Performance Implications of Cache Affinity on Multicore Processors

Vahid Kazempour, Alexandra Fedorova, and Pouya Alagheband

Simon Fraser University, Vancouver Canada
vahid_kazempour@sfu.ca, fedorova@cs.sfu.ca,
palagheb@cs.sfu.ca

Abstract. Cache affinity between a process and a processor is observed when the processor cache has accumulated some amount of the process state, i.e., data or instructions. Cache affinity is exploited by OS schedulers: they tend to reschedule processes to run on a recently used processor. On conventional (unicore) multiprocessor systems, exploitation of cache affinity improves performance. It is not yet known, however, whether similar performance improvements would be observed on multicore processors. Understanding these effects is crucial for design of efficient multicore scheduling algorithms. Our study analyzes performance effects of cache affinity exploitation on multicore processors. We find that performance improvements on multicore *uniprocessors* are not significant. At the same time, performance improvements on multicore *multiprocessors* are rather pronounced.

Keywords: multicore processors, cache affinity, performance evaluation, scheduling.

1 Introduction

Our study investigates performance effects of cache affinity on multicore processors. Cache affinity between a process and a processor (or a processing core) is observed when the processor cache has accumulated some amount of the process's state, i.e., its data or instructions. Affinity may be high or low depending on how much state has been accumulated. In modern multiprocessor operating systems, schedulers exploit high cache affinity by scheduling a process on a recently used processor whenever this is possible [11]. When a process runs on a high-affinity processor it will find most of its state already in the cache and will thus run more efficiently [14]. While exploitation of cache affinity is known to improve performance on *unicore* multiprocessors, its effect on *multicore* processors has not been studied. Understanding this effect is crucial for building efficient multicore scheduling algorithms. Many of these algorithms (aiming to improve performance [2,6,9,13], reduce energy consumption [8] or improve thermal regulation [4,12]) work via frequent migrations of processes among CPU cores. Frequent migrations prevent the scheduler from exploiting cache affinity and may thus hurt performance. To understand whether performance may in fact suffer, we study how variations in cache affinity affect performance of applications on multicore processors. Specifically, we compare performance of benchmarks

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 151–161, 2008.

when they run in conditions of high and low cache affinity. These data tell us to what extent performance can be improved by exploiting high cache affinity and to what extent performance may suffer if affinity is not exploited. Our results will help multicore OS designers built better systems.

We analyze performance effects of cache affinity on *multicore uniprocessors* and *multicore multiprocessors* (see Figures 1(a) and 1(b)). The difference between the two is the physical placement of cores with respect to processor caches. In multicore uniprocessors all cores placed on a single chip, typically sharing the L2 cache. In multicore multiprocessors, there are several multicore chips, and the cores on the same chip share a per-chip L2 cache. The importance of studying both kinds of processors is that on the multicore uniprocessor, only the L1 cache affinity may be exploited (because there is only a single L2 cache), so any performance differences would come from L1 affinity. In contrast, on a multicore multiprocessor, both L1 and L2 cache affinity can be exploited because there are multiple L2 caches. Our goal is to understand the impact of both kinds of cache affinity. Processors used in our study are Sun Microsystems UltraSPARC T2000 "Niagara" (a multicore uniprocessor) and Intel Quad-Core E5320 Xeon "Clovertown" that can be configured both as a multicore uniprocessor and a multicore multiprocessor.

In the multicore uniprocessor study, we analyze how performance of applications changes if at each new scheduling quantum they run on a high-affinity core vs. a low-affinity core. Running on a high-affinity core lets the application capitalize on its L1 cache state; running on a low-affinity core requires that the L1 cache state be reloaded. Reloading the L1 cache state on multicore uniprocessors processors is generally inexpensive, however, because it can be usually restored from the low-latency L2 cache. Therefore, we propose a hypothesis that ***failure to exploit L1 cache affinity on multicore uniprocessors has a small effect on performance***.

On multicore multiprocessors, there are two kinds of affinities: affinity to the L1 cache and affinity to the L2 cache. In our multicore multiprocessor study, we focus on the effects of L2 cache affinity, i.e., how the performance of applications changes if they are always scheduled near a high-affinity L2 caches as opposed to being arbitrarily moved between high and low-affinity cores. Here, we are interested in evaluating differences between multicore multiprocessors and conventional (unicore)

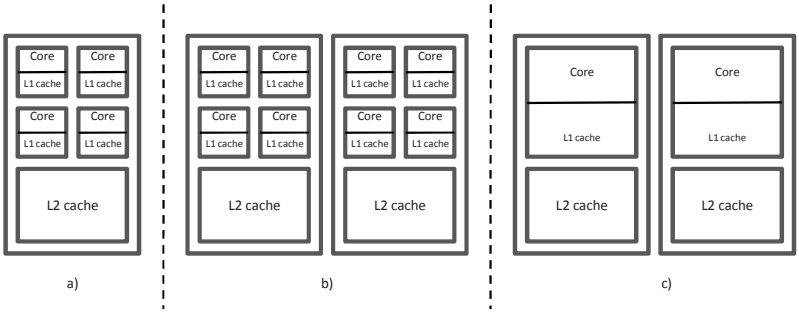

Fig. 1. Schematic view of (a) a multicore uniprocessor consisting of a single chip and four cores, (b) multicore multiprocessor consisting of two multicore chips, and (c) a conventional multiprocessor consisting of two unicore chips

Table 1. Experimental hardware

	Niagara: UltraSPARC T2000	***Clovertown:*** Intel Quad-Core Xeon E5320
Clock Frequency	1.2 GHz	1.86 GHz
L2 cache groups, Cores/group	1, 8	2, 2
L1 caches	8KB D-cache, 16KB I-cache per core	32KB D-cache, 32KB I-cache per core
L2 cache	3MB unified	2x4MB unified

multiprocessors (Figure 1(c)). We hypothesize that ***performance improvements on multicore multiprocessors will be insignificant in comparison with conventional multiprocessors***. On multicore multiprocessors, L2 caches are shared and competition for them can be high. As a result it is harder to retain cache state across invocations of a process. Since the degree of L2 cache state retention will be small, the benefits of exploiting L2 cache affinity will be insignificant.

Our study confirmed the first hypothesis: exploitation of L1 cache affinity has virtually no effect on performance (4% at most in an isolated case). Our second hypothesis, however, was refuted. Although we do find that L2 cache affinity has a smaller performance impact on multicore multiprocessors than on conventional (unicore) multiprocessors, we still observe that the impact is quite significant, especially for applications with large working sets. Those applications experience as much as 27% performance degradation when L2 cache affinity is not exploited on multicore multiprocessors.

The rest of the paper is organized as follows. In Section 2 we present our study on multicore uniprocessors, and in Section 3 on multicore multiprocessors. In Section 4, we discuss related work. In Section 5 we summarize our findings.

2 Multicore Uniprocessors

2.1 Methodology

In this section we restrict our study to multicore uniprocessors (Figure 1(a)). We use two hardware platforms for our study (Table 1). One is Sun Microsystems UltraSPARC T2000 "Niagara" with eight cores and a shared L2 cache. Although each core is multithreaded, we run only one thread per core. Another system is Intel Quad-core Xeon E5320 "Clovertown". Although this processor is built of two dual-core chips, in the experiments of this section we use only one of the chips for running benchmarks, hence we have a multicore uniprocessor.

We experiment with benchmarks from the SPEC CPU2000 suite [1]. We selected eight benchmarks with varied temporal reuse patterns of cached data, because temporal reuse behaviour determines the impact of cache affinity on performance. Our selected benchmarks are *art, crafty, gap, gcc, gzip, mcf, parser, twolf.* Temporal reuse behaviour of benchmarks like *gzip* and *mcf* is good. This causes them to be sensitive

Table 2. Summary of experiments ran for each benchmark

	Low L1 cache affinity		*High L1 cache affinity*
	D-cache	*I-cache*	*D-cache or I-cache*
High L2 retention	Run benchmark interleaved with BV_L1D;	Run benchmark interleaved with BV_L1I;	Run benchmark interleaved with BV_0;
Low L2 retention	Run benchmark interleaved with BV_L1D; Run BV_L2 concurrently	Run benchmark interleaved with BV_L1I; Run BV_L2 concurrently	Run benchmark interleaved with BV_0; Run BV_L2 concurrently

to their cache states, so cache affinity may play an important role in their performance. On the other hand, benchmarks such as *art* have a poor temporal locality. As a result they are less sensitive to their cache states and cache affinity may not be as important.

We run each benchmark in two experimental scenarios: (1) *low affinity* – when no state is retained in the core's L1 cache at each new scheduling quantum, and (2) *high affinity* – when the benchmark's state is almost entirely retained in the L1 cache at each new scheduling quantum. We measure the instructions per cycle (IPC) completed by the benchmark and its cache miss rate. Comparison of these metrics in the low and high-affinity scenarios allows us to gauge the *upper bound* on potential performance improvements from exploiting cache affinity. As such, this experiment's results tell us whether affinity scheduling would matter under *any* affinity-aware scheduling policy, all else being equal.

Here is how we create low and high cache affinity for this experiment. We run the SPEC benchmark together with a *base vector application* [5]. Base vector (BV) is a simple application that can be configured to use a pre-defined cache footprint. We bind the SPEC benchmark and the base vector to the same virtual processor, so their scheduling quanta are interleaved. Therefore, the base vector may displace the cache state of the SPEC benchmark at each quantum. We use two base vectors, BV_L1 and BV_0. BV_L1 has the cache footprint equal to the size of the L1 cache, so it completely displaces the cache state of the SPEC benchmark at each scheduling quantum. This creates the conditions of low cache affinity. BV_0 has the cache footprint of a negligible size, so it leaves the cache state of the SPEC benchmark almost intact. This creates the conditions of high cache affinity.

In addition to varying the degree of L1 cache affinity, we vary other parameters of the experiment, which are discussed below.

The size of the L1 cache. The larger the L1 cache the longer it takes to rebuild its state. Therefore, performance impact from exploiting cache affinity will be higher on systems with larger caches. To account for that, we experiment on systems with different L1 cache sizes (see Table 1).

Retention of state in the L2 cache. Performance effect of L1 cache affinity will depend on if the process reloads its L1 cache state from the L2 cache or from the main memory. To force the reloading from the L2 cache we run our experiments in conditions of *high L2 cache retention*. To force the reloading from the main memory, we run with *low L2 cache retention.* To create low L2 cache retention we run the SPEC

benchmark concurrently with a base vector application configured to have a working set equal to the size of the L2 cache (BV_L2). BV_L2 is run on a different core than the main benchmark, but that core shares the L2 cache with the core on which the SPEC benchmark is run. Therefore, when the SPEC benchmark is de-scheduled, BV_L2 completely displaces its L2 cache state. For high L2 cache retention, we run *no* BV_L2 with the SPEC benchmark.

Type of cache. Performance effects of cache affinity may differ for L1 *data* cache and L1 *instruction* cache. Therefore, we experiment with these caches separately. We vary cache affinity using different base vectors for L1 I-cache and L1 D-cache. BV_L1I is a base vector configured with the I-cache footprint equal to the size of the I-cache and a negligible D-cache footprint. BV_L1D is a base vector configured with the D-cache footprint equal to the size of the D-cache and a negligible I-cache footprint.

Scheduling time quantum. When affinity is not exploited, the process must reload its cache state at each new scheduling quantum. Per-core L1 caches tend to be small, thus reloading them is cheap. So the failure to exploit L1 cache affinity will hurt performance only when the L1 cache state must be reloaded very frequently (i.e., when the scheduling time quantum is small). Therefore, while a large time quantum amortizes the penalty of reloading the cache, a small quantum causes the negative performance effects to be more pronounced. The time quantum assigned to a thread by the scheduler depends on the thread's workload characteristics. I/O intensive or interactive programs will have shorter time quanta (often as short as a few milliseconds), while CPU-bound programs may be assigned time quanta of hundreds of milliseconds [11]. To account for this variation, we used three different scheduling quanta in our experiments: two, ten and 200 milliseconds.

Pipeline architecture. Due to the ability of masking the cache miss latency with dynamically scheduling instructions, deep out-of-order pipelines have a higher tolerance to L1 cache misses than shallow in-order pipelines Therefore, performance effects of cache affinity could vary depending on the pipeline architecture. In our study we consider processors with both deep super-scalar out-of-order pipelines (Clovertown) [10] and shallow in-order single-issue pipelines (Niagara) [7].

For each SPEC benchmark we run six experiments shown in Table 2. Further, we run each experiment from Table 2 with the three scheduling quanta on the two experimental machines. This gives us 288 experiments in total. We run each experiment three times.

2.2 Results

We found that in most cases retaining L1 cache affinity had no measurable impact on performance. There was only a single experimental scenario where exploiting L1 cache affinity resulted in measurable performance difference. This was the case where the cache was large, the scheduling quantum was small, and the L2 cache retention was low. We observed it in the experiment with the Intel system (that has a larger 32KB instruction cache) with low L2 cache retention and a timeslice of 2ms (the experiment from the second row and the second and third columns in Table 2). Figure 2 shows the average IPC and Figure 3 the misses per instruction (MPI) of this

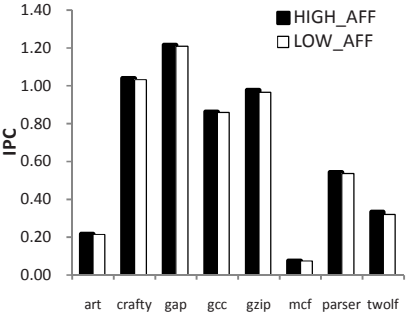

Fig. 2. IPC on the Intel system. Low L2 retention, 2ms time quantum.

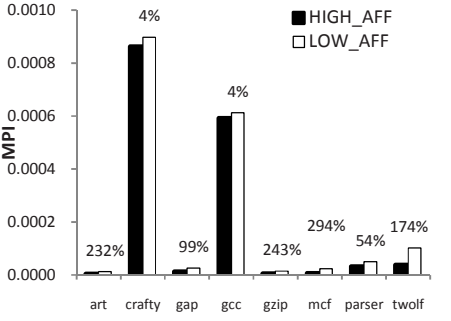

Fig. 3. MPI on the Intel system. Low L2 retention, 2ms time quantum.

experiment. Black bars show the high-affinity scenario (the benchmark runs with BV_0), white bars show the low-affinity scenario (the benchmark runs with BV_L1I). *Twolf* is the only benchmark that experienced a statistically significant decrease in the IPC when the cache affinity was low. *Twolf*'s IPC dropped by 4%, accompanied by a 174% increase in the miss rate. All other benchmarks experienced IPC degradation of less than 1%.

Referring now to Figure 3, where the difference in the I-cache miss rates between the low and high-affinity scenarios is shown on top of each pair of bars, we note that although five out of eight benchmarks experienced dramatic increases in miss rates (from 99% for gap to 294% for mcf), their IPC stayed unchanged. The reason is that the cache miss rates of these benchmarks were small in absolute terms, so they had little effect on performance. These benchmarks have a good reuse of their I-cache, and that is why their cache miss rates are low. That is also why their miss rates skyrocket if cache affinity is not exploited – good cache reuse implies sensitivity to variations in the cache affinity. Benchmarks whose cache miss rates are high, such as *crafty* and *gcc*, are not sensitive to variations in the cache affinity, because their cache reuse is low.

As to the rest of our experimental scenarios, on both systems, with all scheduling quanta and L2 retention levels, we observed no statistically significant effect on the IPC as the degree of affinity varied. We did not observe any positive effects on the Intel system due to the ability of its out-of-order pipeline to mask cache latency (Niagara has an in-order pipeline). Both on Intel and Niagara systems cache affinity effects were negligible with the exception of the case reported above.

We have two explanations for these results: in the case when the L1 cache is reloaded from the L2 cache (high L2 retention) reloading the entire L1 cache is cheap, even with a small scheduling timeslice. In the case when the L1 cache is reloaded from the main memory (low L2 cache retention), the performance impact is small as well, because L1 caches tend to be small, and reloading them is cheap. On the Intel platform with larger L1 caches performance penalty is also masked by aggressive prefetching.

These results confirm our hypothesis that exploiting L1 cache affinity has negligible performance effect on multicore uniprocessors.

Table 3. Benchmark combinations

MMP		*UMP*	
Main	*Interfering*	*Main*	*Interfering*
2x art	crafty	4x art	2x crafty
2x crafty	gap	4x crafty	2x gap
2x gap	gcc	4x gap	2x gcc
2x gcc	gzip	4x gcc	2x gzip
2x gzip	mcf	4x gzip	2x mcf
2x mcf	parser	4x mcf	2x parser
2x parser	twolf	4x parser	2x twolf
2x twolf	art	4x twolf	2x art

3 Multicore Multiprocessors

In this section we study multicore multiprocessors (Figure 1(b)) and compare them to conventional multiprocessors (Figure 1(c)). For a multicore multiprocessor (MMP), we used the Intel system shown in Table 1, but unlike in the previous section, we configured it to use both chips. For a conventional unicore multiprocessor (UMP), we configured the Intel system to use only one core per chip, effectively creating a conventional two-way multiprocessor. We could use Niagara for the experiments in this section, because (by virtue of having only a single chip) it cannot be configured as a multicore multiprocessor.

We compare performance of multiprogram workloads running in conditions of high core/chip affinity and in conditions of low core/chip affinity. To create high-affinity conditions, the measured benchmark is bound to a processing core for the duration of the run. Therefore, it is always rescheduled to run on a high-affinity core. To create low-affinity conditions, the benchmark is not bound to a particular core and may thus be rescheduled to run on any core or chip, including the ones of low affinity. In the low-affinity conditions, we use the Solaris time-sharing scheduler with affinity settings disabled. This experiment thus lets us compare performance achievable by the ideal affinity-aware scheduler with the performance achieved by the affinity-oblivious scheduler. As we learned from the previous section, L1 cache affinity makes no difference for performance, so if there are any performance gains they would be due to L2 cache affinity. Therefore, in the rest of this section we talk about evaluating the effects of L2 cache affinity.

In each experiment, we use two groups of benchmarks: main benchmarks and interfering benchmarks. Main benchmarks are those whose performance we measure. Table 3 shows main and interfering benchmarks for the MMP and UMP experiments. Each benchmark gets to be in the main and in the interfering role. This creates a wide range of experimental scenarios with respect to cache reuse patterns and degrees of contention. We run several copies of the *same* main benchmark, to avoid measuring any effects due to cache contention and isolate the effects of cache affinity only.

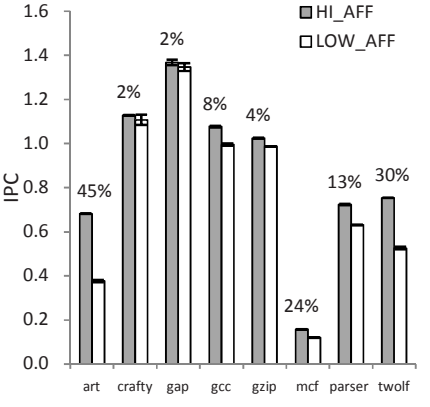

Fig. 4. IPC on UMP

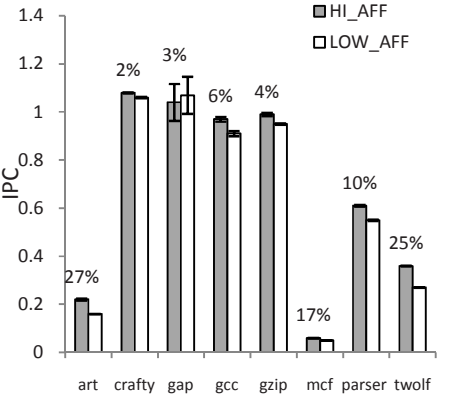

Fig. 5. IPC on MMP

An experiment consists of running a group of benchmarks concurrently (from Table 3), on the MMP and on the UMP configuration. For each configuration we run two experiments: the high-affinity experiment, where the main benchmarks are bound each to its own core, and the low-affinity experiments where they are not bound. Interfering benchmarks are not bound to any core in any experiment.

We run the benchmarks with a 2ms scheduling time quantum, because this is where cache affinity has the most impact – our goal is to measure the upper bound. We run each main benchmark three times to measure the IPC and another three times to measure the L2 cache miss rate. (Separate runs were needed because only a single hardware counter was operational on our Intel system.)

Figures 4 and 5 show the IPC of the main benchmarks on UMP and MMP systems with high affinity (HI_AFF) and with low affinity (LOW_AFF). (Note the difference in scale on the Y-axis.) The bars indicate average values, and the whiskers denote two standard deviations. The percent values at the top of each pair of bars show the decrease in performance between the high-affinity and low-affinity scenarios. Figures 6 and 7 show the corresponding L2 cache miss rates. (Again, note the difference in scale on the Y-axis). The percentage values show the difference in misses per instruction between the high-affinity and low-affinity scenarios.

We note three things about the data. First, on both systems performance noticeably degrades when affinity is low. (Even though the *gap* benchmark on MMP appears to run more quickly with low affinity, this result is not statistically significant as indicated by the whiskers). The most significant degradation in performance is experienced by memory bound benchmarks: *art*, *mcf*, *parser* and *twolf*. Those benchmarks are more dependent on good L2 cache performance than the rest of the benchmarks.

Second, the performance impact of low affinity on the UMP system is greater than on the MMP system for all benchmarks without exception. On the UMP system, performance decreases due to low affinity by as much as 45% (for *art*) and by 16% on average for all benchmarks. On the MMP system, corresponding performance degradation is 27% (for *art*) and 11% on average for all benchmarks. This difference is explained by more dramatic increases in the L2 cache miss rates on the UMP system.

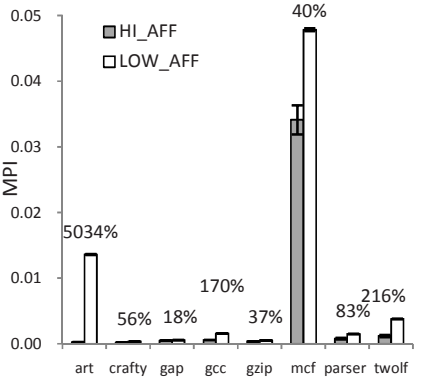

Fig. 6. MPI on UMP

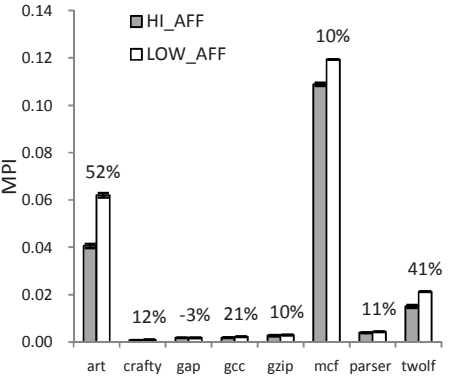

Fig. 7. MPI on MMP

Art's miss rate increases by a factor of 51, and *gcc*'s and *twolf*'s miss rates increase by more than a factor of two. On the MMP system, all benchmarks' miss rates increase by less than a factor of two. A smaller impact of cache affinity on the MMP system could be explained by the presence of contention for the L2 cache (in contrast with the UMP system), and thus there is a lower retention of the L2 cache state across process invocations and a smaller cache miss degradation when cache affinity is not preserved. As a result, the benefits from exploiting cache affinity are lower.

Finally, we note that although performance effects of high L2 cache affinity are smaller on the MMP system than on the UMP system, they are still significant to merit inclusion of affinity awareness in multicore scheduling algorithms.

4 Related Work

Torrellas et al. studied performance effects of affinity aware scheduling on conventional (unicore) multiprocessors [14]. Affinity aware scheduling reduced cache miss rates by 7-36% and improved performance by as much as 10%, according to their study. We pursued a similar goal, but targeted multicore processors. In addition, we answered a slightly different question. Unlike the Torrellas's study that measured performance impact of a particular affinity-aware scheduling algorithm, we evaluated the *upper bound* on performance gains achievable by exploiting cache affinity.

Constantinou et al. considered performance effects of migrating a process among cores on a multicore processor [3]. They studied performance effects of warming up L1 instruction and data caches on the new core before migrating the process to that core (as opposed to leaving the caches cold). Warming up the caches creates affinity between the core and the migrated process. Therefore, experiments in the Constantinou's study effectively measured the effects of exploiting L1 cache affinity, which is similar to what we did. The key differences of our study are in the experimental conditions. While Constantinou's study varied cache affinity by warming up the caches using special hardware, we varied it by means of interfering applications in a multi-program workload. Therefore, while Constantinou's study served the purpose of evaluating migration-friendly hardware architectures, our results are applicable to

scheduling. Constaninou's study evaluated deep out-of-order pipelines *only*. These pipelines are more tolerant to cache misses than in-order pipelines. We evaluated systems with both a shallow in-order pipeline and a super-scalar out-of-order pipeline. Finally, we also compared performance gains from exploitation of cache affinity on multicore multiprocessors to conventional multiprocessors. To the best of our knowledge, this has not been done in the past.

5 Conclusions

We evaluated performance effects of exploiting cache affinity on multicore processors. We studied both multicore uniprocessors and multicore multiprocessors, and evaluated both the effects of exploiting L1 cache affinity and the effects of exploiting L2 cache affinity. We hypothesized that cache affinity does not affect performance on multicore processors: on multicore uniprocessors — because reloading the L1 cache state is cheap, and on multicore multiprocessors – because L2 cache affinity is generally low due to cache sharing. Our first hypothesis was confirmed. Exploiting cache affinity on multicore uniprocessors has no measurable impact on performance even when the L1 cache is relatively large, scheduling time quantum is small and L2 cache retention is low. Our second hypothesis, on the other hand, was refuted. Even though upper-bound performance improvements from exploiting cache affinity on multicore multiprocessors are lower than on unicore multiprocessors, they are still significant: 11% on average and 27% maximum. This merits consideration of affinity awareness on multicore multiprocessors.

We conclude that affinity awareness in multicore scheduling algorithms will make no difference on multicore uniprocessor systems, but will improve performance on multicore multiprocessors. We hope that our results will help multicore OS designers build better systems.

References

[1] SPEC CPU, web site (2000), http://www.spec.org
[2] Becchi, M., Crowley, P.: Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In: Proceedings of the Conference on Computing Frontiers (2006)
[3] Constantinou, T., Sazeides, Y., Michaud, P., Fetis, D., Seznec, A.: Performance Implications of Single Thread Migration on a Chip MultiCore. In: Proceedings of the Workshop on Design, Architecture and Simulation of Chip Multi-Processors (2005)
[4] Coskun, A., Rosing, T.: Temperature aware task scheduling in MPSoCs. In: Proceedings of the DATE (2007)
[5] Doucette, D., Fedorova, A.: Base Vectors: A Potential Technique for Microarchitectural Classification of Applications. In: Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-34 (2007)
[6] Fedorova, A., Vengerov, D., Doucette, D.: Operating System Scheduling On Heterogeneous Multicore Systems. In: Proceedings of the PACT 2007 Workshop on Operating System Support for Heterogeneous Multicore Architectures (2007)

[7] Kongetira, P.: A 32-way Multithreaded SPARC(R) Processor. In: Proceedings of the 16th Symposium On High Performance Chips (HOTCHIPS) (2004)
[8] Kumar, R., Farkas, K., Jouppi, N., Parthasarathy, R., Tullsen, D.M.: Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (2003)
[9] Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N., Farkas, K.: Single-ISA Heterogeneous Multicore Architectures for Multithreaded Workload Performance. In: Proceedings of the 31st Annual International Symposium on Computer Architecture (2004)
[10] Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A., Upton, M.: Hyper-threading Technology Architecture and Microarchitecture. Intel Technical Journal 6(1), 4–15 (2002)
[11] McDougall, R., Mauro, J.: Solaris™ Internals: Solaris 10 and OpenSolaris Kernel Architecture. Prentice Hall, Englewood Cliffs (2006)
[12] Powell, M.D., Gomaa, M., Vijaykumar, T.N.: Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System. In: Proceedings of the ASPLOS (2004)
[13] Snavely, A., Tullsen, D.M.: Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2000)
[14] Torrellas, J., Tucker, A., Gupta, A.: Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. Journal Of Parallel and Distributed Computing 24, 139–151 (1995)