Performance Analysis of Thread Mappings with a Holistic View of the Hardware Resources

Wei Wang, Tanima Dey, Jason Mars, Lingjia Tang, Jack W. Davidson, and Mary Lou Soffa
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
Email: {wwang, td8h, jom5x, lt8f, jwd, soffa}@virginia.edu

***Abstract*—With the shift to chip multiprocessors, managing shared resources has become a critical issue in realizing their full potential. Previous research has shown that thread mapping is a powerful tool for resource management. However, the difficulty of simultaneously managing multiple hardware resources and the varying nature of the workloads have impeded the efficiency of thread mapping algorithms. To overcome the difficulties of simultaneously managing multiple resources with thread mapping, the interaction between various microarchitectural resources and thread characteristics must be well understood.**

This paper presents an in-depth analysis of PARSEC benchmarks running under different thread mappings to investigate the interaction of various thread mappings with microarchitectural resources including, L1 I/D-caches, I/D TLBs, L2 caches, hardware prefetchers, off-chip memory interconnects, branch predictors, memory disambiguation units and the cores. For each resource, the analysis provides guidelines for how to improve its utilization when mapping threads with different characteristics. We also analyze how the relative importance of the resources varies depending on the workloads. Our experiments show that when only memory resources are considered, thread mapping improves an application's performance by as much as 14% over the default Linux scheduler. In contrast, when both memory and processor resources are considered the mapping algorithm achieves performance improvements by as much as 28%. Additionally, we demonstrate that thread mapping should consider L2 caches, prefetchers and off-chip memory interconnects as one resource, and we present a new metric called L2-misses-memory-latency-product (L2MP) for evaluating their aggregated performance impact.

I. Introduction

Compared to traditional uniprocessors, chip multiprocessors (CMPs) greatly improve system throughput by offering computational resources that allow multiple threads to execute in parallel. To realize the full potential of these powerful platforms, efficiently managing the resources that are shared by these simultaneously executing threads has become a critical issue.

In this paper, we focus on managing CMP shared resources through thread mapping. Previous research has shown that thread mapping is a powerful tool for managing resources [6, 8, 17, 22, 26]. However, despite the intensive and extensive research on this topic, properly mapping threads to achieve the optimal performance for an arbitrary workload is still an open question. Finding the optimal thread mapping is extremely difficult because one must consider all relevant resources and the interaction between these many resources is workload dependent. Previous research has shown that L2 caches, front-side-bus and prefetchers have to be considered when managing memory hierarchy resources [29]. However it remains unclear whether there are additional resources that should be considered, and how to *holistically* improve their utilization based on the workload characteristics.

To holistically manage multiple resources with thread mapping, there are three major challenges.

1) The first challenge is to identify the key resources that need to be considered by thread mapping algorithms. Neglecting the key resources would result in suboptimal performance.
2) The second challenge is to determine how to map threads to improve the utilization of each key resource. The best thread mapping also depends on the thread run-time characteristics. For each key resource, we need to identify the related thread run-time characteristics and determine how to map threads when they exhibit these characteristics.
3) The third challenge is to handle situations where no thread mapping can improve the utilization of all key resources. Under such circumstances, thread mapping algorithms must prioritize the resources and focus on improving the utilization of resources that can provide the maximum benefit.

Previous research on thread mapping focused on improving the utilization of the resources within the memory hierarchy [29] or only focus on individual resource [6, 17, 26]. Although the proposed approaches are successful in improving the utilization of these resources, the best application performance is not always guaranteed [28]. Moreover, most previous work has been done using single-threaded workloads, while emerging workloads increasingly include multi-threaded programs. Multi-threaded workloads have different run-time characteristics, thus require different mapping strategies.

As a first step towards overcoming the challenges of holistically managing multiple resources, we provide an in-depth performance analysis of all possible thread mappings for a set of workloads created using applications from the multi-threaded PARSEC benchmark suite [4]. While other work has looked at the memory hierarchy, in this work we take a holistic look at both the memory resources and processor resources (e.g., branch predictors, memory disambiguation unit, etc.), and evaluate their relative importance. In this analysis, we identify the key resources that are responsible for performance differences.

The analysis also determines the thread characteristics related to each key resource, and studies how to map threads with these characteristics to improve the utilization of the key resources. Additionally, to help make trade-off decisions, we analyze the relative importance of the key resources for each workload, and investigate the reason for prioritizing some resources over the others. We observe that, by focusing on multiple resources, proper thread mapping can improve an application's performance by up to 28% over current Linux scheduler, while consideration of only memory resources provides improvement of only 14%.

Specifically, the contributions of this paper include:

1) An in-depth analysis that identifies the key hardware resources that must be considered by thread mapping algorithms, as well as the less important resources that do not need to be considered. Unlike previous work that considered only shared memory resources for mapping single-threaded applications, our paper demonstrates that for multi-threaded applications, thread mapping has to consider more resources, and thread characteristics for better performance.
2) An analysis of how to improve each key resource's utilization with thread mapping when managing threads with different run-time characteristics. To the best of our knowledge, this analysis is the first that investigates the characteristics of multi-threaded workloads and their implications for managing both memory and processor resources with thread mapping.
3) An analysis shows that L2 caches, prefetchers and memory interconnections should be considered as one resource because of their complex interactions. We also propose a new metric L2-misses-memory-latency-product (L2MP) to measure their aggregated performance impact.
4) An analysis that identifies the ranking of the key resources for each workload, and the reason for the ranking.

The remainder of this paper is organized as follows: Section II provides an overview of hardware resources and the thread characteristics considered in our analysis. Section III identifies the key resources for thread mapping algorithms. Section IV analyzes how to improve the utilization of individual resources via thread mapping. Section V discusses using the resource rankings to simultaneously managing multiple resources. Section VI summarizes the performance results. Section VII discusses related work and Section VIII concludes the paper.

II. Performance Analysis Overview

To address the challenges mentioned above, we perform a comprehensive analysis of how an application's performance is effected when threads with various characteristics share multiple hardware resources. This section gives an overview of the resources, the metrics, the run-time characteristics, and the thread mappings that are covered in this analysis.

A. Hardware Resources

We address the resources that are commonly available on current CMP processors. Fig. 1 gives a schematic view of the resources provided by an Intel quad-core processor. The resources we consider can be classified into two categories: the memory hierarchy resources and the processor resources.

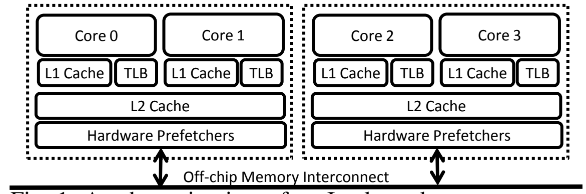

Fig. 1: A schematic view of an Intel quad-core processor.

Memory hierarchy resources include L1 instruction caches (I-cache), L1 data caches (D-cache), instruction and data translation look-aside buffers (I/D TLB), L2 caches, hardware prefetchers and off-chip memory interconnect. We use an Intel Core 2 processor, which has two prefetch mechanisms, Data Prefetch Logic (DPL) and L2 Streaming Prefetch [16]. DPL fetches a stream of instructions and data from memory if a stride memory access pattern is detected. L2 Streaming Prefetch brings two adjacent cache lines into an L2 cache.

Processor resources include the cores and components that require training to function, such as branch predictors and memory disambiguation units. In this paper, we use the term **Resource_Cores** when we discuss the core as a resource.

B. Metrics

The number of cache misses, the amount of memory transactions and memory access latency are used to evaluate the utilization of the memory hierarchy resources. The number of mispredictions and stalled CPU cycles due to these mispredictions are used to evaluate the training-based processor resources.

Processor utilization is used to evaluate the utilization of Resource_Cores. Note that, this processor utilization is viewed from the OS perspective. For example, suppose there is one thread that runs solely on a core. Due to I/O operations or synchronizations, half of the execution time of this thread is staying in the OS waiting queue. Then the core that executes this thread has a processor utilization of only 50%. In this paper, the processor utilization refers to the overall processor utilization of all cores in the system.

We use two metrics to evaluate the performance of the applications: the number of CPU cycles consumed and the execution time. Memory resources and the training-based processor resources impact the total cycles consumed by an application. Accordingly, we use executed CPU cycles to estimate the performance impact of these resources. The execution time, on the other hand, is affected by both processor utilization and the executed cycles. We use execution time when evaluating the performance impact of all the resources. The relation of execution time, executed cycles and processor utilization is described by equation (1).

$$Exec\_Time = \frac{Exec\_Cycles}{Num\_Cores \times Proc\_Util \times Frequency} \quad (1)$$

Num_Cores and $Frequency$ refer to the number of cores and the processor frequency, respectively. Suppose there is an application with four threads running on a quad-core processor of 1GHz. Each of the four thread requires 500 million

PMU Counter	Description
L2_LINES_IN:DEMAND	L2 misses
L2_LINES_IN:PREFETCH	Prefetched cache lines
ITLB_MISSES	ITLB misses
DTLB_MISSES	DTLB misses
PAGE_WALKS:CYCLES	TLB miss penalty
L1D_REPL	L1 D-cache misses
L1I_MISSES	L1 I-cache misses
CYCLES_L1I_MEM_STALLED	L1I miss penalty
BR_MISSP_EXEC	Branch mispredictions
RES_STALLS:BR_MISS_CLR	Br misprediction caused stalls
MEMORY_DISAM:SUCCESS	Success mem-disambiguation
MEMORY_DISAM:RESET	Mem-disambiguation mis-penalty
BUS_TRANS_ANY	Total memory transactions
BUS_REQUEST_OUTSTANDING	Outstanding mem-transactions
BUS_TRANS_BRD	Read memory transactions
BUS_TRANS_IFETCH	Instruction fetch transactions

TABLE I: PMUs used in our analysis.

CPU cycles to execute, and each thread has 50% processor utilization due to I/O operations and synchronizations. Then for this application, its execution time is $(500M(cycles) \times 4(threads))/(4(cores) \times 50\% \times 1GHz) = 1\ second$.

All of the metrics mentioned in this section can be acquired from performance monitoring units (PMUs) [16]. TABLE I gives the name of the PMUs we used in our analysis.

We compute memory access latency from PMUs using the equation (2) proposed by Eranian [13]. Essentially, in equation 2, memory latency is computed by dividing the total cycles of all memory read transactions by the number of memory reads.

$$Mem\_latency = \frac{BUS\_REQUEST\_OUTSTANDING}{BUS\_TRANS\_BRD - BUS\_TRANS\_IFETCH} \quad (2)$$

C. Thread Characteristics

Thread characteristics include the properties of a single thread and the interactions among multiple threads.

For single thread properties, we consider a thread's cache demand, memory bandwidth demand and I/O frequency. Additionally, to describe how threads utilize prefetchers, we introduce three metrics: prefetcher effectiveness, prefetcher excessiveness and prefetch/memory fraction. We define a thread's **prefetcher effectiveness** as the percentage of the L2 cache misses that are reduced when the prefetchers are turned on compared to when they are turned off. We define a thread's **prefetcher excessiveness** as the percentage of the additional cache lines that are brought into the L2 cache when the prefetchers are turned on than when they are turned off. **Prefetch/memory fraction** is defined as the fraction of prefetching transactions in the total memory transactions when prefetchers are on. Prefetcher effectiveness measures how much the application benefits from the prefetcher; prefetcher excessiveness measures how much extra pressure is put on memory bandwidth due to prefetching activity; and prefetch/memory fraction illustrates the overall impact of prefetchers on memory bandwidth.

For multiple thread interactions, we consider data sharing, instruction sharing, and the frequency of synchronization operations. These interactions usually happen among threads from the same application. We call such threads **sibling threads**.

Mapping	Core0	Core1	Core2	Core3
	L2 cache		L2 cache	
OSMap	Any thr.	Any thr.	Any thr.	Any thr.
IsoMap	*a1,a1*	*a1,a1*	*a2,a2*	*a2,a2*
IntMap	*a1,a1*	*a2,a2*	*a1,a1*	*a2,a2*
SprMap	*a1,a2*	*a1,a2*	*a1,a2*	*a1,a2*

TABLE II: Thread mappings used in our analysis. *a1* and *a2* are threads from application 1 and application 2 respectively. Each application has four threads. Threads from the same application are assumed to have similar characteristics. Note that we do not consider SMT there, so when two threads are pinned to one core, they share that core in a time multiplexing manner.

D. Thread Mappings

In our experiments, we examined all possible thread mappings when running two multi-threaded applications each with four threads. Therefore, there are eight threads in total, more threads than cores. Using more threads than cores allows thorough evaluation of the resources, including L1 caches, TLBs, branch predictors, memory disambiguation units and Resource_Cores. Moreover, because real application threads have synchronizations and I/O operations, they can not use all of the cores allocated to them. Thus using more threads than cores can improve the overall processor utilization.

To guide our analysis, we choose four thread mappings that cover all resource sharing configurations (either sibling threads share a resource or non-sibling threads share a resource). All other thread mappings could be viewed as hybrid versions of these four mappings. TABLE II shows the four thread mappings on a quad-core processor running Linux. Except for the OS mapping, all mappings are done by statically pinning threads to cores using the processor affinity system call. How these four thread mappings use resources is described below.

OS Mapping (OSMap): This thread mapping is determined by the Linux scheduler. The OS tries to evenly spread the threads across the cores in the system to ensure fair processor time allocation and low processor idle time. Under this mapping, as long as there is an available core and a runnable thread, that thread is mapped to run on that core. As a result, any thread can run on any core and share the resources associated with these cores. The OSMap is used as the baseline for performance comparison.

Isolation-mapping (IsoMap): Under IsoMap, sibling threads are mapped to run on the two cores that share one L2 cache. In other words, they are isolated on that L2 cache. L1 caches, TLBs, L2 caches, hardware prefetchers and cores are shared by siblings.

Interleaving-mapping (IntMap): Under IntMap, threads from different applications are mapped to the cores in an interleaved fashion. L1 caches, TLBs and cores are still only shared by sibling threads. L2 caches and prefetchers are shared by threads from different applications.

Spreading-mapping (SprMap): Under SprMap, four threads of each application are evenly spread on the four cores. As a result, every core executes two threads which come from two different applications. L1 caches, TLBs, L2 caches and cores are shared by threads from two applications.

Note that, although we assume that sibling threads are identical here, some PARSEC benchmarks have sibling threads

Benchmark	Data sharing	Working set	Bandwidth Req.	Synch. Ops.	I/O time (%)	Pref. Eff.	Pref. Exces.	Perf./Mem Frac.
canneal (CN)	low	2 GB	2.1GB/s	34	0%	6%	49%	44%
facesim (FA)	low	256 MB	3.9GB/s	17K	0.40%	95%	0%	95%
fluidanimate(FL)	low	128 MB	1.5GB/s	17771K	0.14%	67%	42%	57%
streamcluster (SC)	low	256 MB	6.5GB/s	129K	0%	84%	64%	90%
x264 (X2)	low	16 MB	1.3GB/s	17K	10%	-270%	861%	55%
blackscholes (BS)	high	2 MB	40MB/s	8	2%	98%	0%	98%
bodytrack (BT)	high	8 MB	118MB/s	116K	31%	41%	32%	47%
swaptions (SW)	high	512 KB	10KB/s	23	0%	-29%	347%	88%
vips (VP)	high	16 MB	137MB/s	40K	25%	86%	35%	92%

TABLE III: Thread characteristics of PARSEC benchmarks.

with different characteristics. However, we observe that the difference between the sibling threads is negligible. Therefore, thread mappings that only differ in the placement of sibling threads usually have similar performance.

III. Key Resources Identification

This section identifies the key resources for thread mapping algorithms. A resource should satisfy two criteria to be considered as a key resource for thread mapping algorithms:

1) The utilization of this resource varies considerably among different thread mappings.

2) Thread mapping caused utilization variations of this resource result in considerable variations in an application's performance.

Criterion one can easily be determined directly using PMUs. However, the second criterion requires two approaches. Although experimenting on a real machine provides more accurate understanding of thread mappings, the ability to precisely account each resource's performance impact is limited by the types of PMUs available in the hardware. For example, for branch predictors, there are PMUs that count the number of mispredictions, as well as the number of cycles stalled due to these mispredictions. However, for L1 D-cache, there are only PMUs that give the number of L1 D-cache misses. There is no PMU that tells the number of cycles spent on L1 D-cache misses. Therefore, for different resources, different approaches have to be taken:

1) **Direct Approach** For resources that have PMUs to measure their performance impacts, we use the reading from these PMUs directly.

2) **Indirect Approach** For L1 D-caches, L2 caches and off-chip memory interconnects, there are no PMUs to directly measure their performance impact. For these resources, we first verify with PMUs that the performance variations across mappings are caused by memory stalls. Then we compare the performance of the thread mappings. If the application's performance is improved in one mapping, and only one resource's utilization is improved in this mapping, then we can conclude that it is this resource that causes the performance improvement.

A. Experimental Design

To find the key resources, we performed experiments on a real CMP machine. Here we introduce the experimental design. We use PARSEC benchmarks suite version 2.1 (with native input set) to create our workloads because these benchmarks have a large variety of thread characteristics. TABLE III gives the run-time characteristics of PARSEC benchmarks. In Table III, data sharing, working set and synchronization operations are collected with a simulator by the PARSEC authors [4]. The amount of data sharing (high or low) refers to the percentage (high or low) of the cache lines that are shared among sibling threads. The working set here is an architectural concept which means the total size of memory touched by a benchmark. We use working set size to estimate the cache demand of a benchmark. Synchronization operations measures the total number of locks, barriers and conditions executed by a benchmark. All other characteristics are collected on an Intel Q9550 processor. The I/O time is collected by instrumenting the I/O functions. Bandwidth requirement of a benchmark is acquired by dividing the total amount of memory accessed by the execution time of the benchmark. The total amount of memory accessed equals to the total number of memory transactions times the size of each transaction, which is 64 Bytes. Prefetcher effectiveness, prefetcher excessiveness and prefetch/memory fractions are computed following their definitions in Section II-C. The negative values of prefetcher effectiveness for *swaptions* and *x264* suggest that these two benchmarks experience more L2 cache misses when hardware prefetchers are turned on.

Each workload consists of a pair of benchmarks. Therefore, we can compare the mappings where sibling threads share the resources with the mappings where non-sibling threads share the resources. We use nine PARSEC benchmarks and thus there are 36 pairs (workloads) in total. Four benchmarks, *ferret, dedup, freqmine, raytrace*, are not used in our analysis due to compilation errors, configuration errors or execution errors.

For each mapping, each workload is executed until the longest benchmark has finished three runs. Shorter benchmarks are restarted if the longest benchmark has not finished. The average of the results of the first three runs are presented. The variation of the results for the same mapping and workload is very small. For IsoMap, IntMap, and SprMap, the variation is less than 2%. For OSMap, the variation is higher, usually between 2% and 4%. However, since we only use OSMap as a baseline, the higher variation would not affect our conclusions.

All experiments are conducted on a platform that has an Intel quad-core Q9550 processor. Each core of this processor has one 32KB L1 I-cache and one 32KB L1 D-cache. Every two cores share one 6MB L2 cache. (Fig. 1). This platform has 2GB memory and runs Linux kernel 2.6.25. Readings from PMUs are collected with PerfMon2 [12].

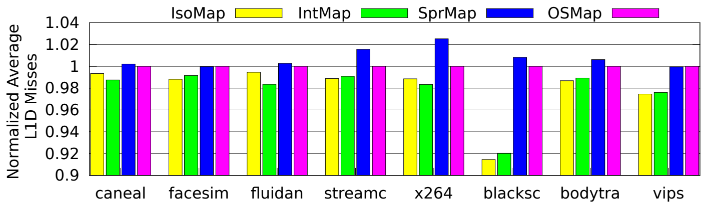

Fig. 2: Average L1 D-cache misses each benchmark experiences under the four mappings. Normalized to OSMap. Result of *swaptions* is not shown here (nor in the following memory resources figures) because it has very few memory accesses.

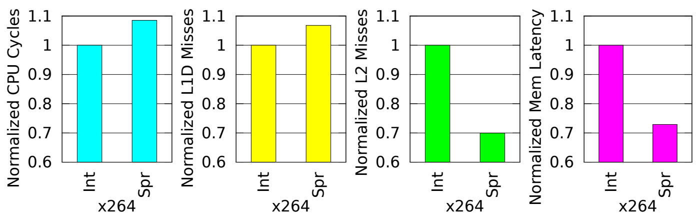

Fig. 3: Comparison of the performance of *x264* running with *streamcluster* under IntMap and SprMap. Lower bar is better. We show only two mappings here to highlight the comparison. Comparing four mappings does not change the conclusion.

B. L1 D-Cache

We first evaluate the importance of L1 D-cache. Fig. 2 shows the normalized average L1 D-cache misses of each benchmark under the four mappings. For each PARSEC benchmark B, there are eight workloads (or pairs) that contain benchmark B. For each mapping, we run the eight workloads, and read the L1 D-cache misses of B. Then we compute the average of the L1 D-cache misses of B for each mapping, and report the results in Fig. 2. We repeat the same process for all benchmarks and all four mappings. As Fig. 2 shows, L1 D-cache misses vary from 2% to 14%, depending on the thread mapping.

We evaluated L1 D-cache's impact on performance with the indirect approach. Fig. 3 shows the CPU cycles and memory resource utilization of *x264* running with *streamcluster* under IntMap and SprMap. Because no other resources' utilization have changed from one mapping to another, only memory resources are shown in the figure. Fig. 3 shows that although IntMap has more L2 misses and higher memory latency, its performance is still better than SprMap due to fewer L1 misses. Therefore, thread mapping induced variation of L1 D-cache misses can cause considerable performance variation.

In conclusion, L1 D-cache misses vary depending on thread mappings. Furthermore, this variation can cause considerable performance variation. Consequently, L1 D-Cache should be considered as a key resource.

C. L2 Cache, Hardware Prefetchers and Off-chip Memory Interconnect

Previous research has demonstrated that thread mapping can significantly impact the utilization of L2 caches, hardware prefetchers and off-chip memory interconnect, and consequently impact application performance [6, 17, 22, 26, 29]. Thus, these resources should be considered as key resources. Results of our experiments corroborate this conclusion. However, unlike previous research, our experiment results show that these memory resources are better viewed as one resource by thread mapping algorithm, and we provide a new metric called L2MP for evaluating their aggregated impact. The detailed discussion on this subject can be found in Section V-A.

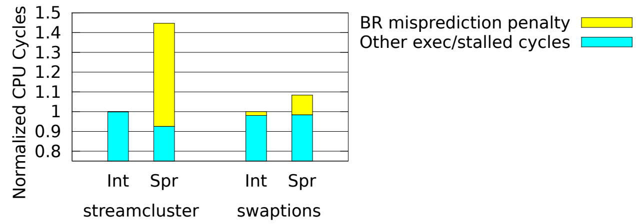

Fig. 4: Comparison of the performance of *streamcluster* running with *swaptions* under IntMap and SprMap. Lower bar is better.

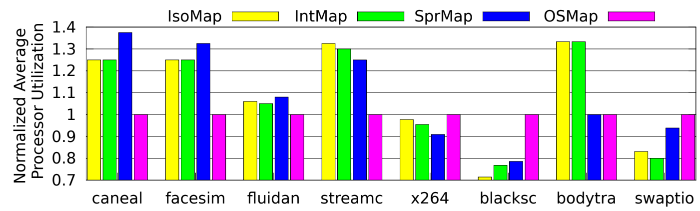

Fig. 5: Average processor utilization of each benchmark running under the for mappings. Normalized to OSMap.

D. Branch Predictors

In our experiments, we observe that one thread mapping could have 15 times more branch mispredictions than another mapping, which suggests that branch predictors' mispredications vary significantly depending on thread mappings.

We evaluate the performance impact of branch predictors with the direct approach. Fig. 4 shows the performance of *streamcluster* and *swaptions* running together. *Streamcluster* consumes 48% more CPU cycles under the SprMap than the IntMap, and *swaptions* consumes 8% more CPU cycles under SprMap. Fig. 4 also shows that 99% of the increased CPU cycles are caused by branch mispredictions. Therefore, the variation of branch mispredictions can produce considerable application performance variation.

In conclusion, branch mispredictions vary depending on thread mappings. Furthermore, this variation can cause considerable performance variation. Consequently, branch predictors should be considered as a key resource.

E. L1 I-cache, I/D TLBs and Memory Disambiguation Units

Different thread mappings have a great impact on the utilization of L1 I-caches, I/D TLBs and memory disambiguation units. One thread mapping can have more than ten times more misses/mispredictions from these resources than another mapping. Yet the absolute amount of time spent in serving these extra misses/mispredictions (acquired from PMUs directly) accounts for less than 2% (in most cases less than 0.5%) of the total execution time. Therefore, we conclude that these resources should receive low priority when mapping threads of the PARSEC benchmarks.

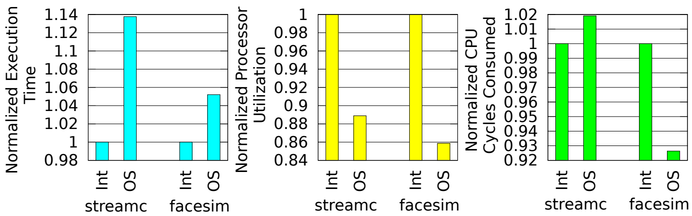

Fig. 6: Comparison of the performance of *streamcluster* running with *facesim* under IntMap and OS mapping. For execution time and cycles consumed, the lower bar is better. **For processor utilization, higher bar is better.**

Key resources	L1 D-caches, L2 caches, prefetchers, off-chip memory interconnect, branch predictors and Resource_Cores
Less important resources	L1 I-cache, I-TLBs, D-TLBs, and memory disambiguation units

TABLE IV: Key resources and less importance resources.

F. Resource_Cores

Fig. 5 gives the average processor utilization of each benchmark under the four mappings. It shows that processor utilization varies significantly across thread mappings.

We evaluate the Resource_Cores's performance impact using the direct approach. Equation (1) in Section II-B shows that any improvement in the processor utilization yields an improvement in execution time. Therefore, the variation of the processor utilization caused by thread mapping can produce considerable application performance variation.

Fig. 6 shows the performance of the pair of *streamcluster* and *facesim* running under IntMap and OSMap. *Streamcluster* consumes 2% fewer CPU cycles under IntMap. *Facesim* consumes 8% more cycles under IntMap. However, both benchmarks execute at least 5% faster under IntMap than OSMap. The improvement of execution time for both benchmarks are primarily due to increased processor utilization.

In conclusion, processor utilization varies depending on thread mappings. Furthermore, this variation can cause considerable performance variation. Consequently, Resource_Cores should be considered as a key resource.

G. Summary

TABLE IV summarize the importance of the resources discussed in this section.

IV. Resource Thread Mapping Guidelines

Having identified the important resources, this section provides guidelines for mapping threads to improve the utilization of individual resources when managing threads with different run-time characteristics.

A. L1 D-cache

Fig. 7 shows the maximum and minimum L1 D-cache misses of each benchmark under the four mappings. For all benchmarks examined, IsoMap and IntMap produce lowest L1 D-cache misses, which suggests that it is better to share L1 D-cache among sibling threads that exhibit the characteristics of data sharing.

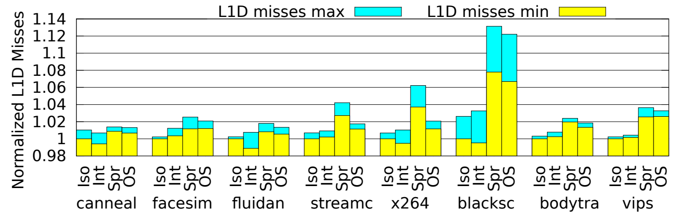

Fig. 7: Maximum and minimum L1 D-cache misses each benchmark experiences under the four mappings. Normalized to IsoMap minimum. Lower bar is better.

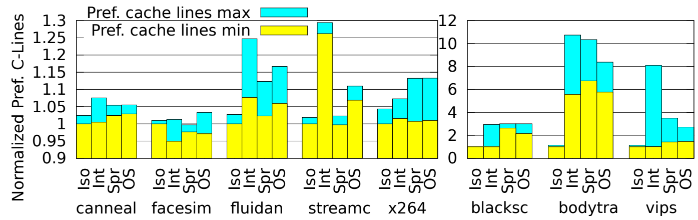

Fig. 8: Maximum and minimum numbers of prefetched cache lines of each benchmark when running under the four mappings.

B. L2 Cache

Our experimental results regarding L2 caches corroborate previous studies. For threads that have high amounts of shared data, mapping them to the same L2 cache can reduce L2 misses [22]; for threads with low or no data sharing, it is best to avoid mapping threads with high cache demands to the same L2 cache [17]. Because L2 cache-aware thread mapping is well studied, we do not discuss L2 caches further.

C. Prefetchers

There are three cases that must be considered when mapping threads to improve prefetcher performance.

Case 1: For threads that share high amounts of data, mapping those threads to share the same prefetcher results in fewer memory prefetches because the L2 caches are also shared. Fewer memory prefetches translate into a lower memory latency because there are fewer memory transactions waiting in the queue. The overall effect is better performance. For example, *Bodytrack*, *vips* and *blackscholes* have high amounts of data sharing. In Fig. 8, these three benchmarks have the fewest prefetched cache lines under IsoMap. The performance results also confirm that the IsoMap yields the best performance. Under IsoMap, these three benchmarks consumes as much as 11% fewer cycles than under other thread mappings.

Case 2: For a benchmark that has low or no data sharing but high prefetcher excessiveness, mapping its threads to share

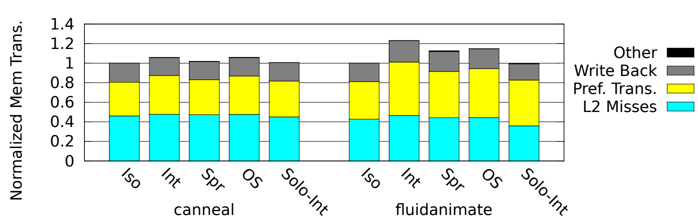

Fig. 9: Breakdown of the memory transactions of workload *canneal_fluidanimate*. The last mapping "Solo-Int" represents running each benchmark solo (without co-runner) using IntMap.

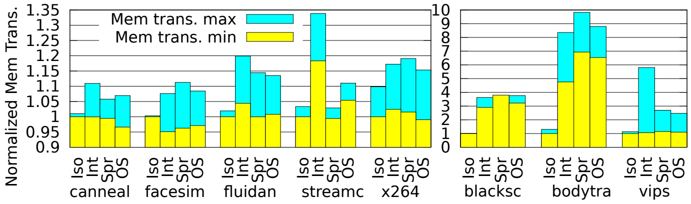

Fig. 10: Maximum and minimum memory transactions each benchmark generates under the four mapping. Lower bar is better.

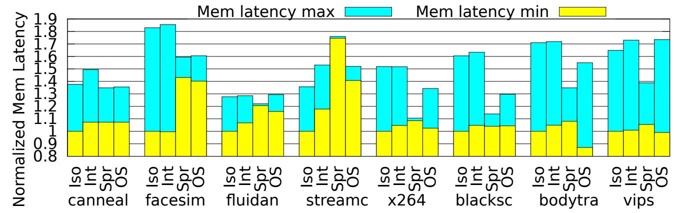

Fig. 11: Maximum and minimum memory latency each benchmark experiences under the four mappings. Lower bar is better.

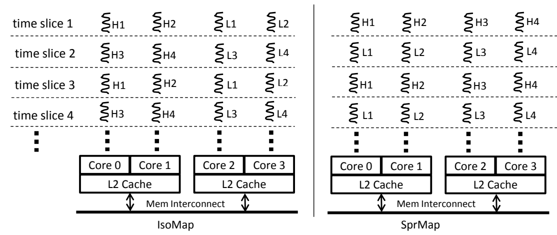

Fig. 12: SprMap tends to have four threads of the same benchmark to run simultaneously, whereas IsoMap (and IntMap) have threads from different applications to share the memory interconnect all the time. H1-H4 are the threads of a high-bandwidth application, and L1-L4 are the threads of a low-bandwidth application.

the same prefetchers results in fewer memory prefetches. For example, *streamcluster*, *canneal*, *fluidanimate* and *x264* have low data sharing but high prefetcher excessiveness. In Fig. 8, these benchmarks have fewer prefetching memory transactions in IsoMap. The performance results also show that IsoMap yields the best performance. Under IsoMap, these four benchmarks consumes as much as 14% fewer cycles than under other thread mappings from improved memory utilization.

Fig. 9 further illustrates this phenomenon with total memory transactions of the pair of *canneal* and *fluidanimate*. Both benchmarks have the fewest prefetching transactions (cache lines) using the IsoMap. Fig. 9 also shows that, when running under IsoMap, the prefetching transactions are even fewer than when they are running alone using IntMap ("Solo-Int" in the figure). One possible explanation of this phenomenon is that the prefetchers fetch the same cache lines for the sibling threads. Therefore, these cache lines can be fetched only once when prefetchers and L2 caches are shared. As this phenomenon only happens for high prefetcher-excessive threads, we suspect that these cache lines are not actually needed, but rather mispredicted to be useful by the prefetchers. However, since we could not find detail information of Intel's prefetching algorithm, we cannot confirm this explanation.

Case 3: For threads with low data sharing and low prefetcher excessiveness, the number of memory prefetches have a strong correlation with the number of L2 cache misses, and fewer prefetches and L2 cache misses both benefit performance. For these threads, thread mappings that reduce the L2 cache misses can also reduce prefetched cache lines. *Facesim* is a PARSEC benchmark that has these characteristics. For *facesim*, our results show that the thread mapping that produces the fewest L2 cache misses also produces fewest memory prefetches and best performance.

D. *Off-chip memory interconnect*

Here, we discuss how to map threads to reduce total memory transactions and memory latency.

1) Total Memory Transactions: Most memory transactions of the PARSEC benchmarks are from L2 misses and prefetches. Therefore, thread mappings that can reduce L2 cache misses and prefetched cache lines would also reduce the total memory transactions. Similar to managing prefetchers, there are three cases to consider.

Case 1: For threads that have high data sharing, IsoMap produces fewest L2 cache misses and prefetching transactions, and hence, fewest memory transactions. For the three benchmarks (*bodytrack*, *vips* and *blackscholes*) that have these characteristics, Fig. 10 shows that they have fewest memory transactions under IsoMap.

Case 2: For threads that have low or no data sharing and high prefetcher excessiveness, IsoMap has the fewest prefetching transactions, with no or slightly increased L2 cache misses. Accordingly, for these threads, IsoMap also has the fewest memory transactions. For the four benchmarks (*streamcluster*, *canneal*, *fluidanimate* and *x264*) that have these characteristics, Fig. 10 shows that they have fewest memory transactions under IsoMap.

Case 3: For threads that have low data sharing and low prefetcher excessiveness, thread mappings that have the fewest L2 misses also have the fewest prefetched cache lines and memory transactions (as discussed in Sections IV-C). For *facesim*, our results (Fig. 10) show that the thread mapping that produces the fewest L2 misses also produces fewest memory transactions.

2) Memory Latency: Fig. 11 shows the maximum and minimum memory latencies of each benchmark under the four mappings. Memory latencies depend on two factors: (1) whether high-bandwidth-demand threads are sharing the memory interconnect, and (2) the number of total memory transactions.

High-bandwidth-demand threads sharing the memory interconnect results in high queuing delays which prolong the memory latency. Fig. 12 depicts how IsoMap and SprMap use the memory interconnect. For SprMap, most of the time, it has four sibling threads running simultaneously, which also means four sibling threads sharing memory interconnect. IsoMap and IntMap always have threads from different applications running simultaneously and share the memory interconnect. For an application A, if it is running with a relatively low-bandwidth-demand application, IsoMap and IntMap are better than SprMap, because they avoid sharing the memory intercon-

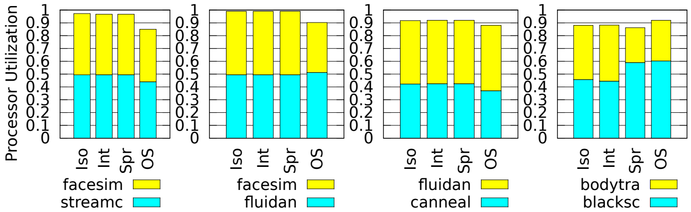

Fig. 13: Processor utilization (total and breakdown) of four pairs of PARSEC benchmarks.

nect by only A's threads. If A is running with a relatively high-bandwidth-demand application, SprMap is the best, because A's threads do not share the memory interconnect with the other application's high-bandwidth-demand threads. In Fig. 11, every benchmark has the lowest minimum latency under IsoMap. The minimum latency happens when a benchmark runs with another relatively light-bandwidth-demand benchmark, where IsoMap is better than SprMap as we explained. Meanwhile, every benchmark has the lowest maximum latency under SprMap. Maximum latency happens when a benchmark runs with another relatively heavy-bandwidth-demand benchmark, in which case SprMap is the best.

Furthermore, reducing memory transactions can also reduce memory latency. In Fig 11, IsoMap has the lower memory latencies than IntMap because most benchmarks generate fewer memory transactions under IsoMap.

E. Branch Predictors

Our experiment results show that IsoMap and IntMap have lower branch mispredictions than SprMap and OSMap. IsoMap and IntMap map sibling threads to share branch predictors. Because sibling threads usually execute the same pieces of code, they may also share branches. Therefore, the branch execution history of one thread can help its siblings have fewer mispredictions.

F. Resource_Cores

Fig. 13 shows the processor utilization of four workloads running under the four mappings. Compared to the OSMap, IsoMap, IntMap and SprMap have up to 12% better processor utilization for the first three workloads in Fig. 13.

To investigate the reason that OS has lower processor utilization, we acquired Linux scheduling history with kernel profiler. We discovered that the reason for the low processor utilization of OSMap is the synchronizations. Under OSMap, any thread can run on any core, which creates contention for the CPU time. The consequence of this contention is that some threads get delayed because they do not get enough CPU time to execute. Many multi-threaded programs use barriers frequently to synchronize their threads (such as the benchmarks used in the first three workloads of Fig. 13). If one thread get delayed, it becomes a bottleneck and all its sibling threads have to wait for it. Therefore, there are not enough runnable threads in the system, and the processor utilization drops. For the first three workloads of Fig. 13, we observed approximately a 100% increase in waiting time on synchronizations for every thread under OS mapping. The other three mappings, on the other hand, successfully eliminate possible bottlenecks by guaranteeing each thread a fair share of processor time with thread pinning, and thus improve processor utilization.

On the other hand, OSMap has the best processor utilization for workloads with frequent I/O operations (as illustrated by the last workload in Fig. 13). An application with frequent I/O-operations spends much time waiting for I/O completion. Consequently, it may not be able to fully utilize the cores to which its threads are pinned. When mapping threads with frequent I/O operations, allowing threads to execute on any free core has the best processor utilization.

G. Summary

In summary, there are seven thread characteristics that should be considered when mapping threads, which are data sharing, prefetcher excessiveness, bandwidth demand, cache demand, instructions sharing, synchronization frequency and I/O frequency. The thread mapping guidelines for improving each important resource as summarized as follows.

L1 D-cache When threads have data sharing, mapping them to the same L1 D-cache can reduce L1 D-cache misses.

L2 cache For threads with high data sharing, mapping them to the same L2 cache can reduce L2 misses. For threads with low or no data sharing, it is better to avoid putting threads with high cache demand on the same L2 cache.

Prefetchers For threads with high data sharing, mapping them to share the same prefetchers can reduce prefetching transactions. For thread with low or no data sharing, and high prefetcher excessiveness, mapping them to the same prefetchers can reduce prefetching transactions. For threads with low or no data sharing, and low prefetcher excessiveness, thread mappings that can reduce L2 cache misses can also reduce prefetching transactions.

Off-chip memory interconnect For lower memory latency, high-bandwidth-demand threads should not be mapped to use the same memory interconnect at the same time. Furthermore, thread mappings that reduce total memory transactions can help improve memory latency. Thread mappings that can reduce the L2 cache misses and prefetching transactions can also reduce the memory transactions.

Branch Predictors Sharing branch predictors between sibling threads can reduce branch mispredictions.

Resource_Cores For threads with frequent synchronizations, it is better to restrict the cores that these threads can use with thread pinning. For threads that have frequent I/O operations, it is better to let them to use any free cores like the OSMap.

V. Managing Multiple Resources

Thus far, we have discussed each resource individually. However, in a real system, thread mapping algorithms have to consider all the key resources altogether because they are all influencing the application's performance. We observe that there are many cases where no thread mapping can improve every resource at the same time. Under such circumstances, thread mapping algorithms have to improve the utilization of the resources that can provide the maximum benefits. This section provides an analysis about the relative importance of

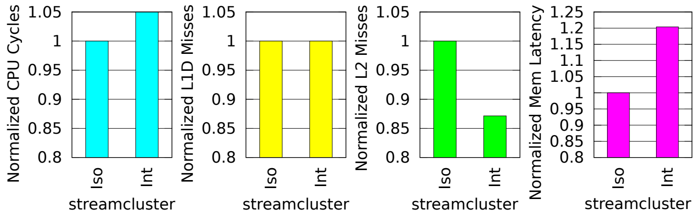

Fig. 14: Comparison of the performance of *streamcluster* running with *blackscholes* under IsoMap and IntMap. Lower bar is better.

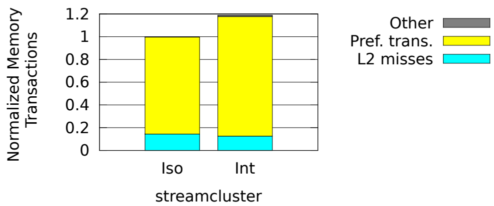

Fig. 15: Breakdown of the memory transactions generated by *streamcluster* when running with *blackscholes* under IsoMap and IntMap.

the resources. It also gives a new metric, called L2MP, for evaluating the aggregated performance impact of L2 caches, prefetchers and off-chip memory interconnects.

A. L2 Caches, Prefetchers and Off-chip memory interconnect

First, we discuss the relative importance of L2 caches, the off-chip memory interconnect and prefetchers. Fig. 14 gives an example that improving the utilization of L2 caches and off-chip memory interconnect are two conflicting goals. Fig. 14 shows the performance and utilization of memory resources of *streamcluster* when running with *blackscholes*. The figure shows that *streamcluster* has a lower memory latency under IsoMap, but it has fewer L2 misses under IntMap. For overall performance, *streamcluster* executes faster under IsoMap than IntMap despite that it has more L2 misses.

The cause of this conflict is the prefetcher. Fig. 15 shows the breakdown of the memory transactions generated by *streamcluster* under IsoMap and IntMap. As discussed in Section IV-C, IsoMap can reduce the prefetcher excessiveness of *streamcluster*. Therefore, when running under IsoMap, *streamcluster* generates fewer prefetching transactions, and fewer memory transactions. Since there are fewer memory transactions to process, the memory latency is reduced. However, mapping these threads to the same prefetchers also means mapping them to the same L2 cache. Because *streamcluster*'s threads have large cache demands and low data sharing, they contend for the cache space and experience more cache misses. However, since the majority of the memory transactions are from the prefetchers, the total memory transactions still drop under IsoMap, which leads to lower memory latency and better application performance.

Although memory latency and prefetchers are more important for *streamcluster*, for some benchmarks, such as *facesim*, L2 caches are more important. It is very hard to argue the relative importance of these three resources because their performances are closely related. The majority of memory transactions are due to L2 misses and to prefetches. Therefore, memory interconnect performance is affected by the performance of

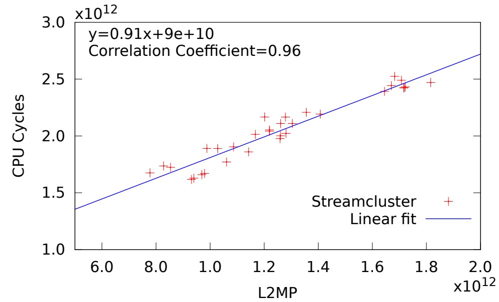

Fig. 16: Correlation between L2MP and executed cycles for *streamcluster*.

L2 caches and prefetchers. Moreover, the cycles stalled due to L2 misses depend on both the number of L2 misses and the memory latency. More L2 misses may not produce more stalled cycles if the memory latency is reduced. For these reasons, thread mapping algorithms should use a single aggregate metric to encapsulate the L2 caches, prefetchers, and memory interconnect as a unified resource.

To evaluate the aggregated performance impact of these three resources, we propose a new metric **L2-misses-Memory-latency-product (L2MP)**, which is defined as the product of L2 misses and memory latency. Because prefetchers impact the application's performance through L2 misses and memory latency, this metric also implicitly considers prefetchers. For memory-intensive applications, L2MP has a strong correlation with the application's performance. Fig. 16 shows the correlation between L2MP and the total cycles of *streamcluster*. Fig. 16 has 32 points. Each point represents the L2MP and total execution cycles of *streamcluster* when it runs with one other benchmark under one of the four mappings. Since there are eight workloads (pairs) that has *streamclsuter*, and there are four mappings, the total number of points is $8 \times 4 = 32$. Fig. 16 shows that correlation coefficient of L2MP and total execution cycles is 0.96, suggesting a strong correlation. Other memory-intensive benchmarks, *canneal*, *fluidanimate* and *facesim*, the correlation coefficients are 0.94, 0.95 and 0.93 respectively. Furthermore, as both L2 misses and memory latencies can be acquired on-line from PMUs, L2MP can be used by on-line thread mapping algorithms. In the following sections, we refer the group of L2 caches, prefetchers and memory interconnect as **L2-Prefetcher-Mem**.

B. L1 D-cache and L2-Prefetcher-Mem

First we discuss the relative importance of L1 D-cache and L2-Prefetcher-Mem. For applications with high bandwidth demands, L2-Prefetcher-Mem is more important than L1 D-caches. Nonetheless, for applications which have high L1 D-cache hit ratio and low L2 misses, L1 D-caches are more important. One example is *x264*. 94% of *x264*'s total memory accesses hit in L1D cache, and only 0.2% of them cause L2 misses. Thus *x264* is more sensitive the performance of L1 D-cache, as illustrated by Fig. 3.

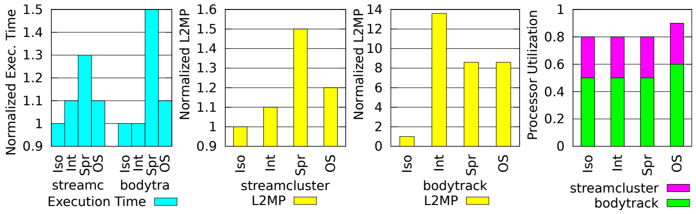

Fig. 17: The performance of *bodytrack* and *streamcluster* running together. For execution time and L2MP, lower bar is better. For processor utilization, high bar is better.

	SC	CN	FL	FA	SW	BS	VP	X2	BT
SC		LM	LM,P	LM,P	B,LM	LM	LM	LM	LM
CN	LM,P		LM,P	LM,P	LM,P	LM,P	P	P	P
FL	LM,P	LM,P		LM,P	LM,P	LM,P	P	P	P
FA	LM,P	LM,P	LM,P		LM,P	LM,P	P	P	P
SW	B	P	P	P		P	P	P	P
BS	P	P	P	P	P		P	P	P
VP	P,LM	P	P	P	P	P		P	P
X2	P,L1	P,L1	P,L1	P,L1	P,L1	P,L1	P		P
BT	P	P	P	P	P	P	P	P	

TABLE V: The most important resources for each of the 36 pairs of benchmarks we examined. "P" refers to Resource_Cores, "B" refers to branch predictors, "LM" refers to L2-prefetcher-Mem, "L1" refers to L1 D-cache. Empty cells represent workloads that created using only one benchmark that we did not examine. Two resources in a cell means that both resources can be improved simultaneously with thread mapping.

C. Resource_Cores and L2-Prefetcher-Mem

In this section we evaluate the relative importance of Resource_Cores and L2-Prefetcher-Mem. For threads that have extremely high bandwidth requirements, such as *streamcluster*, L2-Prefetcher-Mem trumps Resource_Cores. Fig. 17 shows the performance of *bodytrack* and *streamcluster* running together. For better processor utilization, the OSMap allows *streamcluster* to run on all four cores since *bodytrack* can only use 30% of the processor time due to its frequent I/O operations. Accordingly, 3 or 4 *streamcluster*'s high bandwidth threads run simultaneously, which increases the memory latency. On the contrary, IsoMap and IntMap allocate only two cores to *streamcluster* which avoids stressing the memory interconnect with high-bandwidth threads running simultaneously. However, these two mappings suffer from low processor utilization because *bodytrack* cannot fully utilize the two cores allocated to it. When all resources are considered, IsoMap and IntMap have better overall performance.

However, if the workloads contain no extremely high bandwidth requirement applications, it more important to improve the utilization of Resource_Cores than L2-Prefetcher-Mem.

D. Relative importance of all key resources

Although we only discussed three groups of resources in this section, the trade-off between resources actually happens among all resources. TABLE V summarizes the most important resources for the 36 pairs of benchmarks we examined. Cell(row, col) = R in the table represents that when benchmark row runs with benchmark col, mapping threads to improve resources R can provide the maximum performance gains for benchmark row. TABLE V shows that the relative importance of the key resources varies depending on the benchmark characteristics. Even for the same benchmark, the most important resources vary depends on its co-runners.

E. Summary

In summary, a thread's most important resources depend on its own characteristics, as well as on its co-runners'. This observation implies that, not only do thread mapping algorithms have to adapt their mapping decisions to the workloads when improving the utilization of one resource, but also have to adapt their mapping decisions to the workloads when determining the most important resources to improve.

VI. Performance Results

When L2-prefetcher-Mem or L1 D-caches or branch predictors are the most important, either IsoMap or IntMap performs best. When only Resource_Cores is the most important, OSMap usually performs best. Among the 36 workloads, we can improve the execution time of 29 workloads over the OSMap. The average improvement of the execution time for all 36 workloads over OSMap is 4%. For the 26 workloads that contains the four memory-intensive and synchronization-intensive benchmarks, *streamcluster*, *canneal*, *fluidanimate* and *facesim*, the average improvement is 8%.

The workload *streamcluster* and *fluidanimate* has the highest performance improvement over OSMap. Under IsoMap, *streamcluster*'s total cycles are reduced by 14% through reduced prefeteched cache lines and memory latency. And its processor utilization is improved by 20%. Enjoying improvement from both types of resources, the execution time of *streamcluster* is improved by 28%. The execution time of *fluidanimate* is also improved by 2% under IsoMap.

VII. Related Work

There has been prior work on performance analysis and resource management in CMPs via thread mapping. Zhuravlev et al. conducted various performance analyses for memory hierarchy resources and provided several scheduling algorithms [28, 29]. Dey et al. described a methodology to analyze an application's performance when the application threads share the memory hierarchy resources [9]. Mars et al. synthesized last-level cache sharing by thread mapping and analyzed cross-core performance interference on two architectures [19]. Tang et al. studied the impact of memory subsystem resource sharing on data center applications [23]. Chandra et al. and Xie and Loh used thread mapping to address cache contention [6, 26]. Knauerhase et al. developed an idea to evenly spread the threads by mapping to mitigate shared cache contention [17]. Several studies investigated mapping threads based on cache line sharing to reduce cross-chip cache accesses [8, 22]. Snavely et al. proposed the method to find the best thread mapping by sampling all possible thread mapping configurations on a hyper-threaded processor [21]. Most of these works mainly provide performance analysis for shared resources in the memory hierarchy, whereas we examine the usage of processor resources as well as the shared-memory resources. Moreover, most of previous work used single-threaded workloads. In this paper, we examine multi-threaded workloads which require different thread mapping strategies. Teng et al. studied the migration cost of multi-threaded Java applications [24]. Their results show that for real Java applications, the migration overhead is very small.

There also has been prior work on the PARSEC's characterization. The authors of PARSEC characterized the benchmarks from working set, locality, cache utilization, off-chip traffic, programming models and the scaling trend of the inputs [4]. Barrow-Williams et al. described communication characteristics among the PARSEC benchmarks' threads [1]. Bhaduria et al. described thread scalability and micro-architectural design choices for the benchmarks over a wide variety of real machines along with cache performance, sensitivity with respect to DRAM speed and bandwidth [2]. Lakshminarayana and Kim characterized and categorized PARSEC benchmarks into three classes based on execution time variability [18]. Zhang et al. analyzed the data sharing of PARSEC benchmarks [27].

Several studies investigate the performance impact of the hardware resources and how to improve their designs. Cain et al. studied prefetchers on IBM Power architectures [5]. New designs are proposed to address the over-aggressiveness of current prefetchers [10, 11, 15]. Hsu et al. studied three shared cache partitioning policies for different objectives [14]. Bhattacharjee et al. studied the implication of current TLB design for PARSEC benchmarks [3]. Wu et al. investigated the importance of prefetchers and TLBs in terms of intra-application contention [25]. Simultaneously managing multiple resources have been studied before using approaches other than thread mappings [7, 20]. While we focus on the implications of these hardware resources in terms of thread mappings, the solutions and insights from these studies could be used in conjunction with our findings on thread mappings to further improve performance.

VIII. Conclusion

To overcome the challenges of simultaneously managing multiple hardware resources with thread mapping, this paper provides an in-depth performance analysis of different thread mappings. The analysis suggests that thread mapping algorithms should give high priority to improve the key resources, which are L1 D-caches, L2 caches, hardware prefetchers, off-chip memory interconnect, branch predictors and Resource_Cores. Mapping threads to improve the utilization of these resources can significantly improve the application's performance. The analysis also concludes that thread mapping algorithms should consider more thread characteristics than cache demands and data sharing, which include prefetcher excessiveness, memory bandwidth requirements, instruction sharing, synchronization frequency and I/O frequency. We also analyzed how the relative importance of the key resources varies depending on the workload characteristics. By managing both memory and processor resources based on priority and workload characteristics, thread mapping can improve an application's performance by up to 28% over contemporary Linux scheduler, while considering only memory resources only provides performance improvement by up to 14%.

Acknowledgment

This research is supported by National Science Foundation grant number CCF-0811689. We appreciate the insightful comments and constructive suggestions from the anonymous reviewers. We would also like to thank Ryan W. Moore, Mahmut Aktasoglu, Bruce R. Childers, Mary Jane Irwin and Mahmut Kandemir for their valuable input.

References

[1] N. Barrow-Williams, C. Fensch, and S. Moore. A Communication Characterization of SPLASH-2 and PARSEC. In *Int'l Symp. on Workload Characterization (IISWC)*, 2009.

[2] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Int'l Symp. on Workload Characterization (IISWC)*, 2009.

[3] A. Bhattacharjee and M. Martonosi. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] H. Cain and P. Nagpurkar. Runahead execution vs. conventional data prefetching in the IBM POWER6 microprocessor. In *Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-thread Cache Contention on a Chip Multi-Processor Architecture. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2005.

[7] J. Chen and L. K. John. Predictive Coordination of Multiple On-Chip Resources for Chip Multiprocessors. In *Proc. of the Int'l Conf. on Supercomputing (ISC)*, 2011.

[8] 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 *ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, 2007.

[9] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *Int'l Symp. on Performance Analysis of Systems Software (ISPASS)*, 2011.

[10] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Int'l. Symp. on Microarchitecture (MICRO)*, 2009.

[11] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[12] S. Eranian. Perfmon2: A flexible performance monitoring interface for linux. In *Linux Symp.*, 2006.

[13] S. Eranian. What can performance counters do for memory subsystem analysis? In *Proc. of the 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness*, MSPC '08, 2008.

[14] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on

CMPs: caches as a shared resource. In *Proc. of the 15th Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2006.

[15] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *Int'l. Symp. on Microarchitecture (MICRO)*, 2006.

[16] Intel. Intel 64 and IA-32 Architecture Software Developer's Manual, 2009.

[17] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[18] N. Lakshminarayana and H. Kim. Understanding performance, power and energy behavior in asymmetric multiprocessors. In *IEEE Int'l Conf. on Computer Design*, 2008.

[19] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross core interference through contention synthesis. In *Int'l Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2011.

[20] J. F. Martinez and E. Ipek. Dynamic multicore resource management: A machine learning approach. *IEEE Micro*, 2009.

[21] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[22] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *European Conf. on Computer Systems (EuroSys)*, 2007.

[23] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA '11: Proc. of the 37th annual Int'l Symp. on Computer Architecture*, 2011.

[24] Q. Teng, P. Sweeney, and E. Duesterwald. Understanding the cost of thread migration for multi-threaded Java applications running on a multicore platform. In *Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2009.

[25] C.-J. Wu and M. Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems and Softwarea (ISPASS)*, 2011.

[26] Y. Xie and G. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *In Proc. of Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[27] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proc. of the 15th Symp. on Principles and practice of parallel programming*, 2010.

[28] S. Zhuravlev, S. Blagodurov, and A. Fedorova. AKULA: A toolset for experimenting and developing thread placement algorithms on multicore systems. In *Parallel Architecture and Compilation Techniques (PACT)*, 2010.

[29] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.