Scheduling Threads for Constructive Cache Sharing on CMPs

Shimin Chen[†] Phillip B. Gibbons[†] Michael Kozuch[†] Vasileios Liaskovitis[*]
Anastassia Ailamaki[*] Guy E. Blelloch[*] Babak Falsafi[*] Limor Fix[†]
Nikos Hardavellas[*] Todd C. Mowry[*,†] Chris Wilkerson[‡]

[*]Carnegie Mellon University [†]Intel Research Pittsburgh [‡]Intel Microprocessor Research Lab

ABSTRACT

In chip multiprocessors (CMPs), limiting the number of off-chip cache misses is crucial for good performance. Many multithreaded programs provide opportunities for *constructive* cache sharing, in which concurrently scheduled threads share a largely overlapping working set. In this paper, we compare the performance of two state-of-the-art schedulers proposed for fine-grained multithreaded programs: Parallel Depth First (PDF), which is specifically designed for constructive cache sharing, and Work Stealing (WS), which is a more traditional design. Our experimental results indicate that PDF scheduling yields a 1.3–1.6X performance improvement relative to WS for several fine-grain parallel benchmarks on projected future CMP configurations; we also report several issues that may limit the advantage of PDF in certain applications. These results also indicate that PDF more effectively utilizes off-chip bandwidth, making it possible to trade-off on-chip cache for a larger number of cores. Moreover, we find that task granularity plays a key role in cache performance. Therefore, we present an automatic approach for selecting effective grain sizes, based on a new working set profiling algorithm that is an order of magnitude faster than previous approaches. This is the first paper demonstrating the effectiveness of PDF on real benchmarks, providing a direct comparison between PDF and WS, revealing the limiting factors for PDF in practice, and presenting an approach for overcoming these factors.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*threads, scheduling*

General Terms

Algorithms, Measurement, Performance

Keywords

Chip Multiprocessors, Scheduling Algorithms, Constructive Cache Sharing, Parallel Depth First, Work Stealing, Thread Granularity, Working Set Profiling

SPAA'07, June 9–11, 2007, San Diego, California, USA.
Copyright 2007 ACM 978-1-59593-667-7/07/0006 ...$5.00.

1. INTRODUCTION

Chip multiprocessors (CMPs) are emerging as the design of choice for harnessing performance from future multi-billion transistor chips. All the major chip manufacturers have made the paradigm shift to focus on producing chips with multiple cores. Unlike wide-issue single-core designs that have run into power, thermal flux, and instruction-level parallelism (ILP) limitations, CMPs allow for both performance and power scalability across future-generation semiconductor fabrication processes. Projections indicate that by 2015, there will be 64 to 128 cores integrated into a single chip [11].

To effectively exploit available parallelism, CMPs must address contention for shared resources [18, 42]. In particular, CMPs share two precious hardware resources among the cores: (i) on-chip memory, and (ii) pin bandwidth. CMPs are limited by a fixed chip area budget that is divided mainly between processing cores and memory (i.e. cache). Consequently, techniques which improve the efficiency of cache resources enable processor architects to devote less chip area to caches and more to processing cores, which, in turn, enables the CMP to exploit more parallelism. Similarly, in CMPs, re-use of data cached on-chip is especially important in reducing contention for the limited memory bandwidth which must be shared among a number of processing cores— with a continued increase in the processor/memory performance gap, the resulting off-chip accesses incur higher penalties, making performance increasingly sensitive to effective on-chip caching.

Many multithreaded programs provide opportunities for *constructive* cache sharing, where concurrently scheduled threads share a largely overlapping working set, as opposed to *destructive* competition for cache spaces among threads. In this paper, we evaluate the impact of thread scheduling algorithms on on-chip cache sharing for multithreaded programs. Many parallel programming languages and runtime systems use greedy thread scheduling algorithms to maximize processor utilization and throughput. We compare the performance of two state-of-the-art greedy schedulers: Parallel Depth First (PDF) [5, 6], a recently proposed scheduler designed for constructive cache sharing, and Work Stealing (WS), a popular scheduler that takes a more traditional approach. Analytical bounds [5, 10, 8] imply that PDF *should* outperform WS on CMPs with shared caches—however, there has been no direct comparison on real benchmarks. We study a variety of benchmark programs on projected future CMPs through simulations. Our experimental results show that:

- For several application classes, PDF enables significant constructive cache sharing among threads, thus improving performance and reducing off-chip traffic compared to WS. In particular, PDF provides a performance speedup of 1.3–1.6X and an off-chip traffic reduction of 13–41% relative to WS for parallel divide-and-conquer programs and bandwidth-limited irregular programs. Perhaps surprisingly, in such cases, PDF running on a CMP architecture with a relatively slow monolithic shared L2 cache retains its advantage over WS even when WS is run on a CMP architecture with a faster distributed L2 cache.
- For several other application classes, PDF and WS have similar performance, either because there is only limited data reuse that can be exploited or because the programs are not limited by off-chip bandwidth. In the latter case, PDF reduces the working set size, which has other benefits (see Section 2).
- Task granularity plays a key role in CMP cache performance. To help programmers and system designers cope with this issue, we present an automatic approach for selecting effective task grain sizes, based on a new working set profiling algorithm that is an order of magnitude faster than previous approaches.

This is the first paper demonstrating the effectiveness of PDF for constructive cache sharing on real benchmarks, providing a direct comparison between PDF and WS, revealing the limiting factors for PDF in practice, and presenting an approach for overcoming these factors.

The rest of the paper is organized as follows. Section 2 elaborates the benefits of constructive cache sharing and discusses related work. Section 3 describes PDF and WS schedulers in detail. Section 4 provides experimental methodology, then Section 5 presents our detailed experimental study. Focusing on the task granularity issue observed in the study, Section 6 proposes and evaluates a technique to automatically select task granularity for a class of parallel applications. Finally, Section 7 concludes the paper.

2. CONSTRUCTIVE CACHE SHARING

In this section, we motivate the need for constructive cache sharing on CMPs, then discuss related work on achieving it. Our discussion in this section, as well as the properties of the techniques we study, are in many ways *agnostic* to the particulars of the CMP implementations. This generality is important, given the ongoing debate over on-chip cache organizations, interconnect designs, and capabilities of cores in a CMP with many cores. Our argument relies on only two features. First, there is considerable on-chip cache that can service requests by any core. This can be a shared L2 cache or even private (L1 or L2) caches that can service misses from other cores. Second, the off-chip latency and bandwidth is significantly worse than the on-chip latency and bandwidth. The on-chip cache organization can be flat or hierarchical, uniform or heterogeneous, static or dynamic, etc.—as long as the worst of the on-chip latency and bandwidth is still many times better than going off chip, our discussion and results apply.

2.1 Constructive Sharing is Critical for CMPs

Mitigating the latency and bandwidth gap. In CMPs, there is a large latency and bandwidth gap between the on-chip and off-chip storage. The latency to off-chip storage is worse because of the much slower and longer inter-chip buses and the much larger off-chip storage (e.g., main memory) with significantly higher latency than on-chip storage (e.g., L1 and L2 caches). The bandwidth to off-chip storage is severely constrained by pin limitations. As a result, it is not uncommon to have an order of magnitude gap between the latency and bandwidth on-chip versus off-chip. To make matters worse, the number of cores on chip is rapidly increasing as a result of Moore's Law. Concurrently executing program threads on P cores may speed up the on-chip part of the computation P-fold. However, this may also result in a P-fold demand for the precious off-chip bandwidth, which is the case with program threads processing large *disjoint* working sets (as in many of the benchmarks we study). The threads *compete* for the same limited on-chip cache storage and off-chip bandwidth. In contrast, constructive cache sharing aims to have concurrently executing threads share a largely overlapping working set and therefore can reduce the aggregate working set size by up to a factor of P.

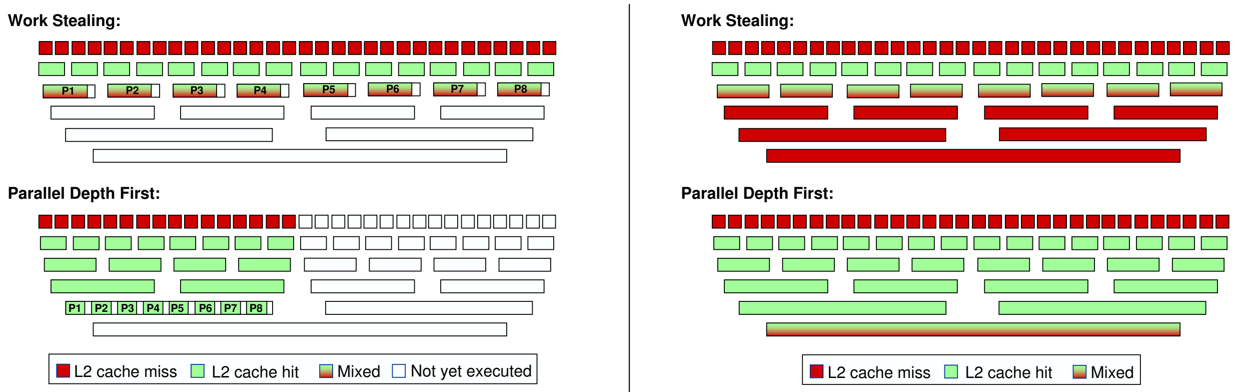
Enabling better use of on-chip real estate. One approach to attacking the off-chip bottleneck is to use larger and larger on-chip caches. If the number of cores doubles, double the cache size. While seductive, this approach suffers from the well-known fact that increasing the cache size often provides diminishing returns on reducing the miss rate. Moreover, it overlooks the fact that the same area may be more profitably devoted to other components, such as adding more cores. Constructive sharing removes the requirement that the cache scales directly with the number of cores. As a result, a given semiconductor fabrication technology generation can support more cores within the same area, while providing better miss rates.

Reducing power consumption. Because constructive cache sharing reduces the amount of cache needed by multithreaded programs (by up to a factor of P), it provides new opportunities to power down segments of the cache [25, 4, 41]. Consider, for example, a cache architecture that supports eight 1 MB on-chip caches that can be powered on or off as needed. If constructive cache sharing reduces the working set from 8 MB to $<$ 1 MB, then 7 of the 8 caches can be powered down. Moreover, constructive cache sharing saves power by reducing the off-chip traffic—studies have shown that an L2 miss serviced off-chip incurs 35X the power of an on-chip L2 hit [26].

2.2 Related Work

Much of the previous work that considers shared cache performance focuses on concurrent or interleaved *independent* computations [39, 2, 36, 31, 17, 38]. In particular, recent years have witnessed a number of investigations into the scheduling of such tasks to improve the utilization of various platform resources, including caches, for SMT [35, 29] and CMP [13, 37, 24, 22] processors by reducing *destructive* interference. In contrast, our work focuses on promoting *constructive* cache interference by taking advantage of the potential overlap in memory references among *cooperating* threads that share an address space, particularly multithreaded programs, for the reasons cited above.

Interestingly, Anderson and Calandrino [3] have a similar objective of encouraging the co-scheduling of cooperative threads—but in the context of real-time systems. While their approach is not particularly well-suited to non-real-



(a) Part way through

(b) Completed

Figure 1: Scheduling parallel Mergesort using WS and PDF: Picturing the misses. Each horizontal box is a sorted array of records, where at each level, pairs of arrays from the previous level are merged until all the records are sorted. The L2 hits and misses are shown for sorting an array of $\mathcal{C}_P$ bytes, where $\mathcal{C}_P$ is the size of the shared L2 cache, using 8 cores.

time systems, their micro-benchmark results do indicate that intelligent co-scheduling of cooperative threads can reduce the number of L2 misses substantially.

Philbin *et al.* [30] studied the possibility of reducing cache misses for *sequential* programs through intelligent scheduling of fine-grained threads. Their approach relies on memory access hints in the program to identify threads that should execute in close temporal proximity in order to promote cache re-use. Although the scheduler is not directly applicable to parallel scheduling, the approach may be a useful preprocessing step for the PDF scheduler, which relies on the program having a cache-friendly sequential schedule.

3. WORK STEALING AND PARALLEL DEPTH FIRST SCHEDULERS

In this paper, we compare the performance of two greedy schedulers proposed for fine-grained multithreaded programs: Work Stealing (WS) and Parallel Depth First (PDF).

Threads and the dependences among them are often described as a *computation DAG*. Each node in the DAG represents a *task*, which is a thread or portion of a thread that has no internal dependences to/from other nodes. A weight associated with each node represents the task's runtime. We refer to the longest (weighted) path in the DAG as the *depth* D. The DAG unfolds as the computation proceeds, and the job of the scheduler is to assign nodes of the DAG to processor cores over time so that no node is assigned at a time before its ancestors in the DAG have completed.

Work Stealing (WS) is a popular greedy thread scheduling algorithm[1] for programs, with proven theoretical properties with regards to memory and cache usage [10, 8, 1]. The policy maintains a work queue for each processor (actually a double-ended queue which allows elements to be inserted on one end of the queue, the top, but taken from either end). When forking a new thread, this new thread is placed on the top of the local queue. When a thread completes on a processor, the processor looks for a ready-to-execute thread by first looking on the top of the local queue. If it finds a thread, it takes the thread off the queue and runs it. If the local queue is empty it checks the work queues of the other processors and *steals* a thread from the bottom of the first non-empty queue it finds. WS is an attractive scheduling policy because when there is plenty of parallelism, stealing is quite rare and, because the threads in a queue are related, there is good affinity among the threads executed by any one processor. However, WS is not designed for constructive cache sharing, because the processors tend to have disjoint working sets.

Parallel Depth First (PDF) [6] is another greedy scheduling policy, based on the following insight. Important (sequential) programs have already been highly tuned to get good cache performance on a single core, by maintaining small working sets, getting good spatial and temporal reuse, etc. In PDF, when a core completes a task, it is assigned the ready-to-execute task that the sequential program would have executed the earliest.[2] As a result, PDF tends to co-schedule tasks in a way that tracks in some sense the sequential execution. Thus, for programs with good sequential cache performance, PDF provides good parallel cache performance (i.e., constructive cache sharing), as evidenced by the following theorem:

THEOREM 3.1. *[5] Let M_1 be the number of misses when executing an arbitrary computation DAG G sequentially with an (ideal) cache of size $\mathcal{C}$. Then a parallel execution of G using PDF on P cores with a shared (ideal) cache of size at least $\mathcal{C} + P \cdot D$ incurs at most M_1 misses, where D is the depth of G.*

This compares favorably to the comparable upper bound for WS, where the cache size must be at least $\mathcal{C} \cdot P$ to guarantee roughly M_1 misses [8, 1]. However, these analytical guarantees leave open a number of problems. For example, how does the number of misses *decreases* as the size $\mathcal{C}_P$ of the on-chip cache *increases* past $\mathcal{C} + P \cdot D$? In this paper, we address the problems through experimental studies, where $\mathcal{C}_P$ is determined by technology factors, and increases roughly linearly with P in the default configurations part of our study.

[1] In a *greedy* schedule, a ready job remains unscheduled only if all processors are already busy.

[2] Note that [6, 7, 28] show how to do this on-line without executing the sequential program.

Table 1: Parameters common to all configurations.

Processor core	In-order scalar
Private L1 cache	64KB, 128-byte line, 4-way, 1-cycle hit latency
Shared L2 cache	128-byte line, configuration-dependent
Main Memory	latency: 300; service rate: 30 (cycles)

Table 2: Default configurations.

Number of cores	1	2	4	8	16	32
Technology (nm)	90	90	90	65	45	32
L2 cache size (MB)	10	8	4	8	20	40
Associativity	20	16	16	16	20	20
L2 hit time (cycles)	15	13	11	13	19	23

Table 3: Single technology configurations with 45nm technology.

Number of cores	1	2	4	6	8	10	12	14	16	18	20	22	24	26
L2 cache size (MB)	48	44	40	36	32	32	28	24	20	16	12	9	5	1
Set associativity	24	22	20	18	16	16	28	24	20	16	24	18	20	16
L2 hit time (cycles)	25	25	23	23	21	21	21	19	19	17	15	15	13	7

Figure 1 depicts pictorially the L2 cache hits and misses when using WS and PDF to schedule a parallel Mergesort (which is detailed in Section 4.2). Mergesorting an n byte (sub)array uses $2n$ bytes of memory, because after completing a merge of two sub-arrays X and Y of size $n/2$ into a sub-array of size n, the buffers holding X and Y can be reused. In (a), we see a snapshot in which WS is starting to encounter capacity misses because each core, P1–P8, is working on a sub-array of size $n = \mathcal{C}_P/8$, and hence their aggregate working set of $2 \cdot \mathcal{C}_P$ does not fit within the L2 cache. In contrast, PDF has P1–P8 performing a parallel merge into a sub-array of size $\mathcal{C}_P/2$, and hence is incurring no capacity misses. In fact, the only misses thus far are the cold misses in bringing in the first half of the input array. From (b) we see that with P cores there are $\log P$ levels in which PDF incurs no misses while WS incurs all misses. This is a general phenomenon for the common recursive divide-and-conquer paradigm where the problem sizes decrease by (roughly) a factor of 2 at each level of the recursion: PDF eliminates the misses in $\log P$ levels (only).

As apparent in Figure 1, for Mergesort using PDF, the number of misses is $M_{pdf} \approx \frac{N}{B}\log(N/\mathcal{C}_P)$, where N is the number of items being sorted and each cache line can hold B items. A standard (recursive) sequential Mergesort incurs $M_1 = \frac{N}{B}\log(N/\mathcal{C})$ misses, where $\mathcal{C}$ is the size of the cache. Note that because $\mathcal{C}_P > \mathcal{C}$, we have that $M_{pdf} < M_1$. For Mergesort using WS, the number of misses is $M_{ws} \approx \frac{N}{B}\log(NP/\mathcal{C}_P) = \frac{N}{B}\log P + M_{pdf}$. These results hold for any $\mathcal{C}_P \geq \mathcal{C} + P \cdot D$, including $\mathcal{C}_P = P \cdot \mathcal{C}$ as well as the configurations in our study.

4. METHODOLOGY

In this section, we describe our experimental methodology, focusing on the CMP design space to explore and the benchmarks to use in our study.

4.1 CMP Design Space

We evaluate the performance of the WS and PDF schedulers across a range of realistic (future) CMP configurations. We assume area-constrained scaling and use a proportional chip area allocation [20]. All area factors used are based on the 2005 ITRS edition [32]. We consider, in particular, the 90nm, 65nm, 45nm, and 32nm technologies.[3] Although to be concrete the configurations described below are based on specific technologies, our results hold more generally across a wide range of cache parameters.

We focus on CMP designs with private L1 caches and a shared L2 cache. For our purposes, the most important configuration parameters are (i) the number of processing cores (P), and (ii) the size of L2 cache ($\mathcal{C}_P$). (We consider a private L1 cache as a component in a core design and keep the L1 cache size per core fixed.) The die size is fixed at $240mm^2$. 75% of the total die area is allocated to the processing cores, shared L2 cache, and the processor interconnect, leaving the rest for other system-on-chip components. Of the core-cache area, 15% is used by the processor interconnect and related components, leaving approximately 65% of the total die area ($150mm^2$) for cores and caches. We model a single-threaded in-order core. We compute its area requirement by using the data of IBM PowerPCRS64 ([12]), which is an in-order, dual-threaded core, and by assuming a 5% area decrease for removing the second hardware thread context [19]. Then we use the logic area factors from ITRS to compute the core area under various process technologies. Given a P, we can determine the area occupied by all cores, and the remaining area is allocated to the L2 cache.

Our L2 cache design assumes a rectangular cache layout in which cache banks are connected through switches on a 2D-mesh network, similar to S-NUCA-2 [23] but with a uniform access delay. We calculate $\mathcal{C}_P$ for each technology using ITRS estimates of SRAM cell area factors and efficiency. The cache access latency is the network round-trip latency to access the furthest away bank, plus the bank access delay.

Cacti 3.2 [34] is used to determine optimized cache designs and their latencies. Our optimized cache designs employ 1MB or 2MB cache banks. These bank sizes balance network delay with bank access latency. Using realistic signal delay models [14], we calculate the bank-to-bank hop latency to be 1 cycle for the cache sizes and technologies considered. We optimize the overall bank access latency by using Cacti recursively on each bank, where each recursion step determines whether dividing this sub-bank even further will result in lower access latency. Our optimized 1MB cache bank design employs 4 x 256KB sub-banks with split tag and data arrays, with an access latency of 7 cycles and wave pipeline time of 3 cycles at 45 nm technology, while our 2MB cache bank design employs 4 sub-banks each divided into 4 x 128KB sub-banks with split tag and data arrays, resulting in 9 cycles access latency and 2 cycles wave pipeline time for the same technology. We assume conservatively that those latencies are the same for the 90nm, 65nm and 32nm geometries.

Given the above methodology, we generate realistic configurations in two different design spaces: *scaling technology* and *single technology*. The non-varying configuration parameters of our experiments are summarized in Table 1.

[3] By the end of 2006, major microprocessor manufacturers have already been shipping or started shipping products based on 65nm process technology. Intel has announced plans to start 45nm production in the second half of 2007.

Scaling technology. Under scaling technology, we assume that the process technology will change as we increase the number of cores. Such an assumption represents realistic trends over time, as designers tend to increase the number of cores with subsequent process generations. One effect of scaling technology is that configurations with more cores tend to also accommodate larger caches. The six settings as shown in Table 2 were selected as the default configurations for the given number of cores.

Single technology. In contrast, the single technology design space represents the trade-offs associated with a particular technology. Microprocessor designers typically must design for a particular process generation and, consequently, must evaluate trade-offs within a particular technology. We chose the 45nm process technology as a contemporary design space and the selected configurations for this technology are given in Table 3.

Given a particular configuration, we evaluate the performance of the two schedulers on each multithreaded program by (1) annotating the program to mark task boundaries, (2) collecting a trace of its computation DAG annotated with the memory references for each task, and finally (3) executing the DAG on the simulated CMP in accordance with the scheduler. (Details are in [16].)

4.2 Benchmarks

We studied the effect of scheduling on a number of benchmarks from a variety of domains. However, here we focus on only three of the benchmarks (LU, Hash Join, and Mergesort), as representative of common classes of benchmarks, deferring discussions of other benchmarks to Section 5.5.

LU. LU is a representative scientific benchmark, with its easy parallelization and small working sets. We used the parallel LU implementation of the Cilk distribution. The benchmark performs a recursive factorization on a dense N×N matrix. The input matrix is partitioned into four quadrants recursively, until the size of the quadrant is equal to the block size B. A smaller block size creates a larger number of smaller threads. The block size effectively controls the grain of parallelism, thus we did not have to modify the benchmark. (Due to trace size limitations, the largest input size we were able to factorize is a 2K×2K matrix of doubles, or 32MB input data. As this is smaller than the L2 cache in the 32-core default configuration, we only report LU results for up to 16 cores.)

Hash Join. Hash Join is representative of many (commercial or otherwise) programs that use large irregular data structures and benefit from large caches. We use a state-of-the-art database hash join code [15]. In an initial *I/O partition phase*, each of the two large input tables is partitioned into fragments that fit within the memory buffer allocated for the join (1GB in our study). We study the second, *join* phase, which joins pairs of partitions; in current database systems, this is the most time-consuming phase. Each partition is divided into sub-partitions that fit within the L2 cache. For each sub-partition, the keys from the "build" table are placed in a hash table, which is then probed for matches from the "probe" table. The matching build and probe records are concatenated to produce outputs. While the original code used one thread per sub-partition, we further divided the probe procedure (which typically dominates the join phase [15]) for each sub-partition into multiple parallel tasks to produce finer-grained threading. Here, we report representative experiments that join a pair of build and probe partitions that fit tightly in a 1GB memory buffer. Every build record matches 2 probe records where each record is 100B and the join attribute is 4B.

Mergesort. Mergesort is representative of many programs that use a recursive divide-and-conquer paradigm. Our Mergesort benchmark is structured after libpmsort [40], a parallel recursive mergesort library, but with the serial merging of two sorted sub-arrays modified to use a parallel merge instead. We select a total of k splitting points from the two sorted sub-arrays, for a suitable value of k seeking to optimize the cache performance at the given level of recursion (details in Section 5.4 and Section 6.2). For each chosen value from one array, we locate the closest value in the other array using binary search. In this way, we create k pairs of array chunks, which can be merged in parallel. An example is given in Figure 1.

5. EXPERIMENTAL STUDY

In this section, we present a detailed experimental study of the PDF and WS schedulers. We explore the CMP design space in order to answer the following questions: (i) Is the choice of PDF vs. WS significant in practice? If yes, what types of applications benefit most from PDF? (ii) How does the performance of PDF vs. WS change across the CMP design space? How does this impact the choice of CMP design points? (iii) Are the results sensitive to changes in architectural parameters? (iv) What is the impact of thread granularity on the performance of PDF vs. WS?

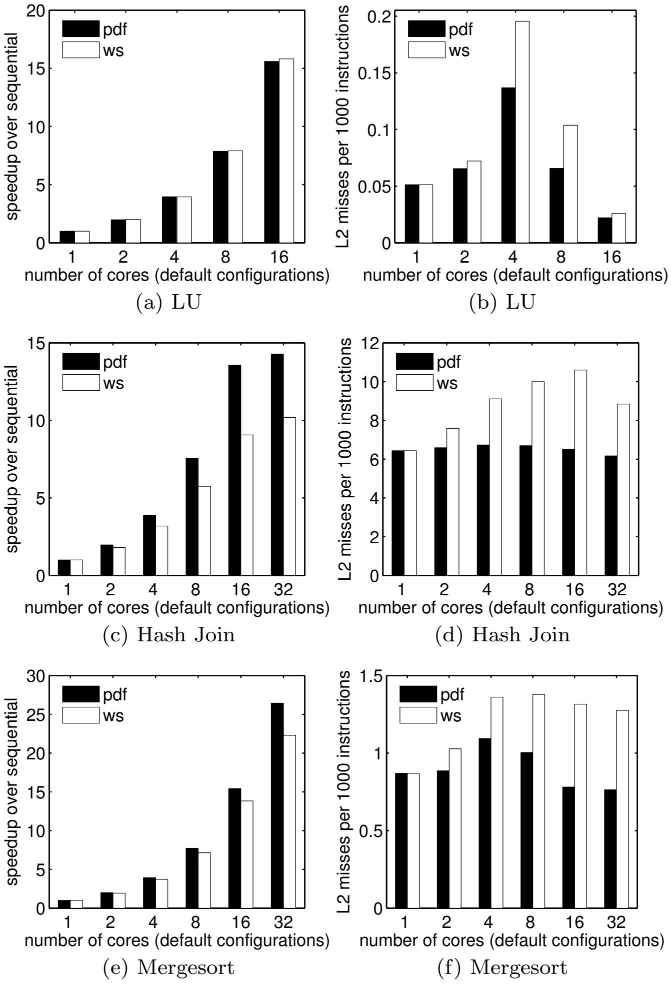
In the following, we begin in Section 5.1 by comparing PDF and WS using the default CMP configurations. Then we explore the CMP design points under the 45nm technology in Section 5.2. We perform sensitivity analysis in Section 5.3, and study the impact of thread granularity in Section 5.4. Finally, Section 5.5 summarizes our findings.

5.1 Default Configurations: PDF vs. WS

Figure 2 compares the performance of the PDF and WS schedulers for the three application benchmarks running on our default CMP configurations with 1 to 32 cores. Each row of sub-figures in Figure 2 shows the performance results of a single application. From left to right, we report (i) the speedup of running the application using all the cores compared to sequential execution of the application on one of the cores with the same CMP configuration, and (ii) the L2 cache misses of the application with different schedulers.

From Figure 2, we see that the comparisons between PDF and WS vary significantly for the three applications. Therefore, we analyze the results for each application in turn.

LU. Figures 2(a)-(b) depict the performance results of LU for five default configurations, using PDF and WS. We can see that PDF reduces up to 36.8% L2 misses per instruction relative to WS. However, the miss per instruction ratio is very low to begin with because of the algorithm's small working set. In fact, the average memory bandwidth utilization for LU is only 0.15%-1.57% for PDF and 0.15%-2.41% for WS, with PDF having slightly smaller utilization as expected. Therefore, the reduced L2 misses by PDF scarcely affects performance, and the absolute speedups are practically the same for both PDF and WS.



(a) LU (b) LU

(c) Hash Join (d) Hash Join

(e) Mergesort (f) Mergesort

Figure 2: Parallel Depth First vs. Work Stealing with default CMP configurations.

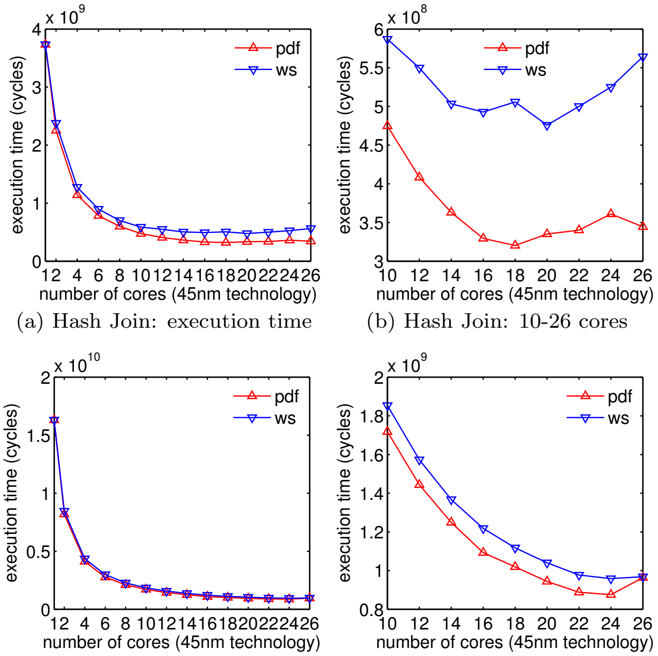
Hash Join. Figures 2(c)-(d) report the performance results of joining a single pair of memory-sized partitions for all six default configurations. We can see that PDF achieves significantly better performance than WS. For 2-32 cores, PDF achieves a factor of 1.97-14.28 fold speedups over sequential execution, while WS obtains only a factor of 1.81-10.19 fold speedups. These result in a factor of 1.09-1.50 fold relative speedups of PDF over WS. The good performance of PDF comes from effective constructive cache sharing to avoid off-chip cache misses. As shown in Figure 2(d), PDF reduces 13.2%-38.5% L2 misses per instruction compared to WS. Interestingly, the performance increase by doubling the number of cores is significantly smaller from 16 to 32 cores than in other cases. This is because Hash Join is main memory bandwidth-bound for the 16-core and 32-core configurations: it utilizes 89.5%-90.1% of the available memory bandwidth with PDF and 92.2%-97.3% with WS.

Mergesort. Figures 2(e)-(f) show the performance results of sorting 32 million integers using Mergesort, for the six default CMP configurations. For 2-32 cores, PDF achieves a factor of 2.00-26.44 fold speedups over sequential execution, while WS obtains a factor of 1.93-22.30 fold speedups. These lead to a factor of 1.03-1.19 fold relative speedups with 2-32 cores of PDF over WS. Figure 2(f) depicts the L2 misses per instruction ratios. Similar to Hash Join, PDF successfully reduces 13.8%-40.6% L2 misses per instruction compared to WS. Comparing Figure 2(b), Figure 2(d), and Figure 2(f), we see that the L2 misses per instruction ratio of Mergesort (around 0.1%) is much lower than Hash Join (around 0.6%), but is still significant enough compared to LU (around 0.01%) to make a difference on performance. We can clearly see the trend that the larger the ratio of L2 misses per instruction, the larger impact constructive cache sharing may have, and therefore the larger relative performance benefits of PDF over WS. Moreover, unlike Hash Join, Mergesort experiences only up to 71.0% memory utilization due to the lower misses per instruction ratios, and thus the absolute speedup continues to increase dramatically from 16 to 32 cores.

Considering the performance results of the three benchmarks, we conclude that PDF achieves significantly better performance than WS for a group of important applications that have non-trivially large working sets, as evidenced by the L2 misses per instruction ratios. Because LU does not differentiate PDF and WS, we focus on Hash Join and Mergesort in the rest of the experimental study.

5.2 Single Technology Analysis

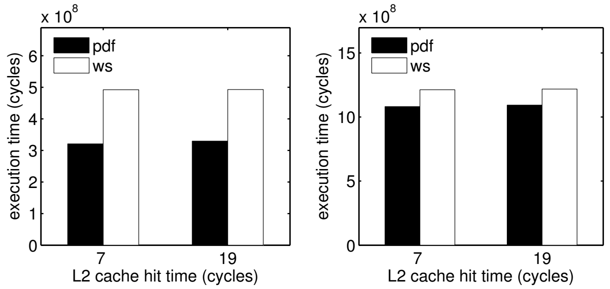
Figure 3 shows the execution time of Hash Join and Mergesort using PDF and WS, for 1-26 cores under the 45nm process technology. As shown previously in Table 3, the L2 cache size decreases from 48MB with 1 core to 1MB with 26 cores. We examine Figure 3 for two purposes: (i) comparing the performance of PDF and WS; and (ii) understanding the impact of PDF on the choices of CMP design points. For the first purpose, as shown in Figure 3, we see that PDF wins across all the CMP configurations, achieving over WS



(a) Hash Join: execution time (b) Hash Join: 10-26 cores

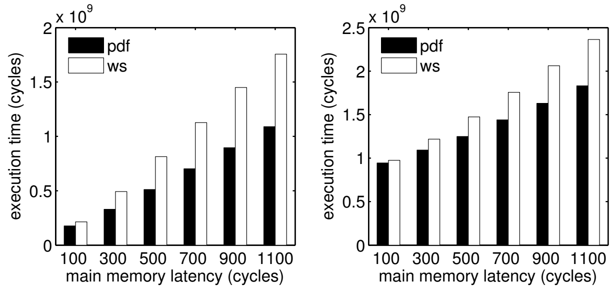
(c) Mergesort: execution time (d) Mergesort: 10-26 cores

Figure 3: Parallel Depth First vs. Work Stealing under a single technology (45nm).



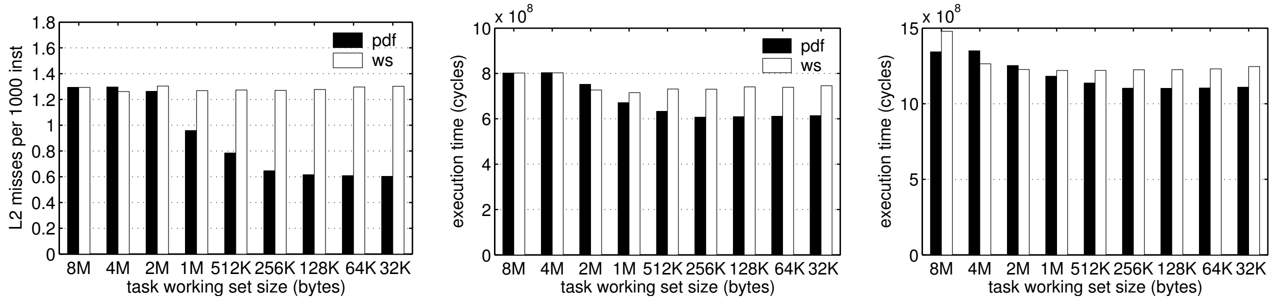
(a) Hash Join (b) Mergesort

Figure 4: Varying L2 cache hit time with the 16-core default configuration.



(a) Hash Join (b) Mergesort

Figure 5: Varying main memory latency with the 16-core default configuration.



(a) L2 misses on 32 cores (b) Execution time on 32 cores (c) Execution time on 16 cores

Figure 6: Varying the task granularity of parallel Mergesort for default configurations.

a factor of 1.06-1.64 fold speedup for Hash Join and a factor of 1.03-1.11 speedup for Mergesort.

For the second purpose, as shown in Figures 3(a) and (c), we can see that the major trend of all the curves is to generally decrease as the number of cores increases, meaning that application performance generally improves with more cores. When there are 10 or more cores, the curves seem flat. However, when we zoom into the 10-26 core performance in Figures 3(b) and (d), we see that the curves actually vary. The Hash Join curves reach the lowest points around 18 cores then go up, while the Mergesort curves continue to decrease until 24 or 26 cores. With 18 or more cores, Hash Join utilizes over 95% of the main memory bandwidth. Increasing the number of cores while decreasing the cache size only makes the situation worse, leading to the worse performance. In contrast, Mergesort is not bounded by memory bandwidth and its performance improves with more cores.[4]

When making a design choice in the CMP design space, a typical goal is to optimize the performance of a suite of benchmark applications (e.g. SPEC) measured by aggregate performance metrics. Compared to WS, PDF provides larger freedom in the choice of design points in order to achieve a desired level of performance (e.g., more than K times faster than a reference configuration) for a multithreaded application. For example, for Hash Join, 10-26 cores with PDF achieve similar or better performance than the best WS performance. Similarly, 20-26 cores with PDF achieve similar or better performance than the best WS performance for Mergesort. In this way, designers are able to make better trade-offs balancing sequential vs. multithreaded programs, and computation-intensive vs. memory-intensive programs.

5.3 CMP Parameter Sensitivity Analysis

Figure 4 and Figure 5 compare the performance of PDF vs. WS varying the L2 cache hit time and varying the main memory latency. The results are similar to our default configurations. In particular, compared to WS, PDF achieves a factor of 1.21-1.62 fold relative speedups for Hash Join and 1.03-1.29 fold relative speedups for Mergesort.

Interestingly, from Figure 4, we can compare the PDF bar with 19-cycle L2 hit time to the WS bar with 7-cycle hit time. This shows that PDF running on a CMP architecture with a relatively slow monolithic shared cache (19-cycle hit time) retains its advantage over WS even when WS is run on a CMP architecture with a faster distributed on-chip cache (7-cycle hit time to a core's local bank). This is because for Hash Join and Mergesort, in our experiments, the number of L2 hits is on par with the number of L2 misses, so the L2 miss time dominates any differences in L2 hit times.

5.4 Impact of Thread Granularity

In the course of parallelizing the benchmark applications, we found that *task granularity* had a large impact on cache performance and execution times. As discussed in Section 4.2, the original versions of the Hash Join and Mergesort programs both suffered from being too coarse-grained. The original Hash Join code generates only one thread per cache-sized sub-partition. The original Mergesort code employs a serial merging procedure. By parallelizing the probe procedure within the processing of a sub-partition in Hash Join and by parallelizing the merging procedure for Mergesort,

[4] From 24 cores (5MB cache) to 26 cores (1MB cache), Mergesort with PDF experiences a large jump (41% increase) of the L2 misses per instruction ratio. This results in the jump of the execution time.

we removed serial bottlenecks and improved constructive cache sharing among multiple threads.[5] As a result, our fine-grained versions are up to 2.85X faster than the coarse-grained originals.

We further study the impact of task granularities on the fine-grained versions of the code. Here we focus on the deeper, more complicated (and thus more interesting) Mergesort DAG. (In contrast, Hash Join has a simple two-level task structure.) Figure 6 shows the L2 misses per instruction and the execution time of Mergesort as a function of the task working set sizes. Because of the regularity of Mergesort, adapting the Mergesort code to use a desired task working set size is straightforward; by choosing the sorting sub-array size to be half the desired working set size[6], the sub-array can be sorted efficiently within a single sequential task.

Figure 6 shows that while the cache performance of WS is relatively flat across the range of task sizes, the cache performance of PDF improves considerably with smaller task sizes. As a result, PDF's cache performance advantage increases with smaller task sizes (e.g., PDF incurs fewer than half as many misses as WS for 32KB task sizes with the 32-core default configuration). Thus, as the figure shows, thread granularity has a large impact on the relative performance gains of PDF vs. WS. When each scheduler gets its *optimal* task size, PDF is 1.17X faster than WS because of this improved cache performance.

5.5 Summary of Our Benchmark Study

Although in this section we have focused on only three benchmarks, in all, we have studied additional benchmarks from a variety of domains: numeric (Cholesky [9], Matrix Multiply), scientific simulation (Barnes, Heat [9]), data mining (Hmmer [21]), sorting (Quicksort), meshing (Triangle [33]) and classification (C4.5 [27]). We now summarize the key findings from the experimental results in this section as well as briefly describe the lessons we learned from our extended benchmark study.

First, there are benchmarks (Hash Join, Mergesort) for which PDF's advantage translates into up to 1.3–1.6X performance improvement over WS. As discussed in Section 4, Hash Join is representative of many (commercial or otherwise) programs that use large irregular data structures and benefit from large caches. Mergesort demonstrates the benefits PDF provides for benchmarks with a recursive divide-and-conquer paradigm. Many other benchmarks (Quicksort, Triangle, C4.5) follow this paradigm. However, unlike Mergesort, their "divide" steps may break a subproblem into two highly imbalanced parts because the "divide" point is often chosen for specific algorithmic needs not for balancing the two parts. Fortunately, PDF can effectively handle irregular parallel tasks that are dynamically spawned [5].

Second, many benchmarks (LU in the above study, and other benchmarks such as Matrix Multiply, Cholesky, Barnes) can achieve good cache performance with a very small amount of data in cache. In such cases, the fact that WS increases the working set by the number of cores P does not effect performance, because the aggregate working set still fits in the shared cache, and hence WS matches PDF's performance. As discussed in Section 5.1, the small working set often manifests itself as low L2 misses per instruction ratios. Our results support that this ratio should be on the order of 0.1% or more for PDF to make a significant difference in execution time. However, even for these applications, PDF can be valuable. As pointed out in Section 2, PDF's smaller working set can translate into better use of on-chip real estate and reduced power consumption. Additionally, when multiple programs are run at the same time, the PDF version is less of a cache hog and its smaller working set is more likely to remain in the cache across context switches.

Third, PDF achieves significant performance gains over WS across a large number of CMP design points and architectural parameters as shown in the above subsections. Because of this, PDF provides larger freedom in the choice of design points in order to achieve a desired level of benchmark performance.

Finally, most benchmark programs, as written, use such coarse-grained threading that there is no opportunity for cache-friendly scheduling. In fact, many programs use a scheduler only at the beginning of the program: for P processors, P threads are spawned at the beginning of the program and no further spawning is done. Thus, in our study, we incorporated much finer-grained threading in the programs (e.g. Hash Join and Mergesort) we studied. Our work quantifies the benefits of more fine-grained threading. Because of its importance, in the following section, we focus on the problem of selecting task granularities.

6. AUTOMATIC SELECTION OF THREAD GRANULARITY

One of the important findings in Section 5 is that *task granularity* has a large impact on cache performance and execution times. On the one hand, coarse-grained tasks may have serial bottlenecks and large disjoint working sets that hinder constructive cache sharing. On the other hand, threading that is too fine-grained increases scheduling and synchronization overheads, as well as any instruction overheads for running parallel code versus sequential code.[7] Thus judiciously choosing task granularity is an important yet challenging problem.

In general, selecting good task sizes is challenging because it requires predicting how P tasks that *might* run concurrently *would* interact in the L2 cache. To guide the selection of appropriate task grain sizes, we have developed an efficient working-set profiler for multi-threaded programs, which can be used for profile-based feedback during software development to set appropriate task sizes. In this approach, programs are first written with fine-grained tasks, and the profiler suggests groups of tasks to combine into larger tasks based on their working sets for better performance.

In the following, we present our working set profiler in Section 6.1, then in Section 6.2 we describe how to use the profiling information to automatically choose task granularity for a class of applications, and evaluate our approach with Mergesort.

[5] In all the above experiments in Section 5, we manually chose the number of splitting points k in the parallel merge to obtain sufficient parallelism. Within the sub-DAG of sorting a sub-array A that is half the L2 cache size, we choose k so that the aggregate number of merging tasks per DAG level is 64, which is larger than the number of cores.

[6] Note that the working set size for mergesorting a sub-array of size n is $2n$.

[7] In Mergesort, for example, it is well known that sorting small sub-arrays sequentially is faster than continuing to apply parallel Mergesort recursively.

6.1 Efficient One-Pass Profiling for Groups of Consecutive Tasks

We call a group of consecutive tasks (corresponding to a sub-graph in the DAG) a task group. Consider the case of parallel Mergesort. The task group for sorting an entire sub-array consists of three smaller task groups: sorting the left half, sorting the right half, and merging the two parts. Parallel merging may be formed into multiple levels of task groups by recursively dividing a group containing K tasks into two sub-groups containing $\lfloor K/2 \rfloor$ and $\lceil K/2 \rceil$ tasks, respectively. In this way, task groups form a hierarchical structure, where each parent task group is a superset of all the child task groups, sibling task groups are disjoint, and the leaf nodes are the finest-grain individual tasks. In general, given very fine-grained tasks, we would like to know the working set sizes of all task groups and use this information to guide task selections.

To obtain the working set size of a single task group, a straightforward approach is to generate the memory reference trace of the task group then perform trace-driven simulations of set-associative caches across a range of cache sizes (starting with a cold cache). We call this approach *SetAssoc*. However, *SetAssoc* becomes tedious when a large number of nested task groups are to be measured. It has to process the trace of the entire application multiple times corresponding to the number of levels in the task group hierarchy.

We propose a one-pass algorithm that first processes a program's memory reference trace once to collect statistics on every task and then uses the gathered statistics to efficiently compute the working set size of potential groups of consecutive tasks. We restrict our analysis to tasks that are *consecutive* in a sequential run of the program, as these are the tasks that are naturally grouped together when coarsening. Note that our use of predefined hierarchical task groups already captures opportunities to group parallel siblings together. We do not seek to group together *arbitrary* collections of tasks, however.

To collect per-task statistics, we perform a one-pass cache simulation using the LRU stack model, where each cache line data structure is augmented with the last visiting task ID. Given a memory reference R of task i, the cache simulator returns two values: the distance d of the visited line to the stack top and the ID j of the task that last visited the line. Note that whether R is a cache hit or a miss depends on both the *cache size* and the *task group* considered. To understand why the latter is also important, consider a task group including consecutive tasks b, $b+1$, $\ldots$, i, $\ldots$, and e. Then, starting with a cold cache before task b, reference R of task i would be a cold miss if the last visiting task ID $j < b$, but possibly a hit if $j \geq b$ (depending on the cache size). Therefore, the per-task statistics must capture both the distance and the previous task information. We design the per-task statistics as a two-dimensional histogram. The *distance* dimension is divided into buckets $D_1 < D_2 < \cdots < D_k$ corresponding to the list of increasing cache sizes for working-set computations. In the *previous-task* dimension, the bucket ID is the difference between the task IDs of the current and previous visits to the same line. For example, given the above return values for reference R, the algorithm looks for $D_{p-1} < d \leq D_p$, and then increments the count of the bucket $(D_p, i-j)$ by 1.

After obtaining the per-task two-dimensional histogram, the algorithm computes the working set size of any group of consecutive tasks as follows. Given a task group including task b to e, the number of cache hits for a cache size D_p is equal to the sum of all the buckets (D, T) where $D \leq D_p$ and $T \leq i - b$ for every task i in the group.

Let us examine the complexity of the above algorithm. The LRU stack model can be implemented as a doubly linked list of nodes representing cache lines. Moreover, all the nodes can be indexed by a hash table with cache line addresses as hash keys. A typical operation involves four steps: (i) looking up the cache line address of a reference R; (ii) counting the distance from the current node to the stack top; (iii) updating the stack model by moving the visited node to the top of stack; and (iv) retrieving and updating the previous task ID. Steps (i), (iii) and (iv) are all $O(1)$. For step (ii), we build a tree structure on top of the linked list to count the distance from a node to the stack top in $O(\log N)$, where N is the number of cache lines of the largest cache size considered. For good cache performance, we implemented a B-tree structure with cache-line-sized tree nodes. We call this algorithm *LruTree*.

We compare the performance of the *LruTree* and the *SetAssoc* algorithms by processing a trace of Mergesorting 32 million integers. The trace consists of 2.85 billion memory references, over 110,000 tasks, and over 190,000 task groups. We ran the algorithms on a desktop machine with 3.2GHz Pentium 4, 1GB memory, and a Seagate Barracuda 7200rpm IDE disk. We find that the *SetAssoc* algorithm took 253 minutes, while the *LruTree* algorithm ran in only 13.4 minutes—an 18X improvement. Because of the nesting nature of task groups, *SetAssoc* suffers from revisiting each memory record over 22 times on average, whereas *LruTree* is a one-pass algorithm. This performance advantage increases as the problem and DAG sizes grow. Note that *LruTree* can be implemented with on-the-fly trace consumption, thus reducing the cost of generating traces.

6.2 Using Profiling Information for Automatic Task Coarsening

The automatic task coarsening algorithm traverses the task group tree from top to bottom and evaluates a heuristic stop criterion at every node. Suppose node G's working set size is W, and it has K child task groups of similar sizes. We stop at G's children if the following is true[8]:

$$W \leq K \times (cachesize/(numcores * 2))$$

In this way, the child tasks can keep the cores busy. Note that due to task size variability, some child tasks may finish early and other parallel work may be scheduled, leading to sub-optimal cache behavior. The "2" in the criterion is to reduce this effect.

To incorporate the task selection information into parallel programs, we capitalize upon the fact that many parallel programs are written with a general divide-and-conquer structure as shown in Figure 7(a). When the `Parallelize` function is evaluated `True` for the parameter, the task is further divided into child parallel tasks; otherwise, a sequential version of the code is executed. Typically, the `Parallelize` function compares the parameter with an appropriate threshold value T, which must be hand-tuned based on the programmer's knowledge of the program data struc-

[8] Note that G's children may have dependencies as in the case of Mergesort. For each independent set of children, we separately apply the criterion and decide whether to stop.

```
Function parallel_f(param) {
    If (Parallelize(param, __FILE__, __LINE__)) {
        Spawn (parallel_f(Subdivide(param, 1)));
        Spawn (parallel_f(Subdivide(param, 2)));
        ...
        Spawn (parallel_f(Subdivide(param, k)));
        Sync ();
        combine_results(param);
    } Else {
        sequential_f(param);
    }
}
```

(a) Example divide-and-conquer style parallel program

CMP Configuration		Calling Location		Param
L2 Size	# Cores	__File__	__Line__	Threshold
....	....	....	....	....
....	....	....	....	....

(b) Table for implementing the Parallelize function

Figure 7: Incorporating task selection results into parallel programs.

tures and would vary with different cache sizes at run-time. Such tuning is error-prone and further complicated in the presence of constructive cache sharing.

This tuning task can be greatly simplified by including a parallelization table as part of the compiled program, as shown in Figure 7(b). In the table, Param thresholds are indexed by CMP configuration parameters and the locations of parallelization decisions. At compile-time, these thresholds are seeded with default values that correspond to very fine-grain threading. The program is then profiled as described in Section 6.1. Each task group is also annotated with the corresponding param value by recording the value at every spawn invocation. After that, the above analysis is used to determine the stopping task groups for the combinations of CMP configurations and calling locations. Finally, the default threshold values in the table are replaced with the param values that are associated with the stopping task groups in the final executable.

Note that we obtain the working set information once through a single profiling pass, but we need to perform a task coarsening analysis for every CMP configuration because the stopping criterion is configuration dependent. Fortunately, the number of CMP configurations can be reasonably bounded by the expected lifetime of the executable. Moreover, the run-time table lookup costs can be reduced by identifying the CMP configuration at program initialization and replace the lookup with a single memory read with appropriate compiler/system support.

Finally, we evaluate the effectiveness of the automatic task coarsening algorithm. Figure 8 compares three schemes using the Mergesort benchmark while varying the number of cores. The left bar uses the manually selected tasks and corresponds to previous results in Section 5. The middle and right bars both use the same task selections recommended automatically by our algorithm. The difference is how we perform CMP simulation using the task selections. For the right bar, we manually change the Mergesort code to realize the selection. We run CMP simulation based on the new trace. In contrast, for the middle bar, we use the same finest-grain trace in the CMP simulation but simply substitute a new task DAG based on the recommended task grouping. Therefore, compared to the right bar, an individual task of the middle case may be less efficient because it still contains the parallel code (e.g. parallel merging). From Figure 8, we see that the right bars are within 5% of the optimal in all cases, demonstrating the effectiveness of our automatic task coarsening scheme.

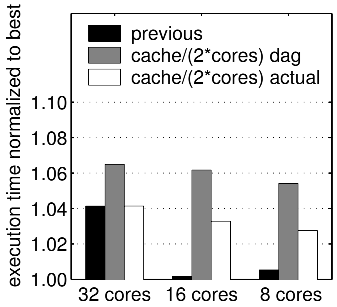


Figure 8: Effectiveness of the task selection schemes.

7. CONCLUSION

The advent of Chip Multiprocessor (CMP) platforms requires a re-evaluation of standard practices for parallel computing. While traditional (Symmetric MultiProcessors) SMP configurations (where the main system limitation was often coherence traffic) encouraged coarse-grained parallelism to reduce sharing, the off-chip bandwidth bottleneck of CMP machines encourages a fine-grained parallelism to increase resource sharing.

This study demonstrates that the Parallel Depth First (PDF) scheduler, which was designed to encourage cooperative threads to constructively share cache, either matches or outperforms the Work Stealing (WS) scheduler on a CMP machine for all the fine-grained parallel programs studied. By making more effective use of cache resources, the PDF scheduler also broadens the design space for microprocessor designers—potentially enabling the inclusion of more cores at the expense of cache resources that are less critical given PDF. Finally, task granularity plays a key role in CMP cache performance, and we present an automatic approach for selecting effective task grain sizes when using PDF.

8. ACKNOWLEDGMENTS

We would like to acknowledge Yevgen Voronenko and Ryan Johnson for their course project on evaluating PDF vs. WS with a set of scientific applications.

9. REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35, 2002.

[2] A. Agarwal, M. Horowitz, and J. L. Hennessy. An analytical cache model. *ACM Trans. on Computer Systems*, 7(2), 1989.

[3] J. Anderson and J. Calandrino. Parallel real-time task scheduling on multicore platforms. Under submission, 2006.

[4] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. A dynamically

tunable memory hierarchy. *IEEE Transactions on Computers*, 52(10), 2003.

[5] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *SPAA*, 2004.

[6] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2), 1999.

[7] G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *SPAA*, 1997.

[8] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *SPAA*, 1996.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiseron, K. H. Randall, and Y. Zhou. CILK: An efficient multithreaded runtime system. In *PPoPP*, 1995.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

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

[12] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM JRD*, 44(6), 2000.

[13] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, 2005.

[14] G. Chen, H. Chen, M. Haurylau, N. Nelson, D. Albonesi, P. M. Fauchet, and E. G. Friedman. Electrical and optical on-chip interconnects in scaled microprocessors. In *ISCAS*, 2005.

[15] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Inspector joins. In *VLDB*, 2005.

[16] 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. *Technical Report IRP-TR-07-01*, Intel Research Pittsburgh, 2007.

[17] Y.-Y. Chen, J.-K. Peir, and C.-T. King. Performance of shared caches on multithreaded architectures. *Journal of Information Science and Engineering*, 14(2), 1998.

[18] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. In *ISCA*, 2005.

[19] J. Clabes, J. Friedrich, M. Sweet, and J. Dilullo. Design and implementation of the POWER5 microprocessor. In *Int. Solid State Circuits Conf.*, 2004.

[20] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing CMP throughput with mediocre cores. In *PACT*, 2005.

[21] S. Eddy. HMMER: profile HMMs for protein sequence analysis. http://hmmer.wustl.edu/.

[22] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX ATC*, 2005.

[23] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS-X*, 2002.

[24] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[25] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. In *ISCA*, 2000.

[26] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy implications of multiprocessor synchronization. In *SPAA*, 2006. Brief announcement.

[27] G. J. Narlikar. A parallel, multithreaded decision tree builder. Technical Report CMU-CS-98-184, Carnegie Mellon University, 1998.

[28] G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Trans. on Programming Languages and Systems*, 21(1), 1999.

[29] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical report, U. Washington, 2000.

[30] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *ASPLOS*, 1996.

[31] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *SPAA*, 1990.

[32] Semiconductor Industry Association. The International Technology Roadmap for Semiconductors (ITRS) 2005 Edition, 2005.

[33] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148. 1996.

[34] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical Report WRL 2001/2, Compaq Computer Corporation, 2001.

[35] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS*, 2000.

[36] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with application to cache partitioning. In *International Conf. on Supercomputing*, 2001.

[37] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, 2002.

[38] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. of Supercomputing*, 28(1), 2004.

[39] D. Thibaut and H. S. Stone. Footprints in the cache. *ACM Trans. on Computer Systems*, 5, 1987.

[40] M. W. Weissmann. Libpmsort. http://freshmeat.net/projects/libpmsort.

[41] S.-H. Yang, B. Falsafi, M. D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA*, 2002.

[42] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA*, 2005.