# Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processors

Matthias Diener[1], Felipe L. Madruga[2], Eduardo R. Rodrigues[2], Marco A. Z. Alves[2], Jörg Schneider[1], Philippe O. A. Navaux[2], and Hans-Ulrich Heiß[1]

[1] Fakultät IV - Elektrotechnik und Informatik
Technische Universität Berlin
Berlin, Germany
{mdiener, komm, heiss}@cs.tu-berlin.de
[2] Institute of Informatics
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
{flmadruga, errodrigues, mazalves, navaux}@inf.ufrgs.br

**Abstract.** Process placement is a technique widely used on parallel machines with heterogeneous interconnections to reduce the overall communication time. For instance, two processes which communicate frequently are mapped close to each other. Finding the optimal mapping between threads and cores in a shared-memory environment (for example, OpenMP and Pthreads) is an even more complex task due to implicit communication. In this work, we examine data sharing patterns between threads in different workloads and use those patterns in a similar way as messages are used to map processes in cluster computers. We evaluated our technique on two state-of-the-art multi-core processors and achieved moderate improvements in the common case and considerable improvements in some cases, reducing execution time by up to 45%.

**Key words:** thread placement, memory access patterns, process mapping, shared cache, multi-core processor

## 1 Introduction

Due to limits on instruction-level parallelism, high power consumption and wire-delay problems of sequential cores, the multi-core architecture is the current choice for high-performance processors. The prediction is that for the next chip generations the number of cores will increase drastically, going from multi-core to many-core [1].

The task of finding the optimal mapping between threads and processors is a NP-hard problem [8]. Usually, graphs are used to represent both architecture and application behavior. In environments where message-passing is the main paradigm to build parallel programs (such as cluster computers), constructing the task graph where vertices represent communication is straightforward. In a previous work [10], it was shown that optimizing thread placement in cluster computers using multi-core machines improves performance. The communication pattern is obtained by monitoring the messages, which contain information about sender and receiver, and using them to calculate the amount of data exchanged between tasks.

However, the thread-placement approach for cluster computers has to be adapted to be used in shared memory applications. In such applications, the communication is not explicit. Therefore, monitoring data accesses is the only way to analyze the interaction between threads and the demands on cache memory.

There are three objectives in optimizing thread placement in multi-core systems. First, make better use of interconnections, i. e. reduce off-chip traffic by using intra-chip interconnections. Second, reduce cache misses when two private caches hold the same data and are continuously invalidated by the respective other cache. These kinds of cache misses are called *invalidation misses*. Third, reduce competition for cache lines between processors that share a cache, which causes cache misses called *compulsory misses*.

Our goals in this work are to investigate whether optimizing thread placement has an influence on performance and to evaluate techniques to place threads. To achieve these goals, we used simulation tools and tests on real machines with a variety of benchmarks. Our focus is to optimize the use of memory hierarchies and interconnections, but not the optimization of the usage of the execution units when threads share them.

Existing approaches analyze cache statistics gathered throughout the execution, therefore making them dependent on the architecture. In our study we observed the memory accesses of each thread, regardless of cache parameters (such as line size and associativity), thus separating the program's behaviour from the architecture. We implemented a new mechanism to transform memory accesses from different threads to communication patterns and used them to place threads that share data on cores that share levels of cache, thereby matching the program's behaviour with the cache organization of the architecture.

In the experiments, using our thread mapping algorithms, the execution time was reduced by up to 45% while also reducing the variance. The reduction of variance is important because it helps to predict the execution time.

The remainder of this document is organized as follows: In Section 2, we present a motivation for optimizing thread placement on multi-core by using a producer/consumer benchmark. We explain the way traces are converted to a sharing metric and the algorithms used to place threads in Section 3. In Section 4, the evaluation methodology, multi-core architectures and tools we used are presented. Results of our tests on two different architectures are shown and analyzed in Section 5. In Sections 6, related work is discussed. Finally, Section 7 summarizes conclusions and outlines future work.

## 2 Motivation

In this section, the significant influence of the thread placement strategy on the performance of multi-core systems is shown. This is done by running a producer/consumer benchmark with different thread placements and observing execution time and cache statistics.

Consider a processor with four cores, where each pair of cores shares a L2 cache. If two threads sharing data are placed on cores that do not share the cache, the overall execution time tends to be higher than when they run on cores which share the cache, because a slower (in terms of both latency and bandwidth) interconnection will be used.

A common situation in shared-memory programs is to have one thread writing to an area of memory and another reading from the same area. In an invalidation-based coherency protocol, like MESI, this can cause one thread to successively invalidate the cache lines of the other thread, thereby causing cache misses due to invalidations.

Another problem is when there are memory hungry threads sharing a cache, therefore evicting lines from each other. These compulsory cache misses are one of the reasons for decreased performance in multi-core systems.

To estimate the impact of thread placement on the execution time and cache misses, we created a synthetic benchmark using the OpenMP API which consists of two pairs of threads, each pair having a writer and a reader thread. The writer thread writes $N$ times a vector of $K$ integers and the reader thread reads each element of that vector. The vector is protected by a lock so that a reader thread only accesses the vector after it has been written to.

When using a multi-core machine with four cores, each two sharing an L2 cache, it is easy to find the best and worst placement in terms of cache sharing: a best configuration, where each pair of threads can make use of the shared L2 cache; and a worst configuration, where each pair of threads is running on cores that do not share the L2 cache.

For the simulations we used Virtutech Simics [9], a full system simulator on the instruction set level. As explained above, we simulated with a focus on two types of cache misses: on cache misses due to invalidation and on compulsory cache misses. The parameters we used for these two configurations are presented in Table 1.

In order to compare the number of cache misses between the best and the worst configuration, we modeled the cache layout for the Simics virtual machine based on the Intel Xeon 5405, a quad-core processor which has 6 MByte L2 cache shared between each pair of cores. The cache and memory latencies were calculated using the CACTI memory modeling tool [14].

The results in Table 2 show that optimizing the thread placement improves the execution time and reduces the number of cache misses and MESI invalidations greatly.

| | Focus on cache misses due to invalidations | Focus on compulsory cache misses |
|---|---|---|
| Vector size | 256 KByte | 4 MByte |
| Number of integers in the vector ($K$) | 65536 | 1048576 |
| Number of iterations ($N$) | 100 | 10 |

**Table 1.** Parameters of the synthetic benchmark configurations

| | Focus on cache misses due to invalidations | | Focus on compulsory cache misses | |
|---|---|---|---|---|
| | best case | worst case | best case | worst case |
| Execution time | 2.576 s | 3.086 s | 4.156 s | 5.127 s |
| Speedup | 19.8% | | 23.4% | |
| L2 cache misses | 1.5% | 91.2% | 5.7% | 98.5% |
| MESI invalidations | 3928 | 417496 | 14539 | 589890 |

**Table 2.** Results of executing the synthetic benchmark in Simics.

## 3 Data Sharing Metric and Placement Algorithms

To be able to place threads according to the amount of data sharing between them, a metric which quantifies this sharing is needed. In this section, we introduce a new metric and propose two new thread placement algorithms using it.

### 3.1 Data Sharing Metric

We propose to use the number of accesses to the same memory locations by two threads as a metric for how much two threads access shared data. We constructed a matrix with the data sharing metric of each pair of threads, referred to in the rest of the paper as a communication matrix.

The communication matrix can be generated as follows: In the beginning, collect the number of memory accesses (both reads and writes) to each address for every thread. Then, calculate the data sharing for each pair of threads by adding up the number of accesses to the same addresses by the thread pair.

For each workload, we generated a communication matrix with the memory access traces generated in the simulation. An example of a communication matrix for the advection workload is shown in Table 3. Each table cell contains the number of accesses to equal memory addresses by two threads, in millions.

| Thread ID | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | | 2.1 | 2.1 | 4.4 | 4.7 | 2.1 | 2.1 | 1.9 |
| 1 | 2.1 | | 2.1 | 2.1 | 2.1 | 3.2 | 2.1 | 1.9 |
| 2 | 2.1 | 2.1 | | 2.1 | 4.8 | 3.5 | 2.1 | 1.9 |
| 3 | 4.4 | 2.1 | 2.1 | | 2.2 | 2.3 | 2.2 | 3.1 |
| 4 | 4.7 | 2.1 | 4.8 | 2.2 | | 4.7 | 4.1 | 2.2 |
| 5 | 2.1 | 3.2 | 3.5 | 2.3 | 4.7 | | 6.9 | 3.1 |
| 6 | 2.1 | 2.1 | 2.1 | 2.2 | 4.1 | 6.9 | | 2.4 |
| 7 | 1.9 | 1.9 | 1.9 | 3.1 | 2.2 | 3.1 | 2.4 | |

**Table 3.** Example of a communication matrix.

In this example, the number of data accesses to the same memory area by threads 5 and 6 was 6.9 million, by threads 2 and 7 it was 1.9 million. Therefore, it would be better in terms of sharing to place threads 5 and 6 on two cores which share the cache than threads 2 and 7.

### 3.2 Placement Algorithms

In this section, we describe the two different placement algorithms we developed to optimize the thread placement on the cores. These algorithms use the communication matrix described in the last section.

**Heuristic Algorithm** The heuristic algorithm consists of two steps: First, order all possible pairs of threads according to the data sharing metric measured between them. Second, for each pair in the sorted list, put the two threads on two cores which share the cache and remove the two threads from the list.

The advantage of this algorithm is that it is very fast; typical execution time is less than one second, even when placing 32 threads. On the other hand, it only considers pairs of threads, not bigger groups. This leads to suboptimal behavior when there are groups of three or four threads sharing lots of memory accesses among themselves. Additionally, this algorithm leads to an uneven distribution of threads when the number of threads is not dividable by the number of cores; this leads to an increase in execution time.

**Exhaustive Search** An approach to find the best thread placement in terms of data sharing is to try every possible placement, which is done in this algorithm. It works as follows: First, generate all possible combinations of thread placements, taking into account the symmetry of the architecture. This has to be done only once for each CPU architecture and number of threads. Then, calculate the gain of each placement by adding up the numbers from the communication matrix for each thread pair. Finally, choose the thread placement with the highest gain.

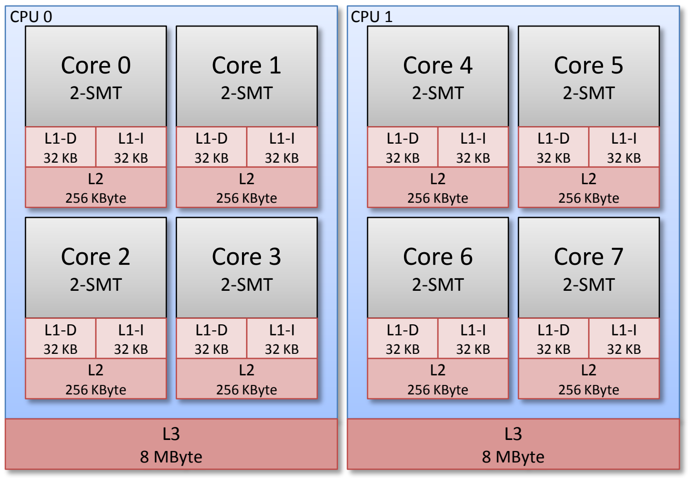


**Fig. 1.** Intel Nehalem architecture

The big disadvantage of this algorithm is that it is only feasible for a small number of threads ($\leq 16$), because the number of combinations is very high. In our tests, it took about 1 hour to find the thread placement for 16 threads, and we estimated that it would take years to find the placement for 32 threads.

We compared the results of these two placement algorithms with the results of running the workloads when the threads were scheduled by the operating systems. In the result section, the heuristic algorithm, the exhaustive search and the operating system scheduler are labeled as HEUR, EXH and AUTO respectively.

## 4 Methodology

For the evaluation of thread placement, we used two state-of-the-art computer architectures, Intel Nehalem (Core i7) [2] and Sun Niagara 2 (UltraSPARC T2) [11]. They are described in this section. In addition, we explain how we generated the memory access traces needed to find the data sharing patterns and introduce the workloads used to measure the impact of optimizing the thread placement.

### 4.1 Architectures

**Intel Nehalem** We used two physical processors with four cores and eight megabytes of L3 cache each. Each core has 256 KBytes of L2 cache and is able to execute two threads in parallel using simultaneous multithreading (SMT). Figure 1 depicts the Nehalem architecture.

We ran the workloads with two threads to measure the influence of sharing the cache on the performance. For two threads, there are three possible placements:

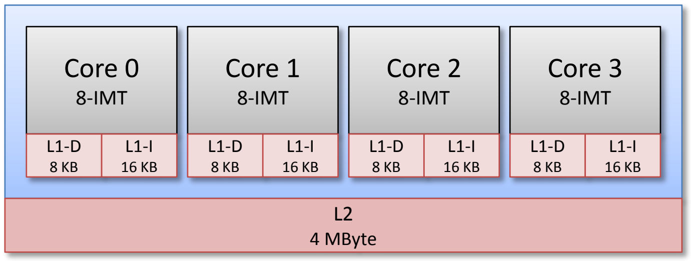


**Fig. 2.** Sun Niagara 2 architecture

- **Different processors** Placing the threads on different processors makes the threads unable to share data via the L2 or L3 cache.
- **Same core** Placing the threads on the same core enables them to share data via the L2 and L3 cache, but makes them compete for resources on the core.
- **Same processor, different core** Placing the threads on the same processor, but not on the same core, enables the threads to share data via the L3 but not the L2 cache.

In the result section, the thread placements for two cores are labeled DIFF CPU, SAME CORE and SAME CPU respectively. The workloads were compiled with GCC 4.3.3 and executed on Ubuntu 9.04 with the kernel version 2.6.28-11.

**Sun Niagara 2** We used one processor consisting of four cores, each of whom is able to execute eight threads using interleaved multithreading (IMT). Each core is divided into two thread groups, which can execute one thread at a time. All cores share a L2 cache of 4 MByte, which is divided into 4 banks. Figure 2 depicts the Niagara 2 architecture.

As in the case, we measured the influence of sharing the cache on the performance by running the workloads with two threads. Again, there are three possible placements for two threads:

- **Different core** When placed on different cores, the threads can share data via the L2 cache.
- **Same core, different thread group** Threads can share data via the L1 and L2 cache, but they start to compete for resources.
- **Same thread group** Threads can share data via the L1 and L2 cache, but since only one thread can run in a thread group at the same time, the competition for resources increases.

In the result section, the thread placements for two cores are labeled DIFF CORE, DIFF TG and SAME TG respectively. The workloads were compiled with GCC 4.3.3 and executed on Solaris 10.

### 4.2 Memory Access Traces

To be able to place threads, we have to find the data sharing patterns between the threads. We obtained these patterns by using Simics, running the workloads in a simulated UltraSPARC machine. We used special instructions provided by Simics (called magic instructions) to register the threads created by the workload with the simulation environment and enable the simulator to track the memory accesses of each thread and record them to a file, which was processed with the algorithm described in section 3.1 to generate a communication matrix.

### 4.3 Workloads

In order to analyze the impact of thread placement, we selected a group of parallel programs with different parallelization schemes and data sharing behaviour. Two well known scientific benchmarks, two emerging applications and a kernel from a weather forecasting model were used. All workloads were compiled with the default compiler flags specified in their respective makefiles.

From the SPLASH2 benchmark suite [15], the LU kernel was used. It comprises the factorization of a dense matrix as a lower triangular and an upper triangular matrix. The matrix is divided in blocks and these are distributed among the threads. We used the two available versions: contiguous (each block is allocated contiguously) and non-contiguous.

Also from SPLASH2, the parallel version of the complex 1-D FFT algorithm was chosen. It shows all-to-all communication throughout the calculation. Since the inter-thread communication graph is not easily separated in clusters, thread placement is not intuitively beneficial.

Dedup from the PARSEC benchmark suite [3] is a kernel that implements a technique called deduplication to compress a datastream. In its parallel phase, it uses three pipeline stages to divide the work. This means that the number of threads created is three times greater than the number of indicated when running the program. Because of that, we did not run Parsec with just two threads (this mapping is impossible) and with the exhaustive search placement (exhaustive search unable to complete in realistic run time).

As another emerging problem from the PARSEC benchmark suite, Streamcluster solves the online clustering, which is a data mining problem. An important characteristic is that it is memory bound for small input sizes.

Advection [6] is a part of the Brazilian Regional Atmospheric Modeling System, a weather forecast program. It uses finite difference methods to compute scalar and vector fields interaction. Basically, the work is evenly divided by the number of threads and they access common ghost zones.

In Table 4 the problem size, which describes the input data, and the memory usage for each benchmark is shown. The memory usage varies between 10 MByte and 3 GBytes, indicating different cache utilization scenarios; in the case of Streamcluster, the application data fits into the cache almost completely, in the other cases, the application data is much bigger than the cache. The memory usage of all workloads was small enough to fit into the main memory of the test machines.

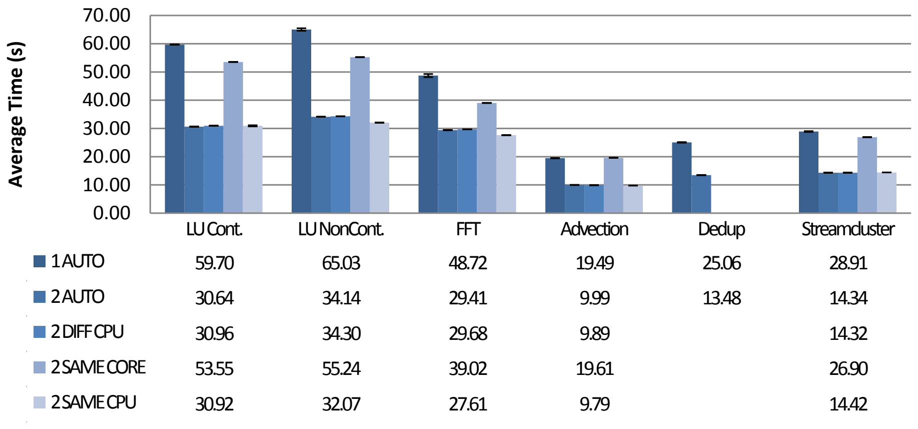
| Benchmark | Problem size | Memory usage |
|---|---|---|
| LU | 3072*3072 matrix | 75 MByte |
| FFT | 67108864 complex doubles | 3.0 GByte |
| Dedup | File of 1 GByte | 1.9 GByte |
| Streamcluster | 16384 points | 10 MByte |
| Advection | 400*400 grid | 1.1 GByte |

**Table 4.** Problem size and memory usage of the benchmarks.

## 5 Results and Analysis

In this section we present the results of our experiments and analyze them. For each architecture, we analyzed first the performance difference of running two threads on two cores with and without a shared cache. Then, we analyze the results of our proposed approach for 8, 16 and, in the case of the Sun Niagara 2, 32 threads, comparing them to the operating system scheduler. We show both the average execution time of 50 runs and the confidence interval for a confidence level of 90% in a Student's t-distribution.

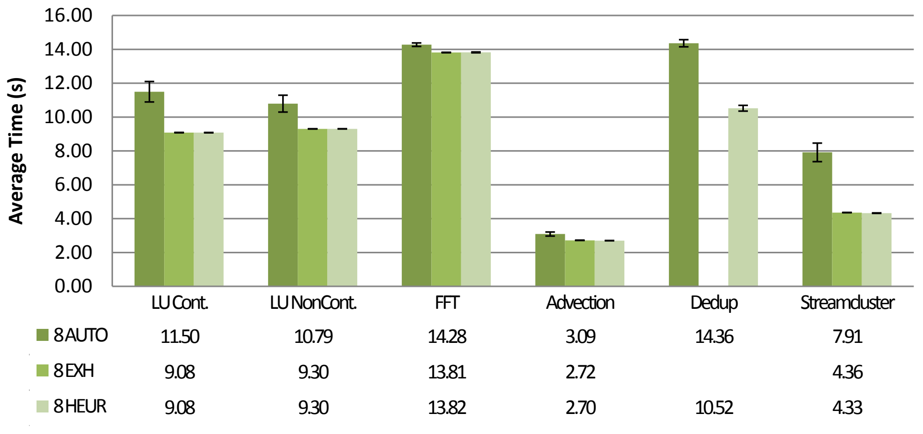
### 5.1 Intel Nehalem



| | LU Cont. | LU NonCont. | FFT | Advection | Dedup | Streamcluster |
|---|---|---|---|---|---|---|
| ■ 1 AUTO | 59.70 | 65.03 | 48.72 | 19.49 | 25.06 | 28.91 |
| ■ 2 AUTO | 30.64 | 34.14 | 29.41 | 9.99 | 13.48 | 14.34 |
| ■ 2 DIFF CPU | 30.96 | 34.30 | 29.68 | 9.89 | | 14.32 |
| ■ 2 SAME CORE | 53.55 | 55.24 | 39.02 | 19.61 | | 26.90 |
| ■ 2 SAME CPU | 30.92 | 32.07 | 27.61 | 9.79 | | 14.42 |

**Fig. 3.** Results for one and two threads on Intel Nehalem

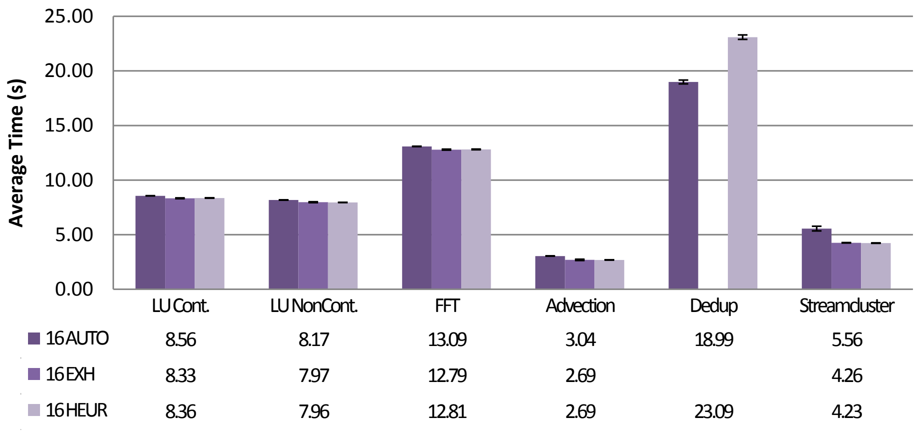
The results shown in Figure 3 show the results for 1 and 2 threads. As expected, when the two threads are running on the same core, the performance decreases drastically because the threads are competing for execution units. The other thread placements perform roughly twice as fast as running with just one thread. The results for the automatic scheduling are very close to the results with optimized thread placement, which suggests that the OS scheduler is aware of this performance problem and does not schedule two threads on the same core unless necessary.



| | LU Cont. | LU NonCont. | FFT | Advection | Dedup | Streamcluster |
|---|---|---|---|---|---|---|
| 8 AUTO | 11.50 | 10.79 | 14.28 | 3.09 | 14.36 | 7.91 |
| 8 EXH | 9.08 | 9.30 | 13.81 | 2.72 | | 4.36 |
| 8 HEUR | 9.08 | 9.30 | 13.82 | 2.70 | 10.52 | 4.33 |

**Fig. 4.** Results for eight threads on Intel Nehalem

Figure 4 shows the result for 8 threads running on Intel Nehalem. In this configuration, we achieved the biggest performance increases in our tests. All benchmarks except FFT show a significant reduction in execution time, Streamcluster reducing it by 45% while also reducing the variance.



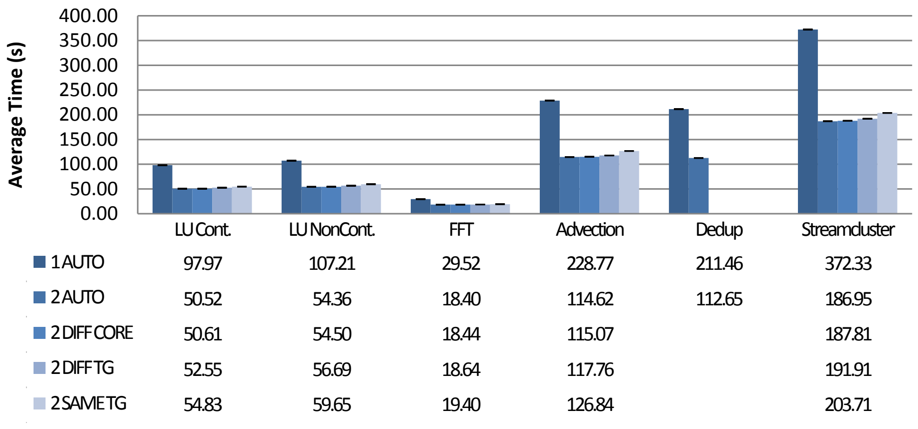
| | LU Cont. | LU NonCont. | FFT | Advection | Dedup | Streamcluster |
|---|---|---|---|---|---|---|
| 16 AUTO | 8.56 | 8.17 | 13.09 | 3.04 | 18.99 | 5.56 |
| 16 EXH | 8.33 | 7.97 | 12.79 | 2.69 | | 4.26 |
| 16 HEUR | 8.36 | 7.96 | 12.81 | 2.69 | 23.09 | 4.23 |

**Fig. 5.** Results for 16 threads on Intel Nehalem

Running with 16 threads, the results as presented in Figure 5 are different. Optimizing the thread placement actually increases the execution time of dedup, while the LU and FFT benchmarks show no difference in performance. Advection and Streamcluster have an improvement of about 10% and 25%, respectively. Additionally, the performance of all benchmarks is about the same or even worse than when running with 8 threads. The reason for this behaviour is that the threads are starting to compete for the execution units on the cores.

To summarize, optimizing thread placement on Intel Nehalem decreased execution time and variance in almost all our tests. However, increasing the number of threads from 8 to 16 has no benefits and can actually lead to worse results. Additionally, our experiments showed that the heuristic algorithm has a similar performance benefit as the exhaustive search algorithm with a drastically lower run time.

## 5.2 Sun Niagara 2



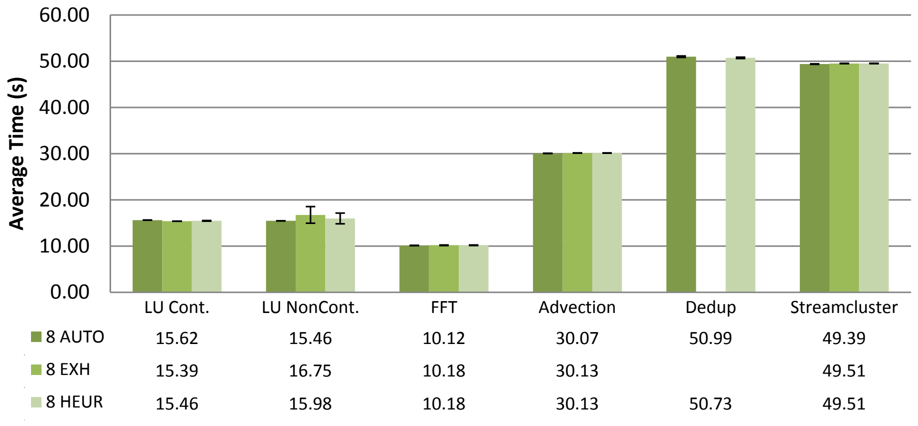
| | LU Cont. | LU NonCont. | FFT | Advection | Dedup | Streamcluster |
|---|---|---|---|---|---|---|
| ■ 1 AUTO | 97.97 | 107.21 | 29.52 | 228.77 | 211.46 | 372.33 |
| ■ 2 AUTO | 50.52 | 54.36 | 18.40 | 114.62 | 112.65 | 186.95 |
| ■ 2 DIFF CORE | 50.61 | 54.50 | 18.44 | 115.07 | | 187.81 |
| ■ 2 DIFF TG | 52.55 | 56.69 | 18.64 | 117.76 | | 191.91 |
| ■ 2 SAME TG | 54.83 | 59.65 | 19.40 | 126.84 | | 203.71 |

**Fig. 6.** Results for one and two threads on Sun Niagara 2

For the Sun Niagara 2 architecture, we also compared the performance when running two threads on cores with and without shared cache. The results in Figure 6 show that the performance difference is not as high as in the Nehalem architecture, suggesting that the results for the optimized thread placement with 8 to 32 threads are not going to be as good as with Nehalem.

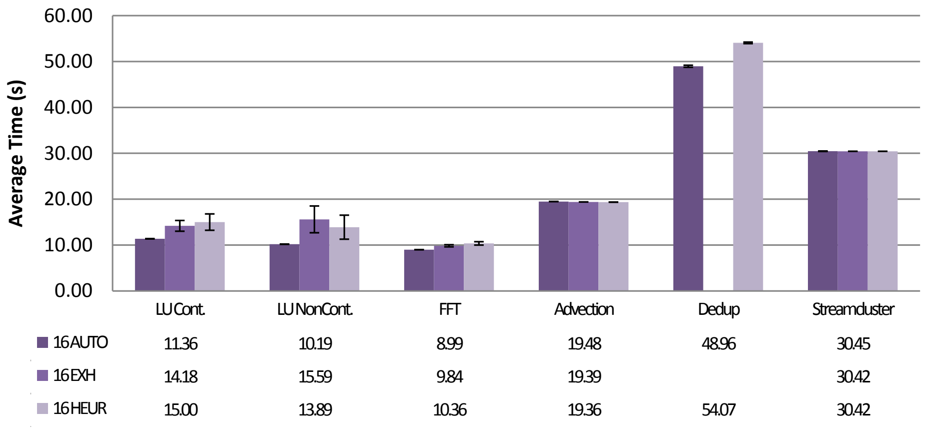
For eight threads, there was no difference in average execution time for any benchmark (Figure 7). There was an increase in variance in the LU (non-contiguous).

For 16 threads (Figure 8), the performance of LU, FFT and Dedup decreased when using optimized thread placement. In addition, the variance of the execu-



| | LU Cont. | LU NonCont. | FFT | Advection | Dedup | Streamcluster |
|---|---|---|---|---|---|---|
| 🟩 8 AUTO | 15.62 | 15.46 | 10.12 | 30.07 | 50.99 | 49.39 |
| 🟩 8 EXH | 15.39 | 16.75 | 10.18 | 30.13 | | 49.51 |
| 🟩 8 HEUR | 15.46 | 15.98 | 10.18 | 30.13 | 50.73 | 49.51 |

**Fig. 7.** Results for eight threads on Sun Niagara 2



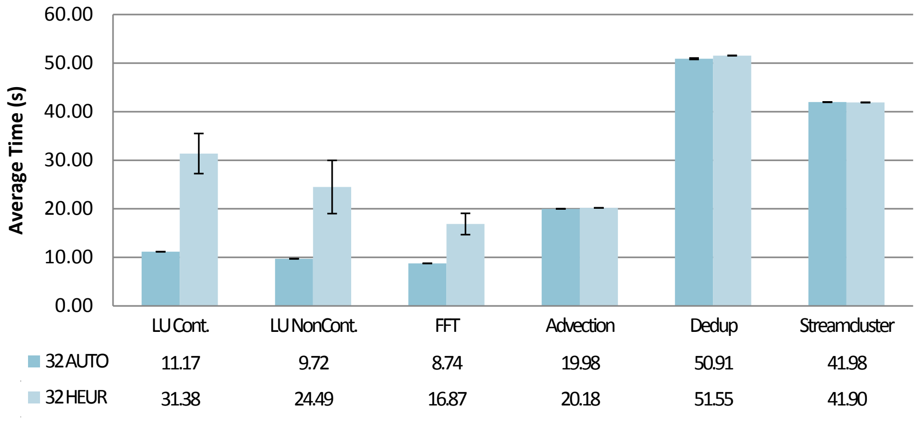
| | LU Cont. | LU NonCont. | FFT | Advection | Dedup | Streamcluster |
|---|---|---|---|---|---|---|
| 🟪 16 AUTO | 11.36 | 10.19 | 8.99 | 19.48 | 48.96 | 30.45 |
| 🟪 16 EXH | 14.18 | 15.59 | 9.84 | 19.39 | | 30.42 |
| 🟪 16 HEUR | 15.00 | 13.89 | 10.36 | 19.36 | 54.07 | 30.42 |

**Fig. 8.** Results for 16 threads on Sun Niagara 2

tion time of LU increased as well. Advection and Streamcluster show no difference in behaviour.

The results of running with 32 threads (Figure 9) are even worse in the case of LU and FFT. Both average execution time and variance increase drastically when running with optimized thread placement in these benchmarks. Advection, Dedup and Streamcluster show the same behaviour with and without optimized thread placement.

To summarize, optimizing thread placement on Niagara 2 did not decrease execution time and actually led to worse results for 16 and 32 threads, both in terms of performance and variance of the execution time. The reasons for this behaviour is that it is only possible to optimize data sharing through the very small L1 data cache, as the L2 cache is shared between all the cores.



| | LU Cont. | LU NonCont. | FFT | Advection | Dedup | Streamcluster |
|---|---|---|---|---|---|---|
| 32 AUTO | 11.17 | 9.72 | 8.74 | 19.98 | 50.91 | 41.98 |
| 32 HEUR | 31.38 | 24.49 | 16.87 | 20.18 | 51.55 | 41.90 |

**Fig. 9.** Results for 32 threads on Sun Niagara 2

## 6 Related Work

In our previous work [10] a process mapping technique for clusters of multi-core processors was presented using MPI traces in order to identify the communication pattern. The impact of placing threads on multi-core is analyzed in combination with placing threads on clusters. Motivated by those results, we extended this approach to multi-core architectures and shared memory.

The MPI Process Placement toolset from Chen et al. [5] aims to find the optimized mapping automatically using a profile-guided approach for SMP clusters and multiclusters, where the communication among the processes is much more easy to obtain because it is made explicit by the MPI primitives. The key difference between this work and ours is the way we obtain the communication patterns by monitoring the application in a simulated environment.

The work from Thekkath and Eggers [13] evaluates the impact of thread placement on multithreaded architectures using placement algorithms fed by trace-driven simulators, but despite the potential of this technique, no performance improvements were achieved due to the memory access patterns of the applications. We used real machines as a testbed which take into account the dynamic behaviour of parallel application, such as lock contention. Moreover we showed improvements in execution time in the Intel Nehalem architecture.

Tam, Azimi and Stumm [12] used performance monitoring units to detect sharing patterns among threads running on a multiprocessor architecture. Even with the performance improvements, the authors show that it is possible to increase the performance even more by using hand optimization. The study of Klug et al. [7] presents a framework that uses the hardware counters in order to find the best thread placement on multi-core machines. These two approaches are tightly coupled to the architecture, while ours is less dependent. In our study we observed the memory accesses of each thread, regardless of cache parameters (for example, line size and associativity), thus separating the program's behavior from the architecture.

Broquedis et al. [4] introduce the hwloc framework which gathers hardware information and exposes it. As a demonstration, they use it change hardware affinities in parallel applications to improve the performance of OpenMP and MPI applications. When using OpenMP, not the memory accesses, but the structure of the program is used, leading to good results. However, the application used creates more than 100,000 threads which is an uncommon situation, while we evaluated a diverse set of workloads with more common configurations.

## 7 Conclusions and Future Work

This work presents an approach to improve execution time of workloads by placing threads according to their communication patterns. In order to find the communication patterns, we executed each workload in a simulator to generate memory access traces and used these traces to place the threads with two different algorithms. Our approach was evaluated by comparing execution times when running with optimized placement and the operating system scheduler.

Our tests show that there are two factors affecting thread placement: the cache architecture of the processor and the data sharing properties of the workloads. The cache architecture of the processor had the biggest influence: on Sun Niagara 2, execution time remained the same or was increased, while on Intel Nehalem it was reduced in almost all cases. On Sun Niagara 2, optimizing thread placement can only improve data sharing on the small L1 cache, because the L2 cache is shared by all cores anyway. Optimizing thread placement on Intel Nehalem can improve data sharing on the L1 and L2 cache, and, since we are running with two physical processors, on the L3 cache as well. The data sharing properties of the workloads influenced performance improvements significantly on Intel Nehalem, where it ranged between 5% in the case of FFT and 45% in the case of Streamcluster. The reason for that is the communication pattern of the workloads: FFT has an all-to-all communication, while the threads of Streamcluster communicate in a pipeline.

To summarize, our results on Intel Nehalem show that execution time can be reduced greatly in a wide variety of workloads by optimizing thread placement. Furthermore, our experiments showed that the heuristic algorithm has a similar performance benefit as the exhaustive search algorithm with a drastically lower run time.

For the future, we intend to use the temporal characteristics of the memory accesses to place threads dynamically, i.e., to change the placement during the runtime of the workload according to temporal changes in the communication patterns. One step further, we intend to evaluate the possibility of moving optimized thread placement into the Linux kernel, thereby making it completely transparent and removing the need to execute the workloads in a simulator beforehand.

## Acknowledgements

This work was partially supported by a grant from the National Counsel of Technological and Scientific Development (CNPq) and was developed under a GMD-CAPES cooperation program.

## References

1. K. Asanovic et al. The landscape of parallel computing research: A view from berkeley. *University of California at Berkeley, Technical Report No. UCB/EECS-2006-183*, 2006.
2. K.J. Barker et al. A Performance Evaluation of the Nehalem Quad-Core Processor for Scientific Computing. *Parallel Processing Letters*, 2008.
3. C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques*, 2008.
4. Franois Broquedis et al. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Euromicro International Conference on Parallel, Distributed and Network-Based Computing.*
5. Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *International Conference on Supercomputing*, 2006.
6. A.L. Fazenda, E.H. Enari, L.F. Rodrigues, and J. Panetta. Towards Production Code Effective Portability among Vector Machines and Microprocessor-Based Architectures. In *International Symposium on Computer Architecture and High Performance Computing*, 2006.
7. Tobias Klug, Michael Ott, Josef Weidendorfer, and Trinitis Carsten. autopin - Automated Optimization of Thread-to-Core Pinning on Multicore Systems. In *Workshop on Programmability Issues for Multi-Core Computers*, 2008.
8. R. Koppler. Geometry-Aided Rectilinear Partitioning of Unstructured Meshes. *Lecture Notes in Computer Science*, 1999.
9. P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer Micro*, 2002.
10. E.R. Rodrigues, F.L. Madruga, P.O.A. Navaux, and J. Panetta. Multi-core aware process mapping and its impact on communication overhead of parallel applications. In *IEEE Symposium on Computers and Communications*, 2009.
11. M. Shah et al. UltraSPARC T2: A highly-threaded, power-efficient, SPARC SOC. In *IEEE Asian Solid-State Circuits Conference*, 2007.
12. David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys European Conference on Computer Systems*, 2007.
13. R. Thekkath and S.J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *International Symposium on Computer Architecture*, 1994.
14. S. Thoziyoor, N. Muralimanohar, J.H. Ahn, and N.P. Jouppi. CACTI 5.1. *HP Laboratories, Palo Alto, Tech. Rep*, 2008.
15. S.C. Woo et al. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture*. ACM, 1995.