LOCALITY-AWARE TASK MANAGEMENT ON MANY-CORE PROCESSORS

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

Richard Myungon Yoo
May 2012

This dissertation is online at: http://purl.stanford.edu/wg639ht3142

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

Christoforos Kozyrakis, Primary Adviser

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

Oyekunle Olukotun, Co-Adviser

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

Mendel Rosenblum

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

The landscape of computing is changing. Due to limits in transistor scaling, the traditional approach to exploit instruction-level parallelism through wide-issue out-of-order execution cores provided diminishing performance gains. As a result, computer architects now rely on thread-level parallelism to obtain sustainable performance improvement. In particular, many-core processors are designed to exploit parallelism by implementing multiple cores that can execute in parallel. Both industry and academia agree that scaling the number of cores to hundreds or thousands is the only way to scale performance from now on.

However, such a shift in design increases processor system demands. As a result, the cache hierarchies on many-core processors are becoming larger and increasingly complex. Such cache hierarchies suffer from high latency and energy consumption, and non-uniform memory access effects become prevalent. Traditionally, exploiting locality was an option to reduce execution time and energy consumption. On the complex many-core cache hierarchy, however, failing to exploit locality may end up having more cores stalled, thereby undermining the very viability of parallelism.

Locality can be exploited at various hardware and software layers. By implementing private and shared caches in a multi-level fashion, recent hardware designs are already optimized for locality. However, this would all be useless if the software scheduling does not cast the execution in a manner that promotes locality available in the programs themselves. Especially, the recent proliferation of runtime-based programming systems further stresses the importance of locality-aware scheduling. Although many efforts have been made to exploit locality on a runtime, they fail to

take the underlying cache hierarchy into consideration, are limited to specific programming models, and suffer high management costs.

This thesis shows that locality-aware schedules can be generated at low costs by utilizing high-level information. In particular, by optimizing a MapReduce runtime on a multi-socket many-core system, we show that runtimes can leverage explicit producer-consumer information to exploit locality. Specifically, the locality on the data structures that buffer intermediate results becomes significantly important. In addition, the optimization should be performed across all the software layers.

To handle the case where the explicit data dependency information is not available, we develop a graph-based locality analysis framework that allows to analyze key scheduling attributes while being independent of hardware specifics and scale. Using the framework, we also develop a reference scheduling scheme that shows significant performance improvement and energy savings.

We then develop a novel class of practical locality-aware task managers, that leverage workload pattern information and simple locality hints to approximate the reference scheduling scheme. Through experiments, we show that the quality of generated schedules can match that of the reference scheme, and that the schedule generation costs are minimal. While exploiting significant locality, these managers maintain the simple task programming interface intact.

We also point out that task stealing can be made compatible with locality-aware scheduling. Traditional task management schemes believed there exists a fundamental tradeoff between locality and load balance, and fixated on one to sacrifice the other. We show that a stealing scheme can be made locality-aware, by trying to preserve the original schedule while transferring tasks for load balancing.

In summary, utilizing high-level information allows the construction of efficient locality-aware task management schemes that make programs run faster while consuming less energy.

Acknowledgments

Same thing can have different meanings to different people. Everybody gets to live a life, but they get different meanings out of it. My life has been about getting to my true self, that green core of mine—my study at Stanford was no exception.

While at Stanford, I was fortunate enough to be endowed with the intellectual, financial, and emotional support necessary to continue that journey. Here, I would like to thank those people who made it possible.

First, I would like to give my great appreciation to my advisor, Professor Christos Kozyrakis. From the moment he recommended my application to the admissions committee, to the very moment I am finishing up this thesis, his support has been instrumental. His advices were highly intellectual and insightful to tackle the most challenging problems, but at the same time, they were honest and supportive that I could keep the courage to carry on. I hate wars, and there never should be one, but if I were to go to a war, I would follow him.

Then I thank my associate advisor, Professor Kunle Olukotun, for starting the Pervasive Parallelism Laboratory (PPL). PPL has been a rare opportunity to discuss problems and exchange ideas with various people both from Stanford and the industry. I take pride that I made contributions to the topic which I think is of utmost importance to PPL.

I am also grateful to Professor Mendel Rosenblum for the discussions on the Common Parallel Runtime. Although it did not result in an actual implementation, some of the key ideas lived on to this project. I hope the findings in this thesis pay due gratitude to his support.

And I thank Professor Fabian Pease for taking the time to serve as the chair on

my defense committee. In fact, my acceptance letter to Stanford bears his signature, so it has been my honor to take the last step under his watch.

Second, I would like to thank Mr. and Mrs. Chyan for their financial support through the Stanford Graduate Fellowship. This project, which was a collaboration with Intel, was once on the brink of cancellation due to financial problems. Their support was crucial to carry out the project regardless of the funding situation.

In that regard, I also thank my collaborators at Intel's Parallel Computing Lab, especially Dr. Yen-Kuang Chen, Dr. Christopher Hughes, and Dr. Changkyu Kim. They were the people who managed to extend an internship project into a full-blown collaboration, and their support and interest in the project made me continue.

Third, I thank my friends at Stanford for their support. I especially thank my labmates for the technical and emotional support: (in alphabetical order) Woongki Baek, Asaf Cidon, Michael Dalton, Christina Delimitrou, Hari Kannan, Jacob Leverich, David Lo, Tomer London, Anthony Romano, and Daniel Sanchez. They were not only inspiring, but also great to share the times with.

I also thank other friends at the Gates building, especially Justin Talbot for collaborating on the Phoenix++ project, and Jared Casper and Nathan Bronson for helping out with technical difficulties. Of course, I have always appreciated that thicker-than-blood support from the Korean Mafia.

I would also like to show gratitude to our fantastic admins, Ms. Sue George, Ms. Darlene Hadding, and Ms. Teresa Lynn for letting us solely focus on solving (unimportant) problems.

Last but not least, I give my heartfelt thanks to my family—Mom, Dad, and my little brother—for their unwavering love and encouragements.

Especially, I would like to dedicate this thesis to my dad, for I was born while he himself was studying abroad to get his Ph.D. One of the reasons I decided to get my Ph.D. was to be in his footsteps, and to understand what he had been through. It seems I got more than what I've bargained for (both in positive and negative terms), but now that the end is near, I would like to claim that it was worth it.

So this one's to you, Dad.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> For my next trick I will make everyone understand me.
>
> — Marc Maron

1.1 The Many-Core Shift

For half a century, transistors have scaled such that the device count doubled for the same area for each process generation. At the same time, key characteristics such as operating voltage, switching speed, and energy efficiency continued to improve. Computer architects invested surplus transistors to exploit instruction-level parallelism (ILP) by creating wide-issue, out-of-order execution cores, and clocked them at higher frequencies. As a result, single thread performance continued to improve.

However, due to physical limits, transistor characteristics ceased to improve as transistor dimensions scale [11]. Especially, the end of voltage scaling posed limits on frequency scaling, and highlighted power as the dominant limiting factor. Coupled with ILP limitations, the traditional microprocessor designs that focused on improving single thread performance provided diminishing performance gains.

Therefore, computer architects started to exploit thread-level parallelism (TLP) to

obtain sustainable performance improvement. On a *many-core processor* [53, 72, 38], surplus transistors are used to replicate simple execution cores across the chip, thereby providing large number of hardware contexts that can execute in parallel; using simple cores can result in better energy efficiency and lower verification costs as well. As a result, the many-core approach has been adopted as the de facto standard: 64-core processors are available in the market [5], and major vendors are following the trend [64, 51]. Both industry experts [38] and academia [4, 32] agree that scaling the number of cores to hundreds or thousands is the only way to scale performance.

1.2 Cache Hierarchy, Locality, and Scheduling

Concurrently executing cores, however, increase the net memory demand. To meet the memory demand, on-chip cache hierarchies have become larger and more complex. As a result, cache transfer latency and energy have increased [20], and we now observe non-uniform memory access (NUMA) effects on a single chip [43, 12]. Therefore, failing to exploit *memory reference locality* will have more cores stalled for necessary data, thus undermining the viability of parallelism. As the number of cores increases, locality will only become more important.

Locality can be exploited at various hardware and software layers [26, 59, 12]. As a matter of fact, by implementing private and shared caches in a multi-level fashion, hardware is already optimized for locality. Rather, it is important that the software schedules cast the execution in a manner that exposes locality within the programs themselves, so that the underlying hardware could capture.

Especially, the recent proliferation of runtime-based programming systems [29, 54, 23, 52, 42, 37, 16, 55, 19] stresses the need to exploit locality through *runtimes*. A *runtime* is a thin layer of efficient software that is located between the hardware (or the operating system) and the language interface, that manages resources and performs scheduling. Specifically, the *scheduling* algorithm of a runtime determines when and where a computation, or a *task*, executes. Therefore, by making the scheduling algorithm *locality-aware*, locality can be exploited.

Exploiting locality through a runtime has a number of benefits. Unlike the operating system, runtimes are lightweight enough to manage fine-grained computation and perform dynamic load balancing. Their closer tie to the language interface also allows the runtimes to exploit high-level information to generate better schedules [15, 25].

1.3 The Challenges of Locality-Aware Scheduling

Exploiting locality on a many-core processor, however, poses unique challenges. As the number of cores scales to hundreds or thousands, it will only become more difficult. Below we list the challenges to developing a practical locality-aware scheduler.

1.3.1 Challenge 1: Complexity and Generality

The complex cache hierarchies of many-core processors imply that the entire hierarchy needs to be considered when generating a schedule. However, the state of the art fails to consider complex hierarchies: Many schemes schedule tasks assuming a flat cache hierarchy [57, 10], and those proposals that do consider cache hierarchy [18, 39, 2, 31] tend to be ad-hoc, due to the NP-hardness of scheduling. Moreover, to efficiently utilize a large-scale many-core chip, the runtime should be able to exploit locality regardless of the programming model or algorithm. However, many proposals are tied to a specific programming model [65, 71] or a particular algorithm class [28, 18, 24]. Lastly, a scheduling algorithm should be scalable so that it does not become the scalability bottleneck itself.

1.3.2 Challenge 2: Management Costs

Basically, to maximize the performance gains of a locality-aware schedule, schedules should be generated and enforced at low costs. The use of fine-grained tasks, however, makes low-cost locality-aware scheduling especially challenging. Given a fixed size input, as the core count increases, computation will have to be broken down into finer granularity. Such fine-grained tasks are more sensitive to cache misses, and many render any task management overhead visible. However, some locality-aware

proposals that are based on heavyweight, managed runtimes [31, 16] are not suitable for fine-grained task management.

1.3.3 Challenge 3: Dynamic Load Balancing

On a large-scale many-core processor, load imbalance can take place frequently. Traditionally, such imbalance occurred due to the imbalance in the software itself or due to the uneven execution of the underlying hardware. However, as core counts increase, load imbalance due to multiprogramming will increase, as well. A runtime can handle dynamic load imbalance through *stealing*—by dynamically redistributing the allocated computation. However, the conventional wisdom is that there exists a fundamental tradeoff between locality and load balance, so many task management schemes fixate on one to sacrifice the other. For example, the most popular load balancing scheme, *randomized stealing* [10], does not consider locality. However, to fully realize the benefits of locality, a stealing scheme should be made compatible with locality-aware scheduling.

1.4 Contributions

In this thesis, we address the above challenges by taking a systematic, framework-based approach. Specifically, by leveraging the high-level information, we were able to develop practical methods to perform locality-aware task management.

- **Manage Complexity with a Generic Graph-Based Framework**
  We provide a generic, graph-based locality analysis framework that is both intuitive and robust. It allows to analyze the key scheduling attributes independent of hardware specifics and scale, and the resulting scheduling scheme can be applied to various cache hierarchies.

- **Apply High-Level Information to Reduce Management Costs**
  We show that an efficient locality-aware task manager can be constructed by

exploiting high-level *task structure* information. When the explicit *data dependency information* is available (e.g., MapReduce [23]), a runtime can exploit locality on the internal data structures used to buffer intermediate results. When such an information is not available (e.g., OpenMP [54], TBB [37]), the workload *sharing pattern information* can be conveyed to the underlying runtime instead, to exploit locality through *pattern-specific task management policies*. Resulting task managers exhibit comparable performance to that from the graph analysis, while maintaining the simple task-based programming interface intact.

- **Perform Locality-Aware Task Stealing**
  We show that task stealing can be made locality-aware by honoring the specified locality-aware schedule as tasks are transfered. Since schedules are generated in a pattern-specific manner, we develop a dedicated stealing scheme for each of the workload sharing patterns.

1.5 Thesis Organization

The rest of the thesis is organized as follows. In the next chapter, we provide detailed discussions on the shift to many-core processors, changes to the on-chip cache hierarchy, and their impact on locality.

In Chapter 3, we start our study on locality-aware runtimes by optimizing a MapReduce implementation on a multi-socket many-core system. We then switch our focus to the locality on a single chip, to provide a graph-based locality analysis framework in Chapter 4. Using this framework, we arrive at a generic scheduling scheme that could be applied to various cache hierarchies. In Chapter 5, we describe the design details of practical task managers, which leverage workload sharing pattern information to approximate the schedules generated from the graph-based framework.

We then discuss task stealing. Chapter 6 starts by analyzing the locality of task stealing under the context of the graph-based framework. We develop a generic locality-aware task stealing scheme, which is then approximated in a pattern-specific fashion. We conclude in Chapter 7.

Chapter 2

The Challenges of Locality

> Or, you'll meet the perfect person who you love infinitely, and you even argue well, and you grow together, and you have children, and then you get old together, and then she's gonna *die*. That's the best case scenario.
>
> — Louis C.K.

2.1 The Changing Landscape of Computing

Transistors maintained exponential scaling for the past decades, where device density doubled for each process generation. At the same time, key characteristics such as operating voltage, switching speed, and energy efficiency continued to improve.

In particular, the excess transistors were used to exploit ILP. To fully extract ILP, computer architects equipped processors with increasingly complex wide-issue, out-of-order execution logic. Then the processors were clocked at higher frequencies to improve single thread performance. Sustained scaling of transistor characteristics justified such a design, and exploiting ILP continued to be profitable.

However, due to physical limits, transistor characteristics ceased to improve as transistor dimensions scale [11]. In particular, the end of voltage scaling put a limit

to frequency scaling, and highlighted power as the dominant limiting factor. Coupled with ILP limitations, the traditional microprocessor design approach that focused on single thread performance provided diminishing performance gains over successive manufacturing process generations.

As a result, computer architects now rely on TLP to obtain sustainable performance improvement. The new approach is to implement a *many-core* processor [53, 72]. On a many-core processor, multiple lightweight cores are implemented on a single die, and the parallelism across threads is exploited to bring about overall performance improvement.

The many-core design approach has other benefits, as well. First, it can achieve better energy efficiency. With the shifting focus on TLP, transistors used to implement complex, power-hogging out-of-order execution logics can be used to implement simple, energy-efficient cores, instead. Sharing on-chip resources among the cores can also save energy. Second, reusing the simple core design across the chip better amortizes debugging and verification costs, and processor vendors can easily re-purpose the same design to various market segments by simply varying the number of cores.

Due to these benefits, many-core designs are becoming prevalent. A startup has already released 64 core processors [5], and major vendors are following the trend [64, 51]. The shift in the design paradigm is also ubiquitous: The core count for mobile processors are also increasing [3].

For the foreseeable future, the many-core approach will continue to be the design of choice, and cores will provide the new abstraction for scaling. Both industry experts [38] and academia [4, 32] agree that integrating hundreds, if not thousands, of cores on a single chip is the only way to scale performance from now on.

2.2 The Challenges of Memory

Concurrently executing cores, however, increase the net memory demand. If threads are stalled fetching required data, execution cores will be kept idle and parallelism will not be exploited. Thus, parallelism on its own is not enough to *efficiently* utilize many-core processors. For scalability, good memory performance is also necessary.

Unfortunately, memory performance has been lagging processor performance for a couple of decades, and the limited number of packaging pins imposes physical limitation to off-chip memory bandwidth. Therefore, it becomes the task of the *on-chip cache hierarchy* to supply necessary data to increasing number of cores. As a result, many-core cache hierarchies are becoming larger, deeper, and more complex.

However, larger and deeper cache hierarchies invariably suffer the following problems. First, as the size of the cache hierarchy increases, the *average* distance that a cache line or a coherence message should travel increases. Since latency and energy consumption is linearly proportional to the traveled distance, access latency and energy efficiency worsens on a larger cache hierarchy.

Second, not only the average, but also the *variance* in distance increases. Due to design and physical considerations, large-scale cache hierarchies are organized in a modular fashion, such that accessing local memory is faster than accessing remote memory (NUMA [45]). And as the cache hierarchy scales, the *skewness* between local and remote latency (thus energy) increases [12, 20]. Therefore, if not careful, same data might have to be brought in at higher latency and energy.

Specifically, there are two common approaches to constructing a large-scale cache hierarchy: *private caches* and *shared caches*. More often, they can be mixed to form a hierarchy, as well. Below we describe each, and show that at scale, both designs suffer the same problems described above.

2.2.1 Private Cache Designs

For a private cache, a cache is dedicated to supplying only one consumer. Figure 2.1 shows an example private cache hierarchy. As can be seen, each core has private L1 and L2 caches, and the L2 caches are connected through a ring network.

Compared to shared cache configurations, which require multiple ports to serve more than one consumers, private caches require only one port. So they assume smaller area (for the same cache size), and can be fast. For this configuration, however, no cache lines are shared, so cache lines tend to be duplicated across different caches. Hence, the effective cache size is usually smaller than the sum of the cache sizes.

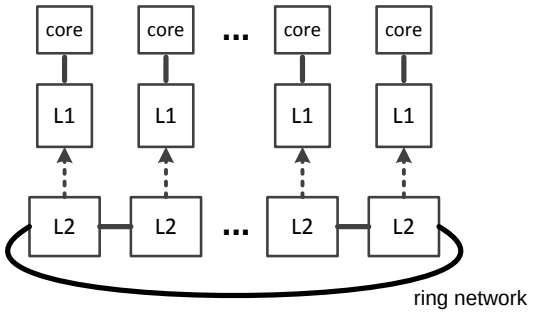


Figure 2.1: Example private cache configuration. Modeled after the cache hierarchy of [64].

While utilizing such a flat cache hierarchy reduces the NUMA effect, it puts every core far apart from the others; as the number of cores increases, the diameter of the network that connects the L2 caches will increase, resulting in higher latency and energy. Therefore, such design exhibits worse cache-to-cache transfer performance when compared to the shared cache configuration.

In fact, the cache hierarchy of Larrabee processor [64] exactly matches that in Figure 2.1. On the processor, accessing a local L2 takes 14 cycles. However, 48 additional cycles on average will be necessary to retrieve a remote cache line (assuming core clock speed, no contention).

2.2.2 Shared Cache Designs

On the contrary, Figure 2.2 shows an example shared cache configuration. As mentioned earlier, private and shared cache configurations can be mixed: In the figure, each core has a private L1 cache. However, beyond the L1, each cache serves multiple consumers: An L2 cache is shared among those cores that form a *tile*, and a last level cache (L3 cache) is shared across all the tiles.

Under this configuration, a cache line brought in by one consumer is shared by the others, resulting in *constructive interference*. As a result, given a shared cache, no cache line is duplicated, and the effective cache size is the same as the size of the cache. Therefore, compared to private cache configurations, shared caches tend to exhibit better storage efficiency. However, *destructive interference* (e.g., cache

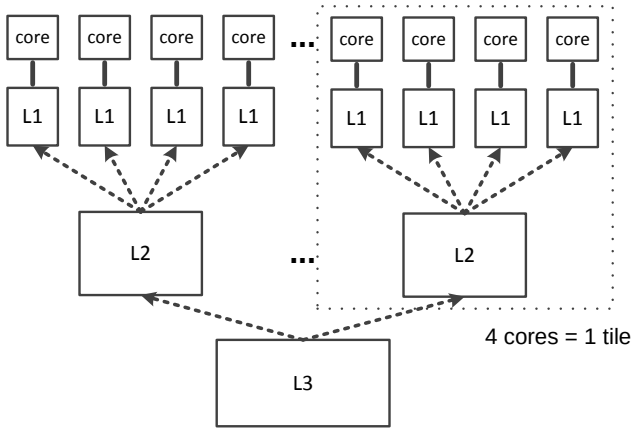


Figure 2.2: Example shared cache configuration. Depicts the cache hierarchy similar to the one in [5].

pollution) could reduce performance, and higher port count increases area budget and access latency.

More importantly, due to physical limits, the degree of fanout across different levels (i.e., the number of consumers per cache) cannot increase without bounds. So as the number of cores scale, the depth of the hierarchy inevitably increases. As noted earlier, increase in travel distance translates into increased latency and energy consumption. Increase in depth also means that the last level cache becomes further apart from the cores, thus increasing NUMA skewness.

TILE64 [5] processor assumes a similar cache hierarchy depicted in Figure 2.2. In our simulation where we assumed 10 cycle local L2 access latency, a single remote L2 miss took 90 cycles on average.

As can be seen, larger and deeper cache hierarchy implies increased latency, NUMA effect, and energy consumption. Regardless of the configuration, at scale, many-core cache hierarchies suffer the same problem. Therefore, locality should be exploited so that the data can be supplied from the L1 or nearby caches, instead of accessing remote caches.

2.3 The Role of Scheduling

A software scheduler determines when and where a computation executes. Therefore, a *locality-aware scheduler* could purposedly place and stage computations, so that the inherent locality in the application is exposed in a way that the underlying hardware can easily capture. Specifically, a scheduler should try to maximally expose the *temporal and spatial locality* of the application, so that the data could be supplied from nearby caches.

Traditionally, scheduling has been performed in the operating system. However, as the number of cores increases, computation has to be broken down into finer granularity, and scheduling these computations requires low overhead, fine-grained control. Due to kernel crossing overheads and their coarse-grained nature, OS-based approaches are not suitable for this task.

Instead, the recent proliferation of runtime-based programming systems [61, 29, 16, 55, 19, 37, 54] stresses the need to exploit locality through a *user runtime*. For these systems, the user denotes code portions that can execute in parallel as *tasks*, and these tasks are scheduled by the underlying runtime for execution. The runtime can *steal* tasks among the cores to perform load balancing as well. Runtime-based approaches are flexible yet lightweight enough to apply fine-grained, dynamic control over the execution. In addition, dynamically discovered information allow to better adapt to the underlying hardware [15], and through tight coupling with the programming model, informed scheduling decisions can be made [25].

Specifically, a runtime scheduler can generate a task schedule by *grouping* tasks to execute on the same core, and by *ordering* the groups and the constituent tasks. Therefore, with proper task grouping and ordering that considers the underlying cache hierarchy, a scheduler could expose temporal and spatial locality.

However, grouping and ordering decisions are usually interrelated, making it hard to isolate the benefits of each. In addition, on a system that can dynamically change task sizes, task size can also impact locality by affecting the flexibility of the scheduler with grouping and ordering. Due to these complexities, the detailed scheduling methodology is typically tied to the programming model being exposed.

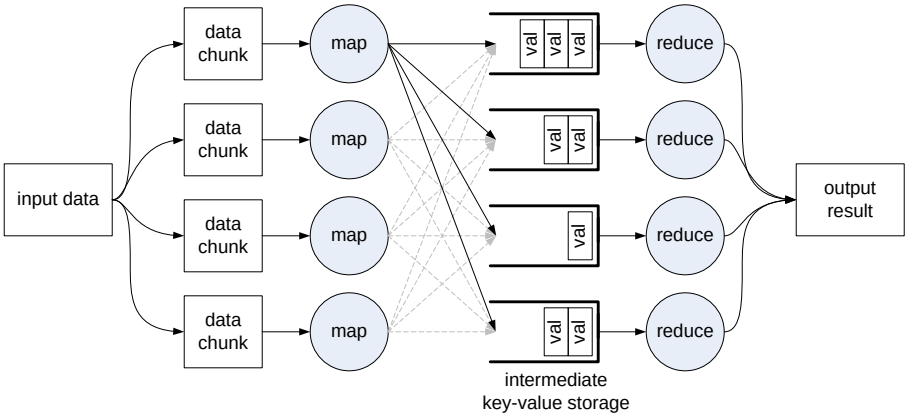


Figure 2.3: Flow of data for a MapReduce implementation.

Below we take a look at two example classes of parallel programming systems, and discuss the interactions between scheduling and locality.

2.3.1 Scheduling Structured Programming Systems

Under a *structured programming system* [23, 27, 66, 52, 42, 71, 21], a workload is expressed as a set of tasks that exhibit *data dependency*. In particular, a program is expressed as a chain of parallel *producer-consumer* task stages. Such a structure is either implied in the programming model itself [23, 27], or a user explicitly conveys the structure to the underlying runtime [66, 52, 71, 21].

For example, for MapReduce [23], a user provides two functions, a *map function* and a *reduce function*. The runtime then automatically *instantiates* these functions in parallel to execute the tasks. Figure 2.3 shows the flow of data during the execution.

The runtime first splits input data into data chunks, and invokes the map function for each chunk to execute *map tasks* in parallel. For each data item in the chunk, a map task emits a key-value pair. Once map tasks have been completed for all the chunks, the runtime groups those intermediate results by keys, and invokes the reduce function for each key to instantiate *reduce tasks* in parallel. A reduce task takes a set of key-value pairs with the same key as the argument, and emits a single key-value pair as reduced output. Key-value pairs emitted from the reduce tasks undergo an

optional sort, and are presented to the user.

As can be seen, for structured programming systems, the tasks exhibit *explicit data dependency*. The dependency information can be either obtained statically [71, 21] or discovered dynamically [23, 66]. By scheduling tasks following the dependency information, task schedules can be generated with relative ease. However, since a producer task and a consumer task may not execute in a successive fashion, a runtime should provide buffering for intermediate results. Therefore, it becomes important to exploit locality on the intermediate buffer structures. Task grouping and ordering decisions should consider the underlying cache hierarchy and NUMA effects.

- Task Grouping: Within a task stage, tasks better be grouped along *NUMA domains*, based on the input and output locality. That is, if a set of tasks exhibit locality on the input or the intermediate buffer structures, they should be assigned to the same NUMA domain to reduce cross-domain accesses. For example, for a MapReduce workload, a set of map tasks that access consecutive regions of input memory could be co-scheduled to the same NUMA domain.

- Task Ordering: As long as the data dependency is not violated, to exploit temporal locality, a consumer task should be executed as soon as possible. This makes it less likely that the intermediate result generated by a producer is evicted from the local cache, thus eliminating potential cache misses and remote cache accesses. For MapReduce, while there exists an implicit barrier between producer tasks and consumer tasks (map-to-reduce barrier), a consumer task could be executed before the barrier if the reduce function satisfies a certain criteria (combiners [23]).

To summarize, a scheduler for the structured programming system could leverage the data dependency information to generate a schedule. However, the locality for accessing and managing buffer structures becomes an important factor, and task grouping and ordering decisions should be made considering the underlying cache hierarchy and NUMA configurations.

2.3.2 Scheduling Task-Parallel Programming Systems

Under a *task-parallel programming system* [54, 29, 37], on the contrary, there is *no explicit data dependency* among tasks. As long as no control dependency exists, these tasks may execute in parallel, and any execution order of tasks will provide a computationally valid result. Examples are the **parallel for** pragmas in [54] and the **foreach** and **ateach** constructs in [16, 24].

Since this model (1) allows to easily express all available parallelism, (2) provides maximum scheduling flexibility, and (3) leads to simple implementations, task-parallel programming model has been selected as the core tasking implementations for many popular parallel programming systems [54, 29, 37, 16, 55, 19, 24].

While being important, however, it is relatively difficult to generate a locality-aware schedule for this category of systems. Unlike the structured programming systems, a runtime need not implement the intermediate buffer structures. But the lack of explicit data dependency information means that the scheduler now should find locality (and structure) where the programmer provided none. As a matter of fact, due to this difficulty, the problem of exploiting locality for task-parallel programming systems largely remains difficult and unsolved.

Instead, to exploit locality, a task manager should *synthesize* a structure. Such structure should methodically expose the temporal and spatial locality among the tasks, while considering the underlying cache hierarchy.

- Task Grouping: Tasks should be grouped so that the locality among the tasks within each group is maximized. The multi-leveled nature of many-core cache hierarchy implies that grouping may need to be performed in a recursive fashion. However, since a task group also determines scheduling granularity, the grouping scheme should also try to maintain balance across the system.

- Task Ordering: Within each task group, tasks should be ordered to maximize temporal locality, so that the underlying cache hierarchy could capture. A set of groups can be similarly ordered, to determine *group ordering*. Group ordering should maximize temporal locality across task groups.

Basically, a locality-aware scheduler should match the inherent workload locality to the underlying cache hierarchy. For example, assigning the two tasks that exhibit high sharing to different cores (thus caches) will incur unnecessary remote cache accesses; ordering high-sharing tasks further apart in the schedule will make it likely that the shared cache lines are evicted in-between.

As will be discussed in Chapter 4, however, performing locality-aware grouping and ordering is NP-hard. So before we devise a locality-aware scheduler for task-parallel programming systems, a *framework* to systematically analyze workload locality should be developed.

2.4 Requirements for a Many-Core Scheduler

Based on the discussions so far, here we list the set of requirements for a many-core scheduler. In addition to exploiting locality, scaling the number of cores imposes unique constraints on the scheduler.

2.4.1 Requirement 1: Locality

As discussed above, a many-core scheduler should exploit locality to reduce execution time and save energy. In particular, the scheduler should be able to exploit locality regardless of scale, and cache hierarchy being private or shared. While the growing complexity of cache hierarchies makes exploiting locality profitable, the penalty for a miscalculated schedule also increases. Therefore, a task manager should be able to generate *high quality* locality-aware schedules.

For structured programming systems, such a schedule could be obtained by leveraging the explicit producer-consumer locality; in particular, by improving locality on the internal data structures used to buffer intermediate task outputs. For task-parallel programming systems, however, schedulers should synthesize locality structures by purposedly grouping and ordering tasks. Such structure should expose locality in a way that it could be easily captured by the underlying cache hierarchy.

2.4.2 Requirement 2: Small Overhead

Given a fixed input size, as the size of the chip scales, computation should be broken down into *finer granularity* to exploit all available parallelism [44, 62, 41]. In particular, *fine-grained tasks* can be used when it is difficult to balance work across cores statically, or when we want faster execution times for a fixed-size problem [41, 62]. Hence, many modern parallel languages and runtimes allow tasks to be as small as a couple hundred to thousand instructions [29, 16, 37, 54, 19, 55]. As the core count increases, computation will be broken down into even finer tasks [62].

However, at few hundred instructions, fine-grained tasks may render any runtime management overhead significant. Specifically, for a locality-aware task manager to be practical, overheads due to locality-awareness should be smaller than the benefits achieved from exploiting locality. A high-quality locality-aware schedule should be generated at low costs, and should be enforced with minimal overheads.

2.4.3 Requirement 3: Scalability and Generality

After all, the scheduler itself should not be the scalability bottleneck. Scheduling and stealing algorithms should be distributed and decentralized, so that they could scale up to hundreds or thousands of cores. It is also important to note that increasing the compute power of a processor typically leads to an increase in input size. For large scale systems, task generation itself might have to be parallelized.

At the same time, Amdahl's law dictates that for a locality-aware schedule to have significant impact, a runtime should be able to exploit locality regardless of programming model and sharing pattern. Constraining the scope of locality exploitation to a specific programming model or a class of algorithms limits the degree to which a locality-aware schedule can improve performance and energy efficiency. In addition, the scheduling algorithm should be generically applicable to many types of cache hierarchies, as well.

2.4.4 Requirement 4: Dynamism

Load imbalance occurs when a few cores (threads) finish executing their initial allotment before other threads finish. When the load imbalance continues for a prolonged period of time, the effective number of worker threads reduces, and results in increased execution time. Such an idle time also translates into wasted energy.

The key to reducing load imbalance is to detect the situation in a prompt manner. However, as the core count increases, it becomes difficult to concurrently monitor all the core executions. As a result, the chance for load imbalance also increases. In general, such imbalance could be introduced due to the imbalance in the workload itself, or due to the uneven execution of the underlying hardware. On a large-scale system, however, since it would be rare for a single application to own the entire chip, interference due to multiprogramming could also introduce significant load imbalance.

Therefore, a many-core runtime should be able to dynamically perform load balancing by stealing tasks from one core to the other. Dynamic load balancing can also help a runtime deal with different system scales and memory hierarchies. However, if not careful, the locality exploited in the original schedule could be disrupted. To maximally exploit locality, stealing should be performed in a locality-aware fashion.

2.5 Related Work

The design shift to many-core processors was predicted as early as in [53, 72], and now both the industry [38, 64, 5] and academia [4, 32] has embraced it as the de facto design paradigm. Simple core designs, however, imply increased reliance on *software* for extracting, exposing, and scheduling parallelism.

As a result, many parallel programming models have been developed [29, 16, 55, 19, 37, 54, 23, 52, 42, 66, 15, 24, 27, 73, 71, 21, 35]. Based on how parallelism is specified, approaches range from manual annotation [35], templated instantiation [54, 23, 52, 42], to full-automation [73]; based on how computation is scheduled, approaches can either be static [73, 71, 21, 27] or dynamic [29, 54, 37, 23, 66, 52, 42].

More recent proposals utilize runtime-based approaches [29, 23, 66, 16, 55, 15].

Through templated task instantiation, runtimes enable simple user interface, and allows for dynamic scheduling and load balancing. Through tighter integration with the language interface, better scheduling decisions can be made, as well [15, 25]. Therefore, a lot of effort has been made to exploit locality through a runtime.

While some schemes exhibit significant performance improvement for specific usage scenarios, overall they fail to meet the requirements outlined in Section 2.4. In particular, many proposals fail to connect how high-level, task locality translates into architectural locality. Coupled with the NP-hardness of the scheduling problem, the proposed schemes [31, 18, 57] tend to be ad-hoc and opportunistic. A systematic framework would be necessary to understand cache locality.

Proposals that rely on managed runtimes [16, 31] allow to transparently improve performance across different architecture platforms. In addition, these approaches may be seamlessly integrated with cross-node locality management. However, due to their heavy-weighted nature, they fail to meet Requirement 2. As the core count scales, tasks will become fine-grained, and the runtime should be able to handle those tasks at low costs.

On the other hand, compiler-based approaches [65, 39, 27] have the potential to produce optimal schedules for a target architecture. In theory, they have access to all the static information necessary, and can amortize the compilation cost across multiple binary executions. However, while they are suitable for specialized and dedicated systems, on a general processor that can exhibit dynamic variances, they cannot handle the imbalance.

Nevertheless, for those approaches that do utilize dynamic task stealing, locality has not always been exploited. For example, the most popular task stealing scheme, *randomized stealing* [10, 29], chooses a victim at random. While it provides good characteristics such as even load distribution, simple implementation, and theoretically bounded execution time, its randomness renders it inherently locality-oblivious.

Hardware-assisted solutions [44, 41] are viable alternatives to software-only scheduling approaches. By using high-speed hardware queues to perform task scheduling, the task management overhead could be made negligible, and additional time could be spent to improve the quality of the schedule. However, as the chip size scales,

their centralized design will eventually become the bottleneck, thus failing to meet Requirement 4. Nevertheless, the hardware schemes could be applied in a distributed fashion to locally reduce the scheduling overhead.

Targeting specific class of algorithms or workloads may reduce complexity in task scheduling. For example, some approaches optimize for the divide-and-conquer paradigm [18, 2] or streaming workloads [71, 65] to generate high quality schedules at low costs. However, to fully utilize large-scale many-core processors, the runtime should be able to exploit locality regardless of the algorithm or workload type.

Chapter 3

Exploiting Locality for MapReduce

In this chapter we explore how locality could be exploited for the structured programming systems, especially on a MapReduce system. Specifically, we use a multi-socket many-core system to verify that the problem of locality already exists. In particular, since the schedulers for the structured programming systems also need to buffer intermediate results, optimizing locality for internal data structures becomes important. We also show that OS interactions could limit the scalability of a runtime system.

3.1 Background and Motivation

MapReduce [23] is a parallel programming framework and runtime system proposed originally for cluster-based systems. Since it is representative of high-level, data-parallel runtimes, the lessons learned from scaling MapReduce can be generalized to other similar frameworks.

3.1.1 The Phoenix MapReduce Implementation

The original MapReduce system was designed for clusters [23], which spawned multiple *processes* to achieve parallelism. In contrast, Phoenix [60] is a shared-memory version of MapReduce targeted for multi-core and multiprocessor systems. Phoenix uses shared-memory *threads* to implement parallelism.

Hardware Settings	
CPU	4 UltraSPARC T2+ chips 8 cores per chip, 8 hardware contexts per core Total of 256 hardware contexts on system
Per Core L1 Cache	8 KB, 16 B lines, 4-way associative Write through, physically tagged / indexed
Shared L2 Cache	4 MB, 64 B lines, 8 banks, 16-way associative Write back, physically tagged / indexed
Memory	128 GB over 16 FBDIMM channels 4 channels local to each chip 300-cycle latency for local access 100-cycle overhead for remote access 85 GB/sec for read, 42 GB/sec for write
Interconnect	Single external coherence hub 130 GB/sec bisection for coherence
Software Settings	
Operating System	Sun Solaris 5.10
Compiler	GCC 4.2.1 with -O3 optimization

Table 3.1: The characteristics of the large-scale shared-memory system used in the study.

Under Phoenix, a user first provides the runtime with the map and reduce functions to apply on the data. The runtime then launches multiple worker threads to execute the computation. In the map phase, the input data is split into *chunks*, and the user-provided map function is invoked on each chunk. This generates intermediate key-value pairs, which reside in memory. In the reduce phase, for each unique key, the reduce function is called with the values for the same key as the argument, to reduce them to a single key-value pair. Results from all the reduce tasks are merge sorted by keys to produce the final output.

Compared to the cluster version, the most striking aspect of Phoenix is that the workers communicate by accessing a shared address space. Unlike the cluster-based system where communication takes place through a distributed file system and remote procedure calls [23, 70], the communication overheads for shared-memory MapReduce are low. On the other hand, because threads contend over the single address space, the way threads access memory and perform I/O can have a first-order impact on the overall performance. As we show in Section 3.4, this can be a significant limitation on a large-scale parallel system.

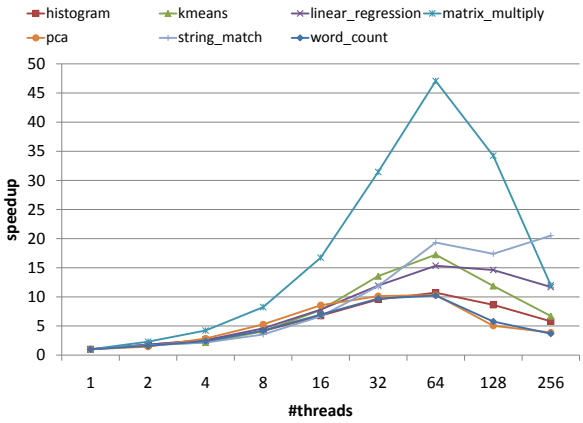


Figure 3.1: Application speedup for the original Phoenix system.

3.1.2 The Large-Scale Shared-Memory System

The most popular form of large-scale shared-memory machines today are multi-socket servers that use two to four many-core chips, with caches and main memory channels physically distributed across those chips. Such a system can readily support hundreds of threads in a single unit, but exhibits variable memory access latencies. Next generation many-core chips with hundreds of cores on a single die will likely exhibit similar NUMA characteristics, so analysis performed on a multi-socket system could be generalized to future many-core chips.

The original Phoenix runtime targeted CMP and SMP systems with uniform memory access characteristics and 24 to 32 hardware threads [60]. In this study, we target the Sun SPARC Enterprise T5440 system [69, 58], summarized in Table 3.1. Each of the four T2+ chips supports 64 hardware contexts for a total of 256 in the whole system. Each chip has 4 channels of locally attached main memory (DRAM) as well. Notice that the accesses to remote DRAM are 33% slower than the accesses to locally attached memory; any program that uses more than 64 threads will experience such non-uniform latency.

Figure 3.1 shows the scalability of the original Phoenix runtime measured on T5440 with the released applications. Despite the parallelism available in these applications, none of them scales beyond 64 threads, and most of them actually slow

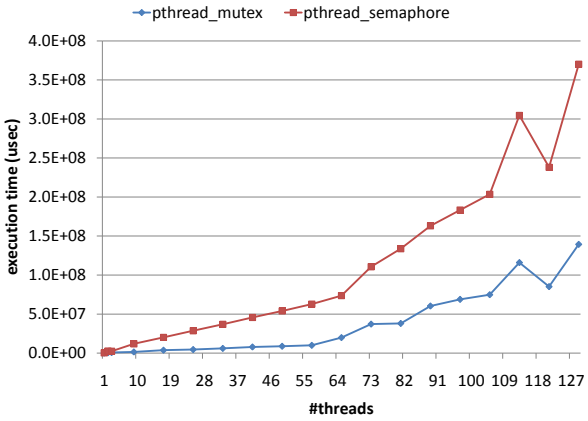


Figure 3.2: The latency of synchronization primitives.

down when more threads are involved. This is the result of the increased memory latency, loss of locality, and high contention when utilizing threads across multiple chips. We explain these issues in detail in Section 3.3.

It is also interesting to note that on T5440, the performance of several OS primitives deteriorates when using more than 64 threads. Figure 3.2 shows the latency of synchronization operations as we scale the number of threads. We measured the time needed for a half million lock acquisitions and releases in order to increment a shared counter. Similar to the behavior in Figure 3.1, the cost for synchronization increases drastically as we cross the chip boundary.

We also observed similar behavior with Phoenix and with low-level OS primitives on an 8-chip, 32-core, 32-thread Opteron system running x86 Linux. This verifies that the challenges observed are fundamental to large-scale systems, and not the artifacts of the T5440 machine we used. We omit the x86 results due to space limitations. However, the optimizations presented in this thesis led to significant performance improvements on this system as well.

3.2 Optimizing Phoenix for Large-Scale and NUMA

A parallel runtime such as Phoenix continuously interacts with the user application and the operating system. Therefore, it is natural that the optimization strategies for

large-scale, NUMA systems are multi-layered. Specifically, the approach we propose comprises three layers: algorithm, implementation, and OS interaction.

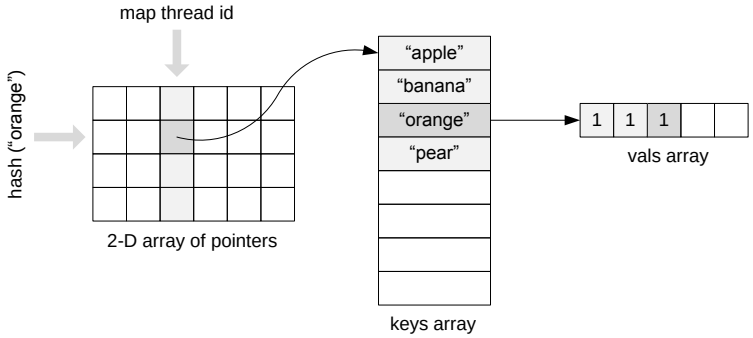
3.2.1 Algorithmic Optimizations

To perform well on a NUMA machine, the basic algorithms used in the runtime must be scalable and NUMA aware. For instance, the original Phoenix algorithm is not NUMA aware in that local and remote worker threads are indistinguishable at the algorithm level. While this is not an issue for small-scale systems, it becomes important for locality-aware task distribution in a large-scale, NUMA environment. When the input data for the application is brought into memory via **mmap()**, Solaris distributes the necessary physical frames across multiple *locality groups*, i.e., chips and their separate memory channels [67]. If Phoenix blindly assigns map tasks to threads, it could end up having a local thread continuously work on remote data chunks, thus causing additional latency and unnecessary remote traffic.

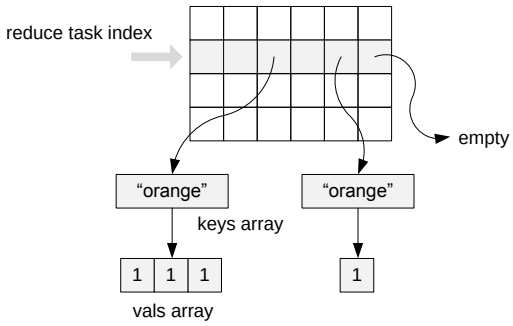
We resolved the issue by introducing a task queue per locality group, and by distributing tasks according to the location of the pertaining data chunk. Map threads retrieve tasks from their local task queues first, and when the queue runs out, they start stealing tasks from remote task queues. Although similar in spirit to the work stealing algorithm utilized in other runtimes [9], the difference here is that we maintain one task queue for each locality group (instead of each thread), hence creating a load balancing approach that is compatible with NUMA memory hierarchies.

3.2.2 Implementation Optimizations

To fully utilize a large-scale machine, applications must include non-trivial amount of work. This typically implies large input datasets that the runtime system must handle efficiently. Meeting this requirement in MapReduce is quite challenging, since a typical MapReduce application also generates commensurate amount of intermediate and output data. Unlike the original MapReduce where data storage and retrieval are limited by the raw bandwidth of the network and the disk subsystem [23], we find that in Phoenix, the design and performance of the in-memory data structures used



(a) Access pattern during the map phase.



(b) Access pattern during the reduce phase.

Figure 3.3: Phoenix data structure for intermediate key-value pairs.

to store and retrieve intermediate data are crucial to overall system performance.

Figure 3.3 gives a simplistic view of the core data structure in the original Phoenix, which is used to store the intermediate key-value pairs generated from the map phase. In particular, Figure 3.3a depicts the typical access pattern during the map phase, where a worker thread is storing an <**"orange", 1**> intermediate pair. Phoenix internally maintains a 2-D array of pointers to *keys array*, where the width is determined as the number of map workers, and the height is fixed by a default value (256). During the map phase, each map worker uses its thread id to index into this array column-wise. Once the column is determined, the element is indexed by the hash value of the key. Therefore, for each thread, a column of pointers acts as a fixed-sized hash table, and one **keys array** amounts to a hash bucket. All the threads use the same hash function.

Specifically, each **keys array** is implemented as a contiguous buffer, and keys are

stored sorted to facilitate binary search. Each entry in the **keys array** also has a pointer to a *vals array*; this structure stores all the values associated with a particular key. To maximize locality, **vals array** is implemented as a single contiguous array as well. During the map phase, both the **keys array** and **vals array** are thread local.

Next, during the reduce phase (Figure 3.3b), each row of the 2-D array amounts to a reduce task; a reduce thread grabs a row from the matrix and invokes the reduce function over all the **keys array** structures in that row. Notice the disparity between how the threads access the 2-D array structure during the map phase (column-wise) and the reduce phase (row-wise); pictorially, every *crossing* in the access pattern signifies that a worker thread has to access data structures prepared by another thread, which could require remote memory accesses in the NUMA environment. On the other hand, since Phoenix is implemented in C, the 2-D array structure is represented as a row-major array in memory. Hence, the above design optimizes for the locality of reduce phase; map phase exhibits little locality since the sequence of keys confronted during the phase is close to random.

With the medium-sized datasets used on small-scale systems, this data structure design was acceptable. But when the input was significantly increased on the 256-thread NUMA system, we observed some performance pathologies. As described earlier, the **keys array** structure was implemented as a sorted array to utilize binary search. Although it provided fast lookup, the downside of this implementation was that when the buffer ran out of space, the entire array had to be reallocated. Even worse, when a new key was inserted in the middle of the array, all the keys coming lexicographically after the new key had to be moved. As we increased the input dataset, this problem became more prominent. Similar buffer reallocation issues occurred on the **vals array** as well.

Improving the **keys array** structure was more complicated than expected. Since massive amounts of intermediate pairs were being inserted into the hash table, slight latency increases such as pointer indirection obliterated any performance gains. For example, we tried a design that implemented **keys array** as a linked list, but failed to improve performance. Replacing the structure with a tree was not an option either, since frequent rebalancing would have been costly. Instead, we settled by significantly

increasing the number of hash buckets so that on average only one key resides in a **keys array**. Detailed reasoning behind this decision will follow (see Section 3.3.3), but as a rule of thumb, increasing the number of hash buckets linear to the number of unique keys was sufficient.

For the **vals array**, we implemented an iterator interface to the buffer and exposed this interface to the user reduce function. This allowed for a buffer that is comprised of a few disjoint memory chunks (good for locality purposes) while still maintaining the sequential accessibility through the iterator interface. This design completely removed the reallocation issue. Additionally, to tolerate the increased memory latencies in the NUMA system, we implemented prefetching functionality behind the interface. Note that unlike the map phase where we can avoid accessing remote memory by assigning tasks based on locality, in the reduce phase a worker might not be able to avoid performing remote accesses since the intermediate pairs with the same key may be produced by workers across all the locality groups. Therefore, sequential accesses and prefetching to **vals array** become important.

We also experimented with combiners [23], where each thread invoked the combiner at the end of the map phase to decrease the amount of reduce phase remote memory traffic. However, with the prefetching in place for the reduce phase, combiners made little difference.

After the data structure was improved, other minor factors such as task generation time were affected by the increased input size as well. We detail the issues in Section 3.3.3.

3.2.3 OS Interaction Optimizations

Once the algorithm and the implementation are optimized, interactions with the operating system are apt to become the next bottleneck. Specifically, Phoenix frequently uses two OS services: memory allocation and I/O.

Phoenix exerts significant pressure on memory allocators, not only due to its large memory footprint, but also its peculiar allocation pattern. In terms of memory footprint, Phoenix generates a significant amount of intermediate and output data, where

the memory needs per key are usually unpredictable. As for the allocation pattern, in Phoenix, the thread that allocates memory (e.g., a map thread) rarely is also the thread that deallocates the memory (e.g., a reduce thread). This mismatch can incur contention on the per-thread heap locks when multiple threads try to manipulate the same thread heap.

We experimented with a sizeable number of memory allocators to compare their performance. However, at high thread count, we noticed that the parallel allocator performance was limited by the scalability of the **sbrk()** system call. We discuss the issues in detail in Section 3.4.

On the other hand, the **mmap()** system call is used to read in input data. Once the user passes the pointer to the memory region allocated through **mmap()** as a runtime input, multiple threads will concurrently fault in the input data while invoking their map functions. Therefore, as we increased the number of worker threads, we observed **mmap()** scalability being crucial to some of the workloads.

More importantly, **mmap()** was also being used to implement thread stacks. Using the SunStudio profiler [68], we found that thread join was a major bottleneck at large thread count. Further analysis revealed that the increased thread join time was due to the calls to **munmap()**, which was used to deallocate thread stacks. To address the issue, we implemented a thread pool that reuses threads across different MapReduce phases and iterations (multiple sets of map and reduce functions), reducing the total number of calls to **munmap()** to a strict minimum.

3.3 Performance Evaluation

3.3.1 Performance Improvements Summary

We implemented the optimizations in Section 3.2 on Phoenix and measured its performance by executing the applications included in the original release. Table 3.2 describes the workloads and their input datasets. We omitted the **reverse_index** workload because it was I/O bound. Compared to the original release [60], we significantly increased the input datasets to stress all the hardware contexts available

Application	Input Dataset	Description
histogram	1.4 GB BMP image	Computes the RGB histogram of an image
kmeans	500,000 data points	Performs k-means clustering analysis over data points
linear_regression	3.5 GB data	Applies linear regression best-fit over data points
matrix_multiply	3,000 × 3,000 matrices	Matrix multiplication
pca	3,000 × 3,000 matrix	Performs principal components analysis on a matrix
string_match	500 MB dictionary	Pattern matches a set of strings against streams of data
word_count	500 MB random texts	Counts the number of unique word occurrences

Table 3.2: The workloads used with the Phoenix MapReduce system.

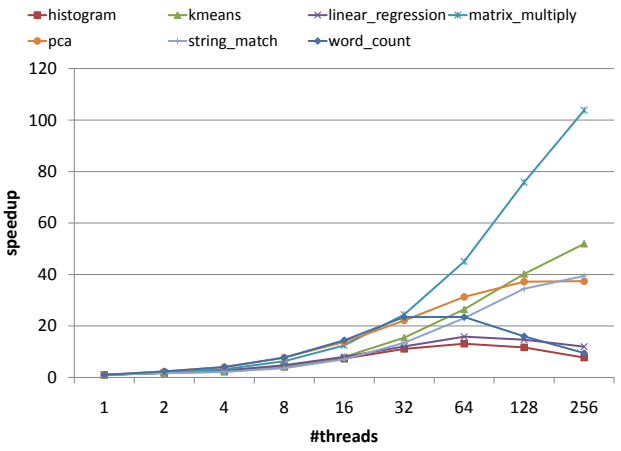


Figure 3.4: Application speedup with the optimized version of Phoenix.

on our system. In all the experiments, we bind threads so that we fill up a chip (64 threads) first before providing for the other chips.

Figure 3.4 summarizes the scalability results for the optimized runtime. Specifically, workloads **matrix_multiply**, **kmeans**, **pca**, and **string_match** scale up to 256 threads. However, some of the workloads still do not scale particularly well. We discuss their bottlenecks in detail in Section 3.4, but in the remainder of this section, we first focus on the optimizations that turned out to be successful.

Figure 3.5 measures the relative speedup of the new runtime over the original. It essentially compares Figure 3.4 and Figure 3.1, using the same dataset. In the figure, the top and bottom of each vertical bar denotes the maximum and minimum performance improvement achieved at a particular thread count, respectively; the horizontal line that connects those bars represent the harmonic means of speedups

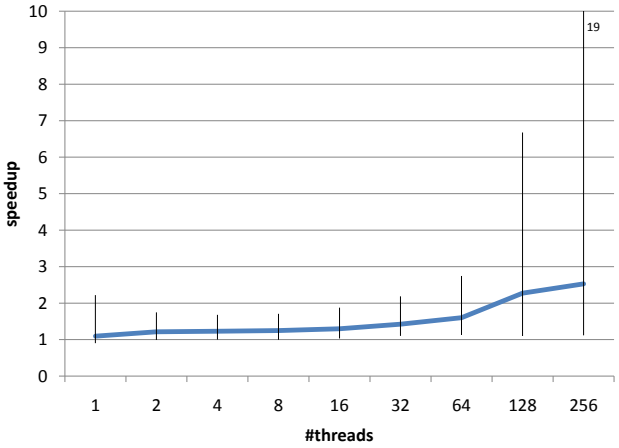


Figure 3.5: Relative speedup over the original Phoenix system.

obtained across all the workloads.

The optimized runtime leads to improvements across all thread counts. For less than 64 threads (single chip), the average improvement is 1.5x, and the variation across applications is rather small (maximum of 2.8x). For large-scale, NUMA configurations with 128 or 256 threads, however, the optimizations are significantly more effective, reaching 19x in maximum and 2.53x on average. We observed similar differences between the two runtimes on the 8-socket, 32-core Opteron system as well.

Since the various optimizations had interrelated effects on the overall performance, e.g., iterators interacting with memory allocation pressure, we could not fully isolate the contributions of each class of optimization for all the workloads. Qualitatively speaking, OS interaction optimizations had the most impact, implementation optimizations next, and algorithmic optimizations the least. In the following subsections, we provide further insights into the impact of each optimization class.

3.3.2 Impact of Algorithmic Optimizations

We implemented the per locality group task queue and task distribution as described in Section 3.2.1. We utilized the **meminfo()** interface in Solaris [67], which returns the locality group of the physical memory that backs the virtual address being queried. Based on the locality group information for an input data chunk, a task is queued

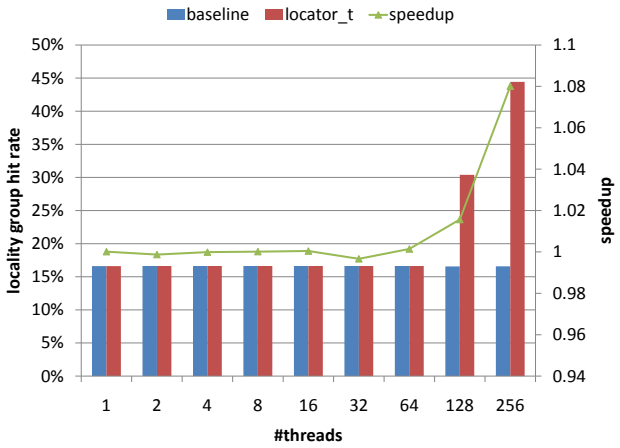


Figure 3.6: Locality group hit rate improvement on **string_match**.

to the pertaining locality group task queue. Note that the locality-aware task distribution is orthogonal to the use of an application specific splitter on the input data; once the input is split in whatever means necessary, map tasks are distributed in a locality-aware manner.

Figure 3.6 compares the *locality group hit rate* of the optimized runtime against that of the baseline for the **string_match** application. The hit rate represents the percentage of map task data that is served by a memory channel attached to the local chip (low latency memory access). When threads are confined to one chip, they are forced to take locality group misses when accessing data on remote memory. Hence, for this case, both schemes exhibit about 17% locality group hit rate. However, when threads are created across multiple chips, careful placement of tasks can readily improve locality group hit rate. The proposed optimization effectively utilizes this opportunity, which leads to 30% and 44% locality group hit rate at 128 and 256 threads, respectively. On average, at 256 threads, this optimization leads to 8% speedup improvement over the baseline.

Workload	Populated Bkts.	Keys / Bkt.	Vals / Key
histogram	256	2	256
kmeans	100	1	78
linear_regression	5	1	905
matrix_multiply	0	0	0 (bypasses reduce)
pca	21 256	2 274	1 (phase 1) 1 (phase 2)
string_match	0	0	0 (bypasses reduce)
word_count	256 244	156 31	34 (phase 1) 1 (phase 2)

Table 3.3: Hash table key distribution.

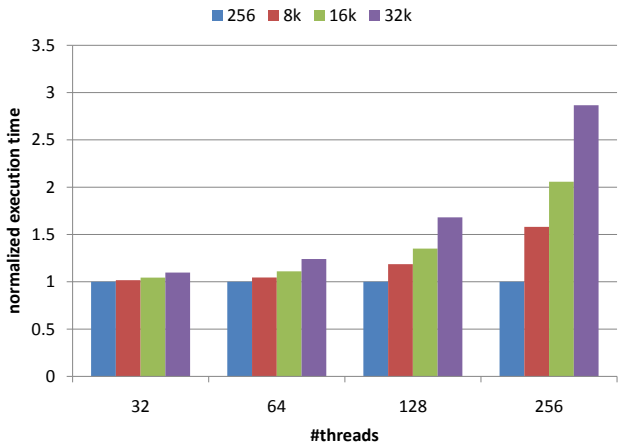


Figure 3.7: **kmeans** sensitivity to hash bucket count.

3.3.3 Impact of Implementation Optimizations

Hash Table Improvements

In Section 3.2.2, we discussed how large datasets lead to performance issues with the core Phoenix data structure that stores intermediate key-value pairs. We addressed the issue by increasing the number of hash buckets, and by implementing the iterator interface. Especially, increasing the hash bucket count avoided buffer reallocation and copying. However, not all the workloads benefited from the optimization. Table 3.3 shows the distribution of keys in the hash table for 64 threads, when the number of hash buckets is set to 256. Note that these are the averages across all the threads; the actual number of keys or values for each thread can be significantly higher.

From the table, it can be seen that although workloads such as **pca** and **word_count**

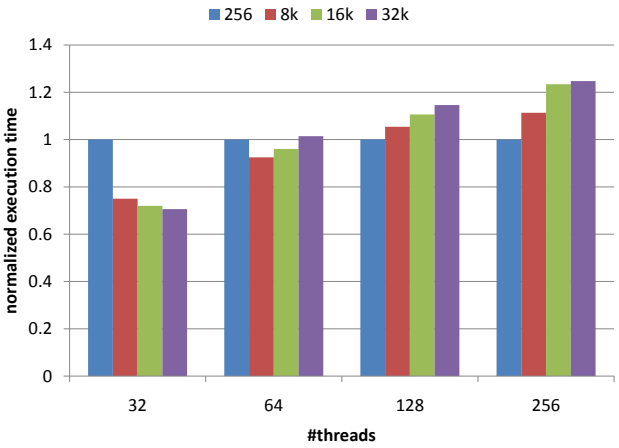


Figure 3.8: **word_count** sensitivity to hash bucket count.

generate significant amount of unique keys, other workloads, e.g., **kmeans** and **linear_regression**, do not.

Those workloads with small key count generate a fixed number of unique keys, regardless of the input. Therefore, increasing the hash table size ended up in a net slowdown for such applications, and this slowdown actually became worse with more threads. Figure 3.7 shows the normalized execution time of **kmeans** increasing with higher bucket counts (time normalized to that with 256 buckets).

Our analysis showed that when a reduce worker traversed down a row of the 2-D array (see Section 3.2.2), there was a fixed cost just to find a remote **keys array** structure empty. Due to NUMA effects, this cost increased when more chips were used. In Figure 3.7, the effect manifests as worse performance with increasing number of threads. Therefore, for workloads that did not generate a large number of unique keys, the hash bucket count had to be kept small.

Even for the workloads that did benefit from the increased bucket count, the improvements were not uniform. Figure 3.8 shows the normalized execution time of **word_count** over various hash bucket counts. Again, the execution time is normalized to that of the 256 buckets case. At low thread counts (∼32 threads), increasing the hash table size results in speedup. However, as more threads are added, the benefit is reversed due to the increased time spent in the *merge phase*.

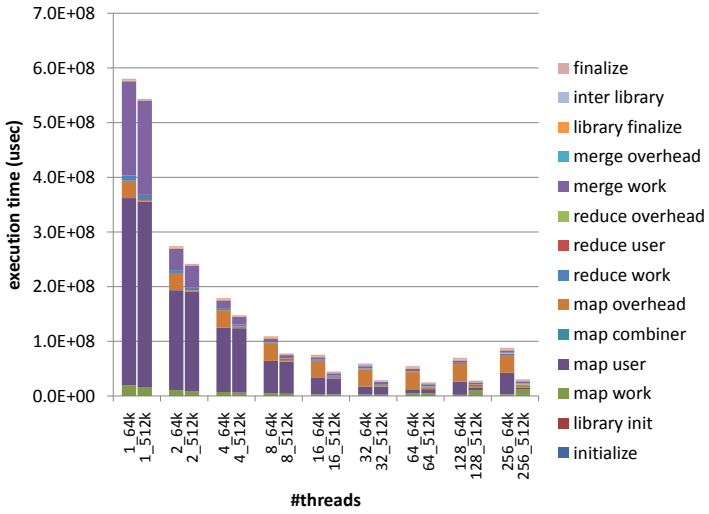


Figure 3.9: **pca** execution time breakdown.

In Phoenix, worker threads output one result structure for each reduce task, which are merged by a tree-based parallel merge sort at the end of the reduce phase. Since each hash bucket amounts to one reduce task, having an excessive amount of hash buckets results in a *wider* merge tree. At the same time, increasing the thread count makes the merge tree *deeper*. When the overhead in the merge phase outweighed the savings from reduced memory reallocation, increasing the number of hash buckets resulted in a net slowdown.

To summarize, no single hash table size worked best for all the workloads. For applications that generated small number of keys, the hash table size had to be kept to a minimum. On the other hand, for workloads with large amount of keys, the hash bucket count could only be increased as long as it did not negatively affect the merge phase execution time. In the optimized version of Phoenix, we made the hash table size user tunable, while providing the default values for efficient execution on each workload. As a rule of thumb, increasing the number of hash buckets proportional to the number of unique keys was sufficient.

Without Thread Pool

#threads	Time (sec)	#calls	Time / Call (msec)
8	0	20	0
16	0.31	1947	0.16
32	0.689	4499	0.15
64	1.695	9956	0.17
128	4.548	14661	0.31
256	8.219	14697	0.56

With Thread Pool

#threads	Time (sec)	#calls	Time / Call (msec)
8	0	10	0
16	0.002	13	0.15
32	0.002	18	0.11
64	0.008	33	0.24
128	0.016	44	0.36
256	0.065	102	0.64

Table 3.4: The effect of the **munmap()** system call on **kmeans**.

Task Generation Time

We also made an interesting observation regarding the effect of map task chunk size on the overall execution time. As reported in [60], it is generally understood that varying the chunk size relative to the cache size does not have a significant impact on locality, due to the streaming nature of MapReduce applications.

However, the chunk size had a direct impact on the map phase task generation time, which started to consume a noticeable amount of execution as the input datasets became larger. Figure 3.9 shows the breakdown of the **pca** execution time as we vary the input chunk size from 64 KB to 512 KB. At the default value of 64 KB (left columns), **pca** generates millions of map tasks, which is captured as the prominent **map overhead** portion in the figure. At higher number of threads, this overhead actually dominates the entire execution time.

In our case, increasing the chunk size up to 512 KB was sufficient to reduce the task generation time to a minimum (right columns in Figure 3.9). However, for systems with even larger scale, the task generation phase might as well be parallelized.

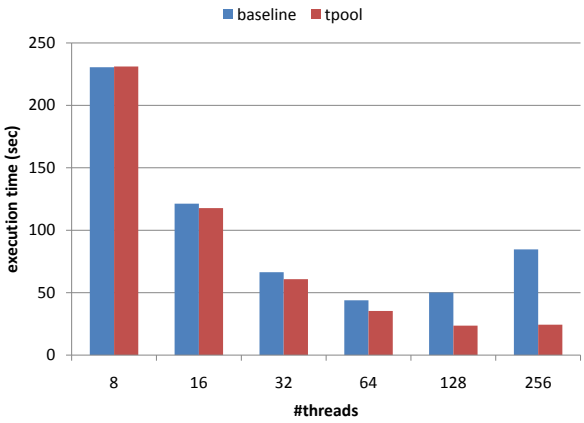


Figure 3.10: **kmeans** performance improvement due to thread pool.

3.3.4 Impact of OS Interaction Optimizations

As described in Section 3.2.3, the **mmap()** and **munmap()** system calls used to implement and destroy thread stack introduce high overhead at large thread counts. Especially, **kmeans** was affected the most since it iterates over multiple MapReduce instances, which results in repeatedly creating and destroying a large number of threads. The upper half of Table 3.4 shows the effect of **munmap()** on **kmeans**, measured with **truss**. It is expected that the number of calls to **munmap()** increases with increasing number of threads; but what is problematic is that each call to **munmap()** takes longer as we use more threads.

To alleviate this problem, we implemented a thread pool. The lower half of Table 3.4 presents the same data when the thread pool is enabled. The thread pool reduces the absolute number of calls to **munmap()**, which makes the time spent inside **munmap()** negligible. As a result, in Figure 3.10, the workload shows notably improved scalability.

However, the time spent for each call to **munmap()** stays more or less the same (see Table 3.4). We discuss the issue in detail in Section 3.4.2.

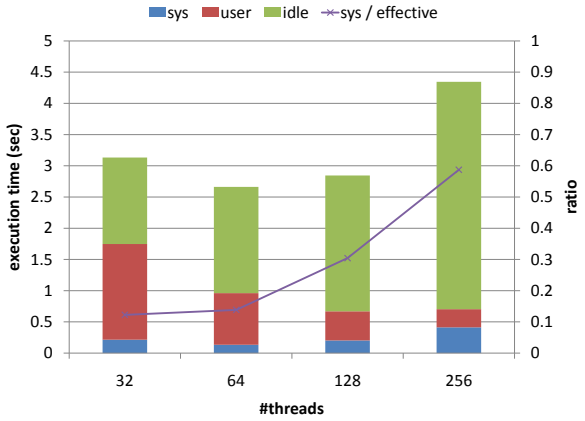


Figure 3.11: Execution time breakdown on **histogram**.

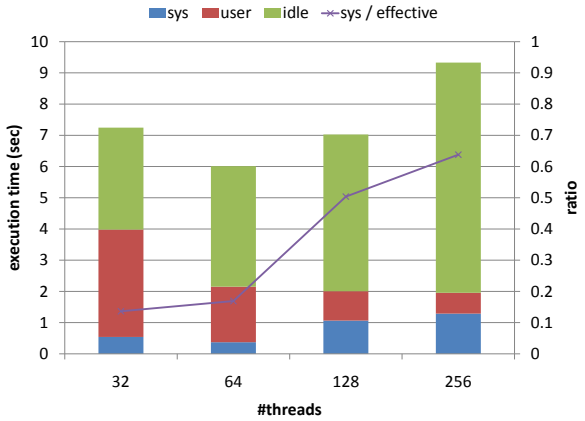


Figure 3.12: Execution time breakdown on **linear_regression**.

3.4 Challenges and Limitations

Although we were able to significantly improve the scalability of Phoenix, workloads **histogram**, **linear_regression**, and **word_count** still did not scale up to 256 threads. We used **/usr/bin/time** to assess where the execution time was being spent. Figures 3.11, 3.12, and 3.13 shows the result for **histogram**, **linear_regression**, and **word_count**, where *effective* time is defined as **user** + **sys** time.

It is clear that the 3 non-scaling workloads share two common trends. First, the idle time increases with increasing number of threads, dominating the total execution

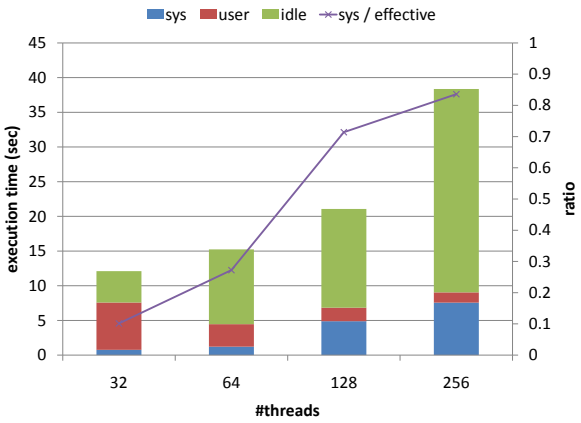


Figure 3.13: Execution time breakdown on **word_count**.

time at high thread count. Second, as we go beyond the single chip boundary, the portion of actual computation time assumed by the kernel code (**sys / effective**) significantly increases.

Through profiler analysis [68], we also found that at 256 threads, **histogram** and **linear_regression** spent 64% and 63% of their execution time idling for data page fault, and that **word_count** spent 28% of its execution time in **sbrk()** called inside the memory allocator. **word_count** spent an additional 27% of execution time idling for data pages as well. In short, scalability on these three workloads were hampered by the two OS services: memory allocation and I/O.

We would like to be clear that these issues are not related to the physical I/O performance. We observed the same issue even when we warmed up the OS file system cache by running the same workload with the same input multiple times. Measurements were taken only when the workload execution time had been stabilized. Hence, all the scalability issues we report here are due to the serialization introduced by the memory management subsystem of the operating system. It is also important to point out that simply substituting **mmap()** with **read()** resulted in unacceptable performance for those workloads that depended heavily on **mmap()**: namely, **histogram** and **linear_regression**.

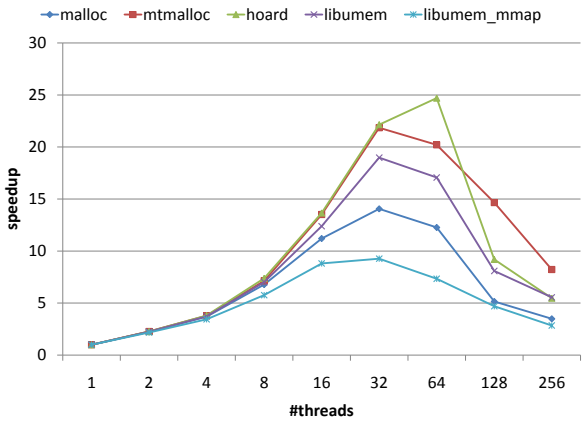


Figure 3.14: Memory allocator scalability comparison on **word_count**.

3.4.1 Memory Allocator Scalability

Compared to the libc sequential allocator, the Solaris concurrent allocator **mtmalloc** provided improved performance. However, when it frequently called **sbrk()**, the allocator exhibited scalability problems. Inside **sbrk()**, a single user-level lock kept other threads from expanding the process' data segment, while per-address space locks protected in-kernel virtual memory object. As more threads relied on **sbrk()** to satisfy allocation requests, **sbrk()** became the point of contention.

We experimented with various allocators[1]. Figure 3.14 compares the scalability of different memory allocators on **word_count**. Solaris alone provided (at least) 5 different memory allocator implementations: **malloc**, **bsdmalloc**, **mtmalloc**, **libumem**, and **mapmalloc**. Among those, **mtmalloc** and **libumem** had concurrency support. Especially, **libumem** could be paired with a backend using **mmap()** instead of **sbrk()**; in Figure 3.14, **libumem_mmap** denotes the performance of **libumem** with **mmap()** backend. Also, the **hoard** trend line denotes the performance of the Hoard [6] memory allocator.

In summary, no allocator successfully scales up to 256 threads—once the **sbrk()** becomes the bottleneck, no allocator is able to scale. We believe this issue opens

[1]We could not experiment with TCMalloc, an open-source memory allocator with NUMA extensions [30], since the library was not ported to SPARC.

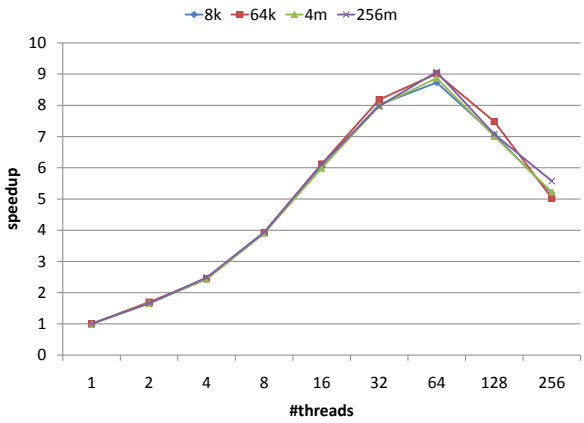


Figure 3.15: **mmap()** microbenchmark scalability results.

up a new research opportunity for concurrent allocators and virtual memory subsystems on large-scale shared-memory systems. Comparing the results of **libumem** and **libumem_mmap** also shows that the **mmap()** backend performs even worse than the **sbrk()** backend: Despite having no user-level lock like **sbrk()**, the **mmap()** backend scales worse than the sequential **malloc**.

3.4.2 mmap() Scalability

The scalability of the **mmap()** system call is critical to shared-memory runtimes like Phoenix, since workloads typically **mmap()** user input data and use multiple threads to fault in. To quantify the problem, we created a microbenchmark that assessed **mmap()** scalability. It simply calls **mmap()** on a user input file, and statically assigns the data chunks across threads. Then the threads compute the total sum of all the data, by streaming through the chunks assigned to them. No user level synchronization primitives are called, and memory allocation is only performed in the sequential portion of the program (an array to hold **pthread_t** structures).

Figure 3.15 shows the microbenchmark results over varying page sizes. In short, **mmap()** exhibits serialization as we cross the chip boundary. The fact that the performance is identical regardless of the page size also suggests that the problem is not related to TLB pressure. The issue is not limited to Solaris, either: Executing

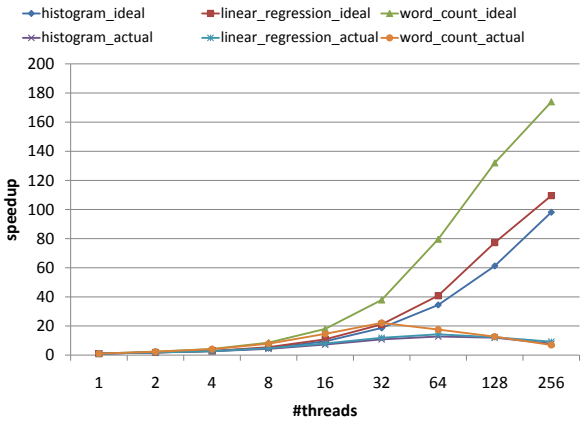


Figure 3.16: Ideal vs. actual speedup for non-scalable workloads.

the same microbenchmark on Linux produced similar results.

3.4.3 Importance of OS Scalability

To investigate to which degree these workloads were affected by the poor scalability of OS primitives, we tried to predict the ideal scalability of each. Figure 3.16 shows the achievable speedup for each workload, when only the time spent executing the user code is considered (including the Phoenix runtime). It is clear that the OS scalability is a significant issue, as the user time on these applications scales well.

To better support our claim, we also created a synthetic benchmark that tries to get rid of **sbrk()** and **mmap()** effects on **word_count** application by, first, pre-allocating all the memory it needs, and second, by reading in all the data into the user address space before it starts executing. In Figure 3.17, we can see that removing each OS bottleneck gradually improves the application performance. When both bottlenecks are removed, the workload scales well up to 256 threads.

3.4.4 Discussions

Scaling operating systems for large-scale, shared-memory systems is an active field of research [12, 63]. Nevertheless, we have shown that the problems are more severe and

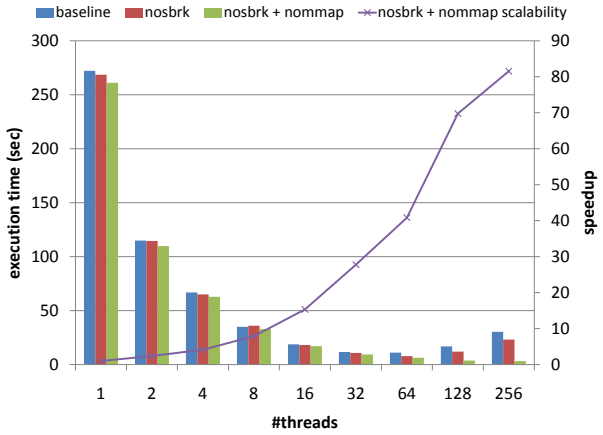


Figure 3.17: **word_count** performance without **sbrk()** and **mmap()** effects.

readily encountered than expected, even with a highly optimized operating system like Solaris. Clearly, further research is warranted.

It is also important to point out that these issues are specific to shared-memory MapReduce. Unlike the cluster implementation, in shared-memory MapReduce, worker threads share a single address space. Therefore, how one thread allocates memory and performs I/O has a direct impact on the overall system performance. In an effort to localize OS interactions as much as possible, we also tried MapReducing-MapReduce: instantiating one MapReduce instance per locality group and crossing the chip boundaries only at the final phase to merge results. However, since those MapReduce instances essentially comprised a single process, they still suffered from similar OS scalability issues.

Another way to overcome the problems we faced on the NUMA system would be to implement each MapReduce worker as a separate process running on a dedicated OS instance (virtual machine) but on the same physical machine. Unlike the cluster environment where workers communicate over the network infrastructure, we could build virtual machine mechanisms that would allow communications across different virtual machines to take place through shared memory, without even invoking the hypervisor every time. We leave this topic as future work.

3.5 Conclusions

We have seen that the problem of locality already exists: On a large-scale shared-memory system comprised of multiple many-core chips, latency and NUMA issues were prevalent. Therefore, to exploit locality on a MapReduce system, we had to optimize the runtime at all possible layers.

In particular, since the MapReduce scheduler provided intermediate result buffering, locality on the internal data structures was important. Other structured programming systems could benefit from our analysis, by applying similar techniques to improve the locality on the intermediate buffer structures. We also showed that the scalability of the OS primitives can impose significant challenges to a scalable runtime. Any runtime that targets large-scale many-core systems should minimize operating system interactions as much as possible.

While our analysis focused on a multi-socket configuration, future many-core processors will exhibit similar NUMA characteristics. Our findings will maintain relevance to those processors as well.

Chapter 4

Graph-Based Locality Analysis Framework

We now shift our focus to a single many-core chip, and study how locality can be exploited for the more challenging case—the task-parallel programming systems. Unlike the structured programming systems (see Section 2.3.1), task-parallel programming systems lack the explicit data dependency information. Therefore, a scheduler should *synthesize task structures* to exploit locality. We first develop a graph-based locality analysis framework to derive insights into how such a structure could be constructed, and study how the structure interacts with the underlying cache hierarchy. We also develop a reference scheduler we later use to quantitatively explore the importance of various aspects of a locality-aware schedule.

4.1 The Task-Parallel Programming Model

We first formally define the task-parallel programming model. Listing 4.1 denotes an example task programming interface.

Under task-parallel programming, a *parallel section* denotes the region of code where parallel execution is performed. For example, in Listing 4.1, to specify a parallel section, a user calls the **taskQEnqueue()** function with a *task function* (**taskfn**) and a *task space* as arguments. A task space, which basically amounts to a

```
// Specify the parallel section by providing
// (1) the task function and (2) task space
taskQEnqueue(taskfn, num_dims, size_arr);

// Execute the parallel section in parallel
taskQWait();
```

Listing 4.1: The task-parallel programming interface.

task index grid, is specified as the number of dimensions (**num_dims**) and the size in each dimension (**size_arr**).

When the **taskQEnqueue()** function is called, the runtime spawns a number of threads; similar to [44, 62], we assume one thread is created per hardware context. It then creates tasks by instantiating the task function with each point in the task space as the argument. These tasks are then scheduled for execution. In particular, we assume a *task queue*-based task management system [10], where the runtime maintains a queue of tasks for each thread. To schedule a task, the runtime enqueues it to one of the task queues.

The tasks are actually executed when the **taskQWait()** function is called. For execution, each thread dequeues a task from its queue and executes it. If the queue is empty, the thread tries to steal tasks from other threads' queues. When all the tasks have been executed, threads are terminated, and the **taskQWait()** function returns to give the control back to the main thread.

As can be seen, for this model, all the tasks are available at the beginning of a parallel section. More importantly, tasks may execute in any order, and any task execution leads to a computationally correct result [54, 37]. While this model does not incorporate tasks with control dependency [29, 16], such model can be reformulated as a series of control-independent parallel sections (e.g., each level of task tree forms a parallel section). At the least, control dependency could be cast as data dependency, and tasks could be scheduled using the methods applied for structured programming systems (see Chapter 3).

Workload	Task	Tasks that Share Data	Access Pattern	#tasks	Task Size	Input
hj	Performs single row lookup	Look up same hash bucket (clustered sharing)	hash tables	4,096	157	4,460
bprj	Reconstructs a sub-volume	Work on sub-volumes mapping to overlapping pixels (clustered sharing)	3-D traversal	4,096	415	3,650
gjk	Operates on sets of object pairs	Work on overlapping object sets (clustered sharing)	pointer access	384	4,950	940
brmap	Maps pixels from a sub-image	Work on sub-images mapping to overlapping pixels (structured sharing)	trajectory-based	4,977	1,298	22,828
conv	Operates on square-shaped image block	Operate on overlapping pixels (structured sharing)	grid-based	67,584	667	26,400
mmm	Multiplies a pair of sub-matrices	Use common sub-matrix (clustered sharing)	grid-based	4,096	49,193	344,064
smvm	Multiplies one row of matrix	Touch overlapping elements in a vector (clustered sharing)	sparse matrix	4,096	891	42,385
sp	Solves a subset of equation lattice	Work on neighboring lattices (structured sharing)	grid-based	1,l56	5,956	439

Table 4.1: Workloads used in this chapter. **Task Size** is reported in terms of number of dynamic instructions. **Input** reports the total sum of task footprints in KB.

4.1.1 Task-Parallel Workloads

Table 4.1 summarizes the task-parallel workloads used in this chapter. Workloads are chosen to cover a variety of locality and access patterns, task sizes, and data set sizes. In particular, notice that the applications are not necessarily organized in a cache oblivious [28] or recursive manner. Most of them are adopted from previous work on task scheduling [44, 62, 41]. As discussed, for these workloads, the execution order of tasks does not affect the computation result.

hj models the probe phase of hash-joining two database tables. A hash join is typically comprised of two phases; in the *build* phase, the smaller table, the *inner table*, is converted into a hash table by applying a hash function to the key of each row; in the *probe* phase, the larger table, the *external table*, is scanned to find the relevant rows by looking up the hash table. Since the external table in general is significantly larger than the inner table, the workload is dominated by the probe phase. A task amounts to a single row lookup, and locality can be achieved by grouping those tasks that lookup the same hash bucket.

bprj maps 3-D scan data back to a 2-D image. The mapped location on the image is determined by the camera position in the 3-D space, and the camera traversal is described by an elliptical function. Single task maps a small volume of 3-D points, and locality exists across those tasks that map to the same block on the 2-D image.

gjk models the collision detection logic used in 3-D graphics and gaming. Individual task performs collision detection for pairs of likely-colliding objects. By grouping tasks that operate on the same objects, locality can be accumulated.

brmap performs interpolation of an input image based on the pixel location. A task operates on a rectangle of input pixels, and row-wise neighboring rectangles share cache lines. Grouping row-wise tasks together benefits from locality.

conv workload performs a 2-D convolution (5x5) on each image pixel. A single task operates on a square-shaped sub-image. Locality exists across those tasks that operate on overlapping image pixels.

mmm implements blocked matrix-to-matrix multiplication. A task performs multiplication over a pair of sub-matrices. Tasks that work on the same sub-matrix can benefit from locality.

smvm performs sparse matrix-vector multiplication. Each row of the matrix represents a noisy observation of pre-determined patterns. Individual task performs a single row-to-vector multiplication, so locality can be obtained by grouping those tasks that touch similar regions of the vector.

sp workload solves a set of nonlinear partial differential equations using the scalar pentadiagonal algorithm. A task solves a subset of equation lattice, and grouping tasks that work on neighboring lattices provides locality.

4.2 Graph-Based Framework Overview

To find and understand the *inherent* locality patterns across our workloads, we develop a *graph theory-based* locality analysis methodology. This methodology is independent of detailed hardware characteristics such as the number of cores and the cache hierarchy.

Specifically, we start with the read and write sets for each task. We profile each

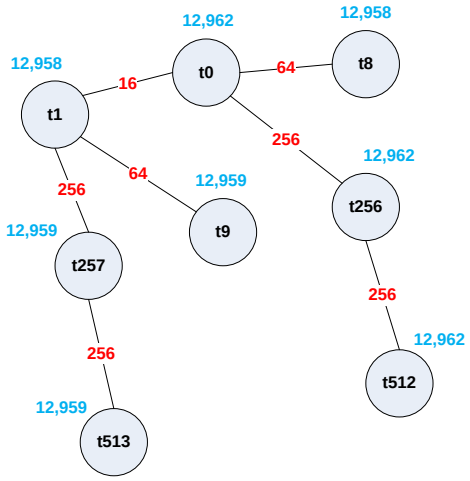


Figure 4.1: Example task sharing graph.

workload to collect heap access traces at cache line granularity[1]; we discard access ordering information to obtain a set representation. Using the read and write set information, we construct a *task sharing graph*. Figure 4.1 shows the part of a task sharing graph collected from one of our workloads[2].

In a task sharing graph $G(V, E)$, a *vertex* represents a task, and an *edge* denotes sharing. A *vertex weight* is the task size in terms of number of dynamic instructions, and an *edge weight* is the number of cache lines shared between the two tasks connected by the edge.

A locality-aware task schedule maps vertices to core resources, taking into account both vertex weights (to balance load) and edge weights (to enhance locality). Two techniques to help construct such a schedule are (1) grouping tasks that share data, and (2) ordering tasks to minimize reuse distance.

Grouping: Executing a set of tasks (a *task group*) on cores that share one or more levels of cache captures the data reuse across tasks. Therefore, careful task grouping can help us construct a schedule that maximizes locality.

Even with profile information about each task's read and write sets, creating an optimal set of task groups is an *NP-hard* problem. We therefore use a heuristic

[1]The line size represents the *logical* cache line size, not the actual, architecture-dependent cache line size. We used 64 bytes.

[2]**mmm** with small task configuration (see Section 4.4).

graph partitioning tool (METIS [40]) to generate quality task groups. METIS divides the vertices from a task sharing graph into a given number of groups while trying to (1) maximize the sum of edge weights internal to each group (i.e., data sharing captured by the task group), and (2) equalize the sum of vertex weights in each group.

Ordering: Executing tasks in an optimal order minimizes the distance between reuses of shared data across tasks. Reducing reuse distance makes it easier for caches to capture the temporal locality because lines are less likely to be evicted between reuses. It may also create incremental, smaller levels of working set. We consider both *task ordering* and *group ordering*. *Task ordering* specifies the traversal order of vertices for a task group. On a task queue-based task management system, task order denotes the *dequeue order* of tasks within the same task group. *Group ordering* specifies the execution order of task groups.

Creating an optimal order for tasks is also an *NP-hard* problem. We therefore use a heuristic to provide a high quality ordering. Specifically, we apply Prim's algorithm to construct a maximum spanning tree (MST), and use the order that vertices are added to the MST. In architectural terms, Prim's algorithm accumulates the read and write sets of scheduled tasks, and picks the task whose read and write sets exhibit the maximum intersection with the cumulative sets as the next task to execute. To construct a task ordering, we apply MST on the task sharing graph. To construct a group ordering, we first coarsen the task sharing graph to create a *task group sharing graph*, where each uber node represents a task group; we then apply MST to the task group sharing graph.

We explore the locality implications of these next, along with the impact of task size. Initially we discuss locality assuming a single core with a single cache. We later extend the analysis to multiple cores with complex, multi-level cache hierarchies.

4.2.1 Implications of Task Grouping on Locality

Table 4.2 shows the statistics collected for some of our workloads, when we use METIS to divide the graphs into different sizes of groups. All of these characteristics are architecture-independent. *Sum of footprint* denotes the average sum of individual

Rel. Task Grp Size	1	1/2	$1/2^2$	$1/2^3$	$1/2^4$	$\mathbf{1/2^5}$	$1/2^6$	$1/2^7$	$1/2^8$	$\mathbf{1/2^9}$	$1/2^{10}$
#tasks / Group	4,096	2,048	1,024	512	256	**128**	64	32	16	**8**	4
Sum of Footprint	42,385	21,192	10,596	5,298	2,649	**1,324**	662	331	165	**82**	41
Union of Footprint	4,946	2,618	1,408	757	400	**211**	110	59	34	**21**	14
Sharing Degree	0.03	0.03	0.04	0.06	0.11	**0.21**	0.38	0.58	0.75	**0.90**	0.99
Cut Cost (E+07)	0.00	2.04	3.13	3.68	3.98	**4.15**	4.27	4.41	4.77	**4.97**	5.08

(a) Statistics collected for **smvm**.

Rel. Task Grp Size	1	1/2	$1/2^2$	$1/2^3$	$1/2^4$	$\mathbf{1/2^5}$	$1/2^6$	$\mathbf{1/2^7}$	$1/2^8$	$1/2^9$	$1/2^{10}$
#tasks / Group	4,096	2,048	1,024	512	256	**128**	64	**32**	16	8	4
Sum of Footprint	3,650	1,825	912	456	228	**114**	57	**28**	14	7	3
Union of Footprint	177	91	51	29	16	**9**	5	**3**	1	1	1
Sharing Degree	0.01	0.01	0.02	0.04	0.07	**0.13**	0.22	**0.38**	0.58	0.78	0.92
Cut Cost (E+06)	0.00	4.19	6.31	7.37	7.91	**8.19**	8.36	**8.48**	8.61	8.71	8.77

(b) Statistics collected for **bprj**.

Table 4.2: Statistics over varying task group sizes for **smvm** and **bprj**. **Relative Task Group Size** of 1 denotes the case where all the tasks are grouped into a single task group. **Sharing Degree** of 1 means on average, when a cache line is shared, it is shared by all the tasks within a task group. All footprints are reported in KB.

task footprints for each task group, while *union of footprint* denotes the average size of the union of individual task footprints (i.e., shared lines are counted only once) for each task group. We discuss the other statistics later[3].

Intuitively, to maximize locality, a task group should be sized so that the *working set* of the group fits in cache. In that regard, the sum of footprint represents an upper bound on working set size for a task group (i.e., when a schedule fails to exploit any reuse across tasks), and the union of footprint represents a lower bound. For a fully-associative cache, the union of footprint should accurately track the working set of a task group. However, due to conflict misses and unaccounted-for stack accesses, the actual working set size is between the union and sum of footprint. For example, for **smvm**, when we have 8 tasks per task group, the actual working set would be between 21 KB (= union of footprint) and 82 KB (= sum of footprint).

Hence, assuming a single cache, task groups should be sized so that the cache size is between the union and sum of task footprints. However, strictly following this rule may lead to other inefficiencies. In an actual implementation, grouping

[3]Since read sharing is dominant in our workloads, we give equal weight to read and write sharing in computing our metrics. It is straightforward to assign different weights to reflect different costs for read and write sharing.

too few tasks together might introduce high scheduling overhead, and too many tasks could introduce load imbalance. Such issues can be avoided by performing multi-level grouping, or by executing tasks within a task group in parallel (see Section 4.3).

4.2.2 Implications of Ordering on Locality

Task ordering and group ordering can maximize temporal locality when reuse distance is minimized. We define two characteristics, *sharing degree* and *cut cost*, that dictate the importance of reuse distance on certain data.

In Table 4.2, given a shared cache line, the *sharing degree* denotes the average number of tasks within a task group that share. The table reports normalized sharing degree, so a sharing degree of 1 means on average, when a cache line is shared, it is shared by all the tasks within a task group. In terms of locality, the sharing degree of an application indicates the potential impact of task ordering. Specifically, a high sharing degree means that data is shared by a large fraction of tasks. For example, in Table 4.2a, it can be seen that for task groups that would fit in a 32 KB cache (8 tasks per group), about 90% of the **smvm** tasks within a group share the same cache line. Therefore, task ordering will have little impact on the reuse distance of shared data. On the other hand, when the sharing degree is low, task ordering could have a significant impact on locality. In Table 4.2b, for **bprj** with 32 tasks per group, the sharing degree is 0.38, which is quite low compared to **smvm**. We can therefore conjecture that **bprj** would be more sensitive to task ordering.

On the contrary, the *cut cost* in Table 4.2 denotes the sum of the edge weights for vertices in different groups (i.e., data sharing not captured within a single task group). Our workloads exhibited two distinct cut cost patterns: *clustered sharing* and *structured sharing*. The relative importance of group ordering depends on this workload pattern.

Workloads with Clustered Sharing: When a task sharing graph is drawn for these workloads, the graph exhibits disjoint clusters of tasks, where each cluster is comprised of a *clique* of tasks sharing the same data structure. For example, groups of **hj** tasks share the same hash bucket, and **smvm** tasks the same vector region.

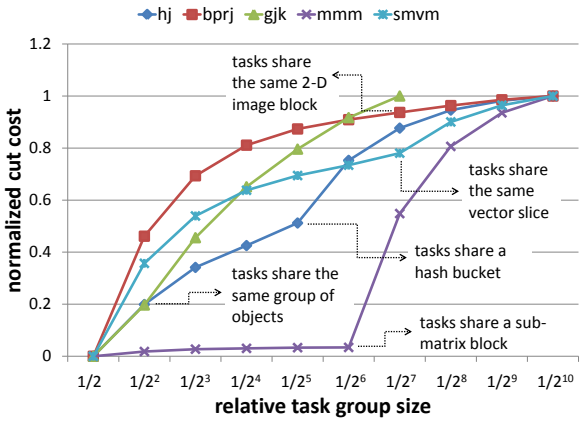


Figure 4.2: Cut cost trend over different sizes of task groups. Cut costs are normalized to fit within the interval [0, 1].

Figure 4.2 plots the cut cost trend for the workloads of this type. As can be seen, these workloads exhibit an abrupt increase in cut cost—a *knee*—as we decrease the task group sizes. For some workloads, other cache lines are sporadically shared, so the knee is not visually striking; in such cases we determine those knees by manual source code analysis. The sudden increase in cut cost means that the task group size has become small enough that tasks sharing their key data structures are now being separated into different groups. Ordering those task groups so that they execute consecutively would increase locality.

So for these workloads, assuming we group tasks so that each task group fits in cache (see Section 4.2.1), the importance of group ordering depends on the *relative size of the cache to the task clique*. For example, we found that a task clique in **hj** exhibits 128 KB footprint (where the knee exists in Figure 4.2). Therefore, on a 32 KB cache, **hj** would benefit from group ordering. However, on a 256 KB cache, group ordering would be less important.

Workloads with Structured Sharing: These workloads exhibit structured, regular sharing patterns. For example, for **conv**, a 2-D stencil operation, a task shares cache lines with its nearest 4 neighbors in the 2-D task space. For these workloads, cut cost is proportional to the number of task groups, and no knee exists. Here, group ordering is important *regardless of the cache size*; but a simple ordering,

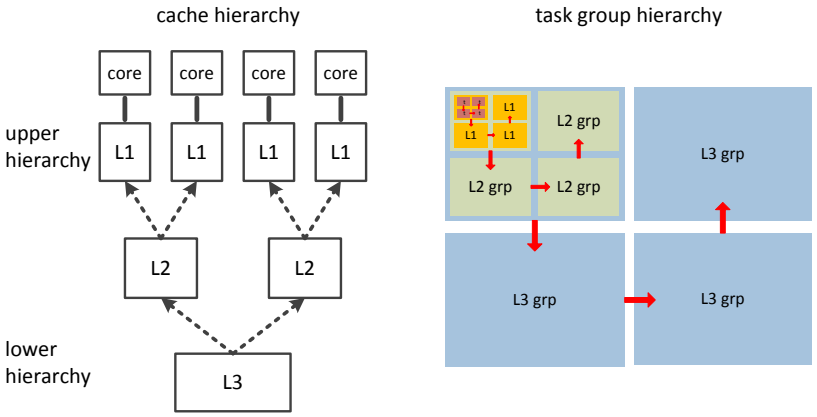


Figure 4.3: Generating recursive task groups. Different levels of task groups are sized to fit in a particular cache level. Colored arrows denote the group ordering determined over those task groups.

such as positioning the consecutive task groups in the task space together, should capture the reuse.

4.2.3 Implications of Task Size on Locality

Task size indirectly changes the relative importance of task grouping and ordering. For example, when the working set of a single task is larger than the cache, grouping tasks has little benefit, but ordering may help capture reuse from one task to the next. In general, smaller tasks give the scheduler more freedom. If a programmer breaks a large task into smaller tasks, and if the memory access pattern was originally suboptimal, better locality may be achieved via proper grouping and ordering.

4.3 Recursive Scheduling

Until now we have assumed a single level of cache. To explore scheduling on systems with complex cache hierarchies, we consider *recursive scheduling*, where the scheduling logic can be applied to arbitrary memory hierarchies. In Section 4.4, we use this scheduler to quantitatively explore grouping and ordering.

Since most applications exhibit multiple levels of working sets, and since cache

hierarchies usually have multiple levels, it is logical to consider grouping and ordering in a hierarchical fashion. Under recursive scheduling, we start from the bottommost level; tasks are first grouped so that a task group's working set fits in the last-level cache, and ordering is applied over those groups. We then recursively apply this approach to each of the task group, targeting one level up in the cache hierarchy each time. Figure 4.3 illustrates the procedure. First, we perform grouping on the full set of tasks to create L3 groups, each of which matches the L3 size. Next, the L3 groups are ordered. For each L3 group, we then decompose it into tasks and create L2 groups to match the L2 size. Then the L2 groups are ordered. We proceed in this fashion until we finally generate L1 groups, order them, and order their component tasks. This results in a hierarchy of task groups.

Because each task group also denotes a scheduling granularity, all of the tasks in a group get executed consecutively. In other words, a task group stays *resident* in its target cache from beginning to end. The existence of a hierarchy among these task groups guarantees that all the groups involved in executing a task stay resident in the cache hierarchy, regardless of the number of levels.

A slight complication arises when a system has private caches or caches shared by a subset of cores. In such a case, a set of task groups should be *pre-grouped*, so that groups with high sharing are assigned to the same cache. To handle this case, recursive scheduling performs an additional operation whenever hitting a *branch* in the cache hierarchy; it first pre-groups the set of tasks according to the number of consumers one level above, and then performs grouping for each partition. For example, in Figure 4.3, with two L2 caches, before generating L2 groups it first divides the set of tasks in an L3 group into 2 partitions, one for each L2 cache. It then constructs a task sharing graph for each partition, and uses the graphs to create 2 sets of L2 groups. Each set of L2 groups is separately ordered. Due to the pre-grouping, the task schedule generated ensures that the tasks from a set of L2 groups go to the same L2—without this property, the group ordering would not be effective. Pre-grouping can be done with the same graph partitioning algorithm that performs task grouping.

Core	32 dual issue in-order x86 16-wide 512-bit SIMD extensions Core-private 32 KB 8-way L1, 1-cycle Core-private 256 KB 16-way L2, 14-cycles Directory slice for L2 coherence
Interconnect	Ring topology connects L2s, directory slices, and memory controllers
Memory	4 memory controllers, 120-cycles DRAM latency

(a) Throughput Processor Configurations

Core	32 dual issue in-order SPARC v9 Core-private 32 KB 4-way L1, 1-cycle
Tile	4 cores on tile Tile-shared 4 MB 16-way banked L2, 10-cycles Directory slice, memory controller, and L3 bank
L3	16 MB per bank, 1 bank per tile 16-way, 21-cycles
Interconnect	2-D flattened butterfly connects tiles
Memory	158-cycles

(b) Tiled Processor Configurations

Table 4.3: Simulated system configurations.

4.4 Evaluation of Locality-Aware Task Scheduling

We now generate recursive schedules for two specific multi-core cache hierarchies and measure performance to quantify the impact of various scheduling decisions.

4.4.1 Experiment Settings

We simulate two different 32-core chip configurations, described in Table 4.3. The first configuration is a throughput computing processor we refer to as *Throughput Processor*. Figure 2.1 shows the configuration. Each core has a private L1 and L2, so all the caches are private. The combined L2 capacity is 8 MB, and coherence is maintained through a directory-based protocol. The instruction set is extended with 16-wide 512-bit SIMD instructions, and the applications have been tuned to use them. For this configuration, we use an industrial cycle-accurate simulator that models a commercial processor [64].

The second configuration is a tiled many-core processor we refer to as *Tiled Processor* (see Figure 2.2). Each core has a private L1, and each tile has four cores and a 4 MB L2, which is shared among the cores on the same tile. All the tiles share a single L3 cache. We used the M5 simulator [8] coupled with the GEMS toolset [47]

Workload	#L2 Grps	Tasks / L2 Grp	Tasks / L1 Grp	L1 Sharing Degree	L2 Cut Cost	L1 Cut Cost
hj	32	128	16	0.91	4.0E+06	7.1E+06
bprj	32	128	32	0.38	8.2E+06	8.5E+06
gjk	32	12	12	0.36	20,010	20,010
brmap	128	39	5	0.53	22,609	61,099
conv	32	33	4	0.55	10,036	27,620
mmm	512	8	1	1.00	1.3E+08	1.4E+08
smvm	32	128	8	0.90	4.2E+07	5.0E+07

Table 4.4: Task groups determined by the recursive scheduler and per group statistics.

for memory latency modeling.

Due to ISA and toolchain differences, we evaluate the Throughput Processor on all workloads except **sp** from Table 4.1, and we evaluate the Tiled Processor on the bottom four workloads.

As discussed in Section 4.1, we assume a task queue-based software task management system [29, 54]. In particular, similar to [44, 62], one task queue exists per core, and all the threads enter and leave each parallel section together. Task queues are pre-populated with offline-generated schedules right before the start of each parallel section. Within the parallel section each thread first works on those tasks enqueued on its local queue, and when they run out, a thread tries to steal tasks from remote queues. By supplying schedules generated from different scheduling policies we can quantify the impact of various scheduling decisions on locality.

Since it uses a simpler cache hierarchy, we focus on the Throughput Processor performance results first. In Section 4.4.3, we contrast the Tiled Processor results to highlight where different memory hierarchies affect scheduling. To isolate locality measurements from task management overheads, we use *the sum of the execution time of the tasks* as our primary locality metric throughout the section. Actual overheads could be mitigated through proposed hardware or hybrid methods [44, 41, 62].

4.4.2 Throughput Processor Performance Results

In this section, we first summarize the performance benefits of recursive scheduling. Then we isolate the benefits of each feature of the recursive schedule. In particular, we try to answer the following questions: (1) How much does locality-aware scheduling

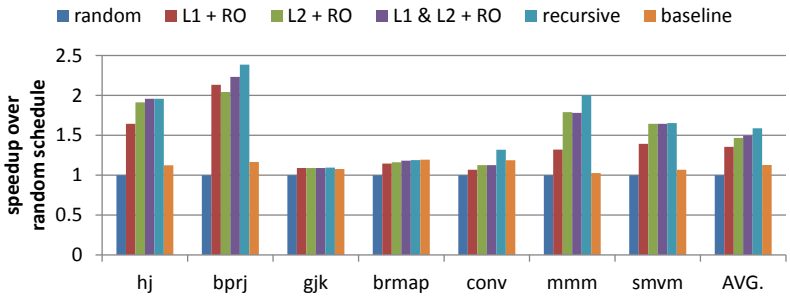


Figure 4.4: Throughput Processor performance summary. Shows the speedup over a random schedule. For each workload, from left to right are: (1) **random**, (2) **L1 grouping only**, (3) **L2 grouping only**, (4) **L1 and L2 grouping**, (5) **recursive schedule**, and (6) **baseline**. Baseline represents the schedule used in [44], and (2)∼(4) use random ordering (**RO**) instead of MST ordering.

Workload	L1 MPKI			L2 MPKI		
	rand	recursive	base	rand	recursive	base
hj	16.05	**6.63**	13.74	12.64	**6.61**	11.36
bprj	12.93	**3.29**	10.35	10.16	**2.72**	6.91
gjk	8.35	7.55	7.31	8.10	7.39	7.16
brmap	18.66	14.97	14.53	18.48	14.60	14.48
conv	8.54	7.73	7.62	8.37	**5.30**	6.47
mmm	28.37	**17.05**	28.38	27.08	**14.29**	26.07
smvm	136.30	132.22	133.86	52.93	**23.29**	48.46

Table 4.5: Measured MPKIs over different schedules. Bold figures denote where recursive schedule improves over baseline.

matter? (2) How much does grouping matter? (3) How much does ordering matter? (4) How does task size affect the schedule? (5) How do single-level schedules compare?

For each workload, Table 4.4 shows the task groups determined by the recursive scheduler for the Throughput Processor, and the corresponding statistics. In Table 4.2, columns corresponding to L2 and L1 groups for the Throughput Processor are also highlighted. We use this data to explain the following results.

Q1: How much does locality-aware scheduling matter?

Figure 4.4 presents the speedup of various schedules over a random schedule, measured in terms of the sum of the execution time of the tasks. A random schedule is obtained by randomly partitioning tasks into 32 chunks, randomly ordering the tasks, and assigning one chunk per core. We report the averages over 3 random instances. For each workload, from left to right, different schedules activate different aspects of

grouping and ordering, to arrive at the recursive schedule. Here we focus on the performance of the recursive schedule; we explain the rest in the following sections. For the baseline schedule, the task space is evenly partitioned into 32 chunks such that consecutive tasks fall in the same chunk. The scheduler then assigns one chunk per core. In fact, this implements the state-of-the-art *parallel depth first (PDF)* schedule [18]. To exploit sequential cache locality, when a thread completes a task, PDF assigns the task that the sequential program would have executed the earliest; this is the baseline scheduler used in [44], and many OpenMP schedulers follow this schedule [54]. Table 4.5 reports the measured misses per thousand instructions (MPKI) for L1 and L2 caches.

The figure shows that a locality-aware schedule (i.e., from the recursive scheduler) improves performance significantly. On average, the speedup over the random schedule is 1.60x, and over the baseline is 1.43x. In particular, **hj**, **bprj**, and **mmm** see large speedups of 1.96x, 2.39x, and 2.00x, respectively. Table 4.5 shows that this speedup is obtained by improving behavior at both cache levels, verifying that multiple levels of scheduling is important. **conv** and **smvm** also see significant speedups of 1.32x and 1.65x, respectively, from improved L2 behavior.

While **gjk** and **brmap** are fairly memory intensive (judging from their MPKIs), they see little benefit from locality-aware scheduling. These workloads have simple locality patterns that the baseline schedule is able to capture—grouping consecutive tasks captures most of the locality.

We now compare the energy consumption of different schedules. Specifically, we compute the energy for the part of the memory hierarchy beyond the L1s, which includes the L2s, ring network, and memory. For each schedule we measured the total L2 accesses, network hops, and memory accesses, and used the model from [33] to derive energy. Figure 4.5 shows the results. For each workload, the first 3 pairs of bars show activity counts—L2 cache accesses, on-die interconnect network hops, and off-chip memory accesses—and the last pair of bars shows the energy consumption. The results are normalized to that of the random schedule.

As expected, the locality-aware schedule significantly reduces all three activity counts to significantly reduce energy consumption: On average, the recursive schedule

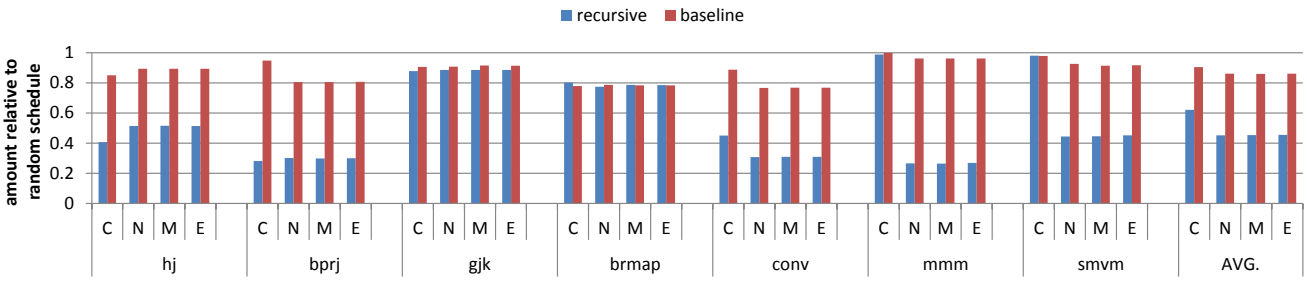


Figure 4.5: Energy consumption and activity counts of the memory hierarchy beyond the L1 caches for various schedules. **C**, **N**, and **M** denote activity counts for L2 cache accesses, network hops, and memory accesses, respectively. **E** denotes the energy consumption computed from the model [33]. All results are normalized to that of the random schedule.

reduces energy consumption by 55% relative to the random schedule, and 47% relative to the baseline. The recursive scheduler reduces L2 accesses by reducing the L1 miss rate (see Table 4.5), and likewise decreases on-die network and main memory activity by reducing the L2 miss rate. This result shows that locality-aware scheduling, or *placing computation* near the caches where data already resides, could be a viable alternative to reducing energy through *migrating data* [34, 33] to the caches near the cores performing the computation.

Q2: How much does grouping matter?

Recursive scheduling provides benefits through both grouping and ordering. Here, we isolate the benefits of grouping by disabling the ordering and pre-grouping parts of the recursive scheduler. We also isolate the benefits of the two levels of grouping by disabling one of the two levels at a time. Figure 4.4 shows the impact of various grouping policies.

Compared to a random schedule, performing both L1 and L2 grouping with random ordering—a *recursive grouping*—provides 1.52x average speedup, capturing most of the benefits of the full recursive schedule (1.60x). This indicates that grouping captures significant locality, and that ordering provides limited additional benefit once recursive grouping is done.

Due to its private-only cache hierarchy, on the Throughput Processor, applying only a single level of grouping can capture much of the locality benefits of recursive

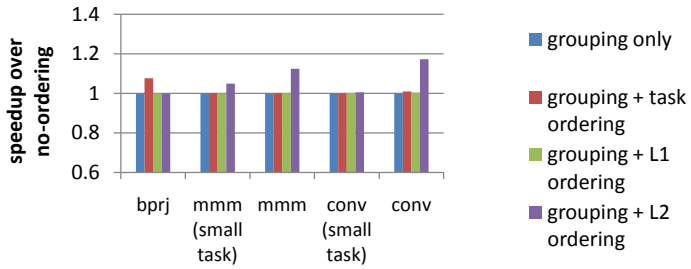


Figure 4.6: Workload sensitivity to task, L1 group, and L2 group ordering. For each workload, speedup is against the case where both L1 and L2 grouping are performed, but random task and group ordering are used.

grouping: L1 grouping with random ordering (**L1 + RO**) or L2 grouping with random ordering (**L2 + RO**) provides 1.36x and 1.49x average speedup, respectively. However, recursive grouping helps when different applications favor different levels.

Q3: How much does ordering matter?

Here we first consider ordering alone, and then when it can provide additional benefits over grouping.

In a separate experiment, we applied MST ordering to each of the 32 task chunks generated by the baseline schedule. This gave a 1.26x average speedup over random. Although not as large as the benefit from recursive grouping alone, this is substantial—it just turns out that METIS, especially when used recursively, is more effective than MST at capturing locality on this processor.

In Figure 4.4, comparing the performance of L1 and L2 grouping with random ordering (**L1 & L2 + RO**) against that of the recursive schedule shows that three workloads see significant benefits from additional ordering: The recursive schedule provides 1.07x, 1.17x, and 1.12x speedup on **bprj**, **conv**, and **mmm**, respectively, over L1 and L2 grouping with random ordering.

Figure 4.6 shows **bprj**, **mmm**, and **conv** sensitivity to task, L1 group, and L2 group ordering. As can be seen, **bprj** benefits from task ordering. **mmm** and **conv** on the other hand, benefit from ordering L2 groups. L1 group ordering, however, is not as effective.

bprj benefits from task ordering because it has a first level working set that is slightly larger than expected (due to stack data), and a low sharing degree (see

Table 4.4). With task ordering, the L1 MPKI for **bprj** reduces from 4.97 to 2.27, while the L2 MPKI reduces from 2.37 to 1.90. **mmm** is a clustered sharing workload—its task groups exhibit affinity for a small number of other groups (see Section 4.2.2). It also has high L2 cut cost (see Table 4.4) meaning the affinity between L2 groups is very strong. Thus, it benefits from ordering L2 groups. **conv** is a structured sharing workload—its task groups exhibit stencil-like affinity. It thus benefits from ordering L2 groups, as well.

Q4: How does task size affect the schedule?

For regular, grid-based workloads such as **mmm** and **conv**, task size can be easily adjusted by changing blocking parameters. For **mmm**, a single task actually overflows an L1. We shrink each **mmm** task so that a dozen tasks can fit in a single L1 group. Likewise, we make each **conv** task smaller so that it fits in an L1. Figure 4.6 shows that the sensitivity to L2 ordering reduces for small tasks (we label the modified versions of workloads with **small task**). Notice that the workloads perform the same computation, regardless of the task size; hence, the locality to be captured should remain the same. The task size *alters at which cache level the locality is captured.*

Also, as discussed in Section 4.2.3, task size can affect performance by changing the scheduler's freedom to exploit locality. For the experiment above, we noticed that the performance improvement of recursive scheduling over random increased from 2.00x to 3.08x as we decreased **mmm** task size. Since a task group amounts to a *scheduler-determined optimal task size*, users should express their tasks in the finest granularity possible to maximize scheduling freedom. Task scheduling overheads may limit task granularity, but these overheads may be reduced by previously proposed hardware and hybrid methods [44, 41, 62].

Q5: How do single-level schedules compare?

A single-level schedule denotes performing grouping and ordering at a single level only: L1 or L2-sized task groups with MST task ordering, but random ordering across groups. Figure 4.7 compares the performance of L1 single-level and L2 single-level schedules against recursive scheduling. Overall, the recursive schedule provides the best performance. Further, for **conv** and **mmm**, neither single-level schedule alone matches the performance of the recursive schedule. For the other workloads, an L2

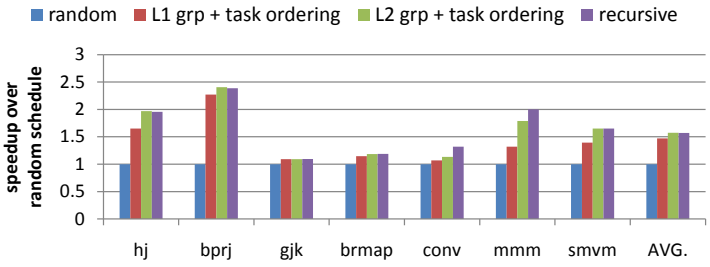


Figure 4.7: Single-level schedule performance. Speedup is over random schedule.

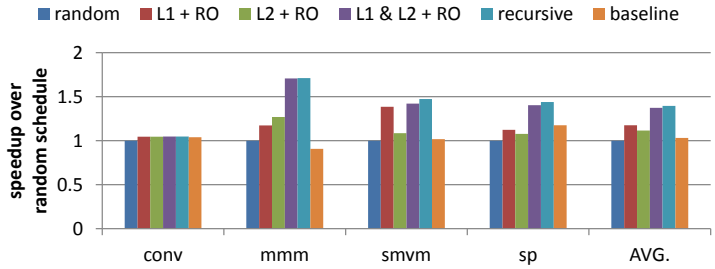


Figure 4.8: Tiled Processor performance summary. Shows the speedup over a random schedule. For each workload, from left to right are: (1) **random**, (2) **L1 grouping only**, (3) **L2 grouping only**, (4) **L1 and L2 grouping**, (5) **recursive schedule**, and (6) **baseline**. Baseline represents the schedule used in [44], and (2)∼(4) use random ordering (**RO**) instead of MST ordering.

single-level schedule is on par with the recursive schedule.

4.4.3 Tiled Processor Performance Results

We ported **conv**, **mmm**, **smvm**, and **sp** to the Tiled Processor[4], and conducted the same set of experiments described in the previous section. In this section we highlight the results that show where different cache hierarchies affect the locality-aware schedule.

Figure 4.8 summarizes the performance results on the Tiled Processor. Similar to the case of the Throughput Processor, the recursive schedule brings about a significant speedup: On average, the speedup over the random schedule is 1.40x, and over the baseline is 1.35x. This verifies that the recursive schedule *effectively exploits locality*

[4]Vector instructions were replaced by scalar loops.

on shared-cache organizations as well.

However, the presence of shared caches *alters the relative importance of grouping and ordering.* Similar to Figure 4.4, Figure 4.8 compares the performance of different grouping schemes to a random schedule. In contrast to the Throughput Processor case where a single-level L2 grouping with random ordering (**L2 + RO**) provided most of the grouping benefits, for the Tiled Processor neither L1 nor L2 single-level grouping (i.e., **L1 + RO** and **L2 + RO**) provides performance close to recursive grouping. With a complex shared cache organization, matching the task group hierarchy to the cache hierarchy becomes more important.

In the same figure, comparing the performance of L1 and L2 grouping with random ordering (**L1 & L2 + RO**) against the recursive schedule also reveals that the additional performance improvement from ordering on top of grouping has diminished. In particular, **conv** and **mmm**, which exhibited some sensitivity to ordering on the Throughput Processor, now barely benefit from ordering. This can be attributed to the increased importance on grouping; recursive grouping amounts to applying *coarse ordering* over smaller task groups, limiting the benefits of additional ordering.

As on the Throughput Processor, the benefits of ordering alone (i.e., applying MST to each of the chunks generated from the baseline schedule) were significant, but smaller than grouping alone (**L1 & L2 + RO**): 1.15x speedup over random, compared to 1.37x.

4.4.4 Conclusions

Locality-aware scheduling gives significant performance and energy benefits on real workloads. Recursive grouping alone captures most of the locality, but for some workloads, ordering can capture additional locality. Meanwhile, task size affects locality by changing the relative importance of grouping and ordering at each cache level. We also saw that recursive scheduling can exploit locality for both private and shared cache hierarchies. The exact degree of improvements, however, is determined by the interaction between the workload and the underlying cache hierarchy.

4.5 Implications for Practical Locality-Aware Task Scheduling

All features of the recursive scheduling can provide significant benefits—recursive grouping to fit the cache hierarchy, task ordering, and group ordering.

However, implementing these features in a practical setting requires a significant amount of application information that is non-trivial to obtain: the amount of sharing between all pairs of tasks, task working set sizes, and task sizes (i.e., dynamic instruction count). Further, a real-world locality-aware task manager must ensure its runtime overheads are smaller than its achieved reduction in task execution time; or a locality-oblivious task manager will be faster. We also expect the direct software implementations of the grouping and ordering algorithms (i.e., METIS and MST) to be too slow.

Therefore, we now propose a set of simplifications to the recursive scheduling based on (1) which features are most effective, and (2) what application information can be obtained or effectively approximated.

Simplification 1: Use a single level of grouping. Our earlier analysis showed that the most effective technique for capturing locality is task grouping. While recursive grouping is particularly effective, we demonstrated that a single level of grouping is almost as good if we also perform ordering (see Figure 4.7). Further, if we only perform a single level of grouping, we can eliminate the need for working set size information (e.g., just create as many groups as threads). Finally, the runtime overhead of a single level of grouping is lower since recursive grouping has runtime overhead that is linear in terms of the number of cache levels.

Simplification 2: Create groups with equal number of tasks. Although creating task groups with *equal amount of work* helps to reduce load imbalance, this requires individual task size information. Instead, an alternative is to create groups with *equal number of tasks*. For those workloads whose task sizes are about the same, this approximation will create task groups with similar amounts of work. For the other, less regular workloads, stealing should be able to balance load—albeit with some loss in locality.

Simplification 3: Use a compact form of sharing information. To perform grouping and ordering, it is necessary to obtain the sharing information between all pairs of tasks. Earlier, we extracted this information from the read and write sets of each task. Collecting the full task read and write sets, however, is not practical.

As we will demonstrate later, for workloads with regular access patterns, we can represent the data touched by each task and the sharing between tasks in a much more compact form. For workloads with irregular access patterns, we could instead collect a *hashed version* of the read and write sets—referred to in some recent work as *signatures* [14, 76, 48]. These signatures could be provided by the software, or could be automatically collected with some hardware support. We can then extract approximate sharing between tasks using signature distance computations.

Simplification 4: Create more task groups than threads, and use that as coarse-grain ordering. If we only use single level of grouping, ordering will become important. However, in the general case, ordering tasks for each group may be expensive. An alternative is to create more task groups than threads, and use that as a form of coarse-grained task ordering. While this approach could be less effective, it should be considerably cheaper.

Following these simplifications, in the next chapter we discuss the design details of practical task managers.

Chapter 5

Pattern-Based Task Managers

In this chapter, we apply the insights obtained from the graph-based framework to develop a *new class of practical, locality-aware task managers*. By leveraging the workload sharing pattern and a simple locality hint, locality-aware schedules can be generated at low costs. We also show that the performance of such schedules are comparable to that of recursive scheduling.

5.1 Methodology Overview

To provide generality to our approach, we first start by collecting workloads that cover a broad parallel workload taxonomy, the Berkeley Dwarfs [4]. The taxonomy classifies a wide range of parallel applications based on their similarity in computation and data movement.

We then analyze each workload's shared data access pattern by using three components: *task space*, *data space*, and *mapping function* (Figure 5.1). The *task space* represents the set of task indices that comprise a parallel section—each task corresponds to a unique, possibly multidimensional index. The *data space* represents the virtual address space used by shared data. The *mapping function* then maps each task index to its set of (important) shared data accesses.

Through this analysis, we identify three common sharing patterns, and the need for a *locality hint* to efficiently capture locality. Figure 5.2 shows a taxonomy of

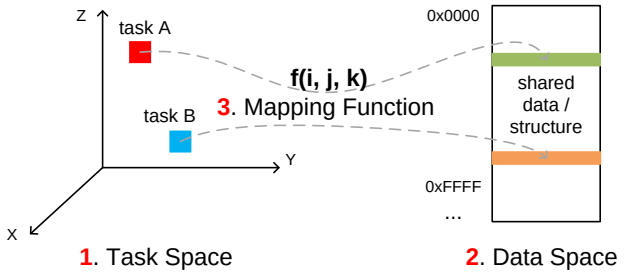


Figure 5.1: Task space, data space, and mapping function. A workload locality pattern can be described by these components.

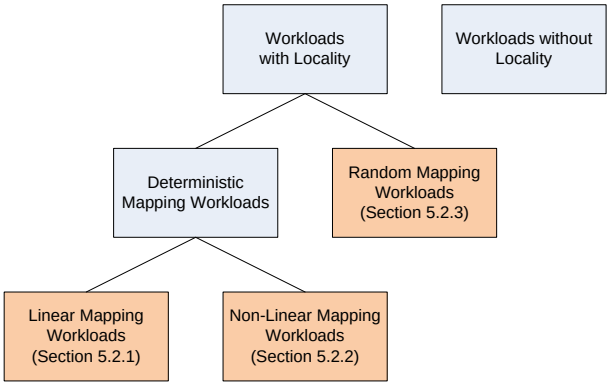


Figure 5.2: Workload taxonomy based on sharing patterns.

workload sharing patterns, showing that the three patterns we identify cover a wide range of possible task space, data space, and mapping function combinations.

In particular, *linear-mapping workloads* operate on regular, grid-based data structures, such as BLAS kernels and stencil-based computations. For this category of workloads, the data space is quite regular, so the task-to-data mapping is typically a *linear function* of the task index. Here, *vicinity* in task space readily translates into locality in data space, so a locality-aware schedule can be generated by *partitioning* the task space. However, to determine the best *shape* of each partition, we need to know the relative amount of sharing done in each direction in task space—we refer to this information as *locality hint*.

Non-linear mapping workloads also operate on regular data structures, but the mapping function is *non-linear* or *complex*. Therefore, vicinity in task space no longer implies data locality. To capture locality, we need the mapping function itself as a

locality hint, so that the data space can be partitioned.

Random mapping workloads operate on regular or irregular data structures, typically in an input-dependent fashion. As a result, a task's index and the data it touches show little correlation. To capture locality for these workloads, we need the explicit information as to which data each task touches as the locality hint.

We then leverage the sharing patterns and locality hints to develop a *practical locality-aware task manager*. It requires *minimal user input*: For each parallel section, an application simply indicates which sharing pattern the section uses, and also provides a locality hint. The manager then transparently employs a different *task management policy* for each of the patterns to generate a schedule that captures significant locality. More importantly, our task manager results in minimal disruption to the programming model itself: It merely *extends* the existing task-parallel API to allow an application to convey a small amount of locality information. This approach alleviates users from performing manual scheduling, and can be easily integrated with many programming models.

5.2 Workload Sharing Patterns

We now start our exploration of practical locality-aware task management by examining a broad set of representative parallel applications. Specifically, we collect workloads covering all the categories of the well-known Berkeley parallel workload taxonomy [4]. Based on their similarity in computation and data movement, the taxonomy classifies parallel workloads into 13 categories[1]. Table 5.1 shows the collected workloads and their corresponding category.

From the high performance computing domain, **mmm** performs 3-D blocking matrix to matrix multiplication, and **smvm** multiplies a sparse matrix with a vector.

[1]We removed 3 categories from our analysis. We dropped the Finite State Machine category since it is inherently sequential. For Backtrack / Branch+Bound workloads, the size of the examined solution space depends on the order of task execution, which makes it difficult to discern the improvements from locality-aware scheduling. The **ep** benchmark, which [4] elects as the representative workload for the MapReduce category, only exhibits global sharing. Locality for other MapReduce workloads can be exploited using the methodology described in Chapter 3.

Workload	Origin	Category	Sharing Pattern	Task Size	#tasks
bt	NPB	Dense Linear Algebra	Linear Mapping	10,204	2,408
mmm	In-house		Linear Mapping	12,962	32,768
conv	In-house		Linear Mapping	17,637	16,384
cg	NPB	Sparse Linear Algebra	Random Mapping	684	1,400
smvm	In-house		Random Mapping	3,383	4,096
ft	NPB	Spectral Methods	Non-Linear Mapping	19,393	4,096
barnes	SPLASH-2 [75]	N-Body Methods	Non-Linear Mapping	6,116	2,048
sp	NPB	Structured Grids	Linear Mapping	5,956	1,156
bprj	In-house		Non-Linear Mapping	3,696	4,096
mg	NPB		Linear Mapping	3,201	4,096
sph	PARSEC [7]	Unstructured Grids	Linear Mapping	5,218	1,024
md6	In-house	Combinational Logic	Producer-Consumer	–	–
semphy	MineBench [49]	Graph Traversal	Producer-Consumer	–	–
nw	Rodinia [17]	Dynamic Programming	Producer-Consumer	–	–
sphinx	ALPBench [46]	Graphical Models	Random Mapping	733	341

Table 5.1: Workload sharing pattern analysis. For each workload, we denote its origin, category [4], and dominant sharing pattern. For those workloads used in our performance evaluation (see Section 5.4), we also report average task size (in dynamic instructions) and the number of tasks in the dominant parallel section. **smvm** stats are based on **pt4096** (see Section 5.4.4).

Then we have a set of solver implementations: **bt** and **sp** are PDE solvers that apply block tridiagonal and scalar pentadiagonal algorithms, respectively. In a similar fashion, **cg** is a conjugate gradient-based linear equation solver, and **mg** applies multigrid method to a Poisson equation.

We also have visual and audio processing workloads. In particular, **conv** performs a 5x5 filter convolution on an image, **bprj** reconstructs a 2-D image by backprojecting a 3-D space, and **ft** applies Fast Fourier transform on the input data. **sphinx**, on the other hand, performs hidden Markov model-based speech recognition.

barnes and **sph** are physics simulation workloads, where **barnes** implements the Barnes-Hut n-body simulation, and **sph** applies smoothed-particle hydrodynamics over a set of particles (**fluidanimate** [7]).

semphy, which performs phylogenetic tree-based structure learning, and **nw**, which performs Needleman-Wunsch sequence alignment, can be classified as data mining workloads. **md6** is a parallel implementation of MD6 secure hash.

Our workloads were originally written in C / C++, and parallelized using Pthreads or OpenMP. We ported the workloads to utilize our task queue library instead (see

Section 5.4). Notice that the applications are not necessarily tied to a specific algorithm class, such as cache-oblivious [28] or recursive decomposition.

For these workloads, same as in Chapter 4, all the tasks are independent (i.e., available at the beginning of each parallel section and can be executed in parallel), and their execution order does not affect the computation result. While our analysis (and our resulting task manager) will therefore exclude tasks with control dependencies, such parallel sections (1) can be reformulated as a series of control-independent parallel sections, or (2) can be scheduled by casting control dependencies as data dependencies (see Chapter 3).

Moreover, we focus our analysis on *within-parallel section locality*. While cross-parallel section locality existed, the workloads were written in a bulk synchronous parallel (BSP) fashion, where one parallel section is executed in full before the next is started. This basically flushes out the cache, making it hard to exploit locality across parallel sections[2].

Lastly, for some workloads, we noticed that parallel sections repeatedly execute (e.g., inside a loop) using the same task space, data space, and mapping function, but with different data values. A task manager could potentially generate a schedule once for such a section, and reuse the schedule over multiple iterations to amortize the scheduling overhead.

Examining the table, we see that a few data sharing patterns cover all of our workloads. Using examples from the workloads, we now describe these patterns.

5.2.1 Linear Mapping Workloads

Linear mapping workloads operate on arrays (1-D, 2-D, etc.) in a regular fashion. Examples are BLAS kernels and stencil-based computations, such as many image processing operations. Listings 5.1 and 5.2 show the task functions of **mmm** and **mg**, respectively. The code has been abbreviated for clarity.

[2]Our producer-consumer locality workloads belong to this category. To efficiently exploit locality, these workloads should be restructured using the programming models that explicitly support pipeline parallelism [66]. However, since our goal was to *augment* the programming model with locality hints while keeping the user code intact, we dropped these workloads from further analysis.

```
void mmm_task (int i, int j, int k) {
  i *= i_2d; j *= j_2d; k *= k_2d;

  while (k < k_max) {
    while (i < i_max) {
      // Compute sub-matrix multiplication
      // C[i][j] += A[i][k] * B[k][j]
      i += i_1d;
    }
    k += k_1d;
  }
}
```

Listing 5.1: **mmm** task function.

```
void mg_task (int i, int j) {
  double u1[M];

  for (int k = 0; k < k_max; k++) {
    u1[k] = u[i][j - 1][k] + u[i][j + 1][k] +
      u[i - 1][j][k] + u[i + 1][j][k];
  }
}
```

Listing 5.2: **mg** task function.

mmm performs matrix-to-matrix multiplication, utilizing cache blocking for the two input arrays and the output array; this leads to a 3-D task space. In Listing 5.1, task indices **i**, **j**, and **k** denote the block index in the X, Y, and Z task dimensions, respectively. A task operates on portions of the shared matrices **A**, **B**, and **C**, according to its task indices multiplied by some constants.

On the contrary, **mg** performs a 2-D stencil operation on a 3-D array. A task (Listing 5.2) with indices **i** and **j** computes the result for a 1-D strip of the output array by accessing the corresponding strip of the shared input array, **u**, along with the Manhattan adjacent neighbors' strips.

These examples illustrate that linear mapping workloads access shared data structures according to a *linear transformation* (or more precisely, an *affine transformation*) of their task indices. Thus, vicinity in task space translates into locality in data space. Locality can therefore be captured by *partitioning the task space* so that adjacent tasks are scheduled on the same thread.

However, to best partition the task space, a task manager needs to determine the

```
void bprj_task (int i, int j, int k) {
  // Perform coordinate transformation
  float tmp0 = c00 * i + c01 * j + c02 * k + c03;
  float tmp1 = c10 * i + c11 * j + c12 * k + c13;
  float tmp2 = c20 * i + c21 * j + c22 * k + c23;

  // Obtain array index
  int index1 = round(tmp1 / tmp2) * WinYs +
    round(tmp0 / tmp2);
  float val = Raw[index1];
}
```

Listing 5.3: **bprj** task function.

```
void barnes_task (int i) {
  bodyptr *p = &btab[i];
  long xp[NDIM] = intcoord(*p), kidIndex = 0;

  // mask is power of two
  for (int k = 0; k < NDIM; k++) {
    if (xp[k] & mask) kidIndex |= 1 << k;
  }
}
```

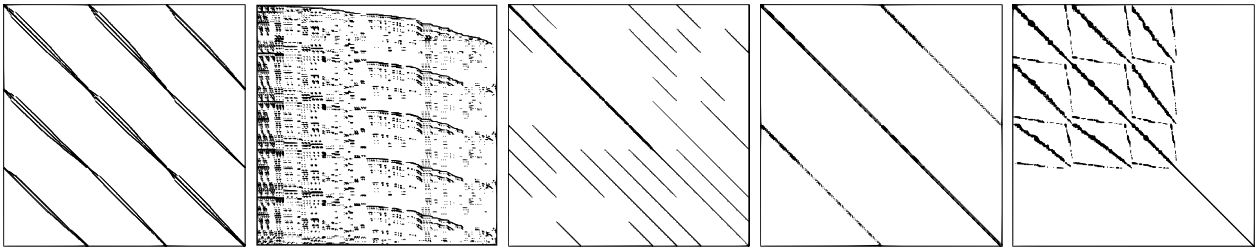
Listing 5.4: **barnes** task function.

right *shape* of a task partition. For example, in **mmm**, it would be beneficial to shape a partition so that the total number of matrix elements reused within each partition is maximized. This, in turn, depends on how many elements are shared when the task index is incremented in each dimension in the task space. Likewise, for **mg**, the equal amount of sharing in both dimensions of the task space suggests that the best partition shape is square. To summarize, the shape of a task partition that maximizes locality depends on the *directional sharing information*.

5.2.2 Non-Linear Mapping Workloads

Non-linear mapping workloads operate on regular data structures in a deterministic, but irregular fashion. Example workloads that fit this pattern are **bprj** and **barnes**. Listing 5.3 shows the task function for **bprj**. In the code, task indices **i**, **j**, and **k** go through a coordinate transformation, and the new coordinates are *divided* to obtain an offset into the shared image, **Raw**.

The task function for **barnes** is shown in Listing 5.4. Here, a task is assigned a



(a) **plat1919** (b) **rbs480a** (c) **fs_541_3** (d) **ck656** (e) **zenios**

Figure 5.3: Sparse matrices used for **smvm**. Black points represent non-zero values. The resolution has been adjusted so that a dot roughly amounts to a cache line. Image credit [50].

particle, and the coordinate of the particle, **xp**, is transformed into a subtree index, **kidIndex**, through *bit sampling*. The task then traverses the subtree at **kidIndex** to insert the particle.

To exploit locality, tasks that access the same address range or those that traverse the same subtree should be scheduled together. However, since the division and bit sampling operation are inherently non-linear, a contiguous partition in the task space will not result in such a schedule. Instead, we need to *partition the data space*. So for non-linear mapping workloads, we need the mapping function itself—for **bprj**, for example, given the task indices **i**, **j**, and **k**, the mapping function will return **index1**; likewise, given the task index **i**, the **barnes** mapping function will return **kidIndex**. With these mapping functions, we can capture data locality by evaluating the function for each task, and grouping together the tasks with the same value.

5.2.3 Random Mapping Workloads

Random mapping workloads access data structures in unpredictable ways, such as through input-dependent indirection. These workloads exhibit little correlation between the task index and data addresses touched. **smvm** is an example random mapping workload. Figure 5.3 visualizes the input sparse matrices used for **smvm**.

In **smvm**, a task multiplies a single row of a sparse matrix with a dense vector—only non-zero elements of the matrix are multiplied by their corresponding vector

elements. Each task thus touches the elements of the shared input vector according to the *fill pattern*, or the set of non-zero elements, of an input matrix row. The fill pattern is input-dependent, and in general cannot be reliably predicted.

To capture locality for random mapping workloads, the scheduler needs to know the data addresses each task touches, and should group together tasks with many overlapping addresses. For example, in **smvm**, we should schedule together tasks that have similar fill patterns. A straightforward way to generate a locality-aware schedule is to *cluster* tasks according to their data addresses, but this approach is very expensive. In Section 5.3.3, we describe an affordable approach that relies on *hashing* to reduce scheduling costs.

5.3 Task Manager Designs

We now try to approximate recursive scheduling with minimal application information, and with fast scheduling schemes. The resulting lightweight task manager can handle multiple data sharing patterns, but requires minimal user involvement. Its APIs can be easily integrated with many programming models.

In Section 5.2, we showed how different workload sharing patterns can be characterized by their mapping functions from task index to data touched. To generate a locality-aware schedule, however, what we actually need is the *inverse mapping function* f^{-1}: Given a data region, which tasks share that data? We can construct a locality-aware schedule by grouping such tasks. Unfortunately, the inverse mapping function is not present in our workloads, or is hard to express in a concise form. Therefore, our key challenge is to efficiently *approximate* this inverse mapping function. As suggested by our sharing pattern analysis, we find that this approximation can be aided with pattern-specific locality hints; thus, our task manager implements *separate policies* for each sharing pattern.

Once the inverse mapping function has been approximated, dynamically generated tasks could also be scheduled by reusing the learned structure. For parallel sections where tasks share more than one region of data, multiple inverse mapping functions could be applied in an iterative fashion to gradually subgroup tasks.

Sharing Pattern	Task Manager Policy	API

Linear Mapping (Sec. 5.2.1)	Flux-Based (Sec. 5.3.1)	`// Set sharing vector`
		`void LMFlux_SetSharingVec (int x, int y, int z);`
		`// Approximate sharing vector (Section 5.5)`
		`void LMFlux_ApproxSharingVec (void);`

Non-Linear Mapping (Sec. 5.2.2)	Binning-Based (Sec. 5.3.2)	`// Set mapping function`
		`typedef void *(*MappingFn3D) (int, int, int);`
		`void LMBin_SetMappingFn (MappingFn3D fn, void *min, void *max);`

Random Mapping (Sec. 5.2.3)	Signature-Based (Sec. 5.3.3)	`// Set signature array`
		`typedef char sig_t[SIGNATURE_BYTES];`
		`void LMSig_SetSigArray (const sig_t *sig_array);`
		`// Set hardware monitor (Section 5.5)`
		`void LMSig_SetMonitor (void *base, size_t range);`

Table 5.2: Workload sharing patterns, task manager policies, and APIs. The API functions convey both the *sharing pattern* and the *locality hint*.

```
// Specify the sharing pattern and locality hint
LMFlux_SetSharingVec(w_x, w_y, w_z);

// Specify the parallel section
taskQEnqueue(taskfn, num_dims, size_arr);

// Execute the parallel section in parallel
taskQWait();
```

Listing 5.5: Task manager API usage example.

In the rest of the section we describe our task management policies, focusing on how they perform task grouping and ordering. For all the policies, we assume that we have no knowledge of the amount of work in each task—we balance load by trying to schedule an equal number of tasks on each thread. Similarly, we try to enforce task ordering when it is cheap; but if not, we try to approximate ordering with the *coarse-grain ordering* implemented by generating many task groups. Table 5.2 lists the sharing patterns, the policies, and the corresponding task manager APIs. Notice that in the table, as we move from top to bottom, we increase the burden on the programmer. But in return, the policy becomes more general.

5.3.1 Flux-Based Task Management

For workloads with linear mapping functions, we propose a *flux-based* policy. As described earlier, for this sharing pattern, tasks that share data are neighbors in

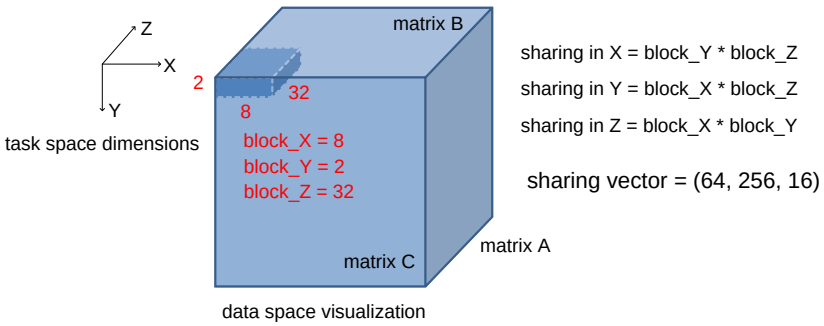


Figure 5.4: Obtaining a sharing vector from **mmm** blocking information. Each task accesses blocks of matrices A, B, and C to compute C += A * B.

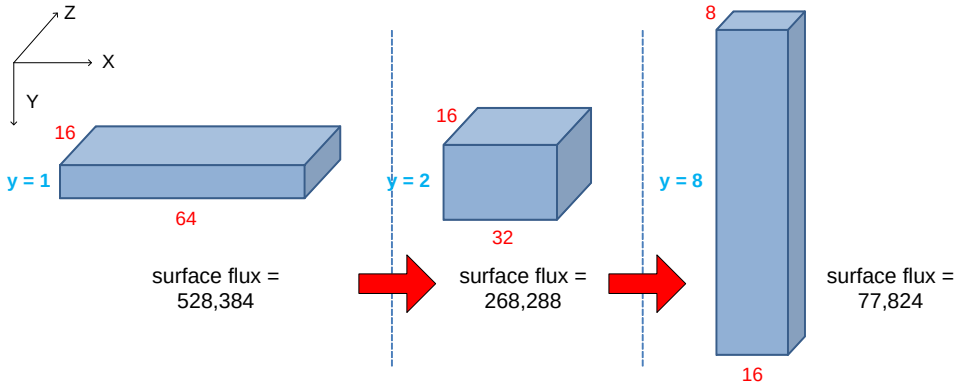


Figure 5.5: Effect of surface flux reduction on **mmm** partition shape. The Y direction has maximum sharing.

task space (see Section 5.2.1). Therefore, the inverse mapping function need not be explicitly approximated, and we directly partition the task space. We determine the best shape of a partition using the directional sharing information.

API: To obtain the sharing information, we rely on the regular nature of the sharing pattern and a simple software hint. For example, Figure 5.4 shows the 3-D blocking performed in **mmm**, where each task accesses blocks of matrices A, B, and C to compute C += A * B. The figure shows that consecutive tasks in the X task dimension reuse **block_Y * block_Z** = 2 * 32 = 64 cache lines of matrix A. Likewise, consecutive tasks in the Y direction reuse 256 lines, etc. We concisely express such sharing as a *sharing vector*, indicating the number of cache lines shared

for consecutive tasks in each direction in task space[3]. For this example, the sharing vector is $(64, 256, 16)$. This information can be provided by a programmer, a compiler, or a hardware monitoring scheme.

Listing 5.5 shows the API usage example. The sharing vector is conveyed to the task manager through **LMFlux_SetSharingVec()** (see Table 5.2). By comparing the code with Listing 4.1, we can see that the code for the parallel section itself need not change. The locality APIs do not affect application correctness.

Given the sharing vector and task space (i.e., number of dimensions and size in each dimension), the task manager performs grouping and ordering as follows.

Grouping: Our scheduler evenly partitions the task space into as many task groups as there are threads. The shape of each partition is determined so that the *surface flux* of the partition is minimized. For example, when the task space is 3-D, and the sharing vector is given as $S = (a, b, c)$, the surface flux of a partition with dimensions $G = (x, y, z)$ will be $F = 2(ayz+bxz+cxy)$. Mathematically, this amounts to an outer product of two vectors, and can be generalized to arbitrary dimensions.

The sharing vector actually represents the edge weights in a task sharing graph (i.e., $(64, 256, 16)$ matches the weights in Figure 4.1). Hence, minimizing the surface flux amounts to minimizing the *cut cost* in the graph. Figure 5.5 shows that as we reduce surface flux, the shape of an **mmm** task partition is elongated along the Y direction, the direction with the highest sharing vector weight, thus increasing the sharing within each task group.

Since the resulting task group is a multidimensional block, instead of explicitly storing each individual task index, we can compactly store the range of task indices assigned to each task queue.

Ordering: Once the manager determines the shape of a partition, it applies *raster-scan* ordering to each partition. The scheduler starts each partition at the task with minimum X, Y, and Z coordinates, and proceeds along the maximum sharing dimension (e.g., Y for **mmm**). Once it hits the end of the partition, the scheduler increments the index in the next most shared dimension (e.g., X for **mmm**), and

[3]Sharing vector may denote the number of elements shared, instead. We use cache lines in this example to show their correspondence to edge weights in a task sharing graph (see Section 4.2).

```
void *bprj_mapping (int i, int j, int k) {
  float tmp0 = c00 * i + c01 * j + c02 * k + c03;
  float tmp1 = c10 * i + c11 * j + c12 * k + c13;
  float tmp2 = c20 * i + c21 * j + c22 * k + c23;

  intptr_t index1 = round(tmp1 / tmp2) * WinYs +
    round(tmp0 / tmp2);

  return (void *)index1;
}
```

Listing 5.6: Example mapping function for **bprj**.

starts over in the maximum sharing dimension. This process is repeatedly applied across all dimensions as in a conventional nested 'for' loop, to order the full set of tasks in a task group. Notice that the manager can generate this order very cheaply and implicitly—it only stores the sorted task dimension order.

5.3.2 Binning-Based Task Management

For workloads with non-linear mapping functions, we propose a *binning-based* policy. For these workloads, a straightforward partitioning of task space may not capture data locality. Instead, we explicitly construct the inverse mapping function by evaluating the mapping function for each task, and then partitioning the data space—more precisely, we partition the *image* of the mapping function.

API: The programmer or compiler conveys the image of the mapping function through **LMBin_SetMappingFn()** (see Table 5.2), which takes (1) the mapping function and (2) its minimum and maximum values as arguments. The latter are necessary so that the task manager can pre-generate some data structures. Listing 5.6 shows an example mapping function for **bprj**. The task manager also obtains the dimensions of the task space through the baseline task-parallel programming API.

Grouping: Our scheduler performs grouping by evenly dividing the mapping function image and making each partition (i.e., a *bin*) a task group. To assign tasks to bins, the scheduler evaluates the mapping function for each task in parallel, and assigns the task according to its return value. Since these bins amount to a quantized approximation of the inverse mapping function, using more bins gives more accurate

approximation, resulting in improved locality for each group. Finally, our scheduler assigns an equal number of bins to each thread, assuming that the mapping function evenly covers its image.

Ordering: Under this policy, the scheduler does not explicitly order tasks, but instead relies on task grouping as a form of coarse-grain ordering. We create more task groups than there are threads to emulate the effect of task ordering; we find that four times as many groups as there are threads is sufficient. The alternative to this approach would be to sort the tasks in each bin, but it can be costly, and may not actually help. Depending on the workload, the mapping function may return an index into an array (see Listing 5.3), or an index of a shared structure (see Listing 5.4). If the latter, sorting may not reduce reuse distance.

5.3.3 Signature-Based Task Management

For random mapping workloads, we propose using a *signature-based* policy that groups tasks based on the inverse mapping function. Since this function cannot be concisely represented for these workloads, we rely on the programmer, compiler, or hardware to provide the explicit inverse mapping in the form of *task signatures*.

API: In addition to the task space dimensions, the task manager takes a *task signature* for each task. A task's signature is a hashed, *bit vector* representation of the data addresses that it touches [14, 48, 76]. For example, for **smvm**, a programmer may convert the matrix fill patterns into an array of signatures. The information is conveyed to the task manager through **LMSig_SetSigArray()** (see Table 5.2).

Grouping: Since each bit in a signature amounts to a region of memory, we can capture locality by clustering tasks with similar bit-patterned signatures. Since we are performing online task scheduling, however, we cannot afford a conventional clustering algorithm that uses pairwise distance calculations ($O(n^2)$, n = # of tasks). Fortunately, we can perform $O(n)$ clustering through probabilistic measures.

Specifically, we use *locality-sensitive hashing (LSH)* [36] to perform clustering.

Simply put, LSH is a set of hash functions H, s.t.,

$$\text{if } ||x - y|| \leq \text{R then } Pr_H[h(x) = h(y)] \geq p_1,$$
$$\text{if } ||x - y|| \geq \text{cR then } Pr_H[h(x) = h(y)] \leq p_2.$$

For an LSH to be useful, it has to satisfy $p_1 > p_2$. That is, if two bit strings x and y are similar, the probability that they will be hashed to the same value should be high. And if they are dissimilar, the probability for them to end up in the same hash bucket should be low.

Different families of LSHs exist for different distance metrics. Thus, we must first choose a distance metric between two signatures.

The most straightforward metric is the *Hamming distance*, as used in [39]. Between two binary strings, the Hamming distance is the number of substitutions required to change one string into the other. For example, for bit strings **01110** and **01101**, the Hamming distance would be 2. Intuitively, the smaller the distance, the more the two strings will resemble each other. However, we find that Hamming distance does not accurately capture the *reuse* of memory blocks (i.e., overlapping 1s in signatures). Continuing with the previous example, the Hamming distance for two bit strings **01110** and **11111** is also two, but they share three 1s, instead of two.

To better reflect the reuse of memory blocks, and also the total number of memory blocks accessed by a pair of tasks, we use *Jaccard distance*. The *Jaccard similarity coefficient* is defined as

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|},$$

which ranges from 0 (no 1s shared) to 1 (all 1s shared). The Jaccard similarity correctly shows that **01110** and **11111** share more (similarity = 0.6) than **01110** and **01101** (similarity = 0.4). *Jaccard distance* is defined as $1 - J(A, B)$.

For the Jaccard distance, the LSH is called *MinHash* [13]. To compute MinHash values for bit strings, we first pick a random permutation of bit positions. Then for each bit string, the *index of the first 1* under the permutation becomes the MinHash value for the string. For example, under a random bit permutation $[3, 1, 4, 2, 0]$, $\mathit{MinHash}(\mathbf{01110}) = 3$, and $\mathit{MinHash}(\mathbf{01101}) = 1$. Grouping bit strings with the same

MinHash value will result in clusters of bit strings with high Jaccard similarity[4].

To increase the quality of the clustering, typically a couple of random permutations are chosen, and multiple MinHash values are used. However, we see sufficiently good clustering even when we do not permute signatures. For example, in Figure 5.3, consider the leading non-zero position for each row in the matrices and how it correlates with the data accessed in that row. Hence, we use the *leading one position* of a signature as its MinHash value.

Computing this MinHash can be very efficient. Some architectures (e.g., x86 SSE, SPARC, Alpha) provide instructions to count leading zeros, which allow us to obtain the leading one position from a bit vector at low cost. Moreover, we could further specialize the task manager for some key data structures such as sparse matrices—a very common representation of sparse matrices, CSR, explicitly stores the index of the leading non-zero for each row (we do not perform this specialization here).

To summarize, our scheduler groups together tasks with the same leading one position in their signatures. Similar to our binning-based policy, a range of leading one positions works as a bin. Notice that this process again amounts to a quantized approximation of the inverse mapping function. However, due to the generality of this policy, we do not assume that each leading one position will have an equal number of tasks. Thus, our scheduler uses an optional step to better balance load: It computes a histogram of the tasks with each leading one position, and uses that histogram to determine the ranges of positions with roughly equal numbers of tasks.

Ordering: Our scheduler uses the same ordering strategy here as it does for the binning-based policy: It applies coarse-grained ordering by generating more task groups than threads (using the same 4x rule). The alternative of sorting the signatures is even more expensive than the binning-based policy, especially when we sort by Jaccard distance ($O(n^2)$).

[4]As a side note, this clustering algorithm was used for AltaVista web document duplication detection [13] and Google news customization service [22].

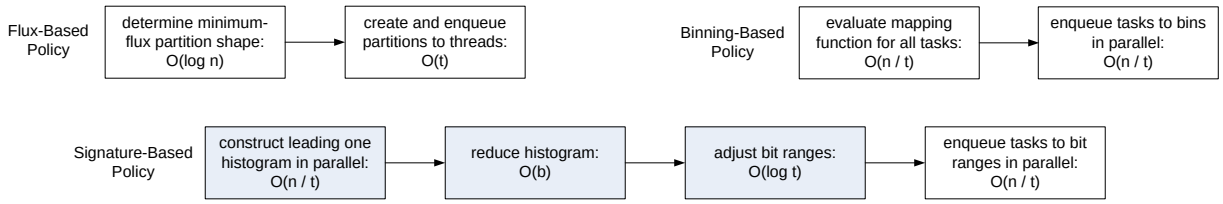


Figure 5.6: Task manager schedule generation cost. Shaded boxes denote the optional load balancing process for the signature-based policy. **n** = number of tasks, **t** = number of threads, and **b** = signature bit count.

5.3.4 Scheduling Overhead Analysis

Our task manager has overheads for creating the task schedule at the beginning of a parallel section, and for retrieving the tasks during the parallel execution. In the absence of contention over shared structures, the overheads for the latter would be constant, and should be very similar to previously proposed task managers. The overhead for creating a schedule, however, depends on the particular task management policy used, and can scale with certain parameters. Figure 5.6 shows the flow of the schedule generation process, and the costs for each stage. The overhead for generating a schedule has two components: (1) determining the schedule, and (2) enqueueing the tasks into the task queues.

For the flux-based policy, we confine the minimum-flux shape evaluation to power-of-two dimensions. Thus for n tasks, evaluating all the shapes for minimum surface flux takes $O(log\ n)$ time. The remaining overheads are constant per partition: Since we generate t partitions (for t threads) and a task partition can be concisely represented as a range of task indices, the cost to create and enqueue partitions are constant per partition, or $O(t)$.

For the binning-based policy, evaluating the mapping function for the tasks in parallel takes $O(n/t)$ time. In this case, since we cannot concisely represent a task group, the manager takes $O(n/t)$ additional time to enqueue individual tasks in parallel.

For the signature-based policy, each thread first computes a local histogram for a subset of tasks, taking $O(n/t)$ time. Then the manager merges the local histograms in $O(b)$ time, where b is the number of bits in a signature. The manager then performs a recursive bi-partitioning on the global histogram to create bit ranges with equal

numbers of tasks. That is, it first determines the *center of mass* of the histogram, and recursively applies the approach to the two sub-histograms on both sides of the center. This takes $O(log\ t)$ time since the number of bit ranges we generate is proportional to the number of threads. Enqueue cost is $O(n/t)$ for this policy as well.

This analysis reveals a *tradeoff* between the generality of the sharing pattern and the overhead of the corresponding policy, for both the scheduler and the programmer. For the most specialized pattern, the linear mapping workloads, our task manager takes a simple sharing vector as a hint, and produces a schedule in $O(log\ n)$ time. For the other, more generic patterns, schedule generation cost increases to $O(n/t)$, with higher constants for the most generic signature-based policy. Further, the programmer burden increases to providing a mapping function or a full set of signatures, for the binning-based and signature-based policies, respectively. We address this burden later (see Section 5.5).

5.4 Task Manager Performance Evaluation

We now evaluate the performance of our locality-aware task manager on a simulated system, examining both the quality of the task schedules and run-time overheads.

5.4.1 Experiment Settings

We simulate the same Tiled Processor described in Section 4.4.1. Each core has a private L1, and four cores form a tile. For each tile is a 4 MB L2, shared among the cores on the tile. All the tiles share a single L3 cache. For our signature-based policy, the simulator supports a single-cycle 1024-bit leading zero count (**LZCNT**) instruction. This setting is aggressive, but recent throughput-oriented processors already support 512-bit operations [64], and several previous proposals utilize signatures larger than 1024-bits [14, 48, 76].

As described earlier, our workloads use a software task queue library. This library includes our task manager from Section 5.3. For the flux-based policy, to match the system's cache hierarchy, our manager assigns consecutive task groups in the

maximum sharing direction to the same tile. Likewise, for the binning and signature-based policy, our manager assigns consecutive bins and bit ranges to the same tile, to exploit potential spatial locality. We determine the sharing vectors for the linear mapping workloads either by expert knowledge or profiling.

For comparison, the task queue library also contains a baseline task manager. The baseline manager linearizes the task space according to the innermost loop, then evenly divides the tasks into as many chunks as there are threads, such that consecutive tasks fall in the same chunk. The manager then assigns one chunk to each thread. Notice that this implements the state-of-the-art PDF schedule [18]. Just like our manager under the flux-based policy, the baseline scheduler also represents each task chunk as a range of task indices (although the baseline always uses 1-D ranges). Therefore, the scheduling and enqueue costs are both $O(t)$.

To assess the quality of the schedules generated from our locality-aware task manager, we also compare against two more schedules: *random* and *recursive*. We obtain a *random schedule* by randomly partitioning and ordering tasks into chunks (one per thread). We obtain a *recursive schedule* by first profiling a workload, and then applying the recursive scheduling scheme from Section 4.3. We generate these schedules offline and supply them to a *static task manager*, which loads in a schedule and populates the task queues accordingly. Since we use it solely to help analyze schedule quality, we do not report overheads for this task manager.

In the rest, since our task manager can specify a different sharing pattern for each parallel section, for each workload we report the performance from the most representative parallel section (both in terms of sharing pattern and execution time). Characteristics of those parallel sections are given in Table 5.1. For applications with parallel sections exhibiting different sharing patterns, the manager will apply different policies for each section.

5.4.2 Linear Mapping Workload Results

Before examining performance, we first verify a key assumption of our flux-based policy for linear mapping workloads: There is a strong correlation between surface flux

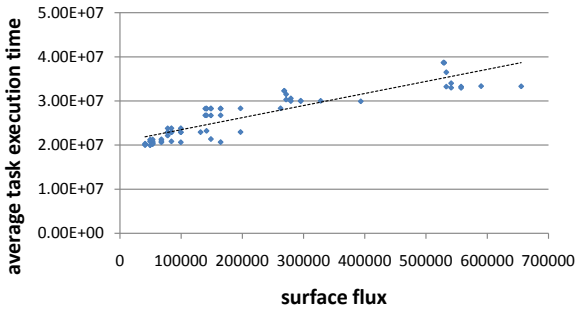


Figure 5.7: Surface flux vs. average task execution time on **mmm**. The dotted line denotes the linear fit of the data points.

Workload	Correl.
bt	1.00
mmm	0.97
conv	0.92
sp	0.99
mg	0.83
sph	0.99

Table 5.3: Flux correlation to task execution time.

and task performance. Figure 5.7 shows the correlation of **mmm** task performance to surface flux. The points represent all the task group shapes that the manager considers. As can be seen, reducing surface flux results in improved performance. Table 5.3 then reports the correlation for all the linear mapping workloads. Correlation is high across all the workloads—the minimum is 0.83.

Figure 5.8 shows the relative task execution time of the linear mapping workloads using a random schedule, the baseline manager, a recursive schedule, and our task manager with the flux-based policy. Here, we measure *just the sum of the execution time for the tasks,* ignoring the task manager itself, and show speedup versus the random schedule. This metric allows to assess the quality of the schedules, ignoring runtime overheads.

The flux-based schedules are on average 1.28x faster than the baseline schedules, which themselves capture significant data locality (i.e., the baseline is 1.13x faster than random on average). The flux-based schedules are also comparable to recursive schedules (1.04x faster on average). For **sph**, the recursive schedule is poor. Here, METIS produces sub-optimal task groups due in part to this workload having both

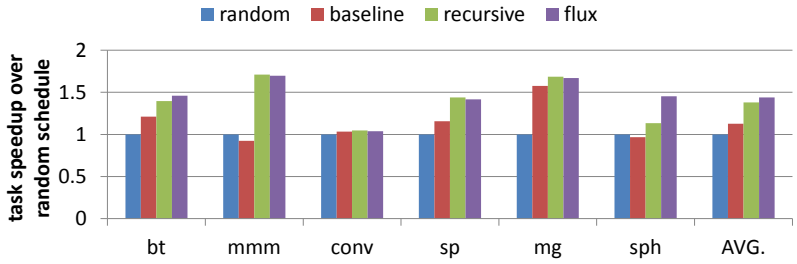


Figure 5.8: Task performance for linear mapping workloads.

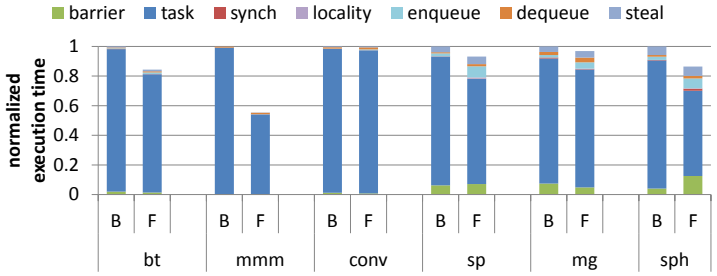


Figure 5.9: Execution time breakdown for linear mapping workloads, normalized to the baseline task manager. **B** = baseline and **F** = flux-based.

regular sharing through neighboring tasks and irregular sharing through particles. Hence the inferior performance of the recursive schedule is a limitation of METIS, not the recursive scheduling itself.

Figure 5.9 compares the total execution time for the baseline task manager (**B**) and our locality-aware manager using flux-based policy (**F**), normalized to the baseline. These times include all the overheads and load imbalances. **task**, **enqueue**, and **dequeue** report the time spent for executing, enqueueing, and dequeueing tasks, respectively. **barrier** denotes the time waiting for other threads to finish. **locality** is the time spent creating a locality-aware schedule; in particular, the time to compute minimum-flux partition shape. **synch** denotes the time spent for lock acquisition and condition variable updates, and **steal** represents the time looking for tasks to steal.

While our task manager has higher overheads than the baseline, its better schedules lead to an average 1.18x speedup. The overhead primarily comes from the enqueue rather than the minimum-flux partition computation (i.e., the **locality** portion). This is because the baseline linearizes the task space and so works with 1-D

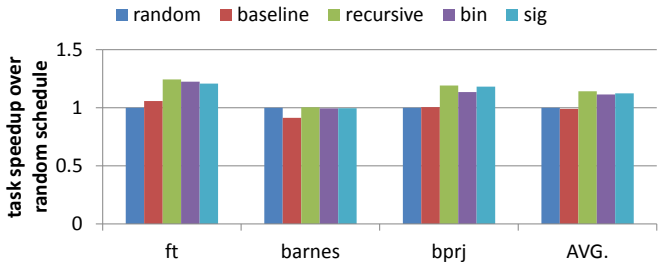


Figure 5.10: Task performance for non-linear mapping workloads.

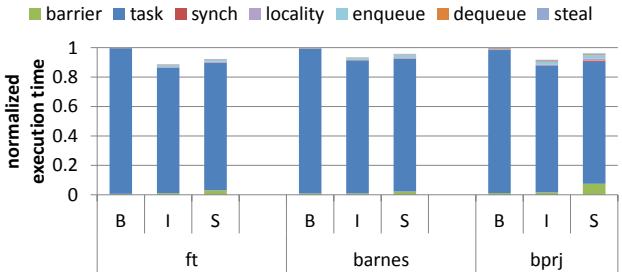


Figure 5.11: Execution time breakdown for non-linear mapping workloads, normalized to the baseline task manager. **B** = baseline, **I** = binning-based, and **S** = signature-based.

task ranges, whereas our manager does not. For 2-D and 3-D task spaces, we incur roughly 2x or 3x the enqueue overhead, respectively.

5.4.3 Non-Linear Mapping Workload Results

Figure 5.10 shows the task speedup of the non-linear mapping workloads using a random schedule, the baseline manager, a recursive schedule, and our task manager with the binning-based policy. It also shows results with the signature-based policy (**sig**), which we discuss in the next section. The binning-based schedules are on average 1.12x faster than the baseline schedules, which capture no locality on average (i.e., are within 1% of the random schedules). For **barnes**, the baseline schedule is actually worse than random because it deterministically *avoids* locality. The binning-based schedules are almost as good as the recursive schedules (1.03x average slowdown).

Figure 5.11 then shows the total execution time for the baseline manager (**B**)

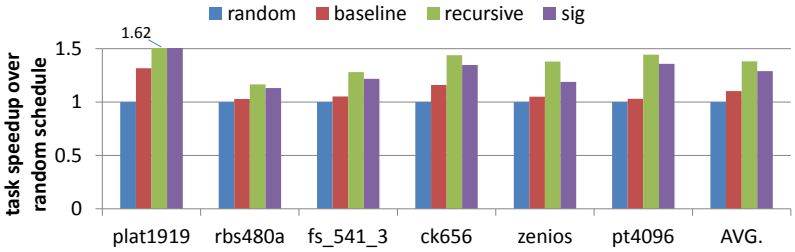


Figure 5.12: Signature-based policy task performance.

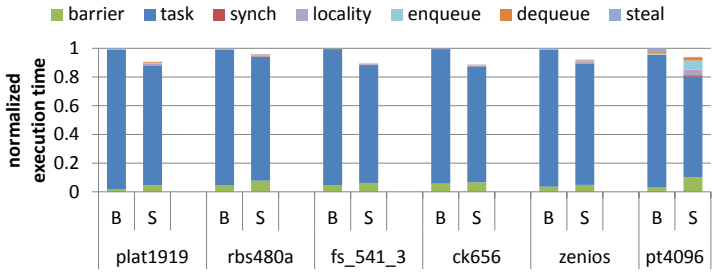


Figure 5.13: Signature-based policy execution time breakdown, normalized to the baseline task manager. **B** = baseline and **S** = signature-based.

and our locality-aware task manager using binning-based policy (**I**), normalized to the baseline. For the **I** bars, the **locality** portion is the time where the manager evaluates the user-provided mapping function to determine the bin index.

As was the case for the flux-based policy, a better schedule translates into overall execution time improvement—our manager brings about an average 1.10x speedup over the baseline. Our task manager with the binning-based policy has low overheads, although the enqueue overhead is visible across all the workloads. Recall that our task manager has to enqueue individual tasks to each bin (see Section 5.3.4), whereas the baseline manager enqueues task index ranges.

5.4.4 Random Mapping Workload Results

Of the three random mapping workloads in our study, **cg**, **smvm**, and **sphinx**, only **smvm** exhibits significant locality. Specifically, **cg** and **sphinx** both have an average

Jaccard similarity of 0.03, compared to **smvm**'s 0.19. For these two workloads, recursive schedule shows 2% performance improvement over a random schedule, demonstrating that it at least does not hurt performance when there is minimal locality.

We therefore focus our evaluation on **smvm**, but we use multiple inputs to ensure we cover a spectrum of locality patterns for random mapping workloads. Specifically, we use five different input matrices from [50], which come from various application domains such as finite element analysis, robotics, and fluid mechanics (see Figure 5.3). However, these matrices were too small to fully utilize the hardware contexts on the simulated system. We therefore duplicated the matrices column-wise, until the entire matrix contained about 2 M non-zero entries. This amounts to *unrolling* the **smvm** loop, while keeping the locality pattern the same. We also use another sparse matrix (**pt4096**) to cover the pattern recognition domain. This matrix comes from an in-house experiment, and it contains about 1 M non-zero entries. We generated signatures by scaling the fill pattern of each matrix row into 1024 bits.

Figure 5.12 shows the task speedup of **smvm** with the six different inputs using a random schedule, the baseline manager, a recursive schedule, and our task manager with the signature-based policy.

The signature-based policy improves task performance over the baseline by 1.17x on average, which itself provides some speedup over a random schedule—some matrices have significant locality between adjacent rows. The signature-based schedules, however, do not capture as much locality as the recursive schedules (1.07x slower on average). Thus, while the leading one-based task grouping and coarse ordering can capture significant locality, there is some room for improvement.

Figure 5.13 compares the total execution time for the baseline manager (**B**) and our locality-aware manager using signature-based policy (**S**), normalized to the baseline. Once again, our manager's improved schedules translate to better overall performance, giving an average 1.09x speedup. The largest overhead for our manager is the barrier time, which is mostly from the multiple barrier phases in the histogram construction (see Section 5.3.4). For **pt4096**, the enqueue cost and histogram construction (**locality**) are also significant, because the tasks are very small (i.e., few non-zero elements per row).

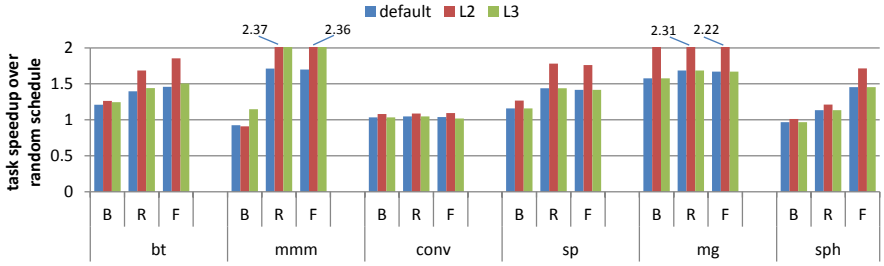
The signature-based policy is our most generic policy, in that it can be used to capture locality for any sharing patterns: albeit not necessarily as well, and with higher runtime overhead and programmer burden. To demonstrate that it can be effective elsewhere, we use the signature-based policy on our non-linear mapping workloads. Here, the user-exposed mapping function is pre-evaluated for all the tasks, and the return value is scaled to $[0, 2^{1024} - 1]$ to form the signature.

Figures 5.10 and 5.11 show the task speedup and execution time with the signature-based policy. The task speedup is comparable to the binning-based policy, demonstrating the generality of the signature-based approach. However, the signature-based policy suffers from higher overheads (e.g., **barrier**) due to histogram generation. Histogram generation could be removed if stealing were cheap enough (e.g., hardware support), and captured enough locality (examined later).

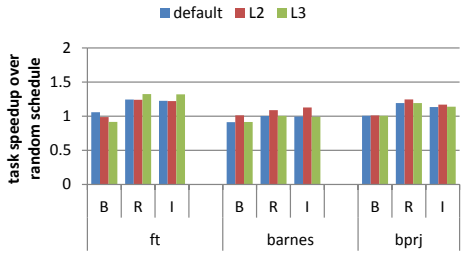
Signature-based policy could also be used to schedule linear mapping workloads as well. We expect the task performance to be comparable to that of the flux-based approach, since the locality information that the signature-based policy conveys is the superset of the flux-based policy. We leave this as our future work.

5.4.5 Sensitivity Study: Signature Bit Width

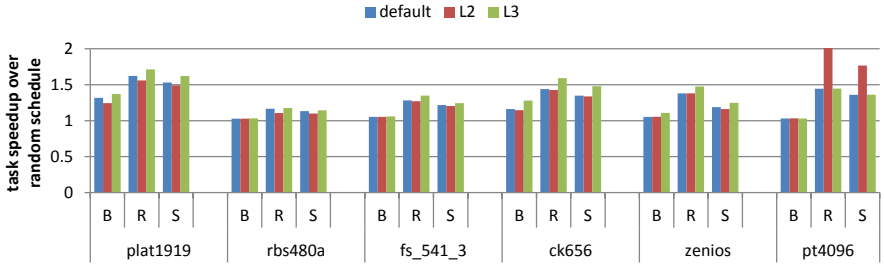
We now summarize the sensitivity of the signature-based policy to its key parameter, the signature size. Signature size, in bits, denotes the maximum *resolution* that our scheduler can identify different leading one positions. So larger signatures capture more locality, up to a point. The cost for the histogram reduction, however, increases linearly with signature size (see Section 5.3.4). Thus, we see an *inflection point* in the execution time as we increase signature size—once the signature is large enough to capture most of the locality, any further increase incurs too much overhead. For most of our workloads, the inflection point is around 1024 bits: On average, 1024 bits gives 1.04x speedup over 512 bits, and 1.03x speedup over 2048 bits.



(a) Linear Mapping Workload Sensitivity



(b) Non-Linear Mapping Workload Sensitivity



(c) Random Mapping Workload Sensitivity

Figure 5.14: Schedule sensitivity to cache latency. The legend shows the cache level slowed. The schedules are: **B** = baseline, **R** = recursive, **F** = flux-based, **I** = binning-based, and **S** = signature-based.

5.4.6 Sensitivity Study: Cache Hierarchy Latency

As discussed in Chapter 2, we expect the number of cores on a chip to grow, which will require larger and more complex cache hierarchies, and therefore increase cache access latencies. Here, we slow down the access time for different cache levels by 10x, and examine the *task performance*, to see what trend we can expect from our

manager. Figure 5.14 shows the results.

For each workload, **B** and **R** denote the speedup for the baseline and recursive schedule, respectively, normalized to a random schedule on the same cache configuration. The other letter (**F**, **I**, or **S** for flux, binning, or signature-based policies, respectively) shows the speedup of our locality-aware task manager. For each schedule, we show three bars indicating speedup on the default configuration, 10x L2 latency, and 10x L3 latency.

As expected, the performance advantage of our locality-aware manager *increases as caches get slower*. In particular, the speedup over the baseline can be as high as 2.26x (**mmm** with 10x L2). Overall, the speedup of our task manager relative to the baseline increases from 1.20x for the default configuration to 1.29x and 1.23x with 10x L2s and L3s, respectively.

Recursive scheduling is able to exploit locality through both the L1 cache (workloads where we see peaks in the L2 bars) and the L2 cache (peaks in the L3 bars). Our task manager, however, closely follows the performance trend regardless of the cache configuration.

5.5 Reducing Programmer Burden through Architectural Support

We have shown that by combining the workload sharing pattern information and locality hints, a task manager can capture significant data locality; in many cases *match* the quality of a recursive schedule. We now explore using hardware support to reduce the burden on the programmer (or the compiler) of providing locality hints.

5.5.1 Approximating Sharing Vector with Cache Hit Counters

For workloads with linear mapping, we use hardware cache hit counters to approximate the sharing vector used by our flux-based policy. Cache hit statistics are available on many modern microprocessors through performance counters [1].

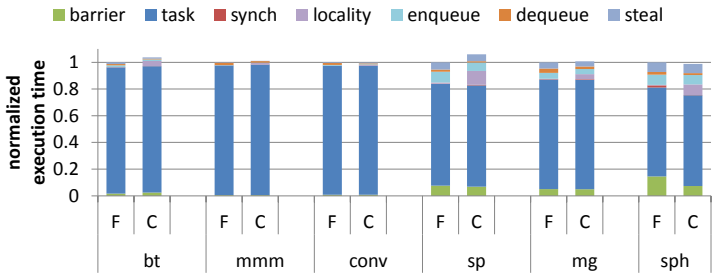


Figure 5.15: Sharing vector approximation with cache hit counters. **F** = flux-based and **C** = counter.

For such a processor, we can obtain the cache hits for *sample tasks* in each dimension of the task space. For example, in a 3-D task space, we execute tasks $(0,0,0) \rightarrow (1,0,0) \rightarrow (1,1,0) \rightarrow (1,1,1)$ on a single thread to obtain the cache hit counts for X, Y, and Z directions, and use those as an approximate sharing vector.

We modify our API (see Table 5.2) to include **LMFlux_ApproxSharingVec()**, which invokes this process, and records the cache hit counts for use by the minimum-flux partitioning algorithm as the sharing vector. The task manager takes note of the executed sample tasks and precludes them from the parallel section.

Figure 5.15 shows the execution time of the linear mapping workloads with the user-provided sharing vector (**F**) and the cache hit counter (**C**), normalized to the former. The **locality** portion of the **C** bars include the time spent on task sampling.

We can see that the performance of the counter-based approach is very close to the flux-based policy. The cache hit counter approximation does not lead to the exact same sharing vector, but small differences in the vector do not drastically alter the partition shape. Thus, the schedules are as good as the flux-based policy. However, for **sp**, the counter-based approach is 1.06x slower; for those workloads with few tasks (see Table 5.1), the sampling overhead could be significant.

5.5.2 Automated Signature Recovery through Hardware

For random mapping workloads, we propose a hardware mechanism to automate signature collection.

As mentioned in Section 5.2, in many applications, such as those using iterative

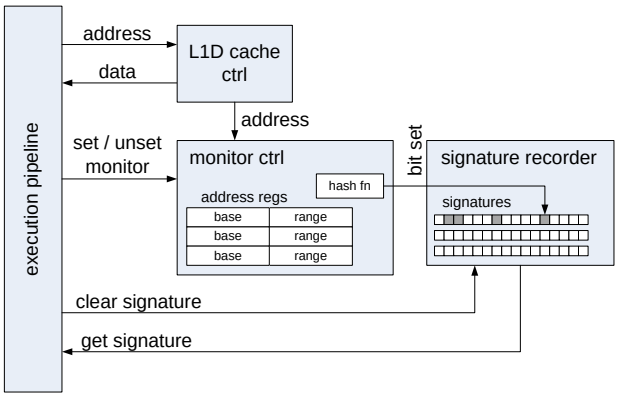


Figure 5.16: Hardware monitor and signature recorder.

solvers, key parallel sections are repeatedly executed—i.e., with the same data structures and sharing among tasks, but with different data values. In such cases, we can use hardware to capture and record the signature for each task during the first execution of the parallel section, the *profiling phase*. For later executions of the section, we can use the previously recorded signatures.

In fact, for such parallel sections, we could even generate and store the schedule for the section right after the profiling phase (regardless of whether we have signature-capture hardware or not), and simply copy that schedule for each of the later executions; this would better amortize the scheduling overheads. Here, we consider and evaluate only the recording of signatures, since we are interested primarily in demonstrating that this can be done effectively.

Our signature-capture hardware is comprised of two components: a monitor for arbitrary memory regions and a signature recorder. An application specifies the shared memory region to track during the task execution as a base address and a range, and the monitor tracks this region. The monitor differs from the x86 **MONITOR** support in that (1) it allows arbitrary sized regions to be watched, (2) it captures read accesses as well, and (3) it keeps monitoring even after an event triggers, until the user explicitly disengages the monitor. It more closely resembles Mondrian memory protection hardware [74].

As shown in Figure 5.16, each core has a monitor controller next to its L1 data cache controller. The monitor holds the base address and range for all active regions,

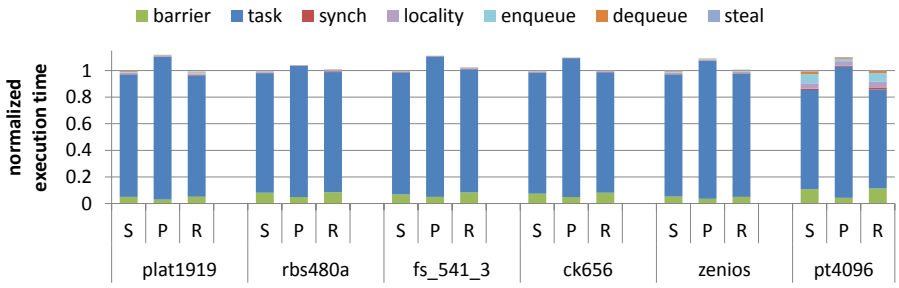


Figure 5.17: Automated signature retrieval with hardware monitors. **S** = signature-based, **P** = profile, and **R** = reentry.

and snoops accesses to the cache. When it is triggered, it hashes the triggering address and sends it to the signature recorder, which sets the corresponding bit in that region's signature register.

We implement the signature-capture hardware in our simulator; we assume 1024-bit signatures, and provide instructions to clear and retrieve signatures. We transfer 64 bits at a time from the signature recorder.

We also modify our signature-based policy to utilize this hardware as follows. For the profiling phase, our manager takes the base address and address range of the shared memory region from the user through **LMSig_SetMonitor()** (see Table 5.2), and sets the hardware monitor logic. For example, for **smvm**, the user provides the start address and size of the shared vector. The manager then executes the parallel section using the baseline schedule, while collecting the signature information: It clears the signature at the beginning of a task, and at the end of the task reads and stores the signature in a global signature array. If the user provides multiple monitors for parallel sections that touch multiple shared memory regions, the manager concatenates their signatures together before storing them. When the same parallel section is encountered again, the task manager uses the stored signatures instead of user-provided ones.

Figure 5.17 shows the execution time for a *single instance* of the **smvm** parallel section, normalized to the original signature-based policy, **S**. In the figure, **P** denotes the profiling phase of the task manager with hardware monitors, and **R** denotes the task manager using the signatures captured with the hardware. In the profiling

phase, the task execution time is much worse, since it uses the baseline task manager schedule. However, the overhead for signature collection, the **locality** portion for **P** bars, is negligible. When the same parallel section is reentered, **R** obtains the same performance as with user-provided signatures.

5.6 Conclusions

By implementing a specialized task management policy for each workload sharing pattern, we were able to effectively approximate recursive scheduling at low costs. Specifically, for each parallel section, a user (or a compiler) specifies the sharing pattern and a simple locality hint. The underlying runtime then transparently selects and applies a different task management policy to generate a schedule that captures significant locality. Since the basic task-parallel programming API is kept intact, our approach can be easily integrated with many programming models.

At this point, it would be interesting to question how these approximation schemes will fair at significantly larger scales: e.g., a thousand cores. It basically boils down to how the single-level approximation will compare against the multi-level recursive scheduling. We believe that the single-level approximation will provide comparable performance as long as it successfully exploits locality on the key working set of a workload. However, combining our single-level approximations in a recursive fashion might be necessary.

Chapter 6

Locality-Aware Task Stealing

> It's gotta stop, you know, 'cause it's.. It's very Peter Pan, you know? It's very 'I'll never grow up,' you know what I mean? I'll just jump around in the Eddie Murphy Raw suit forever.
>
> — Conan O'Brien

Dynamic task management comprises two components: (1) task scheduling, or the initial assignment of tasks to threads, and (2) task stealing, or balancing the load by transferring tasks from one thread to another that is idling. In Chapters 4 and 5, we explored locality-aware scheduling for task-parallel programming systems. When tasks are scheduled in a locality-aware fashion, however, stealing better not disrupt the exploited locality. In this chapter, we explore locality-aware task stealing.

6.1 Motivation

For those results presented so far, we observed minimal task stealing taking place. However, this does not imply that locality is not important for stealing.

Intuitively, task stealing will benefit the most from being locality-aware when many tasks are stolen. This can happen either because the application itself is not

well balanced, or because the underlying system introduces imbalance. As systems with many cores become commonplace, one frequent source of load imbalance will be multiprogramming—software threads from multiple applications will compete for hardware contexts, and will periodically be context switched. In a similar manner, automatic language runtimes, such as our task managers, may generate schedules each assuming it owns the entire chip, only to result in destructive oversubscription [56]. While a hardware context is not available, potentially large number of tasks from the victim thread may be stolen by the remaining threads.

Previously proposed stealing schemes, however, are *locality-oblivious*. The most widely adopted stealing scheme, *randomized stealing* [10], chooses a victim at random, and steals one or more tasks (stealing multiple amortizes the stealing overhead). It provides good characteristics such as even load distribution, simple implementation, and theoretically bounded execution time, but its randomness renders it inherently locality-oblivious. In fact, if the task schedule is also locality-oblivious, we expect locality-oblivious stealing to have little impact on cache behavior. However, for a locality-aware schedule, this stealing policy may significantly decrease the performance of stolen tasks.

We verify this by inducing large amounts of task stealing by emulating context switching—after producing a task schedule for 32 threads, we offline a subset of the threads, and rely on stealing to redistribute tasks from the offlined threads. Then we measure the *sum of the execution time of the tasks* to assess the impact on locality.

Figure 6.1 shows the task performance trend of a random (i.e., locality-oblivious) schedule and a recursive (i.e., locality-aware) schedule for two workloads, normalized to the performance of a random schedule when *no threads are offlined*. We use the same randomized stealing policy for both schedules—it randomly selects a victim and tries to steal half of the victim's queue with a prescribed upper bound (an empirical value of 8 was used [44]). If it fails to steal anything, it visits the other potential victims in a round-robin fashion. The measurements are taken from the simulated Throughput Processor (see Section 4.4.1).

The case where no threads are offlined is the same data presented earlier, where recursive schedules show much better performance than random schedules. However,

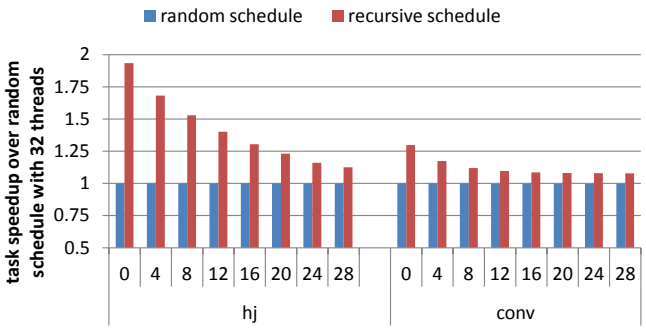


Figure 6.1: Impact of locality-oblivious stealing on a locality-aware schedule. The numbers along the x-axis denote the number of threads offlined. Speedup is over the performance of a random schedule with no threads offlined.

the performance benefit decreases as more threads are offlined, since the locality exploited in recursive schedules is disrupted from randomized stealing. On the other hand, the performance with a random schedule is independent of the number of threads offlined. This highlights the importance of *locality-aware stealing* when the tasks are scheduled in a locality-aware fashion.

6.2 Locality Analysis of Task Stealing

We approach task stealing using a similar analysis methodology we used for task scheduling (see Section 4.2.1). For this discussion, we assume a locality-aware task schedule created from a recursive scheduler. We explore the impact on locality of the two key design decisions for task stealing: *which* tasks to steal, and *how many* tasks to steal. Then we use our insights to develop a reference locality-aware stealing scheme, *recursive stealing*, which we use to quantify the impact of the different aspects of locality-aware task stealing.

6.2.1 Implications of Victim Selection on Locality

We first consider the locality between those tasks that have already executed on a core, and those tasks that are to be stolen. Randomized victim selection fails to capture this locality.

Assuming a multi-level memory hierarchy, it would be the best if such locality is exploited through the highest-level cache (i.e., L1). If no such tasks are available, victim tasks should be chosen among those that will give sharing through the next level (i.e., L2), and so forth. In essence, a thief should look for tasks in a recursive fashion, from top to bottom, so that the *stealing scope* gradually increases from one step to another as we lower the cache level where sharing will take place.

6.2.2 Implications of Steal Granularity on Locality

We next consider the locality among the stolen tasks. A natural steal granularity that would provide good locality among victim tasks is a task group—after all, this is how recursive scheduling constructs groups. Stealing a task group at a time amortizes steal overheads as well.

Stealing a *fixed amount* or *fixed portions* of tasks each time may undershoot or overshoot a task group boundary, and thus break a group. Stealing an already-stolen task (i.e., *secondary stealing*) breaks locality within the stolen task groups as well; performing secondary stealing against a task group that has strong internal sharing, such as an L1 group, may adversely affect performance.

On the other hand, the level of task group stolen affects load balancing. Stealing a coarser granularity task group (i.e., lower-level group) preserves more locality among stolen tasks, but could increase load imbalance, assuming we prohibit secondary stealing. One way to *emulate* the locality of stealing a coarser task group while maintaining flexibility is to steal smaller groups (i.e., upper-level groups) but enforce *steal ordering*: If a thread steals again, it follows the group ordering specified by the recursive schedule.

6.3 Recursive Stealing

In this section we present a reference locality-aware stealing scheme, *recursive stealing*. Our discussion so far suggests that (1) stealing scope should recursively expand through the memory hierarchy, and that (2) stealing should be performed at the

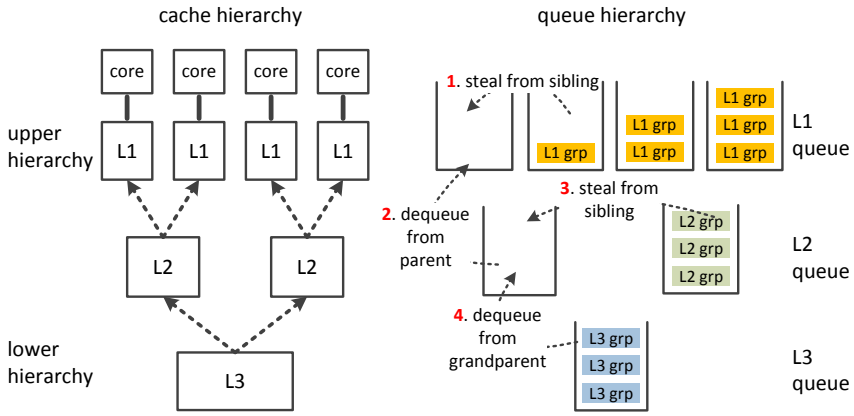


Figure 6.2: Recursive stealing. The top-level queues (that hold tasks) are not shown. Colored numbers indicate the order that the leftmost core visits queues.

granularity of task groups. Figure 6.2 illustrates the scheme.

We maintain a hierarchy of queues, where a queue exists for each cache; these queues may be implemented as software objects or hardware components. A queue at a specific level holds task groups that fit in the cache for that level: an L3 queue holds L3 groups, an L2 queue holds L2 groups, etc. The order these task groups are stored in reflects the *group ordering* determined by the recursive schedule. Not shown are the task queues, which hold the actual tasks. For the example hierarchy, one task queue exists per L1 queue. Once we dequeue a task group and move it to an upper-level queue, we logically break it down into upper-level groups. For example, when we transfer an L2 group to an L1 queue, we decompose it into L1 groups.

Recursive stealing actually *interleaves* regular dequeues and steal operations to exploit as much locality as possible from the original recursive schedule. In our example, tasks are replenished as follows: Once a task queue is empty, a thread attempts to dequeue from its L1 queue. If the L1 queue is empty, it attempts to steal from the sibling L1 queue. We interleave steals with dequeues in this example because the L2 caches are shared. If a thread steals from a sibling L1 queue, it grabs an L1 group that shares data in the L2 cache with (1) the tasks it just executed, and (2) the tasks the sibling core(s) are currently executing; thus, we exploit the shared cache. If all the steal attempts fail, the thread tries a regular dequeue from the L2

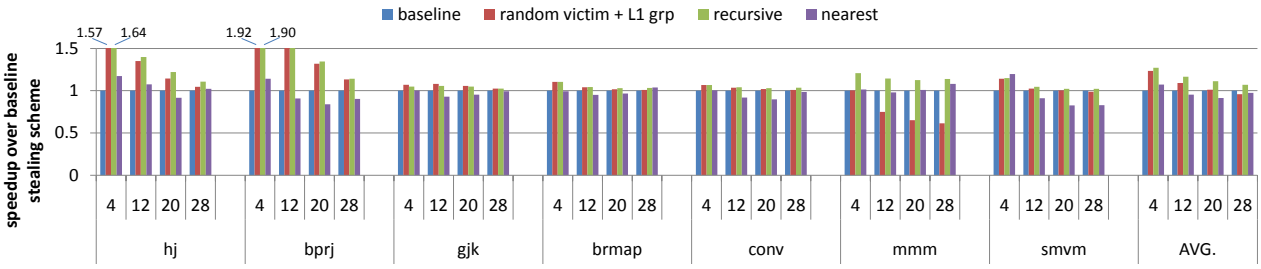


Figure 6.3: Performance improvement of stolen tasks. The numbers along the x-axis indicate the number of threads offlined. At each thread count, performance is normalized to that of the baseline stealing scheme (chooses random initial victim, and tries to steal half of its tasks: maximum of 8 tasks).

queue. If the L2 queue is empty as well, it climbs down the hierarchy and repeats the process: It attempts to steal from the sibling L2 queue, and then visits the L3 queue. When stealing, a thread grabs the task group at the tail of the victim queue, in the same way randomized stealing operates [10].

Note that we do not allow stealing across task queues. This means that the minimum steal granularity is an L1 group, and that a stolen L1 group does not get targeted for secondary stealing. Since a typical L1 group comprises 4 to 8 tasks for our workloads, this does not impose significant load imbalance.

In essence, recursive stealing exploits locality through two features: (1) by performing *recursive victim selection* to exploit locality across potentially multiple levels of shared caches, and (2) by stealing at minimum a whole L1 group to guarantee locality among the stolen tasks.

6.4 Evaluation of Locality-Aware Task Stealing

We implemented the recursive stealing scheme as a software library for our Throughput Processor configuration (see Section 4.4.1), and compared its performance against the baseline randomized stealing (see Section 6.1). We used the same task-parallel workloads introduced in Section 4.1.1. In conducting the experiments, the same recursive schedule was provided for the different stealing implementations.

We first present the performance summary. We then isolate each feature of recursive stealing to assess its relative importance. In particular, we try to answer the following questions: (1) How beneficial is locality-aware stealing? (2) How much does victim selection matter? (3) How much does steal granularity matter?

Q1: How beneficial is locality-aware stealing?

Figure 6.3 compares the performance of *stolen* tasks over various stealing schemes, as we offline some threads. In the figure, **random victim + L1 grp** denotes the performance of recursive stealing when recursive victim selection is turned off; we will discuss its implications later.

Looking at the average, the recursive stealing scheme provides benefit across all numbers of offlined threads, but the average benefit decreases as more threads are offlined. When 4 threads are offlined, the average speedup over randomized stealing is 1.27x. The reason speedup sometimes decreases with more offlined threads is as follows: In general, executing the tasks on fewer cores (i.e., spreading the application's data across fewer caches) naturally captures more sharing. This is especially true for applications with small working sets. For this case, there is less potential for improving locality with fewer cores or threads.

In the figure, when we look at individual workload performance, we can see that those workloads that benefited the most from recursive scheduling (see Section 4.4.2), i.e., **hj**, **bprj**, and **mmm**, significantly benefit from recursive stealing. **hj** and **bprj**'s L1 groups are significantly larger than 8 tasks (the upper bound for the baseline stealing scheme), so stealing an L1 group at a time gives significant locality boost. When 28 threads are offlined, recursive stealing reduces the L1 miss rate by 1.28x and 1.39x, respectively. For **mmm**, however, an L1 group contains only a single task. Recursive stealing exploits locality through L2 instead, and reduces the L2 miss rate by 1.56x (28 threads offlined). These observations show that both steal granularity and victim selection contribute to the benefits of recursive stealing.

smvm presents an interesting case. The workload benefited significantly from recursive scheduling, but the performance improvement due to recursive stealing is not as profound. By coincidence, random stealing's upper bound of 8 tasks matches the number of tasks within an L1 group. However, the baseline performs worse

due to secondary stealing; **conv** behaves similarly. On the other hand, for **gjk** and **brmap**, which exhibit simple sharing through consecutive tasks, just maintaining task ordering is good enough to preserve most of the locality.

The source of randomized stealing's poor behavior is not so much that it chooses its victims in random, *but that its victim selection is locality-oblivious*. To make our point clear, we also show the performance of a nearest-neighbor stealing scheme in Figure 6.3. In this scheme, a thief always chooses its nearest-neighbor as the first victim; if it fails, it visits other victims in round-robin fashion. It uses the same 8 tasks upper bound on stealing as the baseline. As can be seen, this scheme, which is also locality-oblivious, performs as poorly as randomized stealing.

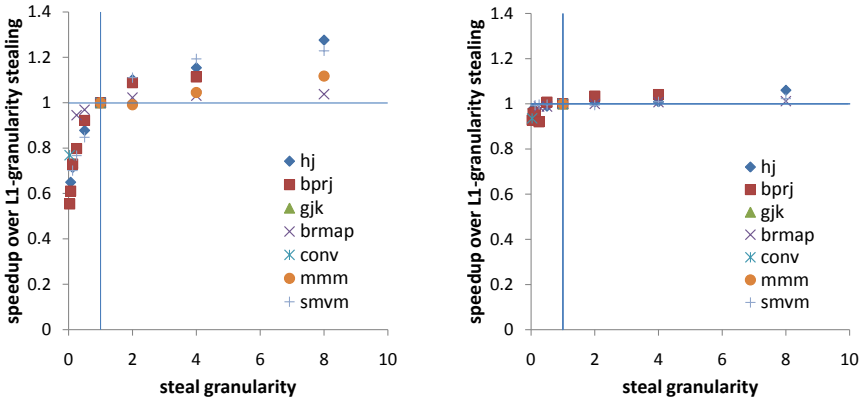
Q2: How much does victim selection matter?

To further isolate the benefits from recursive victim selection, we implement a stealing scheme which (1) selects a victim at random, but (2) steals an L1 group at a time. The **random victim + L1 grp** in Figure 6.3 shows its performance. Most of our applications see the same performance from the **random victim + L1 grp** policy as recursive stealing. However, for workloads exhibiting strong sharing through the L2, recursive victim selection is able to capture the locality. Such effect is the most profound on **mmm**, since its L1 group amounts to a single task; hence locality should be synthesized through recursive victim selection. In particular, we can see that random victim selection exhibits deteriorating performance as more threads are offlined. Offlining more threads means more victims to choose from, making it more likely that random victim selection would not capture the locality. The baseline scheme, however, manages to exploit the locality among the stolen 8 tasks at the least, to further widen the performance gap.

Q3: How much does steal granularity matter?

We now vary the steal granularity. Intuitively, if load balance is not an issue, stealing larger chunks of tasks will better exploit locality. Conversely, stealing a smaller number of tasks will fail to preserve the locality specified in the schedule. However, we find that sensitivity to steal granularity is regulated by the degree to which thieves contend over victim tasks.

In Figure 6.4, the x-axis denotes the normalized steal granularity, where steal



(a) High Contention (4 Threads Off, 28 Threads On)

(b) Low Contention (28 Threads Off, 4 Threads On)

Figure 6.4: Application sensitivity to steal granularity. Steal granularity is normalized to the size of an L1 group, and the performance of a stolen task is normalized to the case where an L1 group is stolen at a time. Vertical and horizontal lines denote $\mathbf{x} = \mathbf{1}$ and $\mathbf{y} = \mathbf{1}$, respectively.

granularity of 1 denotes the case when a single L1 group is stolen at a time. The y-axis is the performance of a stolen task, normalized to that when we steal an L1 group at a time. When multiple L1 groups are stolen, they are stolen in an atomic fashion, and are not potential victims for secondary stealing.

Figure 6.4a shows the performance under high contention. In this configuration, 4 threads are offlined, and 28 online threads compete over the tasks assigned to the 4 offlined threads. When the contention is high, it becomes hard to preserve locality across multiple steal operations from the same thread. Therefore, performance is relatively sensitive to steal granularity. Specifically, we can see that reducing steal granularity below an L1 group results in significant loss in locality, since the strong sharing within an L1 group is now broken. Conversely, increasing the steal granularity beyond a single L1 group gives noticeable locality improvement.

On the contrary, Figure 6.4b shows the performance under low contention, when only 4 threads are online. These threads rarely contend over the tasks originally assigned to 28 offlined threads, and recursive stealing improves locality by preserving the group ordering across different steal attempts from the same thread (see

Section 6.2.2). In fact, recursive stealing effectively collects smaller task groups to emulate the effect of executing a larger granularity group. As a result, sensitivity to steal granularity is much smaller than the high contention case.

6.5 Pattern-Specific Stealing Schemes

We now discuss how to implement a practical locality-aware stealing scheme. Recursive stealing, while being able to capture significant locality, relies on a recursive schedule, which could be non-trivial to obtain online.

Instead, similar to how we approximated recursive scheduling in Chapter 5, we can try to approximate recursive stealing as well. Specifically, we develop a pattern-specific stealing scheme for each workload sharing pattern, so that victim selection and steal granularity can be approximated on a pattern-by-pattern basis. In the rest of the section, we describe how each task management policy can be extended to incorporate locality-aware stealing.

6.5.1 Flux-Based Task Management

For the flux-based policy, we use the shape of a partition to determine the steal granularity. Specifically, to maximize the locality among stolen tasks, we choose the steal granularity to be a 1-D 'strip' of tasks along the maximal sharing dimension. For the example in Figure 5.4, we would steal a set of tasks with consecutive Y values (the dimension with maximal sharing) and constant values for X and Z. For 1-D task spaces, such a strip would be an entire task group, so we just steal a pre-determined amount of tasks. If a thief succeeds, it records the victim (as a *steal hint*), and tries to revisit the same victim at the next steal attempt.

6.5.2 Binning-Based Task Management

Basically, we use half a bin as the steal granularity. This will provide locality among the stolen tasks. However, depending on the mapping function, a bin may contain a large number of tasks. Thus a thief steals *up to* half a bin, with some pre-determined

maximum number of tasks. We maintain steal hint here as well, so that locality can be exploited across different steal attempts.

6.5.3 Signature-Based Task Management

As discussed earlier (see Section 5.3.3), for the signature-based policy, a range of leading one positions amount to a bin. Therefore, following the same rationale as the binning-based policy, a thief steals up to half a bit range, with an empirical maximum value. Steal hint is also maintained.

6.5.4 Performance Results

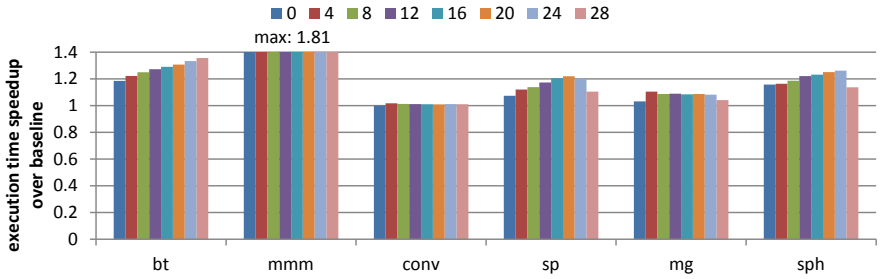
We implemented the above pattern-specific stealing schemes for each of the task management policy, and measured the performance on the simulated Tiled Processor (see Section 4.4.1). Here, we used the workloads discussed in Section 5.2. For all the policies, when the steal hint is empty, a thief first looks for those queues on the same tile and then the remote queues, in random queue order. Stealing is performed at the tail of the victim's queue.

The baseline task manager is also extended so that it utilizes a randomized stealing scheme, which chooses a random initial victim, and tries to steal half of its tasks with a prescribed upper bound (an empirical value of 8 was used [44]). If it fails to steal anything, it visits the other potential victims in a round-robin fashion.

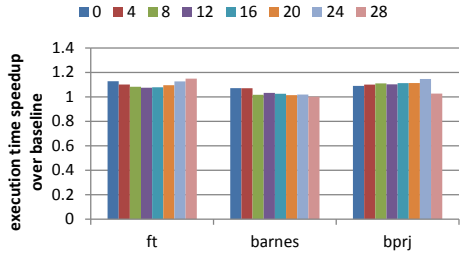
Same as before, we stress the stealing mechanism of the baseline task manager and our task manager by *emulating context switching*: We let each task manager generate schedules for 32 cores, and then offline a subset of cores to force the managers to rely on stealing for task redistribution.

Figure 6.5 shows the speedup of our manager over the baseline (total speedup, including overheads), as we offline different numbers of threads. When 28 threads are offlined, about 88% of the tasks executed are stolen tasks.

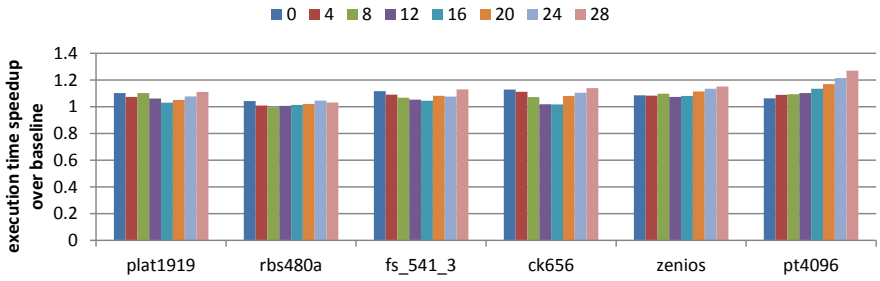
On average, the locality-aware task manager provides a significant performance advantage (1.13x *regardless of the thread count*). This verifies that our pattern-specific stealing schemes can effectively preserve locality while load balancing. However, for a



(a) Linear Mapping Workload Performance



(b) Non-Linear Mapping Workload Performance



(c) Random Mapping Workload Performance

Figure 6.5: Stealing performance trend over increasing number of offlined threads.

given workload, we observe two different performance trends as the number of threads varies: $\cup$ shape and $\cap$ shape.

The $\cup$ shape is more common. In this case, as we offline threads, the advantage over the baseline deteriorates at first, since the stolen tasks have less locality with the originally scheduled tasks (than the original tasks have with each other). However, as more tasks are stolen, the advantage returns because the locality-aware stealing

captures the locality amongst the stolen tasks themselves.

For the workloads that exhibit the ∩ shape trend, however, stolen tasks have a lot of sharing with the originally scheduled tasks. Here, spreading an application's data across fewer caches naturally captures more locality (the same as in Section 6.4). Our locality-aware task manager is more effective than the baseline at capturing this, but as we offline enough threads, the baseline manager sees the same benefits.

6.6 Conclusions

When tasks are scheduled in a locality-aware fashion, task stealing should be locality-aware as well. Ideally, by preserving the task grouping and ordering information specified in the original schedule, locality can be exploited while load balancing. In practice, such stealing can be performed at low costs by implementing pattern-specific stealing schemes.

Again, it would be interesting to see how single-level stealing would perform on thousand-core systems. On systems with even deeper cache hierarchy, stealing schemes may need to be composed in a hierarchical fashion.

Chapter 7

Conclusions

> That would be newsworthy, don't ya think? Today, a young man ... realized that all matter is merely energy condensed to a slow vibration; that we are all one consciousness experiencing itself subjectively. There's no such thing as death, life is only a dream, and we are the imagination of ourselves.
>
> — Bill Hicks

This thesis studied how locality can be exploited for task-based many-core runtime systems. The contributions of the thesis are the following.

- We improved a shared-memory MapReduce runtime system to exploit locality on a multi-socket many-core machine. To efficiently cope with NUMA issues, the runtime had to be optimized across all the layers. The findings could be generalized to other structured programming systems as well.
- We provided a graph-based locality analysis framework that shows how task-level locality translates onto complex many-core cache hierarchy. By utilizing the framework, we also developed a generic, cache hierarchy-aware scheduling scheme for task-parallel programming systems.

- We developed a novel class of practical, locality-aware task managers that exhibit significant performance improvement while keeping the simple task-parallel programming model intact. The task managers effectively approximate the reference recursive scheduling scheme by leveraging workload sharing patterns and simple locality hints.

- We devised locality-aware stealing schemes that preserve locality while load balancing. To exploit locality, a stealing scheme should adhere to the grouping and ordering specified by the original schedule while transferring tasks.

Overall, the key to exploiting locality was to utilize the high-level information to generate efficient schedules at low costs.

For the structured programming systems, the information could be obtained from the explicit data dependency. Through the MapReduce case study, we saw that the constrained programming model reduces the schedule solution space, so locality-aware schedules could be generated with relative ease. Rather, exploiting locality on the data structures that provide intermediate result buffering became more important.

On the other hand, to exploit locality on the more complex, task-parallel programming systems, we needed a framework to quantitatively analyze the impact of different scheduling decisions, and to identify the key attributes necessary to generating a quality schedule. By applying task grouping and ordering in a recursive fashion, locality across the task structure could be exploited.

Using the insights obtained from the framework, we were able to develop practical locality-aware task managers for task-parallel programming systems. Here, the key to reducing the schedule generation costs was again to utilize the high-level information—the workload sharing pattern and locality hint. By developing a dedicated task management policy for each workload sharing pattern, we were able to realize much of the scheduling benefits at low costs.

Lastly, we have shown that task stealing can be made locality-aware as well, by preserving the locality exploited in the original schedule while load balancing. In particular, by developing pattern-specific stealing schemes, locality could be effectively exploited at low costs.

As discussed, it would be interesting to see how single-level scheduling and stealing schemes will perform on future many-core chips with even deeper cache hierarchies. It could be necessary to compose our task management schemes in a hierarchical fashion. Also, on a thousand-core system, using software-only task managers may not be sufficient. Adding hardware support to accelerate our task managers would be the next logical step. We leave these as our future work.

Bibliography

[1] Performance Application Programming Interface, `http://icl.cs.utk.edu/papi`.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, 2000.

[3] ARM. The ARM Cortex-A9 processors. White Paper, 2009.

[4] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[5] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 processor: A 64-core SoC with mesh interconnect. In *Digest of Technical Papers of the 2008 IEEE International Solid-State Circuits Conference*, pages 88–598, 2008.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2000.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[8] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 1995.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, 1994.

[11] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[12] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 43–57, 2008.

[13] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, pages 21–29, 1997.

[14] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 227–238, 2006.

[15] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 35–46, 2011.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, pages 44–54, 2009.

[18] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 105–115, 2007.

[19] Cray. Chapel Language Specification 0.796, 2010.

[20] W. Dally. The future of GPU computing. In *the 2009 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2009.

[21] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.

[22] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web*, pages 271–280, 2007.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 137–149, 2004.

[24] S. J. Deitz, B. L. Chamberlain, S.-E. Choi, and L. Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 155–166, 2003.

[25] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 9:1–9:12, 2011.

[26] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326, 2009.

[27] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2006.

[28] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[29] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[30] S. Ghemawat and P. Menage. TCMalloc : Thread-caching malloc, `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[31] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 341–342, 2010.

[32] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[33] C. Hughes, C. Kim, and Y.-K. Chen. Performance and energy implications of many-core caches for throughput computing. *IEEE Micro*, 30(6):25–35, 2010.

[34] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. *IEEE Transactions on Parallel and Distributed Systems*, 18:1028–1040, 2007.

[35] IEEE. POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995).

[36] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing*, pages 604–613, 1998.

[37] Intel. Thread Building Blocks, `http://www.threadingbuildingblocks.org`.

[38] Intel. From a few cores to many: A tera-scale computing research review. White Paper, 2006.

[39] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. J. Irwin, and Y. Zhang. Cache topology aware computation mapping for multicores. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, pages 74–85, 2010.

[40] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 113–122, 1995.

[41] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 140–151, 2009.

[42] Khronos Group. The OpenCL Specification. Version 1.1, 2010.

[43] C. Kim, D. Burger, and S. W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *IEEE Micro*, 23(6):99–107, 2003.

[44] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 162–173, 2007.

[45] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, 1997.

[46] M.-L. Li, R. Sasanka, S. Adve, Y.-K. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, pages 34–45, 2005.

[47] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, 2005.

[48] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 69–80, 2007.

[49] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. MineBench: A benchmark suite for data mining workloads. In *Proceedings of the 2006 IEEE International Symposium on Workload Characterization*, pages 182–188, 2006.

[50] National Institute of Standards and Technology. Matrix Market, `http://math.nist.gov/MatrixMarket`.

[51] U. Nawathe, M. Hassan, K. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-core, 64-thread, power-efficient SPARC server on a chip. *IEEE Journal of Solid-State Circuits*, 43(1):6–20, 2008.

[52] NVIDIA. CUDA C Programming Guide. Version 4.1, 2011.

[53] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1996.

[54] OpenMP ARB. OpenMP Application Program Interface. Version 2.5, 2005.

[55] Oracle. The Fortress Language Specification. Version 1.0, 2008.

[56] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with Lithe. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, pages 376–387, 2010.

[57] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60–71, 1996.

[58] S. Phillips. VictoriaFalls: Scaling highly-threaded processor cores. In *Hot Chips 19*, 2007.

[59] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, 2006.

[60] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[61] B. Saha, A. Adl-Tabatabai, R. Hudson, V. Menon, T. Shpeisman, M. Rajagopalan, A. Ghuloum, E. Sprangle, A. Rohillah, and D. Carmean. Runtime environment for tera-scale platforms. *Intel Technology Journal*, 11(3):207–216, 2007.

[62] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 311–322, 2010.

[63] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harrisa, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[64] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers*, pages 18:1–18:15, 2008.

[65] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *Proceedings of the ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 115–126, 2005.

[66] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics*, 28(1):4:1–4:11, 2009.

[67] Sun Microsystems. Solaris: Memory and thread placement optimization developer's guide. 2007.

[68] Sun Microsystems. Sun Studio 12: Performance analyzer. 2007.

[69] Sun Microsystems. Sun SPARC enterprise T5440 server architecture. White Paper, 2008.

[70] The Apache Software Foundation. Hadoop, `http://hadoop.apache.org`.

[71] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 10th International Conference on Compiler Construction*, 2001.

[72] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[73] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. The SUIF compiler system: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Departments of Electrical Engineering and Computer Science, Stanford University, 1994.

[74] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, 2002.

[75] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.

[76] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, pages 261–272, 2007.