have helped me in my PhD tremendously. Also, I would like to thank my parents for putting our education above everything else, including their own comfort, professional/personal success, social life, etc.
It was really difficult for my family to first, accept God’s wish to have a physically challenged kid, and second, to raise me in a society where something like that was looked down with pity and fun (ironically). My family has put so many hours in physical exercise that people now hardly notice it, and think it’s a minor issue. Reality is that because of them I really feel proud of this challenge. In fact, sometimes we joke about what doctors had said earlier. We wonder how much more would I have achieved if everything was normal. I am afraid it would have been only less.

Papa, Mammi, Didi, and Bhaiyya: We all often fall short of telling what we felt about something. So, I wrote it like a short story. Because it really means a lot to me. As those who know me well – I am someone who cares the least about what people perceive about me. But, in this case, I want it to be known to everybody, including you all, that your efforts and sacrifices means a lot to me, more than anything else. I want to believe in second life because I want to be born exactly like this, raised exactly like this. Because I want to do it all over again with you four.

My family can thank God for all the courage they had on the way, I can thank them because they are what God is to everyone – knows everything, does all the miracles, but doesn’t expect a thing in return.

My parents have always been extremely selfless and encouraging; they showed it again when it came to my marriage. Arranged marriage is a big part of Indian culture, at least in my generation and my extended family. Just like any other mom in India, my mom had always dreamed of choosing “the one” for me. But, when I proposed my choice, She accepted it happily, and has treated my wife as if she is her own choice. Executing this marriage plan was a very hard task for my parents as this was the first not-arranged marriage in our family of over 300 people. They had to do a lot of work on their front to get everyone happily involved and socially accept it. It may seem easy, or even laughable, but it was impossible given the social and cultural boundaries of my family. My parents made it possible without letting me even feel the heat. Also, I have come to appreciate my mom’s analytical aptitude and her diverse interests/opinions more in last 6 years as we talked long hours on phone, of course, without dad’s expert interruptions. I have been truly enlightened more often than I expected. I would like to thank my extended family, my uncles, aunts and cousins, for their support.

My brother did all the professional and personal sacrifices so that I could live a better life. Somehow he had mastery of all the tricks needed to nurture a "challenged" kid like me, since even when he was very young. He sometimes let me win "crucial" cricket matches that we played in our backyard, just so my morale remained high. But, he beat me often and enough that I realized the value of quality and competition. He was punished in school because I used to hide his drawings the night before, but he didn’t utter a word, let alone scolding. Instead, he taught me how to color a drawing so that I could keep my copy for myself. He took me along in all the things he did; age difference was never apparent. Whether it was studying, sports or even clay-molding. He did what he always does the best: made me believe that I could do anything and to the perfection.

He became the most obedient son so that I could have some leeway. He did not go for graduate school, so that he could stay back in India (to look after family matters) and I could pursue PhD abroad. Till I got married, I didn’t do a single thing: flight/train tickets to my destination places just
showed up in my mail-box and all I had to do was be on-time for flights/trains. He had even backup
tickets in case I missed a connecting train and pulled up all nighters to make sure that I traveled
smoothly. During every visit, he hand-picked everything for next three months that I may need before
I left his house. I still keep his sweet and encouraging snail mails that he used to send, esp. for me,
during his IIT-Delhi days. They were in English so that I could get some exposure to communication
in English. My parents didn’t have enough money to get us educated in English-medium schools, but
to make-up for that he saved enough from his pocket-money to send me those English cards, letters,
magazines etc. I get goosebumps just thinking about his pure intent and how much he has done for
me.

Bhaiyya: This thesis is especially dedicated to you. You are the one I look upto. You are my only inspiration.
You are what Ram is to Hanuman.

Kaustubh and Bhavya, my sister’s son and daughter, get the most special mention for bringing joy
in our lives. It has been a delight to watch them grow, and entertain us more and more. I hope one day
they read this note, and value the importance of knowledge and gratitude. They will understand the
importance of sacrifice that their mom had to make, so that I could reach this point. I hope they will be
able to imagine her contributions in my success and their own. My sister (Mrs. Archana Mishra) has
been very supportive of my actions and decisions, sometimes even more than my parents. My sister’s
family has been extremely nice to me. My brother in-law made sure that my travel plan and stay was
smooth whenever I visited them. His mom has always sent me back with enough pickles of my choice.

Didi: I am humbled with your faith in me. I hope not to disappoint you ever. Kaustubh and Bhavya are as
close to my heart as yours, if not more.

I am thankful to PhD for giving me just the right amount of free time to woo the most lovely lady
in this world, Richa. I am sure my PhD is, in turn, thankful to Richa that she shaped it beautifully
afterward, with minimum delay in graduation. Our relationship has mostly been long-distance, ever
since its inception. When we started, I neither knew how to drive nor I had a car. Consequently, I
have mostly relied on Greyhound bus services to see her. So, a BIG thanks to Greyhound and Richa’s
old Mitsubishi Gallant car. While I may have contributed significantly toward Greyhound’s revenue,
it’s not a big surprise that Greyhound buses are still farthest from being comfortable, esp. given the
schedule for my bus: 3:00am to 9:00am journey, with 1 hour stop at 5:00am right when I would start
to get some sleep in my small seat, you can imagine what happens to my sleep cycle on the following
day (same with the return journey). I have done this continuously over past three years, almost every
other weekend with a graduate student’s salary. This is a small proof of what Richa means to me –
someone who used to become uncomfortable just outside his cozy couch/bed has found new ways to
enjoy Greyhound rides. I have done this because her pure, selfless, undemanding, loving, forgiving
and academically-curious nature has been the source of my happiness for past three years.

Richa has this charismatic personality that makes you believe that everything is perfect and things
can’t get any better than this. But more importantly, she does "a lot of" things behind the scene to make
sure everything is perfect, at least become much better. She can do this 24x7, even on her "relatively more gloomy research" days. But what amazes me the most: her desire to make a real-world impact in the field of human genetics, helping people with genetic disorders and making a difference in their lives. As she often keeps reiterating: her motivation to do science and earn a PhD degree is to make a real-world impact and spread the knowledge, and not just focus on gaining academic reputation by doing more "paper" research. This attitude has always inspired me and helped me remain grounded with my own research ideas and plans. Also, Richa’s parents, to their credit, have placed a lot of faith in me by marrying her in a significantly different culture than their own (Kashmiri). Neither did they know much about my culture/traditions nor about me at the time of marriage, but they did a commendable job in making sure that everything went smooth.

Gudia: I consider myself blessed to have such a partner who challenges me academically and comforts me personally. While you may not appreciate the technical details of this dissertation, it is as much yours as mine, if not more.

For a kid like me, a lot could have gone wrong in school, as often it does for challenged kids in India. But, my teachers were the game-changers; I always felt like I was their top-most priority and my progress mattered to them a lot. They made sure that I was not overly involved in activities where I would be uncomfortable or other kids will have a go at me. Very much like my family, they identified my strengths within and outside academics, and pushed my limits on those fronts. I excelled in academics with their teaching style, while they made sure I was equally good at other fronts where I had potential to excel, e.g. leadership roles, debate competitions, etc. I am highly indebted to my elementary school teachers: Govind Srivastava, Rama Pandey, Anand Srivastava, Vineeta Srivastava. I truly believe that had it not been these four teachers, I would have been no-where and would have restored to home-schooling. I fondly remember my insistence to go to Rama madam’s house and have snacks at my choice of time. Kripanand Tiwari, Ashutosh Kumar, Kakoli Bhattacharya, Santosh Srivastava and Rahul Shukla: I am thankful to all of you for the immense amount of faith you instilled in my abilities. I regret that I have not been able to know your whereabouts in last couple of years.

I am thankful to Harish Chandra Saxena (fondly known as Mamu Sir) for teaching my brother and me most of the concepts in Physics and Mathematics. If you could read it from heavens: we, as a whole family, are highly indebted to your teaching that has brought us so much enlightenment and satisfaction over past 15 years or more. I am thankful to teachers during my undergrad, Bikramjit Basu: for inspiring and teaching me how to do quality research; Debabrata Goswami: for guiding me through my final year; Rahul Verman, Amman Madan, Pinaki Gupta-Bhaya: for your excellent teaching skills. You all showed me that teaching is the most satisfying job and it has that magical power to influence anyone’s thinking process.

I am highly indebted to my PhD advisor, Yan Solihin. Advisor and advisee relationship is much like a martial relationship; of course, with a few notable differences that are not hard to guess. Similar to a martial relationship, adviser and advisee relationship works the best when there is just the right mix of personality similarities and differences. Yan’s advising style and my working style matched
extremely well. Yan had more belief in my capabilities than I did at times. He backed me up even when my papers were repeatedly getting rejected. As an advisee, you can’t ask for a better advisor on those afternoons. Yan was very quick to judge my personality and working style. He provided me a challenging environment to do research, but he set me free to pursue my ideas. He trained me diligently during my first project, but let me explore and navigate through most of my future projects.

Yan makes sure that his students do not have to worry about financial support, no matter how bad is the financial situation in the country or funding prospects with NSF. As he often says: "Your job is to do good research work, focus on that, excel there. I will take care of the rest. Do not distract yourself". While others may not realize, Yan puts a lot of effort towards that. University had stopped supporting me after five years, but I didn’t face any heat because Yan was there. I was teaching in that semester and working on a FAST conference submission along with a project related to my dissertation. Yan knew that with so much load, I wouldn’t be super productive. So, it may not have been the best idea for him to support me. But, he paid a big amount of money out-of-pocket (funding) to continue supporting me for next two semesters. He valued that I always wanted to teach and have teaching experience under my belt, and gain more research maturity with these projects. Yan’s commitment to my professional goals has been excellent.

Yan played "devil’s advocate role" in weekly meetings whenever I proposed new ideas, solutions, and paper drafts. Like a true mentor, he also steered me in the right direction whenever needed. In hindsight, I think I benefited greatly from this approach as he made sure that I rehearsed a real researcher’s life-at-work on multiple projects before I graduated. Thank you, Yan! I hope to emulate your research skills in future, that would be an achievement for me. I regret not being a "true" computer-architect PhD graduate as I did not work on hard-core architecture topics, but rather focused more on analytical performance modeling. Consequently, I didn’t submit what we learned from our projects to the traditional architecture conferences: ISCA, HPCA, MICRO, ASPLOS. I will always feel bad about that.

I learned a lot from other faculty at NC State. I was a rebellious student, a student everyone would turn to see if I had any question at the end of all the presentations and meetings; faculty here at NC State have welcomed that, and helped me become a better researcher. As I threw numerous random research ideas, they have given constructive suggestions and shown that they valued it. When I came to NC State, my English writing skills were pedestrian at the best, largely due to my Hindi-medium schooling. But I am proud to have improved that significantly and write decent papers on my own as shown by some of the paper reviews, I am thankful to my advisor, faculty at NC State and various other people (including Richa) who gave me constant feedback on my writing.

Eric Rotenberg’s passionate teaching style and his attention to detail will always inspire me. Eric’s enthusiastic nature is infectious. He is the most meticulous professor I have known, no matter how small the task may look, e.g., approving the graduate plan of work. James Tuck’s hands-on approach is truly unmatched. He is very quick to expose all the implementation-level devils, and yet not loose the high-level picture. James co-advised me during the initial years of my PhD, I am very thankful for some of the ideas he threw at me in our MMT project. His explanations are always clear and full
of geeky jokes. Whenever I have interacted with him, I couldn’t keep track of time. You come out of the meeting thinking: compiler will save us from all the challenges in multi-core era. Though he never says and claims that directly or indirectly. I am thankful to Xiaosong Ma for agreeing to serve on my PhD committee and collaborating on Active Flash project. Her suggestions have been very helpful and constructive. I would specially like to mention that her teaching style is very helpful, esp. for new graduate students. I took a parallel systems class with her, early in my graduate studies, and learned a lot from her lectures. She listens to her students very patiently. Even during our Active Flash project, she was always patient as we submitted a FAST paper while I had teaching commitments too. She understood my high teaching load that comes with first-time teaching responsibility, and was cooperative.

During my internship at Oak Ridge National Lab, I worked with Sudharshan Vazhkudai and Youngjae Kim. Sudharshan and Youngjae have been very helpful. They both have been very understanding and encouraging. They treated me like a colleague from day-one, and that had a very positive effect on my productivity. I liked my intern-experience so much that I regret not coming to ORNL for my earlier summers; Sudharshan always keeps saying the same. Sudharshan has been very helpful otherwise as well, he has helped me grow professionally and not just ask me to write a few papers. Sudharshan has been very honest and truthful about all the matters, this has only helped me have more faith in him. I couldn’t have asked for a better internship adviser. Thank you, Sudharshan! I am thankful to Simona Boboila and Peter Desnoyers for our productive collaboration on Active Flash project.

Like many other PhD students, initially I struggled in my research work, then started liking it, loved it and then become passionate about it. And right when I was almost certain that this was the pinnacle of joy in graduate school, I took on a new commitment: teaching a class. And, I was proved wrong. Teaching a class was the pinnacle of joy. I taught the same the course that I TA-ed in my first semester at NC State, I taught in the same classroom in which I took my first class at NC state. So, it was a conspired fitting finale. But, it would have not been possible without Greg Byrd and all the students. Thank you, Cheryl Alderman, for taking care of all the administrative issues for my other section (UNC Ashville distance education section). Thank you all the students for encouraging feedback and pleasant experience. You made me feel a void in my busy schedule on Tuesday and Thursday from 1250 to 1405 hrs in the following semester in which I had just enough time to write the dissertation and hunt for a job. I secretly regret not being able to throw a party after the final exam.

Now, the time and context is appropriate to thank the most influential teacher during my tenure at NC State, Greg Byrd. He mentored me during my full-time teaching assignment. He has completely mesmerized me during all his conversations. He hardly needs any testimony for his teaching skills – University has done a good job at that. So, I will not even talk about that; that would be like adding a drop in the ocean. His commitment for any job is exceptional. He stayed late helping me so that I could become a better teacher. He is a prophet when it comes to teaching. He told me with this way of explaining the material, they will do this kind of mistake on exams – and this is what exactly happened. But, the quality that I admire the most in him is "attention-to-fairness". He always makes sure that his policies are fair to each student, and every student has equal opportunity. I have
interacted with him more than one can imagine to interact with a teaching mentor, simply because he was always ready to help. I learned the art of teaching and designing exam papers from him. I would feel satisfied, if I could emulate even a bit of him in my future teaching endeavors. Thank you! I regret making him proctor in my two exams, as he admits proctoring an exam is as boring as it can get.

While one may not realize easily, helping a PhD student to graduate takes a toll on departmental and university administrative staff. We, graduate students, "schedule" all the administrative tasks when nothing is going right in our research. When the whole world looks gloomy, we show up at graduate office and want everything to be done in minutes. I was the farthest from exception in this case. Not to mention, it takes only single missing digit on I-94 card of a student with F1-visa to raise "red alert" everywhere in the system, and risk his "legal" employment status in USA. It's administrative staff who catch and resolve issues like these for a F1-student for no fault or responsibility of their own (a bug in immigration officer's computer software in this case). Sandy Bronson, Elaine Hardin, Linda Fontes, Kendal Del Rio, Jennifer Raab, Katy Wilson, Terri Martin-Moss and OIS staff: I can not thank you all enough. I regret not being able to prove how much I valued your assistance.

During my six years, I have put countless requests to ECE Tech help. My work was real-system oriented and involved experimenting with many system libraries and tools, that had several system/admin issues before I could get them working. I could sleep in night because I knew ECE Tech help desk keeps our servers running. Dan Green, Mike Belangia and Manpreet Singh: Thanks for your super-fast and helpful responses for all of the weirdest requests. I regret not being able to say thank you enough.

My PhD thesis acknowledgment is incomplete without mentioning Cricket and Indian music. Cricket and Indian music has greatly influenced my way of thinking and life over time. I have followed my role models in these fields frantically, got inspired from their performances, learned from their lifestyles and mistakes etc.

I am thankful to "truly timeless" and "the God of Cricket" Sachin Tendulkar for making three (probably four?) generations of India to "dare to dream and believe". I am thankful to Sachin Tendulkar for showing us the value of persistence — that you may be the best batsman in the world, but to win the world-championship as a team of eleven, you may have to wait for "twenty two" years, play six world-cup tournaments and keep giving your best without blaming how poorly your team members may be doing at crucial times.

When Sachin Tendulkar walks out to bat, one billion people and thirty three million Gods in India switch OFF their lives, and switch ON their television sets. Whenever I feel pressured because I am supposed to meet certain expectations set by others, I remind myself that these expectation pale away in comparison to those of one billion people that Sachin has faced every fifth day on an average, for last twenty five years. And, the pressure inside my mind melts away in nanoseconds.

I am thankful to Rahul Dravid for making me realize that you may be the second best in the country, only second to Sachin Tendulkar, but in a crazy country like India you may not get even 10% of the popularity share or cult-following compared to No. 1. But you have to continue giving your best without expecting much in return. Rahul Dravid batted at No. 3 position, so when he got out,
people were not sad or clapped for his efforts, but instead clapped for Sachin Tendulkar who used walked in to bat next. It is tough being Dravid. I am thankful to Amol Muzumdar (even many cricket fans do not know him, please look up) for telling me that you can be as good as the best, but that are few things that are out of your control. I am thankful to MS Dhoni for showing that calm mind is the most dangerous mind. This piece from Siddhartha Vaidyanathan seems to perfectly summarize him: "There was a time when Indian fans turned off the TV when Tendulkar got out (and Dhoni too has admitted to having done the same when he watched the 2003 World Cup final). But the thinking these days seems to have been turned on its head, almost to a point where fans tune into a game when their captain walks in."

I am thankful to AR Rahman for all the music that he has produced. He has never disappointed, from Roja to Raanjhanaa. As he says, creative works are God’s impression on you, so it’s not really your work. This way we can remain detached from both success and failure, and only focus on doing good work. I feel bad that he got two Oscar prizes for Slumdog Millionaire – a work far from the best by his standards. I wonder how much would he have achieved if he was born in USA? I have also taken a lot of inspiration from Mani Ratnam’s movies, who should have won Oscar by now as well. They inspire me because I always tell myself that music and movie directors have only one chance, while I have multiple chances to submit the same work. A music album once rejected is always rejected, so I have got a much easier deal than Rahman or Mani Ratnam.

One unique thing about my PhD is that I didn’t have a car to drive during the entire time, and I ate outside everyday as I never enjoyed/cared enough to start cooking. I am thankful for this decision, because looking back I think it made me manage my time more wisely and plan for the adverse in advance. So, thanks to all the restaurants near-by and sorry, Gas stations!

My six years at NC State have been wonderful largely because of the wonderful company of friends. No order is perfect, but mentioning them in the order I got introduced to them is the least controversial, so I will go with that for a change. Siddhartha Chabbra’s crystal clear concepts were very helpful, esp. in my early years when my knowledge in architecture area was pedestrian. Also, he helped me a lot in pacing my research work, discussing ideas, preparing for meetings, etc. I am thankful to Lavanya Palakshareddy, esp. for making delicious food during my first year at NC State. Company of Salil Pant and Alison Holmes Pant was arguably the greatest source of entertainment outside work. Every evening, Salil dragged me to Cup-A-Joe coffee shop, where he paid $1.25 for regular coffee and kept getting refills, while I paid $3.65 for my signature drink: “tall double vanilla latte”. Perhaps, Allison’s witty nature was enough to get her free drinks? I still feel guilty about our Washington DC trip, where they drove me to a conference and while coming back my only job was to keep them awake while driving as they were tired. I crashed in the back seat, despite two red-bulls.

Sandeep Navada and Manisha Agarwal introduced me to love of my life, Richa. Unarguably, this magic act beats every thing that I did or got help for in graduate school. Unfortunately, this also dwarfs other numerous things that Sandeep and Manisha did for me, including inviting me over for dinner during my early PhD years because I didn’t use to cook (Manisha), helping me overcome some of the rough paper rejections/conference deadlines (Sandeep), offering me to drive his car on the interstate...
when I hardly knew how to drive (Sandeep), etc. Rajeshwar Vanka picked up where Salil and Allison left, always up for a coffee and philosophical chat; no matter how much work was at hand, he never said "NO" to coffee outings. This was as impressive as his generous dinner invitations almost every second day during our last year at NC State. He graduated, left for Boston; then, I felt the void and scheduled my defense date.

Elliot Forbes has been partner-in-crime for my teaching endeavors. He has been kind to enlighten me with his experience. Saurabh Gupta is the most humble person I have met at NC State. He also has this gifted quality of quickly coming up with solutions to the toughest task-scheduling problems in day-to-day life. Also, given his aptitude with mechanical systems, I often think the world has missed on a fine mechanical engineer, since his academic training is in electrical engineering – which he does not seem to be making much use of, anyway. Aditya Deorha’s Colgate-smile is infectious, though I have picked on him more often than not. To his credit, he has been the most sporty character against all odds (my sarcastic comments).

I am thankful to Richa’s adviser, Anand Srivastava and her wife, Fatima, for throwing the best reception party when we returned back from India after our marriage. I am thankful to Alka Chaubey for enlightening me with her own experiences and caring deeply about Richa and me. Thank you, Gaurav, Shruti and Greenwood Fan club, for inviting us to dinner so frequently. I was awestruck by Julianne Collins’s passion for research in human genetics as she was battling with cancer for her life. She lost, but she inspired many!

I am thankful to many other for keeping me sane over last six years (in no particular order): Avik Juneja, Sahil Sabharwal, Siddhartha Sharma, Arpit Gupta, Rahul Ramasubramanian, Raj, Sabina Grover, Sangyeol Kim, Andrew and William Spencer, Amit Gupta, Gaurav Bawa, Pradeep, Aakarshit Dhingra, Adithya, Ankit Jain, Fang Liu, Brian Rogers, Xiaowei Jiang, Yi Yang, Sanghoon Lee, Naser, Abhik Sarkar, Amro Awad and Rami Al-Sheikh. I truly cherish your friendship and help with numerous things over the course of last six years.

It was an honor to notice that there were no seats left in the conference room during my defense, I have never witnessed or heard of anything similar to that in any defense at NC State in last six years. So, thank you for your exceptional support!

Just like scientific publications, no matter how hard you try, some reviewer will always find out one missing reference. I am sure my acknowledgment section is no exception to this, despite having no page-limits. Please let me know how can I make-up, if I missed you here. Also, the "real" dissertation has not started yet. So, please, read on......
# Table of Contents

## List of Tables

<table>
<thead>
<tr>
<th>List of Tables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xv</td>
</tr>
</tbody>
</table>

## List of Figures

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xvi</td>
</tr>
</tbody>
</table>

## Chapter 1 Introduction

1. Exploiting Fine-Grained Parallelism in Dynamic Memory Management ............ 3
2. Modeling and Analyzing Key Performance Factors of Shared Memory MapReduce ... 6
3. Reusing Computation in an In-Memory MapReduce System ............................ 8
4. Organization of the Dissertation ...................................................... 9

## Chapter 2 Exploiting Fine-Grained Parallelism in Dynamic Memory Management

1. MMT Approach and Design ................................................................. 10
2. Experimental Methodology ............................................................... 20
3. Related Work ....................................................................................... 28

## Chapter 3 Modeling and Analyzing Key Performance Factors of Shared-Memory MapReduce

1. Background ......................................................................................... 30
2. Analytical Model Formulation ............................................................ 32
3. Model-Driven Study ............................................................................ 39
4. Experimental Environment ................................................................... 43
5. Experimental Evaluation .................................................................... 48
6. Application Classification ................................................................... 52
7. Related Work ...................................................................................... 53

## Chapter 4 MapReuse: Reusing Computation in an In-Memory MapReduce System

1. Background ......................................................................................... 54
2. MapReduce Programming Model ............................................................ 54
4.2 MapReuse Design

4.2.1 Design Overview

4.2.2 Granularity Choices

4.2.3 Computing the Differential Input

4.2.4 Determining the Deleted Input

4.2.5 The Merging Engine

4.3 Evaluation and Analysis

4.3.1 Modeling the MapReuse System

4.3.2 Model-Driven Results

4.3.3 Experimental Results

4.4 Related Work

Chapter 5 Conclusion

References
# LIST OF TABLES

Table 2.1 Latency of Pthread vs. minimalist synchronization primitives, refer Section 2.2 for machine configuration details. .................................................. 18  
Table 2.2 The benchmarks used for evaluation ................................................. 21  
Table 3.1 Parameters used in the model. ......................................................... 33  
Table 3.2 Comparing asymptotic complexity of various phases. ....................... 39  
Table 4.1 Parameters used in the model. ......................................................... 66
LIST OF FIGURES

Figure 1.1 Fraction of execution time spent in dynamic memory management routines, on
an Intel Core2 Quad-Core platform. .......................................................... 3
Figure 1.2 Latencies of synchronization primitives vs. average memory allocation and time
between consecutive allocation and deallocation requests. .......................... 5
Figure 1.3 Execution time of word count for various inputs, normalized to input A. ........... 6
Figure 1.4 Speedups for word count normalized to the single thread case for each respective
input ................................................................................................. 7

Figure 2.1 Traditional Memory Management (a), Fork-join MMT (b), versus Client-Server
MMT approach (c). .............................................................................. 11
Figure 2.2 Protocol for MMT to perform synchronous allocation and deallocation (a), and
for MMT to perform synchronous allocation but asynchronous deallocation (b). ... 12
Figure 2.3 Preallocation and bulk deallocation (a) conservative preallocation (b) aggressive
preallocation. .................................................................................. 16
Figure 2.4 Illustration and pseudo-code of our lockless protocol. ................................. 19
Figure 2.5 The performance of basic MMT design. .................................................... 22
Figure 2.6 Performance of task partitioning and synchronization implementation. .......... 24
Figure 2.7 Performance of the final MMT design on Doug Lea’s allocator (a), and PHKmalloc
allocator (b). ................................................................................ 25
Figure 2.8 Last level Cache and TLB Misses (Base: Doug Lea allocator). ................. 26
Figure 2.9 Security overhead using MMT approach (Base: PHKmalloc allocator) ........ 27

Figure 3.1 Overall design of shared-memory MapReduce runtime system. ...................... 31
Figure 3.2 Markov model for map-output time. ....................................................... 34
Figure 3.3 Normalized map-output time for 10^6 intermediate output pairs in Phoenix and
Metis (a), UKO vs. SKO in Phoenix (b) and Metis (c), % difference of map-output
time of SKO vs. UKO (d), reduce-comp time for various number of threads and
buckets (e), and Map vs. Reduce time under SKO with 1 million pairs, 250,000
distinct keys, and 8 threads (f). .......................................................... 41
Figure 3.4 Empirically-derived values of w_1, w_2, w_3, w_4, and w_5 (map-phase). ..... 45
Figure 3.5 Empirically-derived values of w_6, w_7, and w_8 (reduce phase). ................. 46
Figure 3.6 Map and Reduce time for randomly generated word-count document with 20
million pairs .................................................................................. 47
Figure 3.7 Map-output time for WC with varying number of keys for Phoenix vs. Metis (a),
and UKO vs. SKO on Phoenix (b) and Metis (c). ........................................ 48
Figure 3.8 Map-output time for various number of threads and hash buckets Phoenix (a),
and Metis (b) ................................................................................ 49
Figure 3.9 Reduce phase execution time for various number of threads and hash buckets
Phoenix (a), and Metis (b) ................................................................... 50
Figure 3.10 Breakdown of execution time with various number of buckets and 8 threads for
Phoenix (a), and Metis(b) ............................................................... 51
Figure 3.11 Classification of MapReduce Applications. Not drawn to scale. ................. 52

Figure 4.1 Overall design of in-memory MapReduce system. ........................................ 55
Figure 4.2 High level diagram of MapReuse. ......................................................... 57
Figure 4.3 A potential problem with fixed size chunking. ......................................... 58
Figure 4.4 Detecting identical map tasks and computing differential input (left), and detect-
ing unmatched map tasks (right). ......................................................... 59
Figure 4.5 Weight values of MapReuse specific parameters. ........................................... 68
Figure 4.6 How the fraction of difference in inputs affect run time for 64K map tasks (a) vs. for 256K tasks (b). ................................................................. 69
Figure 4.7 How the fraction of difference in inputs affect execution time SKO (a) vs. UKO (b). 69
Figure 4.8 The effect of the number of MapReuse threads on performance under uniform key distribution for P/D=1 (a) and for P/D=1000 (b). ................................. 70
Figure 4.9 The effect of the number of MapReuse threads on performance under skewed key distribution for P/D=1 (a) and for P/D=1000 (b). ................................. 71
Figure 4.10 Normalized execution time as affected by various fraction of changes to the original input. ............................................................... 72
Figure 4.11 Speedups of MapReuse with various number of threads vs. IMMR running with eight threads. ............................................................... 73
Figure 4.12 Speedups of MapReuse with various number of threads vs. IMMR running with four threads. ............................................................... 74
Figure 4.13 Breakdown of execution time for Freq. Count input A (left) and B (right), for IMMR using 8 threads and MapReuse using 1, 2, 4, and 8 threads, assuming 25% fraction of input difference. ........................................... 75
Figure 4.14 Breakdown of MapReuse overheads, $P = 10^7$, P/D ratio = 1000, when running with one thread (a) and 8 threads (b), normalized to IMMR running with one and 8 threads, respectively. ........................................... 76
Introduction

Technology scaling has made multicore architectures commercially prevalent [10]. However, exploiting multicore parallelism for performance remains challenging for programmers. This is primarily because programmers often find managing concurrency a daunting task due to data races, deadlocks and other side effects [23, 29, 60]. Therefore, there is a need for solutions that can exploit the computational power of multicore platform without burdening the programmers with concurrency management. We believe that runtime systems are a part of the solution. Runtime systems can take advantage of multicore architecture while hiding the complexity of concurrency management from programmers.

However, there are several challenges in making the runtime systems efficient and easy to use on multicore platforms. First, traditionally runtime systems are sequential in nature and hence, they cannot readily take advantage of multicore architecture. Second, parallelizing traditionally sequential runtime systems may break the legacy code, require compiler hints and changes to the source code or hardware. Third, programmers may not be aware of performance bottlenecks of a runtime system and hence, may gain only suboptimal performance. This dissertation aims to address some of these challenges. In this dissertation, we use two specific runtime systems as a vehicle to demonstrate the value of our proposed techniques. Two runtime systems used in this study are: 1) dynamic memory management library [71, 74], 2) shared memory MapReduce runtime library for parallel programming [41, 51].

The first part of this dissertation aims at reducing the overhead of dynamic memory management runtime system by parallelizing it on a multicore architecture. Dynamic memory management runtime system is responsible for memory management tasks such as `malloc()` and `free()`. However, traditionally this runtime library [71, 74] executes sequentially with respect to the application. Hence, it has not taken advantage of the multicore platform. Moreover, tasks such as `malloc` and `free` are often very small, and hence, executing them in parallel on a separate core may even degrade the performance due to high communication and synchronization cost. In this study, we use a dynamic memory management runtime system as an example to show how to efficiently exploit the fine-grained parallelism in dynamic memory management. We also show the design of such a system that exploits the fine-grained parallelism in the runtime library while remaining transparent to the
application and memory allocation library without modifying any of those. Section 1.1 provides an overview of this work [19, 18].

The second and third part of this dissertation focus on analytical performance modeling and optimizing shared memory MapReduce runtime library. Shared memory MapReduce runtime library [41, 51] is an effort for making parallel programming easier and more deterministic. Shared memory MapReduce or in-memory MapReduce (IMMR) runtime systems\(^1\) allow programmers to express parallelism at a higher level, and provide automatic management of concurrency. Programmers are attracted to MapReduce due to increased programmer productivity, high level programming abstraction, the lack of data races and deadlocks, automatic management of threads, synchronization, and fault tolerance. However, due to the high level of abstraction, programmers are often not aware of performance bottlenecks in the runtime systems. Hence, they may achieve only suboptimal performance gains. Secondly, runtime system designers may not know the key factors that affect performance for different workloads. Therefore, there is a need for solutions that can provide deeper understanding of the key performance factors that affect these runtime systems.

Therefore, in the second part of this dissertation, we build a new analytical model to analyze key performance factors of shared memory MapReduce runtime libraries. Section 1.2 provides an overview of this work [17]. In the third part of this dissertation, we design and implement a new in-memory MapReduce runtime system that exploits an observation that programs are often run multiple times with either identical or slightly-changed input [1, 33], which creates a significant opportunity for computation reuse. We propose a novel technique for computation reuse in IMMR, which we refer to as MapReuse. MapReuse detects input similarity by comparing their signatures. It skips recomputing output from a repeated portion of the input, computes output from a new portion of input, and removes output that corresponds to a deleted portion of the input. MapReuse is built on top of an existing IMMR system, leaving it largely unmodified. We found that MapReuse significantly speeds up IMMR, even when the new input differs by 25% compared to the original input. Section 1.3 provides an overview of this study.

While the two case studies we selected (dynamic memory management and shared memory MapReduce) do not represent the entire spectrum of runtime systems running on the multicore platform, some of the issues that we investigate are likely relevant to other runtime systems. For example, issues we face in offloading memory management to a separate thread include the high cost of thread synchronization, communication overheads, and small task granularities. Such issues are likely present in other systems when programmers want to offload small and frequently executed tasks from the main application to a separate thread through a runtime library. Another example is the analytical modeling that we perform for MapReduce. While the model is specific to shared memory MapReduce runtime systems, the approach for modeling that starts from identifying key factors and their weights, and expressing their relationship mathematically, then deriving the values of those weights through micro-benchmarking, is likely applicable for other runtime systems. While we have not mapped how exactly our approach can be generalized for other runtime systems, we believe that this dissertation presents interesting case studies that programmers and architects can find useful.

\(^1\)We use shared memory MapReduce runtime system and in-memory MapReduce (IMMR) runtime system interchangeably in this dissertation.
1.1 Exploiting Fine-Grained Parallelism in Dynamic Memory Management

Dynamic memory management is one of the most expensive but ubiquitous operations in many C/C++ applications. Many applications, such as factorization algorithms, language processing and translation, object-oriented databases, object-oriented robotics, dataflow constraint solvers, and minimum spanning tree, are highly heap allocation intensive. Previous studies show that some C programs spend up to one third of their execution time in dynamic memory management routines such as malloc and free [7, 75, 86, 89]. While object oriented programming style improves software reusability and extensibility, it has also made applications more heap allocation intensive. For example, some studies reported that C++ programs may use significantly more dynamic memory allocations compared to C programs [12, 16]. Figure 1.1 shows that for heap intensive C/C++ benchmarks that we study, on average 30% of their execution time is spent in dynamic memory management when using GNU C library.

![Chart showing fraction of execution time spent in dynamic memory management routines](image)

Figure 1.1: Fraction of execution time spent in dynamic memory management routines, on an Intel Core2 Quad-Core platform.

Moreover, the time spent in dynamic memory management routines will be even higher if we want to perform safe dynamic memory allocations and deallocations. Adding sanity checks to dynamic memory management routines, such as detecting a deallocation of an invalid pointer and double deallocations, can uncover many dynamic memory management related errors and detect security vulnerabilities [3, 4, 11]. However, such checks incur a high runtime overhead. For example, using the...
widely used PHKmalloc allocator [71] with extra sanity checks, the benchmarks we tested suffer from an average of 21% execution time overhead.

At the same time, continuing progress in process technology enables the integration of multiple processor cores on a single chip, creating a platform that allows threads of an application to run in parallel on different cores. Considering the high overheads of dynamic memory management, it is important to explore the possibility of reducing dynamic memory management cost on a multicore architecture, especially for sequential applications which cannot easily benefit from the multicore architecture otherwise.

Therefore, we would like to explore a new approach to hide the high cost of dynamic memory management by offloading all dynamic memory allocation and deallocation requests to a dedicated memory management thread (MMT) thread.

However, translating the potential parallelism of the MMT approach into performance improvement is challenging. First, memory allocation and deallocation are very fine-grained tasks, compared to thread synchronization primitives available in current systems. Figure 1.2 shows the latency of a pair of Posix thread lock acquisition and release, and a pair of semaphore signal and wait, compared to the average latency for memory allocation, and average time between two consecutive allocation/deallocation requests, for various benchmarks running on an Intel Core 2 Quad system. The figure shows that the latency of lock (uncontended) is roughly the same as the time to perform memory allocation, while the latency of a semaphore is roughly twice as much. Considering that one communication roundtrip between two threads typically involves a pair of semaphores and possibly lock acquisition and release, the overhead of synchronization is multiple times higher than the actual work (memory allocation and deallocation). To make it worse, synchronization latencies are also higher than the time between two consecutive allocation/deallocation requests. Thus, unless these overheads can be reduced significantly, the parallelism from MMT will not offset the overheads to produce a net positive performance benefit.

Assuming we can overcome the performance challenge, the second challenge is whether the application and memory allocation library can benefit from MMT transparently, i.e. without requiring source code modification, hints from programmer, compiler support, library modification, or special hardware support. Achieving such transparency is attractive since the application and the memory allocation library do not need to be redesigned or reprogrammed to enjoy parallelism that MMT provides on a multicore architecture. MMT can be applied, updated, patched, or changed without affecting the application or library code.

Thus, the goal of this work is to explore how MMT approach can (1) provide performance benefits by exploiting parallelism between main application thread and the dedicated memory management thread, and overcoming the fine-grain task granularity and synchronization overheads, and (2) deliver the speedups transparently from the application and memory management library. This work makes the following contributions:

First, we propose a new approach (MMT) to accelerate dynamic memory management for sequential applications on multicore architecture, which can speed up heap-intensive benchmarks through several key techniques: speculative memory allocation, bulk memory allocation and deallocation, and eliminating the use of semaphores and locks. Compared to an average slowdown of $2.7 \times$ without
using these techniques, the techniques combined together deliver a speedup ratio of $1.19\times$, for a set of heap allocation intensive benchmarks running on an Intel Core 2 Quad system.

Secondly, we show that MMT can be designed to be transparent to the application and memory allocation library without modifying any of those. Furthermore, we design MMT to be agnostic to the underlying dynamic memory management library, i.e. MMT does not exploit algorithm or data-structures used by the underlying dynamic memory management library to gain performance improvements. To demonstrate these goals, we use both memory allocation library implemented in GNU C and PHKmalloc library implemented in FreeBSD systems [71]. These two libraries are unrelated, and use significantly different algorithms and data structures. For the same set of benchmarks, MMT delivers an average speedup of $1.19\times$ for both GNU C and PHKmalloc libraries.

Thirdly, we enable security checks in PHKmalloc library that has been useful in detecting many errors related to dynamically allocated objects in many applications like fsck, ypserv, cvs, mountd, and inetd, [82]. Without MMT, these checks incur on average 21% run time overheads. MMT approach reduces such high overheads to just 1% on average, with no overhead in most of the benchmarks.

Finally, in the process of designing an efficient MMT approach, we balance many different tradeoffs and design choices that have strong implications on the MMT design. Due to the fine-grain task nature of MMT, we believe that such discussion may be useful for other researchers in considering what design issues are important for exploiting fine-grain function parallelism. For example, we show how to design contention-resistant data structures and mechanisms, and avoid using locks while exploiting fine-grain task parallelism, where even “lightweight” Posix thread synchronization mechanism might be too expensive.

We believe that this work is a step towards finding useful scenarios in which sequential applications can be sped up through offloading non-essential and meta-computation to separate threads.
1.2 Modeling and Analyzing Key Performance Factors of Shared Memory MapReduce

MapReduce, originally proposed by Google for cluster computing, has enjoyed a widespread commercial adoption by various companies, including Amazon, Facebook, and Yahoo [6, 20, 69]. With MapReduce runtime systems, programmers provide two functions: map() and reduce(), which are executed as separate phases. During the map phase, the map function accepts input data to produce intermediate output in the form of a list of <key,value> pairs. In the reduce phase, the list of key-value pairs are read by the reduce function, and all values having the same key are aggregated. Programmers are attracted to MapReduce programming model due to increased programmer productivity, low barrier to entry, high level programming abstraction, the lack of data races and deadlocks, automatic management of threads, synchronization, and fault tolerance. Recently, IMMR has been shown to be effective in alleviating the I/O bottleneck in peta-scale scientific data analysis and management [64].

![Figure 1.3: Execution time of word count for various inputs, normalized to input A.](image)

There are two versions of MapReduce runtime systems. Cluster- or disk-based MapReduce (e.g. Hadoop) relies on a file system to store the intermediate output. Shared memory MapReduce (e.g. Phoenix), on the other hand, relies on storing the intermediate output in the memory shared by all threads. Shared memory MapReduce is more appropriate for computation that fits in the main memory, but due to avoiding I/O, it outperforms disk-based implementation (e.g. Hadoop) signifi-
Figure 1.4: Speedups for word count normalized to the single thread case for each respective input.

Significantly [26, 36]. Therefore, it is possible that future MapReduce programs may use a hybrid approach, e.g. in-memory version in a shared memory node, and disk-based version across shared memory nodes. This work focuses on shared memory MapReduce.

Shared memory MapReduce’s scalability has been demonstrated for a range of applications [41, 51]. However, the general understanding of what factors affect performance and what performance bottlenecks exist for shared memory MapReduce programs is still at a rudimentary level. Little is understood beyond an observation that computation that can naturally be expressed using key-value pairs tends to perform well [28, 51]. Indeed, there are factors that are not identified yet that significantly affect performance.

Figure 1.3 shows the execution time of word count (WC) application for three equally-sized inputs on an Intel Xeon system using Metis [41], a shared memory MapReduce system. The input text documents contain identical words (Input A), all distinct words (Input B), and mixed (Input C). The figure shows that the execution time of the application hugely varies depending on the input content. WC takes up to $67 \times$ longer with Input B compared to with Input A. Figure 1.4 shows the speedup of WC for up to eight threads, normalized to each input’s single thread case. The figure shows that scalability of the application is also affected by the input content.

These observations demand a deeper understanding into the performance characteristics of MapReduce. Therefore, the goal of this work is to identify and analyze key performance factors of shared memory MapReduce. Developing such an understanding is challenging because the intricate interaction between application’s characteristics, the input type and content, number of threads, MapReduce runtime system implementation, and machine characteristics. However, a deeper understanding is necessary if programmers are to write efficient code with shared memory MapReduce.
To reach our goal, we develop a simple yet powerful parametric analytical performance model to establish mathematical relationship among key performance factors of shared memory MapReduce. The model relies on a mixture of Markov process and algorithm complexity analysis, augmented with empirically-derived parameters. The model is validated against results from real applications running on a real system. To the best of our knowledge, this is the first analytical model for understanding shared memory MapReduce performance. Our study discovers several important findings, among them are:

- Shared memory MapReduce performance is highly affected by the number of intermediate output key-value pairs relative to the number of distinct keys. Our model also discovers that the order in which keys are encountered during the Map phase affects execution time significantly. Inputs of which the content differs in these metrics produce significantly different performance, captured quantitatively by our model.

- Execution time of reduce phase is highly affected by task queue overhead, and this overhead may offset the benefit from increasing the number of reduce worker threads.

- The size and choice of data structure for storing intermediate output affects map and reduce phases differently, and sometimes affecting map phase positively while affecting reduce phase negatively. Our model enables us to quantify this interesting behavior. Furthermore, we show that different input content favors different choices for data structure and algorithms in the map phase, sometimes favoring non-intuitive choices.

Our model quantifies the relative contribution of different components/bottlenecks in the map and reduce phases, and how they are affected by program characteristics and input content. Finally, we propose an application classification framework that can be used to predict likely performance bottlenecks of applications. Given the knowledge about the program and input characteristics, such bottleneck analysis can be performed even prior to coding in some cases.

With the increasing prevalence of multicore systems and increasing adoption of MapReduce programming model, understanding the key factors affecting shared memory MapReduce will become increasingly critical for programmers, performance tuners, and system designers. We believe that our study is an important contribution towards that goal.

### 1.3 Reusing Computation in an In-Memory MapReduce System

Recently, it has been observed that in data-intensive environment, programs are often run multiple times with either identical or slightly-changed input [1]. In a production cluster, researchers have observed that up to 30% of queries are repeated over a period of one month [33]. The same study also shows that identical queries made on the same day use the same input data. Other studies have made a similar observation, and stated that there is a significant redundant computation due to input data reuse [30, 2], where between 20%–40% MapReduce computation instances are redundant [30]. To illustrate one scenario where redundant computation is common, imagine a job that runs everyday
to find the most frequently read news over the past week. Such a job would share 85% of the input from the previous job. There exists a tremendous opportunity to improve performance from avoiding redundant computation.

Recognizing the opportunity, researchers have proposed techniques to reuse computation in disk-based MapReduce [46, 34, 22, 47, 40, 45]. However, we are not aware of any study to apply computation reuse in IMMR. The challenges to reuse computation in IMMR are fundamentally different than in disk-based MapReduce. IMMR deals with the main memory, hence any metadata required for computation reuse must be implemented in data structures, rather than the file system. The main memory is also space-restricted, thus any computation reuse in IMMR must be space efficient. Finally, the latency for detecting input similarity is fully exposed in IMMR, whereas in disk-based MapReduce it can be hidden and overlapped with the (much higher) disk access time, making it imperative to implement very efficient input similarity detection in IMMR.

In this dissertation, we propose a novel technique for computation reuse in IMMR, which we refer to as MapReuse. MapReuse detects input similarity at the granularity of map tasks by comparing their signatures. It skips re-computing output from repeated map tasks, computes output from new map tasks, and removes output that corresponds to deleted map tasks. MapReuse is built on top of an existing IMMR system, leaving it largely unmodified. It does not modify the APIs, hence programmers’ abstraction is unchanged. This makes it possible to change the IMMR system independently of MapReuse, and allows MapReuse to be optimized transparently.

We evaluate MapReuse in two ways. We use an analytical model to understand key factors affecting MapReuse performance, and to stress test MapReuse under a wide range of scenarios. We use empirical evaluation on a multi-core system using benchmark applications such as frequency counting, regular expression matching, and fingerprinting. We found that MapReuse significantly speeds up IMMR. The harmonic mean of speedup ratios of MapReuse over IMMR running with eight threads are 1.3×, 1.7×, 2.0×, and 2.2×, on 1, 2, 4, 8 threads, respectively, assuming the new input differs by 5% from the original input. Even when the new input differs by 25%, MapReduce speeds up IMMR by 1.3× on 8 threads.

1.4 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 discusses MMT approach for exploiting fine-grained parallelism in dynamic memory management. Specifically, we discuss MMT design issues in detail, and evaluate different designs. Chapter 3 discusses a new analytical model for analyzing key performance factors of shared memory MapReduce. In Chapter 3, we use our analytical model to quantify the relative contribution of performance bottlenecks for shared memory MapReduce. In Chapter 4, we describe the design and implementation of our in-memory MapReuse runtime system. Finally, Chapter 5 concludes these three studies and provides the summary of our findings based on this research.
Chapter 2

Exploiting Fine-Grained Parallelism in Dynamic Memory Management

This chapter is organized as follows: Section 2.1 discusses MMT design issues. Section 2.2 gives details about our experimental methodology. Section 2.2 presents the evaluation results. Section 2.3 describes related work.

2.1 MMT Approach and Design

In this section, we will describe our MMT approach, its basic design, and how it achieves the transparency goal. Then, we discuss how we exploit fine-grained parallelism between main application thread and MMT by performing speculative memory management and delaying some memory management tasks to perform them during idle MMT cycles, hiding the cost of such requests from the main application thread, while keeping the MMT design contention resistant and completely agnostic to underlying memory management library implementation. Later, we illustrate how to effectively reduce communication and synchronization costs while offloading fine-grain tasks.

2.1.1 MMT Approach

Traditional memory management lets the application directly interact with memory management library by calling functions such as `malloc` and `free` (Figure 2.1(a)). One way to offload these functions onto a separate thread is by using a fork-join parallelism model in which the application dispatches work to a memory management library thread each time a memory allocation or deallocation request occurs in the application (fork) and the library thread returns the result at the end of processing the request (join), as illustrated in Figure 2.1(b). However, a fork-join approach requires changes to the memory allocation library and the application as they have to be able to communicate and synchronize their progress, as well as it goes against the transparency goal of MMT. In addition, the approach is inflexible since the order and frequency of when the allocation and deallocation requests
are dispatched cannot be changed. Furthermore, the library thread can only be reactive, and perform allocation and deallocation only when requested, instead of being proactive.

(a) Traditional MM:

(b) Fork–Join Approach:

(c) Client–Server Approach:

![Diagram of Memory Management Approaches](image)

Figure 2.1: Traditional Memory Management (a), Fork-join MMT (b), versus Client-Server MMT approach (c).

In order to avoid all the limitations above and achieve the transparency goal of MMT, we propose a client-server model instead, as illustrated in Figure 2.1(c). The figure shows that MMT functionalities are split into the client side and the server side. The MMT client, located at the main thread, acts like a memory management library to the application, accepting regular memory allocation and deallocation requests. On a separate thread, the MMT server acts like the application program, generating calls to memory allocation and deallocation routines of the memory management library. The MMT client and server communicate and synchronize to coordinate which requests should be passed on to the library. With this client-server approach, both the application and the memory management library are completely unmodified and are oblivious to the fact that they run on two different threads. Hence, the application can enjoy the parallelism while using a regular memory management library. In addition, the library can be upgraded, patched, or replaced without affecting the way MMT works.

Finally, the client-server approach is flexible in the sense that the MMT client and server determine the order and timing of when allocation and deallocation requests are passed to the library. This allows MMT to apply interesting techniques to achieve performance benefits, such as having the MMT perform speculative dynamic memory management using the memory management library or
even delay some management tasks to be performed out of the critical path of the main application thread to hide dynamic memory management costs from main thread.

2.1.2 Basic MMT Design

As shown in Figure 2.1(c), an MMT client places memory allocation and deallocation requests into a request queue, and uses a wakeup signal to tell the MMT server that there is a request to serve. The MMT server performs the request, and places the reply in a reply queue, and uses a done signal to tell the main thread of the completion of the work. The reply may contain a pointer to a newly allocated chunk in the case of memory allocation, or no reply in the case of memory deallocation. This is illustrated in Figure 2.2(a).

![Figure 2.2: Protocol for MMT to perform synchronous allocation and deallocation (a), and for MMT to perform synchronous allocation but asynchronous deallocation (b).](image)

This basic design uses a semaphore to signal the MMT server to service a request, and another to signal the MMT client of the completion of the request. In Figure 2.2(a), both allocation and deallocation calls are synchronous, i.e. the main thread is stalled until MMT finishes serving the request. However, since memory deallocation does not return a value to the main thread, it does not need to be synchronous. We can take advantage of this fact by making all memory deallocation calls asynchronous, as shown in Figure 2.2(b). Asynchronous deallocation helps in hiding the cost of deallocation by overlapping it with the main thread and hence, exploiting the parallelism between MMT and main application thread.

While simple, we found that the basic design with asynchronous deallocations slows down, rather
than speeds up, applications by $1.9 \times$ on average (Section 2.2.1). One fundamental reason for this is that the latency of communication and synchronization relative to the time to perform memory allocation or deallocation is very high. For example, since allocation is synchronous, the main thread is delayed by the round-trip communication latencies between itself and MMT, which include semaphore signal/wait, and queue entry insertion and deletion latencies. As shown in Figure 1.2, communication and synchronization latencies exceed the latency of the actual work (allocation and deallocation). Therefore, we need to reduce communication and synchronization costs, and at the same time, look for further opportunities to exploit parallelism between main thread and MMT as asynchronous deallocation alone is not sufficient to achieve any performance benefits.

### 2.1.3 Speculative Memory Allocation

One way to reduce the high communication and synchronization costs is to avoid offloading allocation requests since they are synchronous and incur most of the communication and synchronization costs. If allocation requests are performed at the main thread (by the MMT client), we can avoid the communication and synchronization overheads associated with dispatching them to the MMT. Unfortunately, this scheme results in a new type of synchronization overhead. Since allocation and deallocation requests are performed by different threads, the threads can potentially modify the underlying dynamic memory management data structures (e.g. free lists) simultaneously. For example, while the main thread attempts to recycle a free chunk from the free list, the MMT may be adding newly freed chunks to the free list. Therefore, the free list now needs to be protected by a lock, and this causes extra lock synchronization overheads. Contention for the free list lock can be quite severe because asynchronous deallocation calls at MMT compete for locks with synchronous allocation requests performed at main thread. Moreover, such a fine grain task partitioning might potentially reduce the cache locality because dynamic memory management data structures are shared and modified by two different threads. Finally, requiring accesses to the free lists to be wrapped by lock acquisitions and releases breaks the transparency goal of MMT, since the memory management library code must now be modified.

In order to deal with the communication overheads problem without breaking the MMT transparency goal and causing additional synchronization overheads, we go back to letting the MMT to perform not only deallocations, but also allocations. However, to hide the latency of communication and the actual allocation work, MMT also speculatively preallocates heap chunks proactively, in anticipation of the actual allocation requests by the application, during the MMT’s idle cycles. The key to how preallocation can succeed is our observation that applications tend to allocate many objects of the same size. Therefore, MMT can dynamically preallocate an object before the main thread places the allocation request of the object’s size, and hence, extract more parallelism between MMT and main application thread in addition to asynchronous deallocation. Done correctly, preallocation can completely hide the allocation latency, hide some part of the communication and synchronization overheads, and reduce MMT contention as preallocation is performed during the MMT’s idle cycles.

Note that preallocation requires the MMT to predict the next object size that the main thread will ask to allocate. If the prediction is correct (the size matches), the pointer to the newly created
object is returned to the main thread. If the prediction is incorrect (the size mismatches), the MMT has incurred a fragmentation equals to the size of the preallocated object. However, as long as in the future the predicted size is requested, the fragmentation is only temporary. Thus, in general, the MMT does not need to undo its preallocation when it mispredicts the next allocation size during preallocation.

There are several key design questions that need to be addressed in preallocation: (1) what size of dynamic objects should be pre-allocated at MMT? and (2) when should the MMT preallocate an object?

To answer the first question, we profiled allocation size distribution for allocation-intensive benchmarks. We found that most dynamically allocated objects are of small size (less than 512 bytes). This observation has an important implication on our preallocation scheme design. Small allocation requests \( \leq 512 \) bytes are not only common, but are also repeated many times. On the other hand, large requests \( > 512 \) bytes are rare, less repeated, less predictable, and pre-allocating them result in a higher space fragmentation. Thus, we only preallocate for small allocation request sizes.

To answer the next question of when to preallocate for a particular allocation size, there are several possible approaches that we can consider. In the first approach, preallocation can be triggered conservatively when the MMT has seen a high number of allocation requests for a particular size. Such a conservative approach tries to keep unnecessary preallocations to a minimum. However, the approach may not be an optimal choice because (1) keeping track of the number of allocation requests to different size incurs extra occupancy at the MMT, making MMT contention more likely, (2) the cost of miss-speculation is a small amount of fragmentation if preallocation is only made for small request sizes, and (3) the loss of preallocation opportunities during the learning period.

The second approach is more aggressive. Rather than waiting until a pattern is established, we start preallocating for a given size the very first time an allocation request for that size is seen. Since the cost of miss-speculation for small chunks is very small, the aggressive approach avoids the cost of employing a learning algorithm and avoids missing performance opportunities to preallocate. Thus, we choose the latter approach for the MMT design.

2.1.4 Bulk Memory Allocation and Deallocation

While MMT performing preallocation and asynchronous deallocation are effective techniques for hiding allocation and deallocation latencies through extracting parallelism between MMT and main application thread, unfortunately they are not sufficient for MMT to give a net positive performance benefit. The fundamental reason is that the latency of semaphore signal/wait and lock acquisition is higher than the time between allocation and deallocation requests (see Figure 1.2). Since each request involves a pair of semaphore signal/wait and queue entry insertion and deletion latencies, requests are generated by the application at a much faster rate than they can be dispatched to the MMT. Therefore, we must reduce communication and synchronization frequencies, reduce their latencies, or both. In this section, we discuss how to reduce their frequencies by performing memory management requests in bulk, and leave reducing their latencies to Section 2.1.5.
Bulk Deallocation

Asynchronous deallocation exploits parallelism between MMT and main thread, but to reduce high communication and synchronization frequencies, we aggregate multiple deallocation requests and dispatch them to the MMT server as a single request. We refer to this scheme as bulk deallocation. To support bulk deallocation, we design the request queue such that while one part of it still sends a request to the MMT server on each allocation, another part allows deallocation requests to be accumulated and sent to the MMT server as a single request when they fill up the part of the queue.

The number of chunks that are deallocated in one go (the bucket size) is an important determinant of performance. Recall that deallocation latency is hidden by MMT because it is performed asynchronously during the time period in which the MMT server would otherwise be idle, so it tempting to set the bucket size large. But idle cycles are scattered as holes located between synchronous allocation requests made by the main thread and bulk deallocation may take too long to fit in any contiguous idle cycles. If an allocation request arrives while the MMT is busy performing bulk deallocation (a collision), the allocation request must be delayed until the bulk deallocation completes, which directly exposes the deallocation latency to the main thread. While a too-small deallocation bucket size does not reduce communication and synchronization frequencies sufficiently, a too-large deallocation bucket size exposes deallocation latency. Hence, choosing a good preallocation bucket size is important for optimum performance. Through experiments, we found that, a bucket size of 200 provides a good balance between the competing goals of reducing communication and synchronization frequencies and keeping deallocation latency hidden, across all applications (more discussion in Section 2.2.5).

Bulk Speculative Allocation

The principle of aggregating requests cannot be applied directly to allocation requests because they are synchronous. However, we can apply them to preallocation since it is performed speculatively. To achieve that, MMT can preallocate a bucket of chunks of a particular size in parallel with main application thread rather than preallocating one chunk at a time. Furthermore, this group of preallocated chunks can be sent directly to the MMT client, which will store them locally. When an allocation request of a particular size occurs, and the MMT client has a preallocation bucket for the size, it retrieves one chunk from the bucket and returns it to the application, without involving any communication or synchronization with the MMT server. If the bucket of a particular size is empty, the MMT client sends an allocation request to the MMT server.

In designing bulk speculative preallocation, there are several important issues to consider, such as (1) when preallocation should be initiated, (2) how many buckets there should be, and (3) what preallocation bucket size is appropriate for each bucket. We will discuss each of the issue in more details next.

The first design issue is when preallocation should be initiated. Suppose that an allocation request of size $x$ is received by the MMT server. The MMT server allocates and returns the requested chunk, and with bulk speculative preallocation, it starts to preallocate $N$ chunks of size $x$ by repeatedly calling malloc to the memory management library. If later an allocation request for size $x$ is received again, the speculation is confirmed to be correct, and the MMT server immediately sends off the
preallocation bucket to the MMT client. From this point on, the main thread will start to consume chunks from the bucket residing on MMT client until the bucket becomes empty. During this time, the MMT server can take one of the following actions. In a conservative approach, after half of chunks in the preallocation bucket are used up, the MMT client sends a resume-preallocate signal to the MMT server so that it can start preallocating the next bucket of $N$ chunks. Hopefully by the time the main thread has used up all chunks in the current bucket, the MMT server has finished preparing the next bucket. This conservative preallocation is illustrated in Figure 2.3(a). An alternative is to use a more aggressive approach. The MMT can proactively preallocate the next $N$ chunks immediately after giving a preallocation bucket to the main thread, as shown in Figure 2.3(b).

The conservative approach seems reasonable because it reduces the probability of unnecessary preallocations and can hide the allocation cost if preallocations are completed before the next request arrives. However, it suffers from the risk of not being able to preallocate timely as the MMT may be busy when the resume-preallocate signal arrives, in which case preallocation is delayed until the MMT becomes free. On the other hand, the aggressive approach has a better chance of finishing its next preallocation bucket before the main thread needs it. Through experimentation with both approaches, we confirmed that the aggressive approach always outperforms the conservative approach for the same reason. Hence, MMT uses the aggressive preallocation approach.

The next design issue to consider is whether to associate the preallocation buckets with the allocation request size (which is determined by the application) versus the allocated chunk size (which is determined by the memory management library). The actual chunk size allocated is typically the request size rounded up to the nearest multiple of some (e.g. 8) bytes. Thus, there are a lot more possible request sizes than there are chunk sizes. Keeping one bucket for each chunk size helps in reducing the total number of preallocation buckets to be managed but it breaks the transparency goal because it
requires the knowledge of the chunk size classes as determined by the memory management library. Furthermore, recall that during preallocation, the MMT server calls malloc at the library repeatedly until it obtains the target number of chunks. For preallocation to completely hide allocation latency, the rate at which a bucket is drained by the application must be lower than the rate at which a bucket can be filled by the MMT server, which depends on allocation rate the memory management library can sustain. We found that sometimes there are similar allocation request sizes (e.g. 26 and 28 bytes) made by the application that correspond to a single chunk size (32 bytes). These allocation requests are bursty, and during bursts, the bucket can be drained quicker than a new one can be filled. Keeping two buckets, one for 26 byte request containing 32 byte chunks and another for 28 byte request also containing 32 byte chunks, helps tolerate such burstiness. Therefore, we reduce the contention at MMT by keeping one preallocation bucket for each allocation request size both at MMT client and server side, without breaking the transparency goal of our MMT approach. In order to avoid keeping too many buckets, we only perform preallocation for request sizes smaller than 512 bytes.

Finally, as with bulk deallocation, the choice of preallocation bucket size is important for performance. Choosing too small bucket size might not reduce synchronization cost significantly, on the other hand choosing too large bucket size might expose the preallocation latency to the main application thread. Through experiments, we find that bucket size of 400 offers good performance across all applications we tested, except in few cases where applications generate memory allocation requests in bursts, and the bursts between different allocation sizes coincide. For example, suppose that there are allocation requests for size $x$, $y$ and $z$ performed in a loop. If the same preallocation bucket size is used for all allocation request sizes, preallocation buckets for them will be filled and drained at the same time. This creates a bottleneck at the MMT server due to increased contention at MMT server such that it must refill several buckets at once, making an allocation request for size $y$ wait the completion of preallocation for size $x$. This is obviously detrimental to performance since the allocation latencies are now largely exposed to the main thread. To avoid contention at MMT, we vary the bucket sizes for different allocation request sizes, according to the following formula: $\text{BucketSize}_i = \text{BaseSize} + k \times i$, where $i$ is the allocation request size, $\text{BaseSize}$ and $k$ are constants that are applied to all request sizes. Through experiments, we find that $\text{BaseSize} = 400$ and $k = 4$ provide good performance across a wide range of scenarios without any application-specific tuning.

Note that bulk deallocation and bulk preallocation are only coordinated between the MMT client and server. Both the application and the memory management library are oblivious of the bulk deallocation and preallocation, and all the library sees are individual calls to malloc and free made by the MMT server. MMT approach changes the ordering of allocation and deallocation requests, however that does not affect the correctness or semantics of the programs. Interestingly, changing the ordering of requests and doing them in bulk, on a dedicated thread, improves the cache and TLB performance of the applications we tested (Section 2.2.3). Moreover, we note that cost of miss-speculative preallocations is only small amount of fragmentation and no roll-back is required unlike other speculative techniques.
Table 2.1: Latency of Pthread vs. minimalist synchronization primitives, refer Section 2.2 for machine configuration details.

<table>
<thead>
<tr>
<th></th>
<th>Contention-free</th>
<th>2-thread Contention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pthread lock</td>
<td>0.05µs</td>
<td>0.10µs</td>
</tr>
<tr>
<td>Minimalist lock</td>
<td>0.02µs</td>
<td>0.06µs</td>
</tr>
<tr>
<td>Pthread semaphore</td>
<td>–</td>
<td>0.17µs</td>
</tr>
<tr>
<td>Minimalist semaphore</td>
<td>–</td>
<td>0.07µs</td>
</tr>
</tbody>
</table>

2.1.5 MMT Synchronization Mechanism

In the previous section, we have discussed how to reduce the synchronization frequency. In this section, we will show techniques to reduce the latency of each synchronization in MMT. Bulk speculative preallocation and deallocation decrease synchronization and communication frequency, but increasing the bucket size for them beyond a certain point degrades performance due to increased MMT contention. Therefore, it is important to reduce the latency of synchronization mechanisms to achieve performance improvements.

To reduce the cost of synchronization, we can implement faster synchronization primitives, or avoid the use of semaphores and locks altogether. We will detail them next.

Exploring Minimalist Synchronization Primitives

Our first attempt to reduce synchronization cost is to replace Posix thread (Pthread) synchronization primitives with our own “minimalist” implementation. Pthread primitives can cause a thread to block if it waits for a synchronization event to complete. In a minimalist approach, to implement a lock, we use non-volatile variables, atomic (compare and swap) instructions, and a spin loop for waiting in case the lock is not free. A similar implementation is used for the minimalist semaphores. The synchronization latencies for the minimalist primitives, compared to those of Posix, are shown in Table 2.1. In general, the minimalist primitives’ latencies are 39-60% lower compared to Pthread primitives. However, in addition to the serialization introduced by locks, their costs are still non-negligible for frequent use by MMT (Section 2.2.2 shows results). Thus, we seek to eliminate the synchronization costs entirely.

Avoiding Semaphores

The purpose of the semaphore is to signal the MMT that the main thread has a new request for the MMT to work on. The main thread increments the semaphore, while the MMT decrements it (if positive). The semaphore variable is protected by a lock (or by atomic increment and decrement) and that is where the serialization and contention come from. To avoid this bottleneck, we let each thread keep its own local counter. Each thread can write only to its own local counter, but it can read others’ local counters. The main thread (MMT client) increments its “wakeup” counter when it has a request for the MMT to serve, while the MMT server increments its "served" counter when it has served a request. The MMT (server) discovers about a pending work if the difference between the wakeup
counter and the served counter is positive. Since now each counter only has a single writer, a lock or an atomic increment/decrement over the counter is no longer necessary. It does not assume any kind of support from underlying hardware either.

**Avoiding Locks**

Locks are used by MMT client and server to protect against simultaneous modifications to the request and reply queues, where the MMT client inserts requests into the queue, while the MMT server removes requests from the queue. There are two ways to eliminate the use of locks entirely. One way is to use optimistic concurrency (lock-free) approach such as using software transactional memory (STM). However, such an approach suffers from increased latency under a low contention situation due to the multiple passes involved (marking node addresses, acquiring nodes, and committing). Since the goal of avoiding locks is to improve performance, the extra latencies in optimistic concurrency can potentially defeat the purpose. Thus, we use a different approach. We can eliminate locks entirely if we can ensure that at any given time, only one thread writes to the queue. To achieve that, we split the queue into two, and define a time period (epoch) in which the MMT client always modifies one queue while the MMT server always modifies the other queue. In the next time epoch, the role is reversed. This is illustrated in Figure 2.4. We do not know if such an approach is new (it probably is not), but we argue it is essential for parallelization involving very fine grain tasks, in order to avoid regular lock overheads and critical section serialization.

![Figure 2.4: Illustration and pseudo-code of our lockless protocol.](image)

**2.1.6 Putting it All Together**

**Handling Allocation Requests.** To handle allocation requests, both the MMT server and MMT client keep a bucket (array) of memory pointers of preallocated chunks associated with each allocation request size. On an allocation request, if the MMT client has a non-empty bucket for the request
size, it retrieves an object from the bucket and returns it to the application. Otherwise, it places an allocation request in the allocation request queue and increments the wakeup counter. The MMT server continuously checks the difference between the wakeup and served counters. If it is positive, there is an outstanding allocation request. If it already has a new preallocation bucket filled up for the requested size, it passes the bucket to the MMT client, and signals the MMT client to resume execution. The MMT server then starts to preallocate a new bucket to replace the one it just passed. Preallocation buckets on either side do not need to be protected by locks as at any time each bucket only has one writer. Note that preallocation is only applied to allocation sizes that are 512 bytes or smaller. Larger allocation request sizes are handled synchronously.

**Handling Deallocation Requests.** To handle deallocation requests, two bulk deallocation requests are shared between MMT client and server. These deallocation queues are used for all deallocation chunk sizes. The main thread adds a deallocation request to one queue and MMT serves deallocation requests from the other queue, in order to avoid using locks for synchronization, reduce contention at MMT, and exploit more parallelism between MMT and main thread (Section 2.1.5). The MMT client uses a flag to indicate which queue it has recently filled up. MMT then drains the deallocation requests from one queue while main thread is filling the other queue without stalling. In the rare occasion in which both queues fill up, the main thread waits until the queue it waits for becomes free. With two bulk deallocation queues, not only lock synchronization is avoided, but there is almost no stall since most of the time both the main thread and MMT work in parallel on different queues.

**Interaction between Allocation and Deallocation Request Handling.** The design for preallocation and deallocation affects the performance of one another positively. Bulk deallocation can potentially reduce the cache locality of the program because recently deallocated chunks have a smaller reuse probability, but bulk preallocation technique increases the cache locality of the program by allocating the chunks before they are needed. Bulk preallocation bucket needs to be refilled in one continuous window of idle cycles at MMT which are scattered, fortunately applying bulk deallocation in conjunction increases the probability of finding one large continuous window of idle cycles as MMT does not get constantly interrupted by single asynchronous deallocation calls. Similarly, bulk preallocation helps bulk deallocations in finding large continuous window of idle cycles. Bulk deallocation might make the MMT busy for a long period of time, which increases the waiting time for allocation requests if they have to be serviced synchronously. However, preallocation reduces the occurrence of this case since most allocation requests can be served from the preallocation buckets. Moreover, when the MMT server has an outstanding request to serve, it prioritizes serving an allocation request over deallocation requests since allocation requests are in the critical path of the program execution. Therefore, overall synergy between bulk deallocation and preallocation helps in achieving higher performance in our MMT approach.

### 2.2 Experimental Methodology

**Machine configuration.** We evaluated MMT on a 2.4GHz Intel Core2 quad processor, which runs Linux kernel version 2.6.18. For all experiments, only the application and MMT run, plus regular OS
processes and daemons, and no other user applications run. Each core has small private L1 instruction and data caches, and the cores share a 4MB L2 cache.

**Benchmarks.** We use seven heap allocation intensive benchmarks which perform high number of allocation and deallocation calls per unit time (Table 2.2): cfrac, deltaBlue, gawk, mst, perimeter, roboop, treeadd. These benchmarks have been widely used in past dynamic memory management studies [8, 16, 67, 72, 73, 81, 87]. Each benchmark is compiled with GCC 4.1.2 with -03 optimization flag and with Posix thread version 2.5. For each experiment, each benchmark is run three times, and the average execution time is used for reporting.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Input</th>
<th>malloc/free ops per sec (Doug Lea’s allocator)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfrac</td>
<td>a 35 digit number</td>
<td>9,388,037</td>
</tr>
<tr>
<td>deltaBlue</td>
<td>1000000</td>
<td>37,908</td>
</tr>
<tr>
<td>gawk</td>
<td>large.awk</td>
<td>3,308,227</td>
</tr>
<tr>
<td>mst</td>
<td>8192 nodes</td>
<td>3,441,900</td>
</tr>
<tr>
<td>perimeter</td>
<td>13 levels</td>
<td>6,083,674</td>
</tr>
<tr>
<td>roboop</td>
<td>bench</td>
<td>9,057,914</td>
</tr>
<tr>
<td>treeadd</td>
<td>27 levels</td>
<td>13,357,346</td>
</tr>
</tbody>
</table>

**Memory management library.** To demonstrate MMT’s transparent design, we use two widely used memory allocation libraries. The first library implements Doug Lea’s allocator [74], which is considered to be among the fastest and most space efficient allocators [7, 9], and also forms the basis of GNU C library. The second library is PHKmalloc allocator designed for FreeBSD operating systems [71]. PHKmalloc differs significantly from Doug Lea’s allocator (details in Section 2.2.3).

In addition, PHKmalloc provides memory allocation and deallocation security checks which can be turned on by specifying malloc options, but they result in high performance overheads [71]. However, the security checks have been shown to be highly useful in both identifying both bugs and detecting attacks [71, 82]. We use secure PHKmalloc allocator to illustrate how MMT can reduce the overheads from security checks.

**Detailed profiling.** We use Oprofile [80], a low overhead profiler, for collecting L2 cache statistics, branch mispredictions and data TLB miss statistics for the process we run. These statistics are collected using a different run than the one used for reporting performance numbers, to avoid any profiling perturbation and noise. Evaluation

In this section, we evaluate the performance of MMT on two different memory allocators (Doug Lea’s and PHKmalloc) plus security check-enabled allocator (PHKmalloc), to gain insights into the performance of MMT. Instead of reporting the performance of the final design of MMT, we will show and discuss the performance of various incremental designs of MMT in order to observe the contributions of various design aspects.
2.2.1 Basic MMT Design Performance

In this subsection, we evaluate the following two designs:

<table>
<thead>
<tr>
<th>Design</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>No MMT used. The application calls the Doug Lea’s allocator library directly</td>
</tr>
<tr>
<td>Design 1</td>
<td>Synchronous allocation and deallocation at MMT</td>
</tr>
<tr>
<td>Design 2</td>
<td>Synchronous allocation, bulk asynchronous deallocation at MMT</td>
</tr>
</tbody>
</table>

Figure 2.5 shows the execution time for various benchmarks normalized to the base case in which MMT is not used, broken down into the time spent for computation or for memory allocation and deallocation. The base case uses Doug Lea’s memory allocator. For MMT in Design 1 and Design 2, the execution time is from the main thread, hence the allocation and deallocation time components represent the application waiting for allocation and deallocation requests being performed at the MMT server, or in some cases at the MMT client (e.g., when chunks are extracted from a preallocation bucket). In all designs, we use the minimalist synchronization primitives discussed in Section 2.1.5, unless otherwise noted.

The Base bars in the figure show that on average, the benchmarks spend 30% of their execution time in memory allocation and deallocation routines. Typically, allocation time is much higher than deallocation time. Comparing Design 1 and the Base case, we can see that offloading each allocation and deallocation request to the MMT significantly slows down all benchmarks, increasing the execution time by 173% on average. In Design 1, since each request is serviced synchronously, it is clear that there is no parallel execution between the main thread and the MMT, hence speedups for Design 1 should not be expected. However, the magnitude of slowdown confirms our data in Figure 1.2 in that the additional communication and synchronization overheads for each request greatly outweigh the
latency to perform the actual memory allocation and deallocation.

In Design 2, we use bulk asynchronous deallocation with a single deallocation queue of size 200 to reduce contention at MMT due to frequent deallocation requests. This reduces the average slowdown from 173% in Design 1 to 89% in Design 2. The reduction in slowdown is primarily due to the 88% reduction in the deallocation time component going from Design 1 to Design 2, demonstrating the ability of bulk asynchronous deallocation in hiding deallocation latency. However, even after the reduction, the deallocation time component in Design 2 is still higher than that in the Base case. Overall, even with asynchronous deallocation in Design 2, MMT still does not get any speedup, and the overall slowdown is still very high.

2.2.2 Task Partitioning and Synchronization

From the previous section, we have observed that the main performance problem of MMT in Design 2 is the high allocation time component. Therefore, we need to focus on reducing the allocation cost. One intuitive solution is to perform allocation at the main thread itself instead of at the MMT, as discussed in Section 2.1.3. Thus, in this section, we will evaluate several designs centered in an approach in which the MMT only performs asynchronous deallocation, while allocation is performed at the main thread. The designs are:

<table>
<thead>
<tr>
<th>Design 3, 4, 5</th>
<th>allocation at main thread, asynchronous deallocation at MMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design 3</td>
<td>Pthread synchronization</td>
</tr>
<tr>
<td>Design 4</td>
<td>Minimalist synchronization (Section 2.1.5)</td>
</tr>
<tr>
<td>Design 5</td>
<td>Design 4 + bulk deallocation with a single queue</td>
</tr>
</tbody>
</table>

Figure 2.6 shows the execution time for various designs normalized to the base case in which no MMT is used. Comparing Design 3 with Design 2 from Figure 2.5, it is clear that moving allocation requests back to the main thread reduces the slowdown significantly, on average from 89% to 46%.

Comparing Design 3 and the base case, we can make some interesting observations. First, the allocation time of the base case roughly doubles on average in Design 3 (and quintuples in cfrac and gawk!). Both the base case and Design 3 perform allocation at the main thread by directly calling memory allocation library. Thus, the increase may not seem intuitive. The primary reason is that an allocation request attempts to recycle a recently-freed chunk from the free list. At the same time, deallocation requests at the MMT also attempts to add deallocated chunks into the free list. As a result, the free list must be protected by a lock, and the allocation time suffers from the lock overheads and serialization due to critical sections. This outcome can also be confirmed by looking at the deallocation time which triples, because deallocation also suffers from the same overheads. In addition to these lock overheads, this design breaks the transparency of MMT by making it aware of the internal data structure of the allocation library, i.e. the free lists.

Furthermore, the computation time component increases slightly from the base case to Design 3, especially noticeable in cfrac and roboop, caused by a more subtle phenomenon: the reduction in cache temporal locality when the allocation and deallocation routines run on different cores. For example, the free list is read and written by both the allocation and deallocation routines, so data in the free list
is ping-ponged between the cores through invalidations and subsequent L1 cache misses (note that the L2 cache is shared by both cores).

Since synchronization overheads dominate in Design 3, Design 4 replaces Pthread synchronization with our minimalist synchronization implementation (Section 2.1.5), which roughly halves of the latency of Pthread synchronization primitives. As a result, Design 4’s average slowdown compared to Base is only 33%, through a 21% reduction in allocation time and 13% reduction in deallocation time compared to Design 3. However, Design 4 still suffers from serialization cost of critical sections, and from the bad cache temporal locality. Moreover, even with minimalist synchronization primitives, synchronization latencies are still too high.

Design 5 adds bulk deallocation with a single deallocation queue to Design 4 in order to reduce the synchronization and communication frequencies. The bulk deallocation queue has 200 entries, which provides a good balance between synchronization frequency reduction and MMT contention. Compared to Design 4, the lower synchronization frequency reduces both allocation and deallocation time components significantly such that on average Design 5 performs similarly to the Base case. However, in some programs (cfrac and roboop) the slow down is still significant.

### 2.2.3 Bulk Preallocation and Deallocation

Some important lessons from the previous section are that the synchronization frequency for allocation is still high, and that even minimalist synchronization primitives cannot completely remove synchronization cost since the serialization due to the critical sections still remains. Therefore, the final design to explore is one in which the MMT performs bulk speculative preallocation, and synchronization primitives are avoided. As discussed in Section 2.1.3, preallocation increases parallelism by performing many allocations at once during MMT idle cycles. In this section, our final MMT design employs both bulk deallocation and bulk speculative preallocation, as well as communication...
and access protocols that avoid semaphores and locks (Section 2.1.5 and 2.1.5).

<table>
<thead>
<tr>
<th></th>
<th>Dealloc Time</th>
<th>Alloc Time</th>
<th>Comp Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMT</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
</tr>
<tr>
<td>Base MMT</td>
<td>0.8</td>
<td>1.0</td>
<td>0.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Dealloc Time</th>
<th>Alloc Time</th>
<th>Comp Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMT</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
</tr>
<tr>
<td>Base MMT</td>
<td>0.8</td>
<td>1.0</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Figure 2.7: Performance of the final MMT design on Doug Lea’s allocator (a), and PHKmalloc allocator (b).

To demonstrate MMT’s transparent design, we apply MMT to two widely used memory allocation libraries: Doug Lea’s allocator and PHKmalloc library. The two allocators are significantly different in their design philosophy, algorithms and data structures: (1) PHKmalloc is aware of the virtual memory system while Doug Lea’s allocator is not, (2) PHKmalloc tries to minimize TLB misses and page working set, (3) PHKmalloc does not store metadata for small objects, (4) PHKmalloc rounds up small object requests (less than 2KB) to the nearest power of two versus multiple of 8 bytes in
Doug Lea’s allocator, etc. Using two widely used allocators that are significantly different is suitable for demonstrating the generality of our MMT approach, since MMT is designed without taking into account the underlying algorithms and data structures employed in both allocators. In addition, both allocators are unmodified and are oblivious to MMT and MMT’s parallelization schemes. The performance of MMT for Doug Lea’s and PHKmalloc allocators are shown in Figure 2.7. PHKmalloc’s base case performance is close to that of Doug Lea’s library (within 3% on average across all the benchmarks we tested), except for deltaBlue in which PHKmalloc outperforms Doug Lea by 30% due to a better virtual memory performance. We could not get one benchmark (gawk) to run with PHKmalloc library, therefore we do not report results for gawk.

The figure shows that despite being designed without taking into account the knowledge of the underlying algorithms and data structures of the allocators, MMT reduces the average execution time by 16% in both allocators, resulting in an average speed up ratio of $1.19 \times$. On an average, MMT reduces the allocation and deallocation time components by between 50-70%, which is a very significant improvement. Moreover, the speedups are highly uniform across all benchmarks, except for deltaBlue on the PHKmalloc allocator. The reason for this is that memory management time is no longer significant in deltablue due to the excellent cache and virtual memory performance in PHKmalloc library.

The most important source of performance is the parallelism between MMT and the main thread. However, that is not the only source of performance improvement, since MMT also reduces the computation time component of some applications. To dig into this deeper, we profile the number of L2 cache misses and TLB misses for both MMT and main thread application thread (the number of TLB misses is the sum over both cores) when we apply our MMT approach to Doug Lea’s allocator (Figure 2.8). We find that the number of L2 cache misses is reduced by 23% in the best case, and 5% on average. Similarly, the number of TLB misses is also reduced by 18% on average. The reason why MMT
improves the L2 cache and TLB performance is that the code and cache behavior of regular computation in the benchmarks and the memory allocation library are different enough that they produce mostly negative interference. When they run in separate cores with the MMT approach, they no longer compete and interfere for core resources or cache resources. This observation strengthens the appeal of decoupling the memory management from the main application.

We note that speculative bulk allocation should have positive effect on cache locality, on the other hand bulk deallocations may have negative effect. Interestingly, speculative preallocation and bulk deallocation work synergistically so that overall locality of the program is significantly improved (Figure 2.8). We also experimented with different techniques improving the cache locality by short-circuiting the deferred deallocations, and found that all programs have better or at least same level of locality compared to sequential base case even without short-circuiting deferred deallocations because of positive synergy between speculative preallocations and bulk deallocations. Though two applications do get significant extra benefits due to short-circuiting. More detailed discussion on the same can be found here [18].

### 2.2.4 Safe Memory Management using MMT

In this subsection, we will show that MMT can also be very effective in hiding the high overhead of security checks in memory management library. For that, we enable extra security features in the PHKmalloc library, which include among others zeroing newly allocated chunks, detecting whether pointers being deallocated are valid, detecting double frees, etc. These checks were designed to cover security vulnerabilities that have been reported in many applications [3, 4, 11], and have been shown to detect bugs or attacks on many applications such as fsck, ypserv, cvs, mountd, and inetd, etc [82].

![Dealloc Time vs Alloc Time vs Comp Time](image)

**Figure 2.9:** Security overhead using MMT approach (Base: PHKmalloc allocator)
Note that we do not claim that the security checks are novel, efficient, or complete. They are simply what are provided in the PHKmalloc library. Our focus is that these security checks and features are very expensive, slowing down our benchmarks by 21% on average, and by 44% in the worst case (Figure 2.9). The figure shows that MMT can hide the overheads almost completely, with an average slowdown of only 1%. Only one benchmark (deltaBlue) still suffers from 41% slowdown. All remaining benchmarks are either faster or just as fast as the base PHKmalloc without security features turned on. Overall, this result demonstrates that the parallelism that MMT has relatively broad benefits. It can be used for improving memory management performance, or for hiding overheads of various security functionalities, all achieved without modifications to the memory management libraries.

2.2.5 Sensitivity Analysis

As previously discussed in Section 2.1.4 and Section 2.1.4, choosing the right bucket size for bulk pre-allocation and deallocation is important for performance. We experimented with different bucket sizes for all applications and find that a preallocation bucket size of between 200 and 400 works consistently well for all applications. Similarly, the bulk deallocation queue also gives good performance when the size is between 200 and 400. There is little performance variation for bucket sizes between 200 and 400. However, bucket sizes less than 50 or larger than 1000 show noticeably worse performance. Because of the stability of performance for bucket sizes between 200 and 400 across all applications tested, MMT does not need to be tuned for specific applications.

2.2.6 Memory Footprint

Both bulk preallocation and deallocation may increase the heap memory footprint due to the storage fragmentation of preallocated chunks and delayed storage reclaiming of freed chunks. However, we find that the increased memory footprint for all benchmarks is negligible because the storage overheads of the preallocation and deallocation buckets are small and bounded, compared to the total heap size of the applications.

2.3 Related Work

Dynamic memory management for single core systems. In general, past studies have attempted to improve the performance of dynamic memory management in two ways: designing and implementing better memory management algorithms [71, 74, 78, 85], and increasing the cache reference locality of dynamically allocated objects, through various techniques such as changing the heap layout, predicting lifetime and reference pattern of heap objects, using profiling information, hardware support, etc. [13, 14, 15, 65, 67, 68, 73, 75, 79, 83, 84]. This work looks into an orthogonal question of how dynamic memory management can take the advantage of multicore parallelism.

Custom and pool-based memory allocation techniques have been used for allocation-intensive programs. However, such an approach suffers from significant challenges: (1) customization requires significant programming effort and non trivial source code modifications, (2) programmers must have
a priori knowledge of the allocation and deallocation patterns, (3) memory-related debugging and leak detection become challenging, and (4) many applications are not suited for customized or pool based allocation because of irregular allocation/deallocation patterns, and memory fragmentation and footprint issues due to the coarser allocation and deallocation granularities. Moreover, Berger et al. [7] show that a general purpose allocator such as Doug Lea’s allocator performs competitively with custom memory allocators. Therefore, we believe that it is crucial to speed up applications that rely on common memory allocation libraries, such as Doug Lea’s GNU C allocator and PHKmalloc library.

**Dynamic memory management for parallel and multithreaded applications.** There have been many attempts to design memory allocators for parallel and multithreaded applications, such as Hoard [8], MAMA [70], Streamflow [54], and TCMalloc [66]. Compared to allocators for sequential applications, these allocators attempt to address scalability, avoid false sharing, and restrict memory footprint as first order goals. Currently, MMT targets parallelization between sequential applications and sequential memory allocation libraries. While it is possible to apply our MMT approach for multithreaded memory allocation libraries, it is outside the scope of this work and hence we leave it as future work.

**Heap Server.** Kharbutli et al. [72] proposed splitting the memory management functions into a separate process to decouple the heap meta data and heap data storage in different address spaces in order to improve heap security. While MMT shares the spirit of splitting the memory management functions from the main application, MMT’s goal is to improve performance by exploiting multicore parallelism. Consequently, there are very important differences. First, HeapServer requires the use of a specific (bitmap based) memory allocator that is organized differently than commonly used allocators. In contrast, MMT can be used in conjunction with practically any memory allocators. Secondly, Heap Server slows down, rather than speeds up, the performance of heap intensive applications for better heap security. In contrast, MMT improves the performance of sequential heap-intensive applications. Finally, inter-process and inter-thread synchronization and communication protocols differ significantly in their nature and trade-offs. Therefore, designing and implementing a dedicated thread for memory management functions provides a different set of challenges, opportunities, and design issues.
Chapter 3

Modeling and Analyzing Key Performance Factors of Shared-Memory MapReduce

This chapter is organized as follows: Section 3.1 provides an overview to shared memory MapReduce programming model. Section 3.2 describes our analytical model while Section 3.3 uses the model to explore how performance is affected by various factors. Section 3.4 describes evaluation platform. Section 3.5 shows model validation and evaluation results on real systems. Section 3.6 describes our application classification framework. Section 3.7 discusses related work.

3.1 Background

3.1.1 MapReduce Programming Model

MapReduce program execution is divided into two phases: map and reduce. In the map phase, input records are distributed across map workers in a data-parallel manner. Each map worker performs computation by calling a user-defined map function. The map function produces output in the form of a list of <key,value> pairs, and the output is stored in the intermediate file or buffer. The reduce phase reads from the intermediate file or buffer to perform reduction on the output of the map phase. Each reduce worker is assigned a key or a group of keys and calls a user-defined reduce function, which aggregates all values having the same key. The output of the reduce function are <key, aggregate-value> pairs, which are then sorted to produce the final output. Thus, map workers achieve data parallelism, while reduce workers perform parallel reduction.

An example of MapReduce application is word count, which counts the number of occurrences of each distinct word in a text document. In word count, each map worker takes a chunk from the text...
document input, and for each word that it encounters, it produces a $<\text{word, 1}>$ pair as intermediate output. At the completion of the map phase, the reduce phase is started. Each reduce worker works on a specific key (a distinct word) to sum up the list of values (counts) associated with the key.

Programmers are attracted to MapReduce due to its high level abstraction and simplicity. Programmers define the map and reduce functions, and the MapReduce runtime system automatically manages concurrency and fault tolerance, such as creating, dispatching, and scheduling map/reduce threads, performing synchronization, sorting output, managing locality, restarting crashed threads, etc.

### 3.1.2 Shared Memory MapReduce

Shared memory MapReduce keeps the intermediate buffer as a data structure stored in the main memory, where any map worker thread can write to, and any reduce worker thread can read from. Without shared memory, map workers and reduce workers must communicate through the file system on disks and rely on remote procedure calls for managing the workers, which are expensive. As a result, shared memory MapReduce has been shown to outperform the disk-based MapReduce by up to an order of magnitude on contemporary multicore machines [26, 36]. The drawbacks of shared memory MapReduce are that it is more suitable for smaller computation that fits in the main memory, and fault tolerance cannot be achieved as easily without relying on non-volatile storage. Therefore, it is possible that future MapReduce programs may use a hybrid approach, e.g. disk-based version across shared memory nodes providing fault tolerance on persistent storage, with each computation on a node further divided into finer granularity tasks to exploit in-memory computation [77].

![Figure 3.1: Overall design of shared-memory MapReduce runtime system.](image)

Currently, there are two shared memory MapReduce runtime systems available publicly: Phoenix [51,
and Metis [41]. While Phoenix has been studied more extensively [24, 26, 36, 38, 39, 63], Metis was said to be designed to improve key shortcomings of Phoenix. In addition, while both Phoenix and Metis share an intermediate buffer data structure (2-dimensional array of hash buckets), they differ significantly in what data structure is used for each bucket, and in the key algorithms used in the map and reduce phases. Therefore, we design our model to be parametric, such that a single model can capture both Phoenix and Metis, and future variants.

In both Phoenix and Metis, input data is split into equal-sized chunks that form tasks, and tasks are queued and assigned dynamically to map worker threads. Figure 3.1 illustrates a generic intermediate data structure (used in both Phoenix and Metis), which is a two dimensional array with number of rows equaling the number of map workers, and number of columns equaling the number of hash buckets. Each thread works independently on a row of the matrix and stores its intermediate output in hash buckets in that row. When a thread produces a key-value pair, the key is hashed into a bucket using a hash function, and is added to the bucket. Each bucket maintains a list of distinct keys in a sorted order (to facilitate fast key searching), with each key having a list of values associated with it. Since different threads use different matrix rows to store their intermediate output, there is no data sharing or synchronization among map worker threads.

In the reduce phase, each column of bucket forms a task, and tasks are queued and assigned dynamically to reduce worker threads. Each reduce worker performs reduction by aggregating values that share a common key, both within a bucket and across buckets in the same column. Since reduce workers work on separate columns of buckets, they do not share data. However, map workers and reduce workers share data through the intermediate buffer. After aggregating values, a reduce worker produces \(<key, aggregate-value>\) pair for each distinct key assigned to it, and participates in a merge sort to combine the output from all reduce workers so that the output in the final output buffer is sorted based on key order, as shown in the figure.

### 3.2 Analytical Model Formulation

#### 3.2.1 Assumptions and Scope

The goal of our modeling is not to predict execution time accurately. Rather, it is to capture mathematical relationships of how various factors are related and affect performance, and quantify their relative magnitude of impact on performance.

The scope of our study is shared memory MapReduce performance. Hence, we do not model I/O and network performance. The aggregate effects of processor and cache performance are not explicitly quantified, but are captured as parameters in our parametric model. The values of these parameters can be empirically derived on a target machine.

We assume a standard MapReduce programming model where the reduce phase starts after all map tasks have fully completed. Some variants of MapReduce break up the map task execution and interleave (but not overlap) it with the reduce phase [26]. While this model may not be readily applicable to such a variant, it can be adapted for it if desired.

We instantiated our model for Phoenix and Metis, two shared memory MapReduce libraries that
are publicly available at the time of this study. Taking into account the need for the model to be extensible for future shared memory MapReduce systems, we designed the model to be parametric, such that minor variations only differ in parameter values. For example, most of the differences between Phoenix and Metis are contained in the parameter values that are input to the model.

### 3.2.2 Modeling Approach

Table 4.1 shows parameters and notations we use in our model. During the map phase, all map tasks are executed and the intermediate output is stored into the intermediate buffer. Therefore, we can divide the map phase into two sub-phases: computing the map task (map-comp), and storing the intermediate output (map-output). The map-comp phase is inherent to the algorithm, and its performance is relatively independent of whether it is executed with MapReduce or other models (e.g. PThreads, OpenMP). However, the map-output phase is unique to MapReduce and is an important determinant of performance. Hence, we will focus on modeling map-output performance.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N_I)</td>
<td>Number of Input records</td>
</tr>
<tr>
<td>(N_{mapt})</td>
<td>Number of map worker threads</td>
</tr>
<tr>
<td>(N_{redt})</td>
<td>Number of reduce worker threads</td>
</tr>
<tr>
<td>(C)</td>
<td>Chunk size (#input records grouped as a task)</td>
</tr>
<tr>
<td>(P)</td>
<td>Number of intermediate output (&lt; k, v &gt;) Pairs</td>
</tr>
<tr>
<td>(D)</td>
<td>Number of Distinct keys</td>
</tr>
<tr>
<td>(B)</td>
<td>Number of hash buckets (columns)</td>
</tr>
<tr>
<td>(w_1, w_2, \ldots, w_5)</td>
<td>Map execution time component weights</td>
</tr>
<tr>
<td>(w_6, w_7, w_8)</td>
<td>Reduce execution time component weights</td>
</tr>
</tbody>
</table>

During the reduce phase, all reduce tasks are computed by aggregating values from a common key (reduce-comp) and the output from the reduce-tasks are put into final buffer in key-sorted order (reduce-output).

### 3.2.3 Map Phase

To simplify the discussion, we will start with a simple case where there are \(P\) key-value pairs to be inserted into a single hash bucket in the intermediate buffer. Of those \(P\) key-value pairs, there are \(D\) unique keys. We will later expand the model to take into account multiple buckets.

**Modeling map-output Time**

Suppose that an intermediate output with key-value pair \(< k, v \>\) is generated and the bucket already has \(d\) keys prior to this point. The map-output phase needs to determine whether the key \(k\) is already
present in the bucket. If it is, \( v \) is appended to the list of values for key \( k \). Otherwise, a new entry for key \( k \) is inserted into the bucket, while keeping all keys in the bucket sorted.

Determining whether a key \( k \) is already present in the bucket requires searching. Scalable searching algorithm, such as binary search, requires \( O(\lg d) \) time, assuming that keys are already sorted. Inserting a value to an existing list of values takes a constant time (\( O(1) \)), because values do not need to be kept sorted. Finally, the complexity of inserting a new key into a sorted collection of keys may be \( O(\lg d) \) in the best case (e.g. keys are kept in a balanced binary tree, as in Metis) or \( O(d) \) in other cases (e.g. keys are kept sorted in an array and shifted to make room for a new one, as in Phoenix).

While keeping keys in a balanced tree (e.g. B+ Tree in Metis) gives the best asymptotic complexity, searching and insertion are expensive, as they involve pointer de-referencing, recursion, and potentially tree rebalancing. Sorted array provides much faster key searching and insertion, at the expense of worse asymptotic key insertion time. Choosing the right data structure is not trivial since there is a fundamental trade-off that is not easy to reconcile.

Let us further assume that the average key searching, key insertion, and value insertion take \( w_1 \), \( w_2 \), and \( w_3 \) on average. Processing \( <k,v> \) in the map-output phase takes \( w_1 \lg d + w_3 \) if the key was in the bucket, or \( w_1 \lg(d-1) + w_2 d + w_3 \) if the key was not in the bucket. Assuming that the probability of each of \( D \) distinct keys appearing is equal, we can capture map-output time using the following Markov model.

Let \( (d) \) be a state in the Markov model, representing that currently \( d \) out of \( D \) distinct keys have been added to the bucket. Subtracting the last \( <k,v> \) that was processed, the previous state is either \( (d-1) \) if \( k \) was a new key, or \( d \) if \( k \) was not a new key. Since all keys have an equal probability of occurring, the probability of seeing one not among \( d-1 \) keys is \( \frac{D-(d-1)}{D} \), while the probability of seeing one of the \( d \) keys that have occurred is \( \frac{d}{D} \). The resulting Markov model is shown in Figure 3.2.

The bold circle shows the current state, and both circles show possible previous states. Each edge in the Markov model shows the transition probability between two states. The transition probabilities do not add up to one because the transitions originate from different states.

Algebraically, the probability of reaching a state \( (d) \), or \( Pr(d) \), is:

\[
Pr(d) = \begin{cases} 
1 & \text{if } d = 0 \\
0 & \text{if } d > D \\
\frac{d}{D} Pr(d) + \frac{D-d+1}{D} Pr(d-1) & \text{otherwise}
\end{cases}
\]  (3.1)

Figure 3.2: Markov model for map-output time.
By unrolling the recursion in the Markov model, we can find the expected total time needed to complete the map-output phase \( T_{mo} \). Unfortunately, computing this expression is not practical as \( P \) is typically a very large number. Thus, we focus on two special cases derived from the model: worst case and best case map-output time. The worst case occurs when all keys occur early, which implies that all subsequent intermediate outputs will have to search the maximum number of keys (all \( D \) keys). We refer to this case as skewed key occurrence (SKO). On the other hand, the best case occurs when keys occur late. Since we assume that each key has an equal chance of appearing, each key will occur exactly \( \frac{P}{D} \) times before the next distinct key appears. We refer to the latter case as uniform key occurrence (UKO). SKO and UKO provide lower and upper bounds for \( T_{mo} \), under the assumption of uniform key occurrence frequency.

SKO and UKO are also significant special cases because they represent real scenarios. For example, for machine learning and scientific applications [25], typically all keys are emitted in the first map task, and remaining tasks only insert values as all the keys are already inserted into the bucket by the first map task. Such applications exhibit SKO. Other applications with relatively small number of keys also show SKO behavior. UKO is rarer, but may occur in real situations, such as when the input records are already sorted. For example, in word count application, a document may already be sorted in alphabetical order, and in password check application, user IDs and passwords may already be sorted. All other applications can be expected to show map-output performance between those of SKO and UKO.

With a sorted key array under SKO (Phoenix), when keys are initially inserted, we incur a cost of

\[
T_{mo}^{SKO} = w_1 (\lg D! + (P - D - 1) \lg D) + w_2 \frac{D(D + 1)}{2} + w_3 P
\]

Under UKO, each distinct key appears as late as possible, hence keys are uniformly spaced at a distance of \( \frac{P}{D} \). There are as many key insertions and value insertions as in SKO, so the difference is only the amount of time spent for searching keys. Since key insertion is uniformly spaced, \( \frac{P}{D} \) intermediate outputs after the first pair incur \( w_1 \lg 1 \) key searching time, next \( \frac{P}{D} \) intermediate outputs incur \( w_1 \lg 2 \) key searching time, and so on. The last \( \frac{P}{D} - 1 \) intermediate outputs incur \( w_1 \lg D \) key searching time. Therefore, total key searching time can be written as \( w_1 (\frac{P}{D} \lg 1 + \frac{P}{D} \lg 2 + \ldots + (\frac{P}{D} - 1) \lg D) \). This is equivalent to \( w_1 (\frac{P}{D} \lg D! - \lg D) \). Therefore, the total map-output time in UKO is:

\[
T_{mo}^{UKO} = w_1 (\frac{P}{D} \lg D! - \lg D) + w_2 \frac{D(D + 1)}{2} + w_3 P
\]

Asymptotically, \( O(x^2) > O(\lg x!) > O(x) \). Furthermore, with Stirling’s approximation [76], \( \ln x! \approx x \ln x - x \), which means that for a large \( D \), \( \lg D! \approx D \lg D - \frac{D}{\ln 2} \). Now let us consider Equation 3.2
and 3.3. When \( P = D \) (all intermediate output pairs have distinct keys), then key insertion time dominates, i.e. \( T_{mo} = O(D^2) \). However, when \( P >> D \) (key insertions are rare), key searching time dominates. For SKO, \( T_{mo} = O(P \lg D) \). For UKO, \( T_{mo} = O(PD \lg D) = O(P \lg D) \).

With key management using B+ Tree data-structure (Metis), key insertion takes a \( \lg \) of the number of keys already inserted in the tree. Hence, Equations 3.2 and 3.3 change slightly to:

\[
T_{mo}^{SKO} = w_1 (\lg D! + (P - D - 1) \lg D) + w_2 \lg D! + w_3 P
\]  
(3.4)

\[
T_{mo}^{UKO} = w_1 (\frac{P}{D} \lg D! - \lg D) + w_2 \lg D! + w_3 P
\]  
(3.5)

This leads to the following observation:

**Observation 1** The map-output time highly depends on the number of distinct keys \( D \) and the number of intermediate output pairs \( P \). With a sorted array, if \( \frac{D}{P} \to 1 \), then \( T_{mo} = O(D^2) \) for both SKO and UKO. If \( \frac{D}{P} \to 0 \), then \( T_{mo} = O(P \lg D) \) for both SKO and UKO. With a binary tree, \( T_{mo} = O(D \lg D) \) if \( \frac{D}{P} \to 1 \), or \( T_{mo} = O(P \lg D) \) if \( \frac{D}{P} \to 0 \), for both UKO and SKO.

**Significance:** The significance of this observation is that it points out that a key performance factor in shared memory MapReduce is the ratio of the number of distinct keys to number of intermediate output pairs. Since the ratio is affected by the inherent computation algorithm, input content, and chunk size, the observation explains why shared memory MapReduce program performance is sensitive to these factors. In contrast, a program written in PThreads or OpenMP is sensitive to data layout and memory access pattern, rather than input content. Performance tuners can use this insight to improve the performance, e.g. decreasing the number of pairs by increasing the map-task size in some cases.

Furthermore, the observation points out avenues for optimizing shared memory MapReduce. For example, when keys are comparable to total output pairs (\( D \approx P \)), key insertion dominates the execution time. One way to reduce the dominance of key insertion is to use multiple hash buckets so that the number of keys per bucket decreases. In addition, Metis’ use of B+Tree produces better performance under a large number of distinct keys, but produces worse performance under a small number of distinct keys, and incurs significantly more time per instance of key searching and insertion. This interesting trade-off will be quantified in and discussed in Section 3.5.1.

Now let us contrast the map-output time for SKO and UKO (Equation 3.2 and 3.3). Using Stirling’s approximation and taking the difference between them, we obtain:

\[
T_{mo}^{SKO} - T_{mo}^{UKO} = w_1 \left( \lg D! + (P - D - 1) \lg D - \left( \frac{P}{D} \lg D! - \lg D \right) \right)
\]

\[
= \frac{w_1}{\ln 2} (P - D)
\]  
(3.6)

which leads to the following observation:

**Observation 2** Another key performance factor is the order in which keys occur. The gap in performance between the worst case, where all keys occur early in the intermediate output bucket, and the best case, where key occurrence is uniformly spaced, grows with the number of intermediate output pairs but decreases with the number of distinct keys, i.e. proportional to \( P - D \).
Significance: The significance of the observation is providing a link between the performance of shared memory MapReduce program and the order in which keys appear. The link may be new to programmers, as input content is not typically a major performance factor in PThreads or OpenMP programs. Using this information, performance tuners and programmers can optimize their programs, for example by rewriting the input or programs such that keys are emitted in a more uniform order. We also note that the observation is only dependent on the ordering of keys, which is a program/input characteristic, but is independent of the hash bucket data structures.

Impact of Multiple Buckets and Multithreading

The previous section assumes a single hash bucket. Using multiple hash buckets reduces the number of distinct keys per hash bucket, which in turn reduces map-output time, assuming everything else remains the same. If we use a large number of buckets such that only one key maps to a single bucket, the map-output time is reduced to $O(P)$. However, in practice, using too many buckets increases cache and memory footprint and reduces cache locality, hurting performance by increasing the cost of each key searching and insertion. Therefore, the number of buckets is a critical performance factor.

Finding an optimum number of buckets is difficult. However, we can estimate at which number of hash buckets $B$ the probability of key collisions becomes small. To solve this specific question, we note that the probability of finding two keys that map into the same bucket is a birthday paradox problem. Given a particular key, the probability of another key mapping to the same bucket is $\frac{1}{B}$. Since there are $\binom{D}{2}$ possible pairs, the probability to find any two keys mapping to a single bucket is $\binom{D}{2}\frac{1}{B} = \frac{D^2 - D}{2B}$. Suppose that we want to keep the probability $\leq \frac{1}{2}$. Rearranging the inequality, the minimum number of buckets is $\geq D^2 - D$.

However, caution is needed. In some cases, we have no apriori knowledge of how many distinct keys there would be in the program, hence there is a risk of choosing too many buckets. Too many buckets not only lead to lower cache locality, but as we will demonstrate in Section 3.2.4, also increase the reduce phase time.

Finally, we want to model the multithreading effect on the map phase. As discussed in Section 4.1.2, map worker threads do not share data or synchronize until the end of the map phase. Thus, there are only two potential effects of using multiple map threads: the cost of fetching tasks from a shared task queue, and the impact of false sharing in hash bucket structures between threads. The latter is easily avoidable through padding key-value pairs so that they fit multiples of cache line size (employed by both Phoenix and Metis). The former depends on how map tasks are assigned to map worker threads: statically or dynamically. While static task assignment requires no synchronization, it leads too easily to load imbalance. Dynamic assignment incurs synchronization but ensures load balance. Hence, both Phoenix and Metis adopt dynamic task assignment. When map threads simultaneously attempt to fetch map tasks from the task queue, mutual exclusion between threads is required for correctness, which is achieved using a lock or atomic instructions. Since mutual exclusion implies sequentiality, task fetching time is simply number of tasks multiplied by the average time to fetch a task.

Let $w_4$ denote the average time taken to hash a key, and $w_5$ denote the average time to fetch a map task. The total contribution of hashing and map task fetching on map-output time are $w_4 P$ and $w_5 (\frac{NI}{C})$. 

37
respectively, where $N_i$ is total number of input records and $C$ is map task size (Table 4.1). Taking into account the effects of the number of hash buckets and task fetching, Equations 3.2 and 3.3 become:

$$T_{SKO}^{mo} = w_1 \left( B \log \frac{D_B}{B} + (P - D - 1) \log \frac{D_B}{B} \right) + w_2 B \frac{D_B}{B} \frac{(D_B + 1)}{2}$$

$$+ (w_3 + w_4) P + w_5 \left( \frac{N_i}{C} \right)$$

(3.7)

$$T_{UKO}^{mo} = w_1 \left( \frac{P}{D} \log \frac{D_B}{B} - \log \frac{D_B}{B} \right) + w_2 B \frac{D_B}{B} \frac{(D_B + 1)}{2}$$

$$+ (w_3 + w_4) P + w_5 \left( \frac{N_i}{C} \right)$$

(3.8)

### 3.2.4 Reduce Phase

The Reduce phase consists of reduce-comp and reduce-output sub-phases. In reduce-comp, each worker thread calls `reduce()` function to perform reduction or aggregation of all values of a key (or keys) assigned to it. In reduce-output, the output is written to a final output buffer, either unsorted or sorted based on keys.

Recall from Figure 3.1 that a map worker thread works on buckets across a row. In contrast, a reduce worker thread works on buckets across a column. Each column forms a reduce task, and reduce tasks are dynamically assigned to reduce worker threads. Each column consists of many buckets containing the output of map worker threads. With $B$ columns and $N_{redt}$ reduce threads, if load is balanced, each thread will be assigned $\frac{P}{N_{redt}}$ columns to compute. Therefore, the total number of buckets assigned to a reduce worker thread is equal to the number of columns assigned to a reduce worker thread multiplied by the number of map threads, i.e. $\frac{BN_{mapt}}{N_{redt}}$. With dynamic task assignment, task fetching requires mutual exclusion, and the synchronization overhead scales with the number of tasks $B$ and the number of reduce threads $N_{redt}$.

Let us denote reduce-comp time for a key-value pair as $w_6$, and the time to fetch a task from the task queue as $w_7$. Assume key-value pairs are distributed evenly across all buckets, then each bucket will contain $\frac{P}{BN_{mapt}}$ key-value pairs. Therefore, the reduce-comp time is:

$$T_{rc} = \frac{P}{BN_{mapt}} \left( w_6 \frac{BN_{mapt}}{N_{redt}} \right) + w_7 B$$

$$= w_6 \frac{P}{N_{redt}} + w_7 B$$

(3.9)

leading to the following observation:

**Observation 3** The asymptotic complexity of the reduce-comp time is $O\left( \frac{P}{N_{redt}} + B \right)$. It increases with the number of intermediate output pairs and hash buckets, but decreases with the number of reduce worker threads.

**Significance:** Equation 3.9 and Observation 3 show that the relative contribution of $w_7$ on $T_{rc}$ increases when more buckets and more reduce threads are used. This creates a dilemma: on one hand
increasing the number of buckets reduces map-output time, but on the other hand it increases reduce-comp time. Therefore, keeping the reduce task queue overhead low is very important as it enables the map phase to use a larger number of buckets and run faster.

In the reduce-output sub-phase, all values with a common key are aggregated. The final output, consisting of a list of key and aggregate value pairs, may or may not be sorted. Phoenix and Metis use variants of parallel merge sort algorithm which have a asymptotic complexity of $O(D \lg D)$, hence the reduce-output time is:

$$T_{ro} = w_8 \frac{D \lg D}{N_{redt}}$$  \hspace{1cm} (3.10)

where $w_8$ is the average time to perform one merge sort comparison and data movement.

### 3.3 Model-Driven Study

In this section, we will use the model derived in the previous section to gain insights about how various factors affect shared memory MapReduce performance, first using asymptotic analysis and then quantitative analysis based on empirically-derived parameter values.

#### 3.3.1 Asymptotic Analysis

Table 3.2 shows the asymptotic time of various shared memory MapReduce phases for three cases: small number of buckets and equal number of keys and key-value pairs (Phoenix on first row, Metis on second row), small number of buckets and number of keys hugely exceeding number of key-value pairs (third row), or a large number of buckets (fourth row). For now, let us assume that $N_{mapt} = N_{redt}$.

The table reveals several interesting insights. First, when the number of buckets is small, map-output time dominates the total execution time when $D = P$, because $O(D^2) > O(D \lg D) > O(P)$, and also when $D << P$ because $O(P \lg D) > O(D \lg D)$ and $O(P)$.

However, when the number of buckets is large, i.e. $B \approx D^2$, the opposite occurs. Reduce-comp now dominates execution time because $T_{rc} = O(P + D^2) >> T_{mo} = O(P)$, and also $T_{rc} = O(P + D^2) > T_{ro} = O(D \lg D)$. Only if the number of pairs is in the order of the square of the number of keys, i.e.

![Table 3.2: Comparing asymptotic complexity of various phases.](image)

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Case} & T_{mo} & T_{rc} & T_{ro} \\
\hline
\text{P: Small } B, \frac{D}{P} \to 1 & O\left(\frac{D^2}{N_{mapt}}\right) & O\left(\frac{P}{N_{redt}} + B\right) & O\left(\frac{D \lg D}{N_{mapt}}\right) \\
\text{M: Small } B, \frac{D}{P} \to 1 & O\left(\frac{D^2}{N_{mapt}}\right) & O\left(\frac{P}{N_{redt}} + B\right) & O\left(\frac{D \lg D}{N_{mapt}}\right) \\
\text{Small } B, \frac{D}{P} \to 0 & O\left(\frac{P}{N_{mapt}}\right) & O\left(\frac{P}{N_{redt}} + B\right) & O\left(\frac{D \lg D}{N_{mapt}}\right) \\
\text{Large } B & O\left(\frac{P}{N_{mapt}}\right) & O\left(\frac{P}{N_{redt}} + B\right) & O\left(\frac{D \lg D}{N_{mapt}}\right) \\
\hline
\end{array}
\]


\( P = O(D^2) \), then the map-output time has the same asymptotic complexity as the reduce-comp time. Therefore, we can conclude that:

**Observation 4** Increasing the number of hash buckets reduces map-output time complexity from \( O(P\lg D) \) to \( O(P) \), but increases reduce-comp time, to a point where the reduce-comp time may dominate execution time.

So far, our analysis has focused on asymptotic behavior, which is unable to take into account the effect of relative parameter values. Next, we will rely on quantitative analysis to determine which time components become dominant and which factors produce performance bottlenecks.

### 3.3.2 Quantitative Analysis

**Map Time**

Figure 3.3(a) shows map-output time for both SKO and UKO for Phoenix-based model as the number of keys increases along the x-axes, while the number of intermediate output key-value pairs stays fixed at 1 million. Both axes are shown in a log scale, and the curves are normalized to the map-output time for UKO. The curves are obtained from Equation 3.3 and 3.2 using empirically-derived parameter values: \( w_1 = 110, w_2 = 2.95, w_3 = 29, w_4 = 47, w_5 = 2500 \) (Section 3.5.1). Note that parameter values are specific to the platform we used, hence the value of quantitative analysis lies in the performance trends, not the exact performance numbers.

From the figure, we can observe that for both UKO and SKO cases, map-output time increases with a mild slope from 1 to 100,000 keys, then increases with a steep slope from 100,000 to 1 million keys. From our model, we know that the mild slope portion of the curve is when the map-output time is dominated by key searching, while the steep slope portion is when the map-output time is dominated by key insertion. These portions correspond to the two cases stated in Observation 1: one where \( D \) is much smaller than \( P \), and another where \( D \) is approaching the value of \( P \).

We also plot map-output time for Metis-derived model with empirically derived weights \( w_1 = 175, w_2 = 15.90, w_3 = 29, w_4 = 47, w_5 = 2500 \) (Figure 3.3(b)). How these parameter values are derived are discussed in Section 3.5.1. We notice that unlike the Phoenix model, Metis model’s map-output time increases with flattening slope as the number of keys increases. This is because Metis’ data structure has a logarithmic complexity for key insertion. Moreover, the difference in map-output time between UKO and SKO is proportional to \( P - D \), as we observe in Observation 1.

Based on the observation that Metis model shows a slower increase in map-output time compared to Phoenix as the number of keys increases, Metis may seem the preferred choice. In Figure 3.3(c), we plot map-output time for both Metis and Phoenix under SKO, normalized to Phoenix’s map-output execution time. We notice that when the number of keys exceeds 10,000, Metis hugely outperforms Phoenix. However, the opposite is true when the number of keys is less than 10,000: Phoenix significantly outperforms Metis (magnitude not readily apparent on a log scale chart). A similar trend applies for UKO, but not shown due to space limitation. We should emphasize that what matters here is not the number of keys, but rather the ratio between number of keys and pairs. Phoenix outperforms Metis as long as \( D << P \), and underperforms Metis when \( \frac{D}{P} \to 1 \). To summarize:
Figure 3.3: Normalized map-output time for $10^6$ intermediate output pairs in Phoenix and Metis (a), UKO vs. SKO in Phoenix (b) and Metis (c), % difference of map-output time of SKO vs. UKO (d), reduce-comp time for various number of threads and buckets (e), and Map vs. Reduce time under SKO with 1 million pairs, 250,000 distinct keys, and 8 threads (f).
**Observation 5** Comparing Phoenix and Metis, their choice of data structures produce the following performance difference: Phoenix’s map-output time is lower than Metis when $D << P$, but becomes higher as $\frac{D}{P} \rightarrow 1$.

*Significance:* The observation is interesting as it could not be derived using an asymptotic analysis, which would always favor Metis. The observation is also important for programmers and performance tuners because our analytical model can help in deciding which data structure to use given an algorithm with known number of keys and output pairs, even before writing code for the algorithm using different MapReduce runtime systems.

Figure 3.3(b) and (c) also show that initially SKO and UKO differ, but as the number of keys increases, they converge. Note, however, that we should not conclude that the difference between SKO and UKO is minor. In Figure 3.3(d), we plot the difference of map-output time between SKO and UKO, divided by the map-output time of UKO, as the number of keys is increased from 10 to 1 million. The figure shows that the performance gap between SKO and UKO is significant (> 10%) when the number of keys is up to 10,000, and can be as high as 60% for both Phoenix and Metis. This is consistent with Observation 2 which states that UKO and SKO performance difference is proportional to $w_1$ (average key search time) and to $P - D$. However, Metis’ performance gap is consistently higher compared to Phoenix, because of the higher performance weight factor associated with key searching.

Observation 5 is critical for performance because the number of keys and key occurrence order are algorithm-dependent characteristics and may not be changed in some cases. For example, linear regression will always have only five keys.

**Reduce Time**

Figure 3.3(e) shows reduce-comp time as a function of number of hash bucket columns (reduce tasks) for various numbers of reduce threads under Phoenix (similar trends and insights are observed for Metis but are not shown due to space limitation). The reduce-comp time is normalized to the single-threaded case with 1,000 reduce tasks. According to Equation 3.9, the reduce-comp time is a weighted sum of $\frac{P}{N_{map}}$ and $B$. We use $w_6 = 13$ and $P = 10^6$. At 1,000 buckets (leftmost points in the curves), the reduce-comp time is halved as the number of map threads doubles. As the number of hash buckets increases, the contribution of $w_7 B$ on reduce-comp time increases, hence all lines show an increasing trend. However, the slope is steeper for a higher number of threads because synchronization cost increases as more threads are involved (Section 3.5.1 shows how $w_7$ is affected by the number of threads), leading to the following observation:

**Observation 6** While using more reduce worker threads decreases reduce-comp time, the benefit disappears and reverses as the number of reduce tasks (i.e. hash-buckets) increases. Thus, with a sufficiently high number of buckets, using fewer reduce worker threads yields better performance.

*Significance:* The observation opens up an avenue for tuning MapReduce runtime systems: adapting the number of reduce threads and map threads differently (in prior studies they were always equal), and gives a guideline as to how many reduce worker threads should be used.
Comparing Map and Reduce Time

Figure 3.3(f) plots the map time, reduce time, and their sum, as the number of buckets varies. The numbers are normalized to the total execution time with 1K buckets. The figure shows that as the number of buckets increases, map time declines due to fewer hash bucket collisions, but reduce time increases due to reduce task queue contention. We observed the same trends when we use Metis model at different magnitude. This result was captured in Observation 4. Overall, in this section we have shown that a combination of asymptotic and quantitative analyses can explain key MapReduce performance phenomena and contrast two different implementations of MapReduce.

3.4 Experimental Environment

Machine Configurations. For empirical evaluation, we use Intel Xeon X5560 8-core processor with Linux kernel v2.6.31. This system has two chips with each chip having four cores. Each core has a 2.80GHz frequency, a 32KB private L1 cache, a private 256KB L2 cache, and an 8MB L3 cache shared by four cores. Each core also has two hardware contexts, but we do not use one of them for our experiments to avoid simultaneous multithreading effects. The system is lightly loaded for all the experiments. Only the application, regular OS processes and daemons, run. Each run was repeated ten times, the highest and lowest values were removed, and the average was taken and reported, to avoid measurement biases [43].

Applications. We use applications and inputs that come with Phoenix and Metis, which are already well-optimized by Phoenix developers [51, 62]. They include Histogram (HG) with a 400MB image input, KMeans (KM) with 20 clusters and one million points, Linear Regression (LR) with a 100MB input file, Matrix Multiply (MM) with a 8K×8K dimension, String Matching (SM) with a 100MB input, and Word Count (WC) with 100MB input file (input C). We also generated two more inputs with the same number of words: input A (identical words) and B (all different words). We also constructed other inputs for WC for other purposes. For all the applications, the default number of hash buckets is 32K. The inputs are selected to be large enough to have a substantial execution time but without incurring much I/O activity.

3.5 Experimental Evaluation

3.5.1 Microbenchmark and Validation Results

In this section, we validate the model we discussed in Section 3.2. All the equations in our model assume that $w_1, w_2, \ldots, w_8$ are parameters with constant values that give weights to various terms of the execution time. If the equations have captured all important variables correctly, then all the parameter values derived empirically should remain constant as the variables’ values vary. Figure 3.4 and 3.5 show empirically-derived parameter values in number of clock cycles, as all key variables are varied. The parameter values were derived from synthetic benchmarks, which are variants of the
word-count application that we specifically designed for this purpose. We delimit the Map and Reduce phases with hardware counter collection, so that we can measure their execution time separately.

We will first discuss the methodology to infer the weight factor (parameter) values for Map phase, and the Reduce phase next. For \( w_4 \) (weight factor for hashing), we take out the hashing function and take the total execution time for hashing various number keys, and divide it by the number of pairs to get \( w_4 \). We plot \( w_4 \) values separately for the same key and for unique keys.

Then we set the number of hash buckets to one, and remove the hashing function for keys in order to exclude its effect when measuring other parameter values. For \( w_3 \) (weight factor for value insertion), we emit 10 to 1 million pairs having the same key, and divide the total time by the number of pairs to get \( w_3 \).

To estimate \( w_1 \) (weight factor for key search) for a given number of keys \( D \), we first insert \( D \) keys and subsequently search for 1 million randomly-generated keys from among already-inserted keys. The total time is subtracted by the value insertion time and then divided by \( 10^6 \lg D \). The value of \( w_1 \) is not affected by key insertion as it is mostly excluded and amortized over a large number of key searches. We measure \( w_2 \) (weight for key insertion) by inserting a varying number of keys and subtracting the time for key search and value insertion, and then dividing it by the number of keys (for Phoenix implementation) or \( \lg \) of keys (for Metis implementation). Finally, \( w_5 \) (weight factor for map-task queue overhead) is estimated by creating 10 to \( 10^3 \) map tasks with NULL map function and executing them with 1–8 threads.

For \( w_7 \) (reduce task queue overhead factor), we vary the number of buckets from 1K to 128K, but leave the buckets empty (zero pairs and keys). This removes reduce-output (no output produced) and value aggregation part of the reduce phase. For \( w_6 \) (weight for aggregating values), the number of map and reduce worker thread is set to one, the number of keys and hash bucket to one, but the number of pairs is varied. In one case we schedule map and reduce threads on same core and in another case we schedule them on different cores to measure the effect of cache coherence (transferring values across caches) on value aggregation. \( w_8 \) is obtained by keeping the number of hash buckets fixed and varying the number of threads and keys. The resulting execution time is subtracted by the reduce-comp component and then divided by \( \frac{D \lg D}{N_{redt}} \).

Figure 3.4 and 3.5 show all weight factors for Phoenix and Metis. The values of the parameter weight factors reveal the relative importance of performance factors. We can make several interesting observations. First, for both Phoenix and Metis, \( w_1 > w_4 > w_3 > w_2 \), suggesting that a single instance of key searching is the most expensive, followed by a single instance of key hashing, value insertion, and key insertion. This is likely because of value insertion and key insertion enjoy better spatial cache locality than key searching. Key searching jumps over different elements, while key insertion only involves shifting adjacent elements, and value insertion involves adding adjacent to previously accessed elements. Second, since hashing is more expensive than value insertion, it suggests that hashing should be of higher priority for optimization than value insertion. Finally, \( w_5 \) has a large value but is multiplied by a small value (there are only 1K map tasks), hence it is not a bottleneck during the map phase.
Figure 3.4: Empirically-derived values of $w_1, w_2, w_3, w_4,$ and $w_5$ (map-phase).
Now, we discuss the difference between Phoenix and Metis for a given weight factor (Figure 3.4). Phoenix and Metis show similar weight factors for hashing, value insertion and multi-threading overhead, on our test platform. The similarity may extend to other platforms because of similar map task queue design, array data structure for storing values, and an identical hash function. However, key search and insertion are vastly different because of the choice of data structures. Phoenix uses a binary search tree implemented as a sorted array, resulting in low key search time and good key insert cache locality, at a cost of \(O(D)\) key insertion time. In contrast, Metis uses B+Tree which provides \(O(lg D)\) for both key search and key insertion, but suffers from poor cache locality from pointer dereferencing while searching for keys and splitting buckets to balance the tree. Consequently, \(w_1\) and \(w_2\) for Phoenix are smaller than Metis, and Phoenix outperforms Metis until the number of keys is large enough to offset the effect of lower \(w_1\) and \(w_2\).

Next, we discuss weight factor for the Reduce Phase components. \(w_6\) (weight factor for value aggregation) is low for both Metis and Phoenix, even when values must be fetched from other caches. The reason is that value aggregation inherently has good spatial locality if values are kept in an array, enabling the cache’s sequential prefetchers to be effective in bringing data early into the cache. This argues against implementing the value list using data structures that reduce spatial locality, such as linked lists. Second, \(w_6\) is lower than \(w_4\) but multiplied with the same variable \(P\). This suggests that improving hashing function performance is more important compared to improving value aggregation performance.

The figure shows that \(w_7\) (weight factor for bucket overhead) is roughly constant but its value is specific to the number of threads. Since the relationship of bucket overhead and number of threads is
affected by many factors (coherence misses, synchronization implementation, etc.), and is difficult to express mathematically, we simply use different \( w_7 \)'s for different numbers of threads.

We also notice that \( w_7 \) in Phoenix is significantly larger than in Metis. Analyzing and comparing Phoenix and Metis code, we found that Phoenix uses a traditional task queue implementation where each thread competes for the head of the queue using a lock. In contrast, Metis uses atomic update of a counter variable that mimics the behavior of a task queue, incurring less synchronization and fewer dynamic instructions. This difference in performance emphasizes that task queue synchronization should be avoided as much as possible.

So far, we have shown that our model capture key variables affecting performance of the Map and Reduce phases. However, one limitation of our model is that it does not model the reduce-output time very well. Instead of showing a constant value, \( w_8 \) is affected by the number of keys and the number of threads. \( w_8 \) in Metis is lower than in Phoenix because Metis uses sampling-based parallel merge sort which utilizes all cores better compared to Phoenix’s regular parallel merge sort. The higher \( w_8 \) value on a larger number of threads indicates sub-linear speedup (Metis), while a lower value on a larger number of threads indicates super-linear speedup (Phoenix). However, reduce-output time may be an optional step and is important only when number of keys is large enough such that reduce-output time becomes larger than reduce-comp time. Although our reduce-output time modeling is less accurate, we can still use it to predict the trends correctly.

Overall, Figure 3.4 and 3.5 have validated that our model has captured mathematically the key
variables affecting performance. For further validation, we use the model to generate estimates of Map and Reduce phase execution time, and compare it against the actual measured execution time. We generated a random input file consisting 20 million pairs with varying number of keys. Figure 3.6(a) and (b) show that under SKO or UKO, our model captures performance trends closely, although not the exact magnitude in UKO. We compare the predicted map-output time (Figure 3.6(c)) and reduce-comp time (Figure 3.6(d) and (e)) with the actual execution time across different number of threads. The figures show a close match between the time predicted by the model and the time measured on the platform, across different number of threads and hash buckets. Comparing Figure 3.6(d) and (e), a higher number of hash buckets makes fewer number of threads preferable (compare 4 vs. 8 threads with 64K buckets), matching Observation 6. Finally, Figure 3.6(f) shows that the number of coherence misses in the reduce phase is higher in Phoenix than in Metis, and increases with a higher number of threads. This confirms coherence misses as a contributor to the higher value of $w_7$ in Phoenix compared to in Metis.

3.5.2 Empirical Results for Real Applications

In this section, we present experimental results from running real applications using Phoenix and Metis libraries on the test platform described in Section 3.4. We will demonstrate the role of our model in: (1) predicting performance trends in real applications, and (2) explaining the reasons behind such trends.

Map Phase Performance Results

Figure 3.7(a) shows the SKO map-output time for Phoenix and Metis, normalized to Phoenix with 25K keys. The input to WC application is a text document containing 20 million words with varying number of distinct keys. There are two important observations. First, the figure shows that Phoenix’s map-output time increases with increasing slope with the number of keys, while Metis’s map-output time increases with decreasing slope with the number of keys. Second, Phoenix’s map-output time is lower than Metis’ until the number of keys reaches 125K. These two observations are both predicted...
Figure 3.8: Map-output time for various number of threads and hash buckets Phoenix (a), and Metis (b)

(Figure 3.3(c)) and explained (Observation 5) by our model. They are an outcome of the asymptotic key search and insertion cost arising from the choice of data structure (array vs. B+tree) for the hash buckets. Knowing these help programmers in choosing the appropriate MapReduce library early in their project.

Next we consider the effect of UKO vs. SKO. Figure 3.7(a) and (b) show the map-output time for Phoenix and Metis, respectively. The figure confirms the model’s prediction that key ordering is a significant performance factor, with UKO significantly outperforming SKO (Observation 2 and Figure 3.3(a)&(b)). Furthermore, the performance gap between UKO and SKO is higher in Metis than in Phoenix, again consistent with our model (Figure 3.3(d)).

Next, we discuss the effect of the number of hash buckets and threads on the map phase. Figure 3.8(a) and (b) show map-output time with varying number of buckets (shown across groups of bars as indicated by numbers on the x-axes) and number of map threads (shown across different bars in a group). For both Phoenix and Metis, all the bars are normalized correspondingly to a case where one hash bucket is used. Let us first focus on the first bar of each group, corresponding to single-thread execution (Thr 1). We can see that increasing the number of hash buckets from one to 32K decrease the map-output time for most benchmarks (HG, KM, LR, WC-B, and WC-C). Only MM and WC-A do not see any improvement, because MM completely bypasses key insertion, while WC-A only has one unique key. On another extreme, WC-B shows more than 100× improvement from using 32K buckets compared to 1 bucket in Phoenix, hence the bars are not visible in the figure. For the remaining benchmarks, the map-output time improvement of using 32K buckets vary. Our model explains that the improvement from reduced hash collisions stops when the number of buckets is significantly larger than the number of distinct keys (e.g. $B \geq D^2$).

Next, let us compare Thr1 bars across different groups representing different number of hash
Figure 3.9: Reduce phase execution time for various number of threads and hash buckets Phoenix (a), and Metis (b)

buckets. WC-C (in Phoenix and Metis) and WC-B (in Metis) are the only benchmarks that enjoy map-output time reduction as the number of hash buckets increases from 32K to 128K. Again, the reason is explained by our model: these applications are the ones that have thousands to millions of distinct keys, hence the increase in number of hash buckets continues to improve performance.

Finally, let us compare bars in each group, representing the effect of increasing the number of map threads. It is straightforward to observe that increasing the number of threads reduces map-output time inverse-proportionally as the computation load is distributed across more threads.

**Reduce Phase Performance Results**

Now, we present experimental results for reduce phase. Figure 3.9(a) and (b) show the reduce phase execution time as the number of hash buckets and threads vary, normalized to a single thread case with 32K hash buckets. There are several major observations. First, comparing across groups of bars, increasing the number of hash buckets from 32K to 128K increases reduce time (notably for HG, LR, KM, and SM). This phenomena is predicted and explained in Observation 3, which shows how the number of hash buckets (multiplied by weight factor $w_7$) directly influences the reduce time. The reason is due to synchronization overheads for assigning reduce tasks to reduce worker threads.

Second, let us compare bars within each group, which represent the effect of increasing number of reduce worker threads. There are two opposite outcomes from using a larger number of reduce threads: increased reduce time for some benchmarks (HG, LR, KM, and SM) or decreased reduce time for others (WC-B and WC-C). For some benchmark (HG, LR, KM, and WC-A), both outcomes appear, for example going from one to two threads in HG decreases reduce time by 12%, but going from two to four/eight threads the reduce time increases. We note that this performance trend is
captured and explained by our model (Observation 6), which states that using more reduce worker threads allow more parallelism in aggregating values (during reduce-comp) and in sorting the final output (during reduce-output) but at the same time incurs a higher task queue overhead (during reduce-comp), especially when the number of hash buckets is high. This again points out to the need to choose the number of reduce worker threads and map worker threads differently (also taking into account the number of hash buckets), unlike prior studies which always kept them equal.

Finally, WC-B and WC-C enjoy a significant improvement from increased number of threads. A key reason for this is in the output merging in reduce-output phase, the sorting algorithm benefits from a larger number of threads. This is evident in the weight factor $w_8$ that decreases with the increase in number of threads (Figure 3.5(e) and (f)). However, due to the non-linear nature of $w_8$, we cannot capture the exact improvement precisely in our model, but knowing how $w_8$ responds to number of threads in Phoenix and Metis still helps in successfully predicting this trend.

**Comparing Map and Reduce Phase**

We also investigate how the number of hash buckets affects both Map and Reduce phases simultaneously. Figure 3.10 shows map vs. reduce time as fractions of the total execution time for Phoenix and Metis respectively. The figure shows that the fraction reduce time increases with the increase in the number of hash buckets for many applications. Consistent with Observation 4, increasing hash buckets has the opposite effect on map time (reducing it) due to fewer hash collisions and faster key searching, versus on reduce time (increasing it) due to a higher task queue overhead. And for some benchmarks, reduce time may dominate the execution time.

We point out that the fraction of reduce time in Phoenix is larger than in Metis (e.g. HG and LR). This is due to the reduce task queue overhead factor ($w_7$) being higher in Phoenix than in Metis (Figure 3.5). However, Phoenix’s task queue structure is more generic and may be more scalable.
### 3.6 Application Classification

We have demonstrated that the most important performance factor for MapReduce performance is the number of intermediate key-value pairs and number of distinct keys. Interestingly, both metrics are specific to the algorithm and input. Consequently, in some cases they are already known to programmers prior to coding. Figure 3.11 shows a map of where applications in the Phoenix package fall (not drawn to scale). The arrows show the range of parameter values for various applications.

![Figure 3.11: Classification of MapReduce Applications. Not drawn to scale.](image)

At the bottom right corner are applications with a relatively large number of distinct keys but with a relatively small frequency of distinct keys. These applications will have key insertion dominating the map-output time (Observation 1) as well as a high Reduce time (Observation 3). This category includes word count (WC-B), matrix multiplication (MM), and string matching (SM). Effective tuning for these applications includes not sorting the final output, or skipping the Reduce phase altogether (e.g. in Phoenix, MM directly writes to a matrix output). Applications without natural keys can use such tuning techniques.

At the top left corner are applications with a small number of distinct keys but a high frequency of distinct keys, including word count (WC-A) and linear regression (LR). These applications’ performance will be dominated by key searching in the Map phase (Observation 1), while the Reduce phase will not be a bottleneck. Such applications should use a relatively small number of hash buckets which keeps reduce time low (Observation 3) and a small number of reduce worker threads (Observation 6) which produces similar effects.

At the top right corner are applications with a high number of keys and high frequency of distinct keys, which include HG and WC with certain inputs. For such applications, all components of
execution time are stressed, hence the number of buckets should be fine tuned to keep both map and reduce time low (Observation 4 and 3).

3.7 Related Work

MapReduce programming model was proposed by Google [28], and has been adopted by various companies, including Amazon, Yahoo, and Facebook. The typical MapReduce system used is cluster-based (or disk-based) Hadoop and its variants [6, 20, 69]. MapReduce has also been extended to run on various platforms, including GPUs [32, 57, 35], FPGAs [55], and IBM Cell BE [37].

Shared memory MapReduce has also been developed as an alternative to cluster/disk-based MapReduce. In shared memory MapReduce, the intermediate output is stored directly in memory (rather than in the file system), shared by all threads. Shared memory MapReduce is more suitable for computation that fits in the main memory, but due to avoiding I/O, it outperforms disk-based implementation (e.g. Hadoop) significantly [26, 36]. This work focuses on shared memory MapReduce.

The first shared memory MapReduce runtime system (Phoenix) was developed by Ranger et al. [51]. They demonstrated Phoenix’s performance scalability on contemporary CMP systems. Yoo et al. [62] further optimized Phoenix runtime system, which has been used in various other studies [24, 26, 36, 38, 39, 63]. Concurrent to our work, Talbot et al. [58] developed Phoenix++ runtime system, C++ version of Phoenix. Phoenix++ has the ability to adjust some parameters to suit the workload characteristics. Another shared memory MapReduce runtime system, Metis, was developed by Mao et al. [41], to overcome some of the performance problems associated with Phoenix. While the performance and scalability of shared memory MapReduce have been demonstrated, detailed understanding of what factors affect shared memory MapReduce performance has not been discovered, which is the focus of this work.
This chapter is organized as follows: Section 4.1 provides an overview to in-memory MapReduce programming model. Section 4.2 provides the overview of our in-memory MapReuse runtime system. Section 4.2.2 describes the design and implementation of in-memory MapReuse runtime system. Section 4.3 discusses our analytical and experimental results. Section 4.4 discusses related work.

4.1 Background

The background information about MapReduce programming model and In-Memory MapReduce is the same as described in Chapter 3, but repeated here for easy referencing.

4.1.1 MapReduce Programming Model

MapReduce program execution is divided into two phases: map and reduce. The map phase is a data-parallel phase where input records, grouped into map tasks, are distributed across map worker threads. Each map worker thread performs computation by calling a user-defined map function. The map function produces output in the form of <key, value> pairs, and the output is stored in the intermediate storage (i.e. files or in-memory buffers). The reduce phase reads from the files or buffers to perform user-defined reduction operation on the output of the map phase. Each reduce worker thread is assigned a group of keys and calls a user-defined reduce function, which aggregates all values having the same key. The output of the reduce function are <key, aggregate-value> pairs, which are then sorted to produce the final output. Thus, map workers achieve data parallelism, while reduce workers perform parallel reduction.

An example of MapReduce application is word count, which counts the number of occurrences of each distinct word in a text document. In word count, each map worker takes a chunk from the text document input, and for each word that it encounters, it produces a <word, 1> pair as intermediate
output. At the completion of the map phase, the reduce phase is started. Each reduce worker works on a specific key (a distinct word) to sum up the list of values (counts) associated with the key.

### 4.1.2 In-Memory MapReduce

In-Memory MapReduce (IMMR) keeps the intermediate buffer as a data structure stored in the main memory, where any map worker thread can write to, and any reduce worker thread can read from. Without memory buffers, map workers and reduce workers must communicate through the file system on disks and rely on remote procedure calls for managing the workers, which are expensive. Consequently, IMMR has been shown to outperform the disk-based MapReduce significantly, by up to an order of magnitude on contemporary multicore machines [26, 36]. The drawbacks of IMMR are that it is more suitable for smaller computation that fits in the main memory, and fault tolerance cannot be achieved as easily without relying on non-volatile storage. It is possible that future MapReduce programs may use a hybrid approach, e.g. disk-based version across in-memory nodes providing fault tolerance on persistent storage, with each computation on a node further divided into finer granularity tasks to exploit in-memory computation [77].

![Figure 4.1: Overall design of in-memory MapReduce system.](image)

In two of current IMMR systems, Phoenix [51, 62] and Metis [41], input data is split into equal-sized chunks that form map tasks, and map tasks are queued and assigned dynamically to map worker threads. Figure 4.1 shows at a high level the intermediate buffers used in both Phoenix and Metis: a two dimensional array with the number of rows equaling the number of map workers, and the number of columns equaling the number of hash buckets. Each map worker thread works on map tasks assigned to it and emits output into its own row of hash buckets in the matrix. When a thread produces a key-value pair, the key is hashed into a bucket using a hash function, and the pair is added
into the bucket. If there are multiple keys that map to a bucket, the bucket maintains the keys in a sorted order (in order to facilitate fast key searching), with each key having a list of values associated with it. Since different threads use different matrix rows to store their intermediate output, there is no data sharing or synchronization among map workers.

In the reduce phase, each column of bucket forms a task, and tasks are queued and assigned dynamically to reduce worker threads. Each reduce worker performs reduction by aggregating values that share a common key, both within a bucket and across buckets in the same column. Since reduce workers work on separate columns of buckets, they do not share data. However, map workers and reduce workers share data through the intermediate buffer, hence the reduce phase must wait until the map phase completes before it starts. After aggregating values, a reduce worker produces <key, aggregate-value> pair for each distinct key assigned to it, and (optionally) participates in a merge sort to combine the output from all reduce workers so that the output in the final output buffer is sorted based on key order.

### 4.2 MapReuse Design

#### 4.2.1 Design Overview

This section overviews the architecture of our MapReuse runtime system. One of the goals of MapReuse design is to keep the IMMR system unchanged as much as possible. It is important not to modify the IMMR application programming interface (API), so that programmers’ abstraction is unchanged. This makes it possible to change the IMMR system independently of MapReuse, and optimize MapReuse without requiring changes to the IMMR system or the application code. To achieve that, we build MapReuse on top of an existing IMMR system.

Suppose that there are two inputs: an original input, and a new input that is slightly different. MapReuse needs to completely compute the output for the original input, but reuse as much computation as possible when computing the output for the new input. Figure 4.2 illustrates the basic MapReuse design. Suppose that the original input is denoted as \( A \), and the new input is denoted as \( A' \). \( \Delta = A' - A \) represents changes in the new input vs. the original input. An application is wrapped into two separate instances: a server component that computes output for the original input \( A \), and a client component that computes output for \( \Delta \). Initially, the original input (Step 1) is given to the server (Step 2). The server is none other than the original application, interacting with a private instance of an IMMR system. The server’s IMMR keeps the intermediate buffer to store the intermediate output of the application (Step 3), and emits output for the original input (Step 4). When the new input arrives (Step 5), it is input to a \( \Delta \) engine, of which its role is to compute the difference between the original input and the new input. The engine keeps metadata including signatures, tables, etc. that will be discussed in more details later (Step 6). The differential input (i.e. the records present in the new input but not in the original input) is fed into the client (Step 7). The client is none other than the application itself, interacting with a different instance of IMMR system. The client’s IMMR keeps the intermediate output and buffer for the differential input (Step 8). The merger engine receives input from both instances of IMMR systems and merge them in the reduce phase (Step 9) in order to produce the final...
output for the new input (Step 10). Once the final output is available, the metadata in the $\Delta$ engine can be updated so that the new input is now considered the original input.

The design shown in Figure 4.2 offers several benefits. First, the underlying IMMR system is largely unchanged. The server and client use their own private instances of the IMMR system, hence there are no additional mechanisms for synchronization, communication, etc. that are required. Second, the mechanism to detect differences between input and merging of intermediate outputs are implemented as separate components and are not integrated into the IMMR system. The only modification required for the IMMR system is that each key-value pair is augmented with a tag that identifies the original map task that produces the key-value pair (the purpose of this tagging will be explained later).

The third benefit of the design is its efficiency in terms of latency at the client side. We keep two separate instances of IMMR systems for the server vs. client. An alternative design would be to use one IMMR system where the intermediate buffer will be shared by both the client and server, and client emits intermediate output directly into the server’s intermediate buffer. This alternative implementation merges the result from the differential input directly, but suffers from a serious drawback. Under a shared intermediate buffer, searching for a key in a hash bucket or inserting a new key into a hash bucket incurs the complexity of $O(lg D)$ and $O(D)$, respectively (Chapter 3), where $D$ is the number of distinct keys that are already in the intermediate buffer. Thus, searching and inserting a key into a lightly populated intermediate buffer is much cheaper than searching and inserting a key into a heavily populated intermediate buffer. If the client shares the intermediate output buffer of the server, it suffers from a significantly higher key search and insertion latencies. By letting the client use its own instance of IMMR system including a separate intermediate buffer, the client is significantly more efficient.

The fourth benefit of our design is that the IMMR instances can be tuned separately to optimize
the server and client separately. For example, this allows the client to run using a different number of threads than the server (we will evaluate the performance impact of such optimization in Section 4.3.2). Furthermore, we can also tailor the intermediate data structures, such as the number of hash buckets, hash functions, etc. depending on the resource availability and requirements.

### 4.2.2 Granularity Choices

The Δ engine is responsible for calculating the difference between the original input and the new input. The new input may differ from the original input due to insertions of new records, deletions of old records, or change of values in existing records. More specifically, the Δ engine must perform the following steps: (1) compute **differential input**, i.e. discovering parts of the input present in the new input but not present or have been changed in the original input, (2) compute **deleted input**, i.e. discovering what parts of the input present in the original input but not present in the new input. Since the computation lies in the critical path, it must be performed with very low latencies or else it may negate the benefit of computation reuse. Thus, it is not feasible to compare the original input vs. the new input byte by byte. Instead, similarity must be performed at a larger granularity, and indirectly, such as using hash signatures.

One important issue is what granularity the two inputs should be compared at. Note that while it is input records that are added or deleted in the new input, computation is performed by the server at a granularity of map tasks, where each map task is a collection of input records bundled into a single unit of dispatch. While the inputs differ on a per-record basis, results are reflected in the intermediate buffer on a per-map task basis. Thus, in order to reuse computation, we need to identify map tasks that have been added, deleted, or changed.

#### 4.2.3 Computing the Differential Input

In order to compute the differential and deleted input at the map task granularity, one approach is to divide the input into fixed-size chunks, where each chunk is the same size as a map task. Unfortunately, while simple, fixed-size chunking may suffer from a low coverage in detecting identical
map tasks. Figure 4.3 illustrates this. In the figure, each square represents an input record. Multiple squares with the same color represent one map task. Black squares represent new input records that were not present in the original input. For illustration purpose, each map task contains four input records. Map task boundaries are shown as the lightning symbols. Case (a) shows map tasks for the original input, case (b), (c), and (d) show new input records appended at the end, middle, and at the beginning, respectively. If the Δ engine compares map tasks between the two inputs, it detects that all map tasks match in case (b), some map tasks match in case (c), and no map tasks match in case (d). Thus, fixed-size chunking is not appropriate to use.

An alternative approach is to use content-based chunking, such as one used in LBFS [42]. The boundaries of chunks in content-based chunking does not depend on fixed boundaries, and a chunk is not a multiple of map task size. Instead, the boundaries depend on when checksum of the content matches with a specified pattern. Since chunk boundaries are based on the content, it is likely to achieve a better similarity detection coverage. However, this approach has several challenges. First, output computation in IMMR is performed on a per map task basis, hence the chunk boundary may not align with the map task boundary, which is problematic. If the IMMR system itself is modified to work at tasks at content-based boundaries, it has to be substantially modified and conflicts with our goal of transparency. For example, with the content-based map tasks, the size of map tasks will be variable and may introduce load imbalance problems. Second, producing hash checksum is computationally expensive and the latency falls on the critical path of computation, thus the overheads are too significant to make content-based chunking feasible.

![Diagram](image)

**Figure 4.4:** Detecting identical map tasks and computing differential input (left), and detecting unmatched map tasks (right).

Note that using the content-based chunking approach is acceptable in disk based MapReduce systems [22] where the chunk boundary calculation latency is (partially) overlapped and dwarfed by the very high latency of disk accesses. However, such an opportunity is not present in IMMR systems. Chunk boundary calculation requires CPU busy time just as the map and reduce phases, hence they
cannot be hidden nor overlapped.  

In order to achieve the low overhead of fixed-size chunking but obtain a high coverage of content-based chunking, we propose a hybrid design. The \( \Delta \) engine divides the original input into fixed-size chunks where each chunk is the size of a map task, and computes their checksums (signatures). For the new input, the engine calculates the signature of each window, and the window slides byte by byte. The signature of each sliding window is compared against signatures obtained from the original input. When there is a match, we have identified a map task from the original input that has survived in the new input. Any unmatched portions of the new input are either map tasks that have been added into the new input or existing map tasks whose content has changed. This hybrid approach gets the benefits of both prior approaches. First, unlike content-based chunking, using fixed-size chunking for the original input does not incur chunk boundary calculation, and does not require changes to the underlying IMMR systems. Second, unlike fixed-size chunking, the coverage for similarity detection is high because the sliding window allows an existing map task in the new input at any offset to match the corresponding one from the original input.

Note that as long as all map tasks are preserved and their content unchanged, any reordering of map task locations in the input does not change the output, due to MapReduce’s input commutativity and associativity. The signatures that we keep for the original input do not need to be ordered when they are matched against the signatures of the new input. Therefore, it is sufficient to record the signatures on a hash table, which facilitates a fast and direct signature lookup. To cater for map tasks have identical content in the original input, we can associate each signature with its frequency (i.e. number of occurrences).

The \( \Delta \) engine requires the following inputs to detect identical map tasks: 1) the map task size, 2) a Signature Frequency Table (SFT) that records the signature and frequency for all map tasks in the original input, 3) the new input. In a nutshell, signature matching is performed by computing signatures in a sliding window fashion on the new input. For each sliding window, its signature is computed, and is used to look up the SFT to check for a signature match. Upon a match, the frequency for the signature in the SFT is decremented, the window is moved, and signature matching continues. At the end of signature matching, the engine places the differential input in the differential input buffer, which will be the input of the client.

To achieve signature matching, the \( \Delta \) engine starts by processing the new input in a sliding window fashion by maintaining two index pointers: start index and current index (Figure 4.4(left)). Initially both indices are set to zero when the signature for the first sliding window is computed (line 6 in Algorithm 1). We will explain the significance of the arguments to compute_signature() function shortly. The engine then looks up in the SFT using this signature to check for a match (line 7, Step 1 in Figure 4.4). If there is no match, then the current index is incremented (line 9, Step 2(b)), while keeping the start index unchanged. Then, the signature is computed for the next sliding window. This process continues until a match is found in the SFT. On a match, the frequency of that particular signature is decremented (line 11, Step 2(a)).

Note that upon an SFT mismatch, the reason why only the current index is increment while the start index is not changed is that it enables the \( \Delta \) engine to track the portion of the input that did not match. When there is a signature match, the portion of input that did not match starts from the start.
index to the current index minus the size of the sliding window. The engine places this unmatched portion in the differential input buffer (line 13, Step 3). On a match, both the current index and the start index are set to the next byte after the match (line 15 and 16).

At the end of the new input, if the current index and start index are pointing to the different positions (line 21), it implies that the residual input is unmatched, hence it is also placed into the differential input buffer (line 23, Step 4).

Now we will discuss the computation complexity of the $\Delta$ engine. Suppose that the original input and new input have a size of $B$ bytes, and the sliding window size is $W$ bytes. There are $\frac{B}{W}$ fixed-size chunks in the original input and there are $O(B)$ sliding windows. Computing the signature of each chunk or window incurs $O(W)$ complexity. Therefore, a naive implementation of signature matching will require $O(BW)$ complexity for generating signatures for all sliding windows, and $O(B\frac{B}{W}) = O(B^2W)$ for comparing each window with all original input signatures. The total complexity is $O(BW + B^2W)$.

Algorithm 1: Identifying identical map tasks and computing the differential input.

Input: Map Task Size, Signature Frequency Table (SFT), New Input
Output: Differential Input Buffer

1. signature = 0;
2. start_index = 0;
3. current_index = 0;
4. SFT_Entry = NULL;
5. while (Not end of New Input) do
6.     signature = compute_signature(signature, start_index, current_index, Map Task Size);
7.     SFT_Entry = lookup_SFT(signature);
8.     if SFT_Entry is NULL then
9.         current_index++;
10.    else
11.        SFT_Entry.frequency--;
12.        if (current_index > start_index) then
13.            append [current_index−start_index] to differential input buffer;
14.        end
15.        start_index = current_index + Map Task Size;
16.        current_index = start_index;
17.        if (SFT_Entry.frequency == 0) then
18.            delete this SFT_entry from SFT;
19.        end
20.    end
21. end
22. if (current_index > start_index) then
23.    append [current_index−start_index] to differential input buffer;
24. end

Our optimization goal is to lower the complexity of matching the two inputs. We apply several optimizations to reduce the overheads of this signature matching. We maintain SFT as a hash table,
indexed by hashing a signature. By using a hash table, assuming there is no hash collision, each sliding window only needs to be compared against one signature from the original input. This reduces the complexity of signature matching to $O(B)$. The new total complexity is $O(BW + B) = O(BW)$, which is now dominated by the signature generation for the sliding windows.

Another optimization we perform is on a signature match, the corresponding frequency is decremented and if it becomes zero, we delete the entry (line 17 and 18). By deleting such an entry, future lookups become faster and memory overheads are reduced.

To reduce the complexity for computing the signatures of the sliding windows, the $\Delta$ engine uses a rolling checksum method [5], where the signature of the next sliding window is computed by a function that takes the signature of the current sliding window, the old byte value that is removed from the new window, and a new byte value that is added in the new window. This takes $O(1)$ time, hence the signature generation is now unaffected by the size of the window. The signature generation for all sliding windows now only takes $O(B)$, which is also the new total complexity.

However, even with the rolling checksum, the $\Delta$ engine needs to produce a signature at every byte, hence its total computational complexity scales with the input size. To reduce this overhead further, we apply another optimization. We make an observation that immediately after a match, the next match is not possible until the data size equal to one map task has been seen. Therefore, the $\Delta$ engine does not need to calculate signatures for all the intermediates bytes or look up the SFT for a possible match. To implement this, the compute_signature function takes additional arguments, the start and current index (Algorithm 1, line 6). If both the indices are pointing to the same position, then we skip computing the rolling checksum for all intermediate bytes. The more similar the original input and new input, the fewer signature computations and comparisons occur. When the original input and the new input are identical, there will only be $O(BW)$ signatures computed and matched. Thus, with this optimization, the $\Delta$ engine complexity ranges between $O(BW)$ and $O(B)$, depending on how much the original input changes in the new input.

One interesting aspect of our design is how the $\Delta$ engine interacts with the client. The engine fills the differential input buffer as it executes. This raises an opportunity for the client to execute as soon as there is enough input accumulated in the input buffer, rather than waiting the $\Delta$ engine to finish its computation. This allows the $\Delta$ engine and the client to proceed in a pipelined fashion.

One caveat with our design is that because we compare signatures instead of the actual contents of the inputs, there is a tiny probability for non-identical map tasks lead to a signature match. The probability for such a match follows a classical birthday paradox problem. For example, for a 100MB input and 64KB map task size, there are 1600 map tasks. The theoretical collision probability for a 32-bit checksum function as used in this study [5] is only 0.0003. For a much larger randomly generated number of map tasks, e.g. 500K, our result finds only 28 collisions, i.e. 0.000056 collision ratio. For mission critical applications, such a false match problem may not be appropriate. However, many MapReduce applications are not mission critical and can tolerate some small input imprecision. In fact, the same signature/checksum has been adopted in real-life applications such as compression libraries [88]. Furthermore, the checksum size can be increased if one wishes to lower the false match probability further.

Finally, one pathological case may be that if each map task is slightly altered, then signatures
change and mismatches may become common. Such a pathological case causes a performance issue but not a correctness issue. Furthermore, such a case is rare. More likely, new input data is appended or pre-pended [30]. Scattering small changes uniformly across all map tasks are not common. However, if the pathological case appears, there is a fix, for example we can reduce the map task size. In our experiments, we observed that reducing the map task size as low as 4KB does not produce any noticeable performance degradation. In our evaluation (Section 4.3.2), we will show that a higher number of map tasks does not degrade performance by much.

4.2.4 Determining the Deleted Input

So far we have discussed how we generate computation for differential input, i.e. inputs records that were added or modified in the new input. However, there may also be input records that are deleted from the original input in the new input. Such deleted input records are detected when at the end of signature matching, the SFT is not empty, i.e. there are non-zero frequency signatures.

Such deleted input records have already produced output in the server because they were a part of the original input. However, the output corresponding to the deleted input must somehow be omitted in the output for the new input. Thus, the challenge is how the output corresponding to the deleted input should be removed. The client cannot remove such output because the server and client both have private IMMR instances, and they cannot cross-modify each other’s instances. Alternatively, the server may be modified so that it has a new unmap function to remove key-value pairs that were inserted during the map phase. However, that modifies the MapReduce programming model, and programmers may be reluctant to provide an unmap function.

In order to remove the output contributed by deleted input records, we use selective reduction when the intermediate output of the server and client are merged during the reduce phase. Any output contributed by deleted input records are omitted from merging, in effect discarding it during the merging reduce phase. With this approach, the intermediate output only needs to be tagged in order to distinguish output that should be discarded or kept. To implement the approach, each key-value pair in the server’s intermediate buffer is tagged with the map task ID. However, we still need to know which map task IDs contain deleted input records.

Since the SFT structure does not store the ID of unmatched map tasks, we design another structure for this purpose, called the Signature Identifier Table (SIT). Each SIT entry contains a signature and a list of all map tasks (map task IDs) that have such a signature, as illustrated in Figure 4.4(right). Similar to the SFT, the SIT is a hash table, where the appropriate index is found by hashing a signature. The SIT is built simultaneously with the SFT when the original input is available, and before the new input arrives. During the building, every time a new signature is encountered, a new entry is created in both the SFT and SIT, the frequency of the signature incremented in the SFT, and the corresponding map task ID added into the SIT.

During signature matching, the \( \Delta \) engine only decrements the frequency of a signature in the SFT, but leaves the SIT untouched, i.e. map task IDs are not removed from the SIT. This is an optimization based on an observation that any map tasks that have the same signature have an identical content and will contribute in the same way to the output, hence at the signature matching stage, there is no
need to distinguish map tasks that have the same signature.

At the end of signature matching, the SFT may not be empty if some signatures remain unmatched. At this point, we need to discover what map tasks are “unmatched”, because they are ones whose effect on the output must be removed. Algorithm 2 shows how the list of such map tasks is obtained. The \( \Delta \) engine visits each signature in the SFT and reads its frequency. Then it indexes the SIT (line 2 in Algorithm 2), retrieves as many map task IDs as the frequency of that signature (line 3), and places them into the \textit{unmatched map task list} or UMTL (line 4). Recall that map tasks that have the same signature have identical content and effect on the output, so it really does not matter which map tasks are retrieved from the SIT, hence this is why during signature matching the SIT was not updated.

\begin{algorithm}[h]
\caption{Detecting Unmatched Map Tasks}
\begin{algorithmic}[1]
\Input Signature Frequency Table (SFT), Signature Identifier Table (SIT)
\Output Unmatched Map Task List
\ForEach \textit{SFT} \textit{Entry} in SFT
\State Get the corresponding \textit{SIT} \textit{Entry} in Signature Identifier Table (SIT) ;
\For \textit{i} $\leftarrow$ 1 to \textit{SFT} \textit{Entry}.frequency \Do
\State Insert in
\State Unmatched Map Task List(SIT Entry.ID[i]);
\End
\End
\end{algorithmic}
\end{algorithm}

One may notice that the SIT stores a superset of information stored in the SFT, essentially a signature’s frequency information in the SFT can be derived by counting the number of map tasks in the SIT. The reason why MapReuse keeps both of them is for performance reasons. During signature matching, it is faster and easier to decrement the frequency of a matched signature instead of remove a map task associated with the signature.

We make an important observation that identifying unmatched map task list does not affect map phase of the client instance. To start the map phase of the client instance, we only need differential input buffer to be available (as computed by Algorithm 1). Consequently, identification of the unmatched tasks and map phase execution can proceed in parallel. Identifying unmatched map tasks needs to be finished before the start of reduce phase though. However, we took a conservative approach in this work and executed algorithm 2 before starting map phase, and show that it sill yields significant performance gains (Section 4.3.2 and 4.3.3).

### 4.2.5 The Merging Engine

After the map tasks have finished, the merge engine can perform the reduce phase, combining the client’s intermediate output and the \textit{reusable} portion of the server’s intermediate output (Algorithm 3). As discussed earlier, each reduce task is responsible for performing the \texttt{reduce()} function on a set of keys (along a column of the intermediate buffer) (line 1 and 2 in Algorithm 3). As a reduce worker executes a reduce task, it visits all key-value pairs for a key assigned to it. For each key-value pair,
it uses the pair’s map task ID to search the UMTL to check whether this map task ID is present in
the UMTL (line 3). If it is, then it is skipped from the reduce phase, otherwise it added to the reduce
function’s input value list (line 4).

Algorithm 3: Merging Server and Client Intermediate Outputs

Input: Server Intermediate Buffer(SIB), Client Intermediate Buffer(CIB), Unmatched Map Task
List(UMTL)
Output: List of values for a given key(K):K.Value_List[] (used as input to reduce function)
1 foreach Column in Client and Server Intermediate Buffer do
2     foreach Key(K) in either column do
3         if (K.Map_Task_ID is NOT in Unmatched Map Task List(UMTL)) then
4             insert K.Value in K.Value_List[] ;
5         end
6     end
7 end

We note that tagging intermediate pairs with map task ID incurs memory overhead. For a 100MB
input and 64KB map task size, there are 1600 map tasks. This requires map task ID to be 11 bit long.
For a typical word-count application, keys (words) are 5 byte long (in our input) and value (count)
part is 4 byte long, this would result in approx. 15% memory overhead per intermediate pair. For our
experiments, though we used integer type (4 bytes long on our system) for map task ID, we did not
observe any noticeable performance overhead due to map task ID tagging. This is primarily because
of the favorable spatial locality of map task ID tagging process as they appended to the value part of
intermediate pair.

We also point out that our design opens up the opportunities for performing different reduce
functions at the client side, as long as both the client and server execute the same map function. If the
client has a different reduce function, it can still selectively reuse the server’s map output using our
technique.

4.3 Evaluation and Analysis

In this section, we will discuss our evaluation, consisting of results from an analytical model followed
by experimental results. A key reason for developing an analytical model is so that we can find
the fundamental relationships between parameters, and understand insights needed to explain the
experimental results. Before discussing our results, we will first describe the platform used in this
experiment.

Machine Configurations and Applications: We use an Intel Xeon X5560 system with 8-core pro-
cessor. This system has two chips with each chip having four cores. Each core has a 2.80GHz frequency,
a 32KB private L1 cache, a private 256KB L2 cache, and an 8MB L3 cache shared by four cores. The
operating system is Linux with a recent kernel (v2.6.31). Only the application, regular OS processes,
and daemons, run during measurements. Each run was repeated ten times, the highest and lowest values were removed, and the average was taken and reported, to avoid measurement biases [43].

The applications used for evaluation include Regular expression matching (RegEx), Fingerprinting (Fingerprint), Word filtering (Filter), and Word frequency count (Freq. Count). They are benchmark applications commonly used for evaluating the performance of IMMR systems [59, 26, 51, 64]. We also generated three inputs for the word frequency count application with the same number of words: input A (identical words) and B (all different words) and C (mixed words). The Filter application parses though a log user list and filters ones above a certain threshold. The Fingerprint application is a kernel of content based chunking algorithm used in typical deduplication application. The Regular expression matching is similar to grep utility implemented in MapReduce style. The input size is 100 MB. Unless otherwise notes, we fixed the map task size to be 64KB.

4.3.1 Modeling the MapReuse System

Assumptions and Scope: We extend the analytical model for analyzing IMMR system performance as discussed in Chapter 3 to support our MapReuse runtime system, by adding components to account for MapReuse specific operations. The underlying IMMR system we choose is Phoenix [62]. We denote the fraction of the original input that changes in the new input as \( f \), and assume that the number of unmatched map tasks and distinct keys differ by the same fraction. We also assume that as the number of threads increases, both the map and reduce phases achieve a linear speedup. Assuming linear speedups is reasonable here as a first order approximation because we are merely interested in obtaining insights from basic relationships between parameters in our model.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>Number of map tasks</td>
</tr>
<tr>
<td>( P )</td>
<td>Number of intermediate (&lt; k, v &gt;) Pairs</td>
</tr>
<tr>
<td>( D )</td>
<td>Number of Distinct keys</td>
</tr>
<tr>
<td>( B )</td>
<td>Number of hash Buckets (columns)</td>
</tr>
<tr>
<td>( w_1, w_2, \ldots, w_5 )</td>
<td>Map execution time weights</td>
</tr>
<tr>
<td>( w_6, w_7 )</td>
<td>Reduce execution time weights</td>
</tr>
<tr>
<td>New parameters in our model extension</td>
<td></td>
</tr>
<tr>
<td>( f )</td>
<td>Fraction of original input that changes</td>
</tr>
<tr>
<td>( x_1, x_2, \ldots, x_6 )</td>
<td>MapReuse model extension weights</td>
</tr>
<tr>
<td>( r )</td>
<td>Input record size (in bytes)</td>
</tr>
</tbody>
</table>

Modeling and Validation: Table 4.1 shows parameters and notations used in our model. The additional parameters added in our model extension are shown in the last three rows. The overheads of MapReuse include several components: computing signatures for the original input and inserting them to the SFT and SIT, computing signatures for the new input, SFT table lookup for the new input,
SIT table search for the unmatched map tasks for the new input, and merging the two intermediate results into the final output. Computing signatures for the original input is assumed to be performed offline prior to the arrival of the new input, hence the overhead is hidden. The remaining overheads are incurred after the new input arrives, hence they are exposed in the critical path. We will discuss how the latter overheads can be modeled.

First, computing signatures for the new input require a rolling checksum computation. Recall that due to optimizations we apply for signature computation, signature computation is less frequent. Every time a signature in the two inputs matches, the sliding window skips by the map task size before a new signature is computed. In contrast, when no signature matches, the signature must be computed for each sliding window (byte per byte). Since the two cases occur for \( f \) and \( 1-f \) fractions of the inputs respectively, we can assign different cost weights to the two cases: \( x_1f \) and \( x_2(1-f) \), where \( x_1 \) and \( x_2 \) are unknown constants that can be empirically discovered. The overall input size is the multiplication of the number of input records \( P \) and the size of each input in bytes \( r \). Hence, the total cost for computing the signatures is \( (x_1f + x_2(1-f))rP \).

Next, we account for looking up the SFT table. There are \( (1-f)N \) lookups for original map tasks that survive in the new input, whereas for new map tasks in the new input, each byte of advance in the sliding window will cause a SFT lookup to compare the signature of the window against known signatures from the original input. Since there are \( (1-f)N \) original map tasks and there are \( frP \) bytes in the new input that are new, there are a total of \( (1-f)N + frP \) lookups. Each lookup incurs a roughly constant time, assuming there are very few collisions in the SFT. Suppose that the cost of each lookup is \( x_3 \). The total cost for the SFT lookups is \( x_3(frP + (1-f)N) \).

Next, the SIT is only searched after signature comparison is complete, at which time only \( fN \) unmatched map tasks remain. The unmatched map tasks are further inserted into the UMTL. Suppose that each SIT search incurs a cost of \( x_4 \) and each insertion into the UMTL incurs a cost of \( x_5 \).

During merging, the original \( P \) key-value pairs from the server and an additional \( fP \) key-value pairs from the client must be evaluated, either selected or not selected, and merged. Suppose the cost of processing each key-value pair is \( x_6 \), then the total merging cost is \( x_6(1+f) \).

We can sum these components together to derive the total MapReuse overheads:

\[
T_{\text{mapreuse}} = (x_1f + x_2(1-f) + x_3f)rP \\
+ (x_3(1-f) + x_4f + x_5f)N + x_6(1+f)P
\]  

We add the overheads to the basic IMMR performance model [19]. The total execution time, under uniform and skewed key ordering, for the IMMR + MapReuse will be:

67
\[ T^{SKO} = w_1 \left( fP \log(fD) - \log(fD) - \frac{fD}{\ln 2} \right) + \frac{fD(fD + 1)}{2} \]
\[ + (w_3 + w_4) fP + w_5 fN + w_6 P + w_7 B + T_{mapreuse} \]
\[ T^{UKO} = w_1 \left( (fP) \log(fD) - \frac{fP}{\ln 2} - \log(fD) \right) + \frac{fD(fD + 1)}{2} \]
\[ + (w_3 + w_4) fP + w_5 fN + w_6 P + w_7 B + T_{mapreuse} \]  

(4.2)

where \( w_1, w_2, \ldots, w_5 \) are constant weights corresponding to key searching, key insertion, value insertion, key hashing, and map task fetching, respectively. Weight factors \( w_6 \) and \( w_7 \) correspond to the weights assigned to reduce phase computation time and multithreading overhead.

Validation: We validated our model using microbenchmarks and methodology from [19]. Briefly, the validation is obtained by varying key parameter values and testing whether the weight factors that were assumed to be constant indeed hold constant values. If they do not show constant values, there are important factors that our model has not captured. Figure 4.5 shows the results. The figure shows that all the weight factors are approximately constant, validating that the model is a solid first order approximation. For the rest of the model-based study, we will assume the following weight values (derived empirically on the evaluation platform we described earlier): \( w_1 = 110, w_2 = 2.95, w_3 = 29, w_4 = 47, w_5 = 2500, w_6 = 13, w_7 = 500, x_1 = 8, x_2 = 3.36, x_3 = 42, x_4 = 142, x_5 = 70, x_6 = 49. \)

We assume a record size of \( r = 5 \) bytes. We observed high setup overhead cost for \( x_4 \) (i.e. looking up SIT from SFT table), therefore in our model-driven study we included this cost, 58 million CPU cycles, separately (not shown in the equations). Note that the parameter values are specific to the
platform we used, hence the value of analytical model study lies in the performance trends, not the exact performance numbers.

### 4.3.2 Model-Driven Results

In this section, we will use the model derived in the previous section to gain insights about how various factors affect the performance of MapReuse.

**Figure 4.6:** How the fraction of difference in inputs affect run time for 64K map tasks (a) vs. for 256K tasks (b).

Figure 4.6 (a) shows the execution time of MapReuse as affected by the fraction of the original input that changes in the new input, normalized to the single-threaded IMMR system (under uniform key ordering). We fixed the input records $P$ to 10 million, and vary the number of distinct keys $D$ to vary the $P/D$ ratio that correspond to different curves. In Figure 4.6(a), we observe that MapReuse significantly reduces execution time. When the input difference is 10%, it reduces execution time by at
least 80%. When the input difference increases, the reduction in execution time becomes smaller as the overheads for detecting input similarity increase. However, even when the new input is substantially different compared to the original input (e.g. 70%), MapReuse still reduces execution time. Comparing curves from different P/D ratios, the convexity of the curve depends on the P/D ratio. A low P/D ratio results in a more convex curve, yielding a much higher performance improvement at low fraction of input difference compare to the higher P/D ratio. The reason is that at a low P/D ratio, key searches and insertions are more expensive, hence the savings from computation reuse is higher. Under skewed key ordering (SKO), similar results and trends are observed as shown in Figure 4.7.

![Figure 4.8: The effect of the number of MapReuse threads on performance under uniform key distribution for P/D=1 (a) and for P/D=1000 (b).](image)

Figure 4.6(b) shows the same normalized execution time when the number of map tasks is increased fourfold, from 64K to 256K. The figure does not show any noticeable change in performance compared to the case of 64K map tasks.

Figure 4.8(a) plots the execution time for P/D ratio of 1, normalized to the base IMMR system running with 8 threads. For small fractions of input difference (e.g. 20% difference), MapReuse with running with one thread outperforms IMMR with 8 threads. This is substantial improvement in performance and most importantly, power consumption. The maximum fractions of input difference where MapReuse lose its performance advantage are quite high: 35%, 50%, 70%, and 99%, for 1-thread, 2-thread, 4-thread, and 8-thread execution, respectively. However, for a higher P/D ratio (Figure 4.8(b)), MapReuse’s performance improvement significantly smaller because the ∆ engine overheads are larger than the saving from computation reuse.

We repeated the experiments for the case when keys are not uniformly distributed (skewed), and obtain similar observations (Figure 4.9).
Incremental Input Change (P/D=1)
Execution Time (SKO)
0% 20% 40% 60% 80% 100%
Incremental Input Change (P/D=1000)
Execution Time (SKO)
0% 20% 40% 60% 80% 100%
4.3.3 Experimental Results
This section presents results from experiments on real platforms. We use hardware counters to break the execution time of MapReuse into three components: 1) MapReuse overheads, 2) map time, and 3) reduce time. We vary the fraction of input differences by replacing records in the original input randomly with new input records, while at the same time preserving the P/D ratio of the original input. Preserving the P/D ratio is a crucial step, because IMMR performance is highly dependent on the P/D ratio of the input, as the previous section has shown.

Figure 4.10 shows the execution time for different applications as affected by different fractions of input difference \( f = 5\%, 15\%, \text{ and } 25\% \), normalized to the base case of single-threaded execution of the IMMR system. The figure shows that as expected, a higher fraction of input differences leads to a smaller performance gain. One interesting observation is the significant variation in how much MapReuse improves performance across applications. For example, while Freq. Count (B) and Filter show more than 80% reduction in execution time for \( f = 25\% \), Freq. Count (A) shows only 40% reduction. What explains the variation? Our analytical model in the previous section shows that the benefit of MapReuse depends on the P/D ratio. In Freq. Count (A), there is only one key because all input records are identical, hence the P/D ratio is the highest possible. In Freq. Count (B), each input record has a unique key, hence the P/D ratio equals one, the lowest possible. Freq. Count (C) lies between Freq. Count (A) and Freq. Count (B). Freq. Count (A) and (B) are interesting because they bound the best and worst case of P/D ratio. We can see in the figure that MapReuse performance improvement is the highest in Freq. Count (B). This is because key searching and insertion are high when there are a lot of unique keys, hence the benefit from computation reuse is high. In contrast, when there are very few keys, key searching and insertion are already cheap, hence the benefit from computation reuse is lower.

Finally, the figure shows that the reduce phase (where merging occurs) only takes small portions of MapReuse execution time. The portions grow larger with larger \( f \) because of increasing number of
Figure 4.10: Normalized execution time as affected by various fraction of changes to the original input.

unmatched tasks. This is in line with what our analytical model suggests (Section 4.3.2).

Next, we investigate the effect of using different number of threads in MapReuse. Figure 4.11 shows MapReuse speedups with various number of threads over base IMMR running with eight threads. The figure shows that MapReuse’s speedups are significant for most benchmarks, and can be as high as $50 \times$ for Freq. Count (B). Such a speedup is very significant considering that it would require hundreds of threads to achieve the same magnitude of speedup through parallelism alone. In most cases, however, the speedups are roughly several times compared to IMMR. The harmonic mean of speedup ratios of MapReuse over IMMR running with eight threads are $1.3 \times$, $1.7 \times$, $2.0 \times$, and $2.2 \times$, on 1, 2, 4, 8 threads, respectively, assuming the new input differs by 5% from the original input. Even when the new input differs by 25%, MapReuse speeds up IMMR by $1.3 \times$ on 8 threads. MapReuse does not achieve a linear speedup as the number of threads is increased because not all its computation is data parallel. MapReuse shows slowdown for Freq. Count (A) vs. IMMR running with 8 threads. This is because IMMR’s performance improves with thread parallelism, whereas MapReuse’s $\Delta$ engine computation is sequential, so running with more threads does not reduce the overheads. Similar results and trends were observed for IMMR running with four threads (Figure 4.12).

Due to its effectiveness, MapReuse does not need to use as many threads as the IMMR system in order to improve its performance. The speedups are still substantial when MapReuse uses two threads. This matches with the observation from our analytical model (Section 4.3.2). The significance of this is that MapReuse can be used not only to improve performance, but also to free up core resources, either for lowering power consumption or for running additional workloads to increase throughput.
using the same number of cores.

In order to understand MapReuse performance better, Figure 4.13 shows the execution time of MapReuse for the extreme cases (Freq Count (A) and (B)) for various number of threads, compared to and normalized to the execution time of IMMR with 8 threads. We assume \( f = 0.25 \) for the figure.

MapReuse execution time is further divided into the \( \Delta \) engine overheads, map time, and reduce time (which includes the merger engine overheads). Looking at Freq Count (A) on IMMR, we can see that over one third of the execution time (36%) is spent in the reduce phase. This is not because of the unusually high reduce time. Instead, with a maximum P/D ratio (all input records share just one key), no key search and key insertion are incurred in the map phase, making its latency unusually small. MapReuse successfully reduces the map time significantly from 121% with one thread, down to only 14% with 8 threads. However, since there is only one reduce task corresponding to the single key, the reduce phase is sequential and does not benefit from thread parallelism. Finally, the extremely small original map phase latency is too small to offset MapReuse overheads. Thus, the slowdown is entirely due to the \( \Delta \) engine overhead.

We note that the situation that leads to MapReduce underperformance is rare. It only happens when there are many input records sharing a very small number of keys, or when the new input differs significantly from the original input. Fortunately, our analytical model gives a clear guidance. Since the P/D ratio is known after the original input is processed but before the new input arrives, we can use the P/D ratio of the original input to guide MapReuse to make a dynamic decision on whether to apply computation reuse or not. The pattern of fraction of input difference is most likely quite stable or known to users, hence significant changes in the fraction can be anticipated.

Let us look at the other extreme. Freq Count (B) has the lowest possible P/D ratio of 1. This causes a high map time, which manifests in it dominating the execution time. MapReuse’s \( \Delta \) engine overheads
and the reduce time are negligible components in IMMR with 8 threads. With MapReuse, the map time declines by 31% even with only one thread, and further declines by 82%, 92%, and 96% for 2, 4, and 8 threads, respectively. MapReuse overhead is negligible at 0.7% across all cases. The reduce phase benefits from the number of MapReuse threads, declining from 8% to only 1.2% from 1 to 8 threads.

There is an additional interesting observation to make. One may guess that with \( f \) fraction of input difference and \( T \) threads, if the original map time in IMMR is \( a \), the minimum map latency in MapReuse would be \( a f \frac{8}{T} \), because \( a f \) fraction of the input records must be processed by MapReuse, divided over \( T \) threads, normalized to IMMR with 8 threads. For example, we may expect a minimum map time of \( 0.25 \times \frac{8}{1} = 200\% \) for MapReuse with 1 thread, and a minimum map time of 100%, 50%, and 25% for 2, 4, and 8 threads, respectively. However, clearly Freq. Count (B) outperforms that significantly, with a map phase latency of 71%, 18%, 8%, and 4%. The only way such an outcome is possible is that the processing time for each input record in MapReuse’s map phase vs. in IMMR’s map phase is significantly lower. Here our analytical model helps explain the reason: since our client and server approach lets client and server use separate IMMR instances (Section 4.2.1), the client works on a much smaller set of keys, and since key searching and insertion are dependent on the number of keys already in the intermediate buffer, each processing of input record in the client is much cheaper than in the server. This demonstrates the significant benefit of our MapReuse client-server design.

Figure 4.14(a) further shows a breakdown of MapReuse overheads execution: signature computation for the new input, SFT table lookup, UMTL construction, and reduce time (which includes merging overheads), for one thread. The breakdown is not possible to obtain using actual experiments since the code that perform them are tightly intertwined, hence for this numbers, we rely on the analytical model. The figure shows that for small fraction of input difference (20% or less), the most
Figure 4.13: Breakdown of execution time for Freq. Count input A (left) and B (right), for IMMR using 8 threads and MapReuse using 1, 2, 4, and 8 threads, assuming 25% fraction of input difference.

dominant sources of overheads is the reduce time. The next significant sources are signature computation and SFT lookup. For larger fractions of input difference, SFT lookup dominates as it continues to increase along the fraction. However, for 8 threads, the reduce latency is significantly reduced through reduce thread parallelism. Hence, the sequential overheads, primarily the SFT lookup (for fraction of 10% or higher), or signature computation (for fraction less than 10%) dominates.

4.4 Related Work

General Computation Reuse. The idea of reusing computation for identical inputs has been pursued fruitfully in other areas such in microprocessor architecture [56, 61, 53] and in compiler through function reuse [52, 21]. Previous studies in the programming languages and algorithm community develop dynamic algorithms [27, 44, 49] to support incremental computations. Compiler based approaches [50, 48, 31] focus on automatic translation of programs to support incremental computation for sequential programs.

Computation Reuse in MapReduce. There are several studies exploiting computation reuse in disk-based MapReduce: Percolator [46], Comet [34], Incoop [22], DryadInc [47], and [40, 45], among others. However, to our knowledge, there has not been any study to apply computation reuse in IMMR. The challenges to reuse computation in IMMR are fundamentally different than in disk-based MapReduce. IMMR deals with the main memory, hence any metadata required for computation reuse must be implemented in data structures, rather than the file system. The main memory is also space-restricted, thus any computation reuse in IMMR must be space efficient. Finally, the latency for detecting input similarity is fully exposed in IMMR, whereas in disk-based MapReduce it can be hidden and overlapped with the (much higher) disk access time, making it imperative to implement very efficient input similarity detection in IMMR.
Figure 4.14: Breakdown of MapReuse overheads, $P = 10^7$, P/D ratio = 1000, when running with one thread (a) and 8 threads (b), normalized to IMMR running with one and 8 threads, respectively.

In-Memory MapReduce (IMMR) Systems. The first IMMR runtime system, called Phoenix, was developed by Ranger et al. [51]. They demonstrated Phoenix’s scalability on contemporary multicore systems. Yoo et al. [62] further optimized Phoenix, and the optimized version was used as the representative system for IMMR evaluation [26, 36, 24, 38, 39, 63]. Another IMMR runtime system, Metis, was developed by Mao et al. [41], as an alternative to Phoenix. Talbot et al. [58] and Chen et al. [26] further improved Phoenix using better workload optimized data structures and tiling optimizations. None of these studies address the input reuse problem that is the focus of this work.
Chapter 5

Conclusion

As technology scaling makes multicore architectures more common place, exploiting multicore parallelism for performance will become more important. Unfortunately, exploiting multicore parallelism is challenging due to parallel programming side-effects, such as data races, deadlocks etc. We believe that runtime systems that provide automatic parallelism extraction and management will become more useful for achieving higher performance on multicore architectures. Therefore, in this dissertation we investigated topics related to extracting and understanding parallelism of such runtime system.

In the first part of this dissertation, we have presented a new approach for exploiting multicore parallelism in dynamic memory management for sequential applications. Dynamic memory management functions are offloaded to a separate thread called MMT. We have presented a thorough discussion of the design issues that are needed in order to exploit parallelism and improve the performance of sequential heap allocation-intensive applications and provide low-overhead safe dynamic memory management.

Our study resulted in several interesting findings. First, it is possible to accelerate dynamic memory management using a separate thread, despite many challenges that need to be overcome for parallelism involving fine grain tasks. Secondly, we show that MMT can be designed to be transparent to the application and memory allocation library without requiring source code modifications, hints from programmer, compiler support, library modification, or special hardware support for them. Furthermore, MMT design is agnostic to the underlying dynamic memory management library algorithms or data structures.

Using heap allocation-intensive benchmarks, we evaluate MMT on an Intel core 2 quad platform. On average, our MMT approach achieves a speedup ratio of 1.19× for both Doug Lea’s allocator and PHKmalloc allocator. We also showed that for PHKmalloc allocator, MMT approach can allow security features and checks to be provided without any performance loss on average, which otherwise result in 21% execution time overheads.

We believe that the findings in this work can be useful to other researchers in considering what design issues are important for exploiting fine-grain function parallelism. We also believe that MMT
can be useful for offloading other high overhead tasks, such as debugging, tracing, and profiling.

In the second part of this dissertation, we discussed a new analytical model that identifies key performance factors for a shared memory MapReduce system, and reveals how they affect performance and scalability. We demonstrated how the model can explain performance phenomena that MapReduce exhibits and predict likely performance bottlenecks, in some cases prior to coding. We have summarized several important and interesting findings, such as how the number of intermediate output pairs, number of keys, number of hash buckets, number of map and reduce threads, affect performance. We have also showed various components of MapReduce execution time, and under what circumstances each component becomes a performance bottleneck. We have also developed an application classification framework that helps in predicting likely performance bottlenecks and methods to address them, for a given application.

In the third and part of this dissertation, we have presented MapReuse, a novel technique to reuse computation in an In-Memory MapReduce (IMMR) system. MapReuse is built on top of existing IMMR systems, largely leaving them unmodified. MapReuse separates the computation for the original input vs. the computation for the differential input, and merge the two results during the reduce phase. We have discussed various optimizations that allow MapReuse to achieve very low overheads, such as separating IMMR instances for server and client, hybrid of fixed size chunking and sliding windows, signature computation using sliding checksum, and selective reduction.

We evaluated MapReuse using analytical modeling and evaluation on benchmark applications running on a real platform. The results show that except in an extreme case that can easily be anticipated, MapReduce improves performance across the board, often very significantly. The harmonic mean of speedup ratios of MapReuse over IMMR running with eight threads are $1.3 \times$, $1.7 \times$, $2.0 \times$, and $2.2 \times$, on 1, 2, 4, 8 threads, respectively, assuming the new input differs by 5% from the original input. Even when the new input differs by 25%, MapReduce speeds up IMMR by $1.3 \times$ on 8 threads.
REFERENCES


[40] Logothetis et al. Stateful bulk processing for incremental analytics. SoCC, 2010.


