Memory Management in NUMA Multicore Systems:

Trapped between Cache Contention and Interconnect Overhead

Zoltan Majo
Department of Computer Science
ETH Zurich, Switzerland

Thomas R. Gross
Department of Computer Science
ETH Zurich, Switzerland

Abstract

Multiprocessors based on processors with multiple cores usually include a non-uniform memory architecture (NUMA); even current 2-processor systems with 8 cores exhibit non-uniform memory access times. As the cores of a processor share a common cache, the issues of memory management and process mapping must be revisited. We find that optimizing only for data locality can counteract the benefits of cache contention avoidance and vice versa. Therefore, system software must take *both* data locality and cache contention into account to achieve good performance, and memory management cannot be decoupled from process scheduling. We present a detailed analysis of a commercially available NUMA-multicore architecture, the Intel Nehalem. We describe two scheduling algorithms: *maximum-local*, which optimizes for maximum data locality, and its extension, *N-MASS*, which reduces data locality to avoid the performance degradation caused by cache contention. N-MASS is fine-tuned to support memory management on NUMA-multicores and improves performance up to 32%, and 7% on average, over the default setup in current Linux implementations.

Categories and Subject Descriptors D.4.8 [*Performance*]: Measurements; D.4.1 [*Process Management*]: Scheduling

General Terms Performance, Algorithms, Experimentation

Keywords NUMA, multicore processors, shared resource contention, memory allocation

1. Introduction

Multicore multiprocessors create unique challenges for runtime systems and compilers. If multiple cores on a processor share a cache, contention for the shared cache memory is a major performance bottleneck. Moreover, as the number of processor cores per chip increases with every new microprocessor generation, the problems caused by limited main memory bandwidth are further aggravated.

To scale memory system bandwidth, new processors integrate a memory controller on the processor chip. Therefore, in multiprocessor systems the physical memory address space is divided between the processors, with each processor accessing its share of the address space via its on-chip memory controller. Yet in shared-memory multiprocessors, each processor must be able to access the local memory of other processors as well. Such memory accesses happen via the cross-chip interconnect that connects the processors, and the major processor manufacturers have developed their proprietary cross-chip interconnect technology (e.g., AMD's HyperTransport, or Intel's QuickPath Interconnect). Multiprocessor configurations of these systems have a non-uniform memory architecture (NUMA) as remote memory accesses via the interconnect are subject to various overheads. The bandwidth is lower than the bandwidth provided by the (local) on-chip memory controller. The latency is higher as well: memory operations are processed by the local interface to the interconnect (arbitration may be needed if multiple cores access remote memory), a request is transmitted to another processor, and additional steps may be needed on this remote processor before the memory access can be done. Consequently, remote memory accesses suffer penalties of 1.5 to 2 times relative to local accesses. Good data locality is therefore highly desirable, i.e. the computations should take place on the processor that keeps their data. So memory management on these systems cannot be done without paying attention to process mapping, and a process scheduler that determines which processor executes a thread must consider where memory has been allocated.

There are two classes of problems that memory management and process scheduling on a NUMA-multicore must consider. First, the cores of a multicore processor share on-chip memory system resources (e.g., memory controllers, last-level caches, or prefetcher units). *Shared resource contention* can lead to severe performance degradation, as discussed in [3, 4, 9, 11, 18, 19, 23, 26, 29]. In [2, 3, 7, 12, 29] the authors show that the operating system scheduler is in a good position to reduce shared resource contention, especially contention for shared last-level caches (LLCs). Mapping memory-bound processes so that they use different last-level caches increases performance by avoiding inter-core cache misses that co-executing processes cause to each other.

The second class of problems in NUMA-multicores is related to the *data locality* in the system. There exists a large body of work on methods for improving data locality in NUMA systems, either by profile-based [14, 17] or dynamic [22, 27, 28] memory migration. However, none of the previous approaches considers increased shared resource contention that may be caused by data locality optimizations. Additionally, operating system schedulers that target shared resource contention avoidance can compromise data locality by mapping a process onto a processor that does not hold the process's data.

In this paper we argue that memory management and process scheduling must be coupled. We focus on the first and simplest part of memory management, the allocation of a process's data to a specific processor. We show that a process scheduler that aims at maximizing data locality in a system may not always obtain good

ISMM'10, June 4–5, 2011, San Jose, California, USA.

performance, as contention for shared resources may degrade performance. To support our argument, we present in the first part of the paper a detailed analysis of the performance of the memory system of a commercially available NUMA-multicore architecture, the Intel Nehalem. In the second part of the paper we present a novel NUMA–Multicore-Aware Scheduling Scheme (N-MASS) that is an extension of the standard Linux scheduler. N-MASS considers *both* previously discussed performance-degrading factors (shared resource contention and data locality) when deciding on how to map processes onto the hardware. N-MASS increases performance by up to 32%, and 7% on average, relative to default operating system process scheduling.

2. Background

In this section we analyze the performance impact of mapping processes onto a NUMA-multicore computer with data locality constraints with a simple example: mapping two memory-bound processes. For this example we select two programs from the SPEC CPU2006 benchmark suite, `mcf` and `lbm`. Both programs have a high last-level cache (LLC) miss rate, therefore the performance of both programs is highly dependent on efficiently using the memory system of the machine.

For the discussion of the example we assume that the two programs are executed on a NUMA-multicore machine similar to the one shown in Figure 1. The machine has two processors. Each processor is multicore, and the cores of each processor share an LLC. Moreover, the machine is NUMA: each processor is directly connected to a part of the physical memory, and the processors are connected to each other with a cross-chip interconnect.

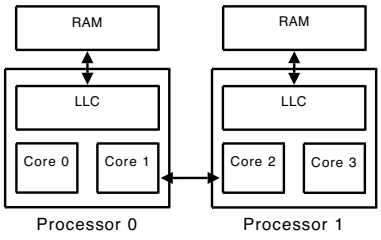


Figure 1: Example NUMA-multicore machine.

In a NUMA system a process's data can be allocated in the memory of any processor in the system. We say that a process p is *homed* on Processor i of the system if the process's data was allocated only on Processor i. If a process runs on its home processor, it is executed *locally*. Similarly, if a process runs on a processor different from its home processor, it is executed *remotely*. For our example we assume that both processes (executing `mcf` resp. `lbm`) are homed on Processor 0 of the machine (we relax this constraint in the evaluation presented in Section 4.3). Because both programs are single-threaded, there are four ways the processes executing the two programs can be mapped onto the system given this memory allocation setup. Figure 2 shows all possible mappings:

(a) **Both processes executed locally.** As both processes execute on their respective home node (Processor 0), they both have fast access to main memory. As Processor 0 has only one LLC, the processes contend for the LLC capacity of Processor 0.

(b) `mcf` **executed locally,** `lbm` **executed remotely.** As `lbm` is executed remotely (on Processor 1), it accesses main memory through the cross-chip interconnect, therefore it experiences lower throughput and increased latency of memory accesses relative to local execution. Additionally, as the two processes execute on two different processors, they do not share an LLC, therefore there is no cache contention in the system.

(c) `mcf` **executed remotely,** `lbm` **executed locally.** This case is similar to case (b), but in this case `mcf` uses the cross-chip interconnect to access main memory instead of `lbm`.

(d) **Both processes executed remotely.** Both processes share the LLC, and both processes execute remotely. This setup is clearly the worst possible scenario for performance, therefore we exclude this case from further investigation.

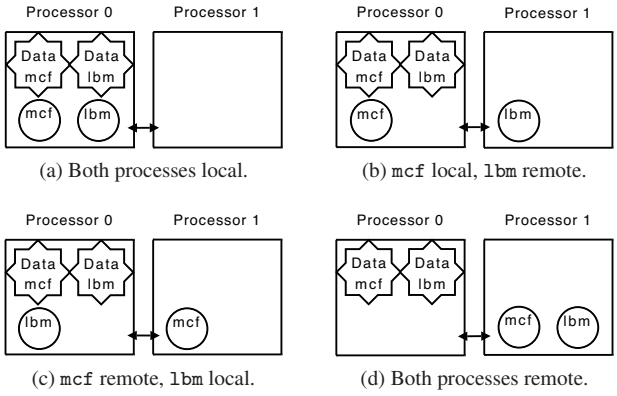
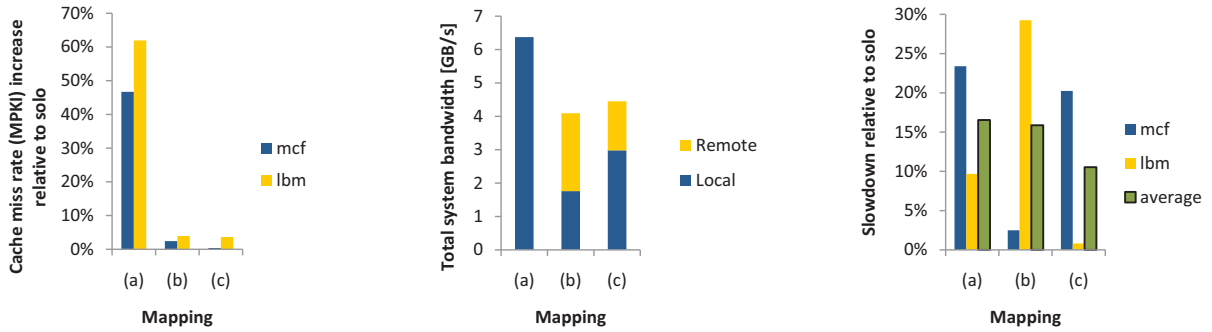


Figure 2: Possible mappings of `mcf` and `lbm`.

In this paper we use the cache miss rate per thousand instructions executed (MPKI) to characterize the memory-boundedness of programs. Figure 3.(a) shows the increase of the MPKI of `mcf` and `lbm` relative to each program's execution in *single-process mode* (executed alone and locally on the system, also referred to as *solo mode*). In case (a) (both processes locally executed), the MPKI increases by 47% resp. 62% due to cache contention. In cases (b) and (c) (when the processes are mapped onto different processors, therefore different LLCs), the MPKI increases by at most 4% relative to solo mode. The reason for this small increase is the contention on the memory controller relative to solo mode.

Good data locality is crucial for obtaining good performance in NUMA systems. Figure 3.(b) shows the distribution of bandwidth over the interfaces of the system. In case (a), when both processes are executed locally, the system has good data locality: 100% of the memory bandwidth in the system is provided by the local memory interface of Processor 0. In cases (b) and (c), when one of the two processes executes on Processor 1, data is transferred also on the cross-chip interconnect of the system: 56% (resp. 33%) of the generated bandwidth is due to one of the two processes executing remotely. Figure 3.(b) also shows the total bandwidth measured on the interfaces in the system. If the processes execute locally and thus share the cache (case (a)), the total bandwidth is approximately 50% more than in cases (b) and (c) (when caches are not shared).

In this paper we investigate which mapping leads to best performance: when cache contention is minimized (cases (b) and (c)), or when data locality is maximized (case (a)). Figure 3.(c) shows the individual and average performance degradation of `mcf` and `lbm` in all three mapping scenarios. The performance degradation of a program is calculated as the percent slowdown in wall clock execution time relative to the solo mode execution of the program. (Generally, if not qualified, performance means wall-clock execution time in this paper.) The average performance of the workload consisting of `mcf` and `lbm` is better in cases (b) and (c) than in case (a). Case (b) shows only a minor improvement over case (a) because remote execution slows down `lbm` by almost 30%. However, in case (c) the degradation is reduced relative to case (a); `mcf`



(a) Cache miss rate increase. (b) Bandwidth distribution in the system. (c) Performance degradation.

Figure 3: Performance of `mcf` and `lbm` in different mapping scenarios.

sees a small improvement, `lbm`'s slowdown is reduced from 11% to 1%, so the average degradation is reduced from 17% to 11%.

In conclusion, in a NUMA-multicore system we must find a compromise between favoring data locality and avoiding cache contention. Good data locality is beneficial in most cases, however when the memory pressure on LLCs is high, it is beneficial to avoid cache contention, even at the cost of compromising data locality in the system. In this paper we show that an architecture-aware process scheduler that is also aware of the memory allocation setup in the system can significantly increase performance relative to default operating system scheduling.

To simplify the discussion, we focus on a setup with 2 processors. We also assume that all the cores of a processor share an LLC. There are systems that do not support this assumption (see multi-socket implementations like the AMD Magny-Cours). In this case you should consider all the cores that share a cache to form a "processor". We consider only the memory allocation aspect of memory management. We restrict our attention as even this simple issue has many interesting aspects. Garbage collectors without doubt add additional complexity (and require more space for a detailed discussion). We assume that all the data of a process is allocated to one processor. This assumption includes scenarios when co-executing processes have their memory allocated on specific, possibly different, processors. We assume only that a single process's memory is not scattered around in the system. We also assume that the home processor of a process cannot be changed (i.e., data cannot be migrated). These limitations are discussed in Section 5.

3. Cache-conscious scheduling in NUMAs

3.1 Principles

In general, the tradeoff between local cache contention and remote execution can be observed with memory-bound programs. We focus in the presentation on the `soplex` benchmark from the SPEC CPU2006 benchmark suite. This program stores large amounts of data in the caches, and its performance is hurt if the available cache capacity is reduced because of other memory-bound programs using the same caches. There are several other memory-bound programs in the SPEC suite that show this behavior (see [25] for details), and the principles we discuss here are valid for these programs as well. We construct a multiprogrammed workload that consists of four identical copies (clones) of `soplex`. We allocate the memory of all clones on Processor 0 of a 2-processor NUMA-multicore system (details about the machine in Section 4.1). We execute the multiprogrammed workload in various mapping configurations with a different number of clones executed locally respectively remotely. The mapping configurations range from all four clones executed locally (on Processor 0) to the configuration where all four clones execute remotely. If a clone finishes earlier than the other clones in the workload, we restart it. We run the experiment until all clones execute at least once.

Figure 4 shows the average MPKI increase of all `soplex` clones relative to the solo mode MPKI of `soplex`. Remember that the MPKI of a program increases if in its execution the program contends for LLC capacity with other programs using the same LLC. When the data locality is maximal in the system (100% of the references are local), the average increase of MPKI peaks at 35% because all clones execute on the same processor and thus use the same LLC. When the data locality in the system is 57%, cache contention is minimal as the MPKI increase is also minimal (19%).

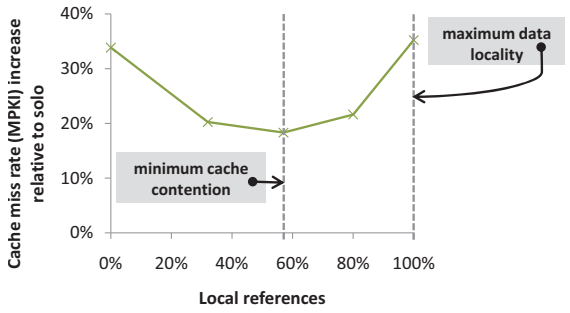


Figure 4: Increase in MPKI vs. data locality of `soplex`.

Figure 5 shows the slowdown of the locally resp. remotely executing `soplex` clones. The slowdown is calculated relative to the solo mode execution of `soplex`. We also plot the average degradation of the clones. Clearly, neither the mapping with minimum cache contention, nor the mapping with maximum data locality performs best. The average slowdown (and also the individual slowdown) of the clones is minimal if there is 80% data locality in the system. Therefore, process scheduling on NUMA-multicores must target a tradeoff between data locality and cache contention avoidance (the optimum performance point on Figure 5).

Our approach builds upon the idea of cache-balancing algorithms for SMPs ([12, 29]). The basic principle of these algorithms is illustrated in Figure 6 (for a system with two LLCs). If the difference D between the pressure on the two caches of the system is large (Figure 6.(a)), some processes (with a pressure of $D/2$) are scheduled onto the cache with the smaller pressure, therefore the difference between the pressure on the two caches is minimized (Figure 6.(b)). Our approach is similar to cache-balancing algorithms in SMPs and relies on two principles. First, we also distribute pressure across caches, however not evenly as in an SMP

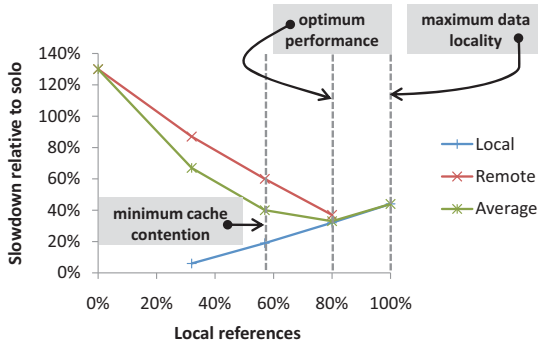
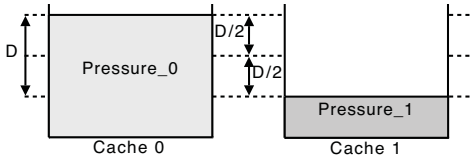
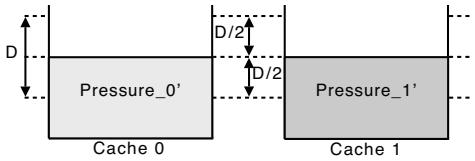


Figure 5: Performance vs. data locality of `soplex`.

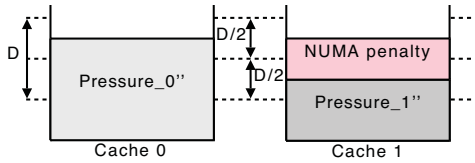
system: The amount of cache pressure transferred to Cache 1 is less than $D/2$ (the half of the difference) – as illustrated in Figure 6.(c). If mapping processes onto a different LLC results in the remote execution of the re-mapped process, then we account also for the performance penalty of remote execution. The pressure on the caches is equal if this penalty is also considered. The second principle of our approach states that overloading the cross-chip interconnect with too many remotely executing processes must be avoided. Therefore, if the pressure on the remote cache is above a threshold, we do not re-map processes for remote execution. Section 3.3 presents the N-MASS algorithm that implements these two principles; Section 3.2 discusses how we calculate LLC pressure.



(a) Initial configuration.



(b) SMP.



(c) NUMA.

Figure 6: Cache balancing in SMP and NUMA context.

3.2 Program characterization

A scheduling algorithm targeting memory system optimizations must be able to quickly estimate the memory behavior of the scheduled programs on runtime. There are two parameters that we want to estimate: the cache pressure of programs, and the performance penalty they experience due to remote execution.

The cache pressure of programs can be estimated with reasonable precision by performance metrics available at runtime via the performance monitoring unit (PMU) of modern CPUs. Knauerhase et al. [12] use the LLC misses per elapsed CPU cycle to estimate cache pressure, Blagodurov et al. [29] use the MPKI. Other synthetic metrics like stack-distance profiles [4, 25] or miss-rate curves [26] can offer better precision in estimating cache behavior, but generating these metrics might result in higher runtime overhead than low-overhead PMU-based measurements, therefore we estimate cache pressure based on the MPKI of programs.

The second parameter we want to estimate is the *NUMA penalty* of a program. This parameter quantifies the slowdown of a remote execution of a program relative to the program's local execution. Let CPI_{local} denote the CPI (cycles per instruction) of a program executing locally, and let CPI_{remote} denote the CPI of the same program executing remotely. Given this notation, the NUMA penalty is defined as:

$$\text{NUMA penalty} = CPI_{remote}/CPI_{local} \quad (1)$$

The NUMA penalty is a lower-is-better metric, and its minimum value is 1 (if a program does not slow down in its remote execution). E.g., if a program has a NUMA penalty equal to 1.3, the program slows down 30% on remote execution. We measure the NUMA penalty of a program by executing the program twice, once locally and once remotely. During the measurements all cores are inactive, except the core that executes the program. Figure 7 plots the NUMA penalty of all programs of the SPEC CPU2006 suite against their MPKI. The chart also plots the linear model fitted onto the data. Although the two parameters are positively correlated (the NUMA penalty increases with the MPKI), the coefficient of determination (R^2) is relatively low, 0.64.

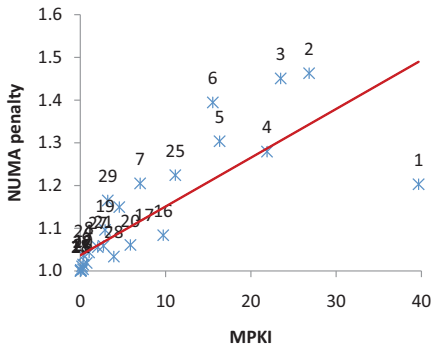


Figure 7: NUMA penalty vs. MPKI.

3.3 The N-MASS algorithm

The NUMA–Multicore-Aware Scheduling Scheme (N-MASS) implements the two principles of cache-aware scheduling in NUMA systems described in Section 3.1. Algorithm 1 presents an outline of N-MASS. The algorithm is designed for a 2-processor NUMA system, but it can be easily extended to handle a higher number of processors as well. The algorithm is invoked after a scheduler epoch has elapsed, and it calculates the mapping M_{final} : Processes $\mapsto$ Cores of processes onto cores. The number of scheduled processes n equals at most the number of cores in the system (this limitation is discussed later in this section). The algorithm uses the following performance data about each scheduled process i: the process's *cache pressure* ($mpki_i$) and an estimate of the process's *NUMA penalty* (np_i). The N-MASS algorithm has three steps. First, for each processor, it sorts the list of processes homed on the processor in descending order of the processes' NUMA penalty (lines 2-3). Second, it maps the processes onto the system using the *maximum-local* policy (line 5). If the

pressure on the memory system of the two-processor system is unbalanced, then in the third step the algorithm refines the mapping decision produced by the maximum-local mapping (line 7). In the following paragraphs we describe Step 2 and Step 3 of N-MASS.

Algorithm 1 N-MASS: maps n processes onto a 2-processor NUMA-multicore system.

Input: List of processes P_0 and P_1 homed on Processor 0 resp. Processor 1.
Output: A mapping M_{final} of processes to processor cores.

```
1: // Step 1: Sort list of processes by NUMA penalty
2: P_sorted_0 ← sort_descending_by_np(P_0)
3: P_sorted_1 ← sort_descending_by_np(P_1)
4: // Step 2: Calculate maximum-local mapping
5: M_maxlocal ← map_maxlocal(P_sorted_0, P_sorted_1)
6: // Step 3: Refine maximum-local mapping
7: M_final ← refine_mapping(M_maxlocal)
```

Step 2: Maximum-local mapping The maximum-local scheme (described in detail by Algorithm 2) maximizes data locality in the system by mapping processes onto their home nodes in descending order of their NUMA penalty. The algorithm has as its input two lists of processes, P_0 and P_1. The processes in list P_0 (P_1) are homed on Processor 0 (Processor 1). The lists are sorted in descending order of the NUMA penalty of the processes they contain. The algorithm merges the two lists. During the merge, the algorithm determines which core each process is mapped onto. The algorithm guarantees that processes with a high NUMA penalty are mapped onto a core of their home node with higher priority than processes with a lower NUMA penalty that are homed on the same processor. The lists P_0 and P_1, and the mapping $M_{maxlocal}$ of processes are double-ended queues. The function $pop\_front(l)$ removes the element from the front of the list l; the function $push\_back(l, e)$ inserts element e at the back of the list l. The function $get\_next\_available\_core(p)$ returns the next free core, preferably from processor p. If there are no free cores on processor p, the function returns a free core from a different processor.

Algorithm 2 map_maxlocal: maps n processes onto a 2-processor system NUMA system so that data locality is maximized.

Input: List of processes P_{sorted_0} and P_{sorted_1} homed on Processor 0 respectively Processor 1. The lists are sorted in descending order of the processes' NUMA penalty (np).
Output: A mapping $M_{maxlocal}$ of processes to processor cores.

```
1: M_maxlocal ← ∅
2: p_0 ← pop_front(P_sorted_0); p_1 ← pop_front(P_sorted_1)
3: while p_0 ≠ NULL or p_1 ≠ NULL do
4:    if p_1 = NULL or np_p0 > np_p1 then
5:       core ← get_next_available_core(Processor 0)
6:       push_back(M_maxlocal, (p_0, core))
7:       p_0 ← pop_front(P_sorted_0)
8:    else if p_0 = NULL or np_p0 ≤ np_p1 then
9:       core ← get_next_available_core(Processor 1)
10:      push_back(M_maxlocal, (p_1, core))
11:      p_1 ← pop_front(P_sorted_1)
12:   end if
13: end while
```

Step 3: Cache-aware refinement If the maximum-local mapping results in increased contention on the caches of the system, Step 3 of the N-MASS algorithm refines the mapping produced by the maximum-local scheme in Step 2. This step implements the two principles of scheduling in NUMA-multicores previously discussed in Section 3.1, and is described in detail in Algorithm 3. First, the algorithm accounts for the performance penalty of remote execution by multiplying the MPKI of remotely mapped processes with their respective NUMA penalty (lines 8, 9, 12). Second, the algorithm avoids overloading the cross-chip interconnect by moving processes only if the pressure on the remote cache is less than a predefined threshold (line 14). We discuss in Section 3.4 how the threshold is determined. By construction (line 6 and 10 of Algorithm 2) $M_{maxlocal}$ contains pairs $(process, core)$ ordered in descending order of the processes' NUMA penalty. The function $back(l)$ returns the last element of list l without removing it from the list; $push\_front(l, e)$ inserts element e to the front of list l.

Algorithm 3 refine_mapping: refines the maximum-local mapping of n processes to reduce cache contention.

Input: Maximum-local mapping of processes $M_{maxlocal}$. For each process i the NUMA penalty respectively the MPKI of the last scheduler epoch is available in np_i respectively $mpki_i$.
Output: A mapping M_{final} of processes to processor cores.

```
1:  M_0 = {(p, core) ∈ M_maxlocal | core ∈ Processor 0}
2:  M_1 = {(p, core) ∈ M_maxlocal | core ∈ Processor 1}
3:  pressure_0 = ∑{mpki_p | (p, core) ∈ M_0}
4:  pressure_1 = ∑{mpki_p | (p, core) ∈ M_1}
5:  repeat
6:     Δ ← |pressure_1 − pressure_0|
7:     (p_0, core_0) ← back(M_0); (p_1, core_1) ← back(M_1)
8:     Δ_MOVE0→1 ← mpki_p0 · np_p0
9:     Δ_MOVE1→0 ← mpki_p1 · np_p1
10:    if Δ_MOVE0→1 < Δ_MOVE1→0 then
11:       pressure_0 ← pressure_0 − mpki_p0
12:       pressure_1 ← pressure_1 + mpki_p0 · np_p0
13:       core ← get_next_available_core(Processor 1)
14:       // Could be on Processor 0 if ∄ free core on Processor 1
15:       if core ∉ Processor 0
          and pressure_1 < THRESHOLD then
16:          pop_back(M_0, (p_0, core_0))
17:          push_front(M_1, (p_0, core))
18:          decision ← MOVE0→1
19:       else
20:          decision ← CURRENT
21:       end if
22:    end if
23:    if Δ_MOVE0→1 ≥ Δ_MOVE1→0
       or decision = CURRENT then
24:       // Similar to the MOVE0→1 case
25:    end if
26: until decision ≠ CURRENT
27: M_final ← M_0 ∪ M_1
```

Limitations The N-MASS algorithm requires the number of processes n to be at most the total number of cores on the system, therefore it can only decide on spatial multiplexing of processes (and not on temporal multiplexing). Nevertheless, if the OS scheduler decides on the temporal multiplexing (the set of processes that will be executed in the next scheduler epoch), N-MASS can refine this mapping so that the memory allocation setup in the system is accounted for, and the memory system is efficiently used.

3.4 Implementation

To verify that N-MASS is capable of finding the tradeoff between cache contention avoidance and optimizing for data locality, we implemented a prototype version of N-MASS as a user-mode extension to the Linux scheduler. We design N-MASS to adapt to

program phase changes. N-MASS samples the PMU to characterize applications at runtime. The length of the sampling interval is determined adaptively to bound the sampling overhead. Each sample includes the MPKI of a process. The scheduler is invoked if a process's MPKI changes by more than 20% relative to the previous scheduler epoch. The costs of re-schedules (moving a process to a different core) can be high. We select an epoch length of 1s. This epoch length almost completely eliminates the costs of re-schedules, while the scheduler is still able to quickly react to program phase changes. We select 60 MPKI for the threshold used by the refinement step of the N-MASS algorithm (line 14 of Algorithm 3). We base our selection on a detailed empirical evaluation of the memory system performance of the STREAM and SPEC CPU2006 benchmarks. We have found that a cache pressure of around 60 MPKI corresponds to the saturation limit of the cross-chip interconnect of our evaluation machine. We omit details of this evaluation because of a lack of space.

The N-MASS algorithm relies on an estimate of the NUMA penalty of processes (in lines 8, 9 and 12 of Algorithm 3). In the first part of our evaluation, we look at performance of N-MASS with perfect information available about the NUMA penalty of scheduled programs. We simulate the availability of the NUMA penalty to the scheduler by using program traces generated on separate profiling runs (for each sample we compute the NUMA penalty using Formula 1). In Section 4.5 we also evaluate N-MASS with an on-line estimation of the NUMA penalty. In our evaluation machine the number of local and remote LLC misses generated by a process cannot be measured at the same time. Therefore, when a process is re-mapped from its home node to a remote node, the PMU must be reconfigured, which results in unacceptably large overhead. Hence we also include the MPKI of programs in the trace file. Nevertheless, the performance measurements of N-MASS include the overhead of performance monitoring, as we sample the number of instructions executed by each process to keep track of the process's execution in the trace file. We also record the number of elapsed processor cycles to measure performance.

4. Evaluation

4.1 Experimental setup

We use a 2-processor system based on the Intel Nehalem microarchitecture. The machine is multicore: each processor has four cores that share an LLC. The machine is NUMA because it has two types of memory controllers: Each processor has half of the physical memory directly connected through an on-chip memory controller (IMC), and cross-processor communication is handled by the QuickPath Interconnect (QPI). Table 1 shows the detailed parameters of the machine. The bandwidth of the IMC and QPI are approximately the same, but while there are two IMCs in the system (one on each processor), there is only one QPI link connecting the two processors. Therefore, if there is good data locality in the system, the full throughput of the two IMCs can be exploited. But if most memory accesses are remote, the QPI link is a performance bottleneck. The latencies of local resp. remote memory accesses also differ significantly, as shown in Table 1 (values based on [10]).

The evaluation machine runs Linux 2.6.30 patched with perfmon2 to allow access to the PMU. We disable frequency scaling, simultaneous multithreading, and the Turbo Boost feature of the machine to avoid measurement variance. We use standard Linux APIs to control the CPU affinity of processes and also to set their preferred memory allocation policy.

Our evaluation methodology is very similar to the methodology used in [2, 3, 6, 7, 12, 29]. We use a subset of the SPEC CPU2006 benchmark suite (14 programs out of the total 29 in the suite). Our selection includes programs 1–14 in Figure 7. We select the subset so that it includes programs with a broad range of memory pressure. The MPKI of a program (the x-axis in Figure 7) roughly characterizes the memory boundedness of the program. The selection includes both CPU- and memory-bound programs. Some memory-bound programs saturate the IMC of the evaluation machine even in solo mode. The programs in the subset have also a broad range of NUMA penalties (between 1.0 and 1.46).

Processor:	2 x Intel Xeon E5520
Cores per processor:	4
L3 cache size:	8 MB
Main memory:	12 GB DDR3
IMC bandwidth:	25.6 GB/s
QPI bandwidth:	23.44 GB/s
Local DRAM access latency:	∼50 ns
Remote DRAM access latency:	∼90 ns

Table 1: Parameters of the evaluation machine.

We are interested in multiprocessor performance, therefore we construct multiprogrammed workloads with the programs of the SPEC CPU2006 benchmark suite. Like [12], we run each multiprogrammed workload exactly one hour. If a program terminates before the other programs in a workload do, we restart the program that terminated early. We use the reference data set and follow the guidelines described in [21] to minimize measurement variance. This setup usually gives us three measurable runs for each workload within the one hour limit. For each run we report the average slowdown of each constituent program relative to its solo mode performance. We also report the average slowdown of the whole workload, as suggested by Eyerman et al. [5].

4.2 Dimensions of the evaluation

There are two dimensions that must be considered to evaluate the interaction between memory allocation and process scheduling in a NUMA-multicore system. The two dimensions (shown in Figure 8) are the memory boundedness of the workloads (y-axis) and the balance of memory allocation in the system (x-axis).

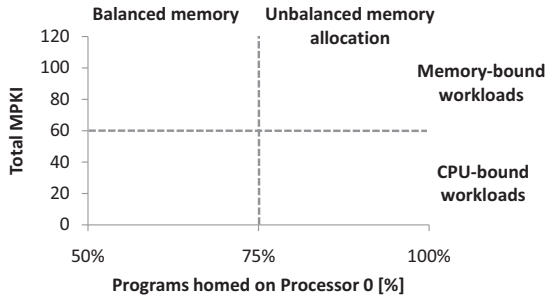


Figure 8: Dimensions of the evaluation.

To show that N-MASS can handle workloads with different memory-boundedness, we use 11 different multiprogrammed workloads (WL1 to WL11). This setup corresponds to evaluating N-MASS along the first dimension (the y-axis in Figure 8). The workloads are composed of different number of compute-bound (C) resp. memory-bound (M) programs. The memory-boundedness of a workload is characterized by the sum of the MPKIs of its constituent programs (measured in solo mode for each program). The total MPKI of each multiprogrammed workload we use is shown in Figure 9. The composition of the multiprogrammed workloads is shown in Table 2. The workloads in the set of 4-process workloads (WL1 to WL9) contain one to four

memory-bound programs. The 8-process workloads (WL10 and WL11) contain three, resp. four, memory-bound programs. In the case of all 11 workloads we add CPU-bound programs so that at the end there are four (resp. eight) programs in each workload.

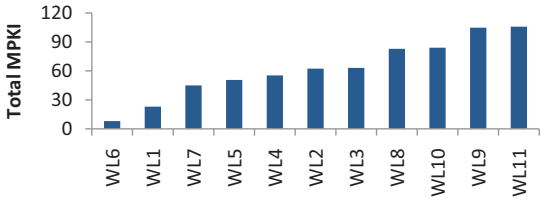


Figure 9: Total MPKI of multiprogrammed workloads.

#	Programs				Type
1	soplex	sphinx	gamess	namd	1M, 3C
2	soplex	mcf	gamess	gobmk	2M, 2C
3	mcf	libquantum	povray	gamess	2M, 2C
4	mcf	omnetpp	h264	namd	2M, 2C
5	milc	libquantum	povray	perlbench	2M, 2C
6	sphinx	gcc	namd	gamess	1M, 3C
7	lbm	milc	sphinx	gobmk	2M, 2C
8	lbm	milc	mcf	namd	3M, 1C
9	mcf	milc	soplex	lbm	4M
10	lbm	milc	mcf	namd	3M, 5C
	gobmk	perlbench	h264	povray	
11	mcf	milc	soplex	lbm	4M, 4C
	gobmk	perlbench	namd	povray	

Table 2: Multiprogrammed workloads.

As the performance of process scheduling closely depends on the memory allocation setup in the system, for each workload we consider several ways memory is allocated in the 2-processor evaluation machine. The second dimension of our evaluation (the x-axis in Figure 8) is the percentage of the processes of a multiprogrammed workload homed on Processor 0 of the system. (Ideally we would like to vary the percentage of memory references to local resp. remote memory, but as we can map only complete processes, we vary along this dimension by mapping processes.) The left extreme point of the x-axis represents the configuration with balanced memory allocation (50% of the processes homed on Processor 0). On the other end of the x-axis we find the most unbalanced configuration (100% of the processes homed on Processor 0). Because of the symmetries of the system there is no need to extend the range to the case with 0% of the processes' memory allocated on Processor 0. This corresponds to 100% of the processes homed on Processor 1, which is equivalent to all processes homed on Processor 0.

In Section 4.3 we evaluate N-MASS with different memory allocation setups (along the x-axis of Figure 8). Then we focus on unbalanced memory allocation setups in Section 4.4.

4.3 Influence of data locality

The second dimension of our evaluation is defined as the percentage of the processes homed on Processor 0 of the system. This percentage however does not specify *which* constituent processes of a multiprogrammed workload are homed on each processor in the system. We define the concept of *allocation maps*. An allocation map is a sequence $M = (m_0, m_1, \ldots, m_n)$, where n is the number of processes in the workload executing on the system, and

$$m_i = \begin{cases} 0, & \text{if the } i^{th} \text{ process is homed on Processor 0;} \\ 1, & \text{if the } i^{th} \text{ process is homed on Processor 1.} \end{cases} \quad (2)$$

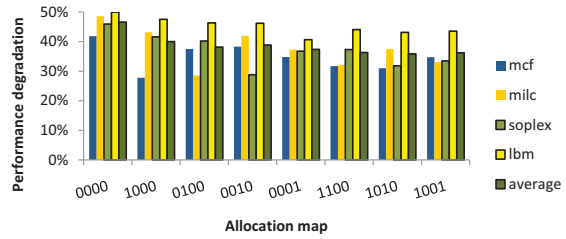
There are $\sum_{i=0}^{4} \binom{4}{i} = 2^4 = 16$ ways to allocate memory for a 4-process workload on the Nehalem system (assuming each program's memory is allocated entirely on one of the two processors of the system). Because of the symmetries of the system the number of combinations is reduced to 8. These allocation maps are shown in Table 3. For example, if 50% of the processes are homed on Processor 0 we must consider three different possibilities. If we look at the performance of a mapping algorithm with a multiprogrammed workload that has a composition (M, M, C, C) (first two processes are memory-bound, the last two compute-bound), then the 50%-allocation maps 1100 and 1010 are different from the point of view of the maximum-local scheduling scheme. Remember that the maximum-local scheme maps processes onto their home nodes if possible. In the case of the 1100 allocation map maximum-local maps the two memory-bound processes onto the same processor, therefore the *same* LLC. This setup results in good data locality but also produces high cache contention. In the case of the 1010 allocation map maximum-local maps the memory-bound processes onto *separate* LLCs. Therefore, the maximum-local policy maximizes data locality and minimizes cache contention in this case.

Processes homed on Processor 0	Allocation maps
50%	1100, 1010, 1001
75%	1000, 0100, 0010, 0001
100%	0000

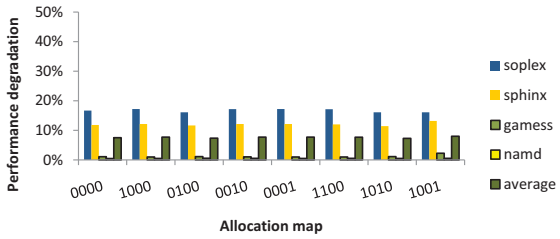
Table 3: Allocation maps for 4-process workloads.

For clarity of presentation we use two workloads from opposite ends of the memory-boundedness spectrum to evaluate the performance of N-MASS with different allocation maps: the compute-bound WL1, and the memory-bound WL9. We compare the performance of three mapping schemes: *default*, *maximum-local* and *N-MASS*. If not stated otherwise, in our evaluation *N-MASS* denotes the version of the algorithm that has perfect information about the NUMA penalty of the programs from profile-based program traces. The *maximum-local* policy is similar to N-MASS, except it does not include the cache-aware refinement step of N-MASS (Step 3 of Algorithm 1). We evaluate this scheme to quantify the improvement of the cache-aware refinement step over maximum-local mapping.

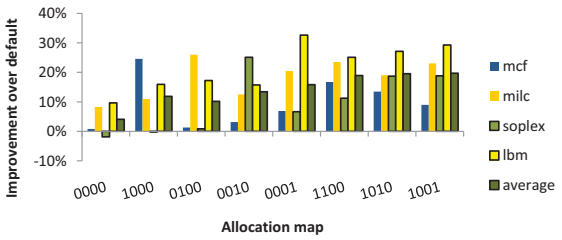
The performance of multiprogrammed workloads varies largely with the default Linux scheduler, and simple factors like the order in which workloads are started influence the performance readings. Because operating system schedulers (including the Linux scheduler) balance only the CPU load and do not account for data locality or cache contention, processes might be mapped so that they use the memory system in the most inefficient way possible. To avoid measurement bias, we account for all schedules that an OS scheduler that balances CPU load would consider. E.g., in the case of 4-process workloads the default Linux scheduler always maps two processes onto each processor so that each processor is allocated half the total CPU load. For each 4-process workload there are $\binom{4}{2} = 6$ equally probable different schedules with the CPU load evenly distributed in the system. Running a single workload in all these schedules takes 6 hours execution time with our evaluation methodology, which is tolerable. Therefore, for each workload we run the workload in each schedule possible for the default scheduler, and then we report the average degradation of the workload in all schedules as the performance of *default* scheduling.



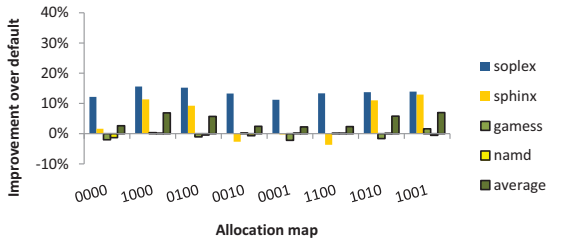
(a) Performance degradation (WL9).



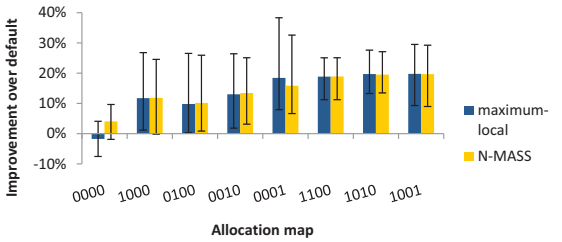
(b) Performance degradation (WL1).



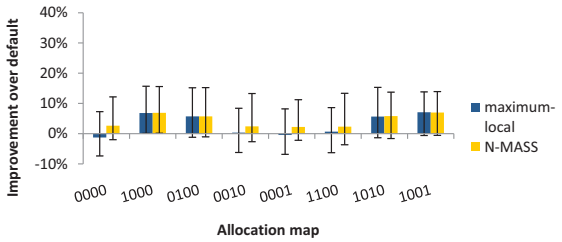
(c) N-MASS improvement over default (WL9).



(d) N-MASS improvement over default (WL1).



(e) N-MASS compared to *maximum-local* (WL9).



(f) N-MASS compared to *maximum-local* (WL1).

Figure 10: Performance evaluation of the *maximum-local* and N-MASS schemes.

Figure 10.(a) (resp. Figure 10.(b)) shows the performance degradation of the programs of WL9 (WL1) with the default scheduler. The degradations are calculated relative to the solo mode performance of the programs. WL9 is composed of more memory-bound programs than WL1, therefore the degradations experienced by WL9 programs are higher (up to 50% vs. 18%). Figure 10.(c) (resp. Figure 10.(d)) shows the performance improvement of N-MASS relative to default scheduling. Performance improvements of individual programs up to 32% are possible.

An interesting question is how much improvement is due to the maximum-local scheme, and how much benefit is due to the final refinement step of N-MASS. In Figure 10.(e) and Figure 10.(f)) we compare the average performance improvement of N-MASS versus the maximum-local scheme. The bars show the maximum and performance improvement of the constituent programs of the workloads; these bars do not show the "standard error". A negative performance improvement means performance degradation. N-MASS performs approximately the same as maximum-local in most of the cases. However, when the memory allocation in the system is unbalanced (allocation map 0000 for both workloads, allocation maps 0001, 0010, and 1100 for WL1), the additional cache-balancing of the N-MASS scheme improves performance relative to maximum-local. In these cases maximum-local results in a performance degradation relative to default, because cache contention on the LLCs cancels the benefit of good data locality. There are also some cases when N-MASS performs slightly worse than maximum-local, but its average performance is never worse than the performance of default scheduling.

4.4 A detailed look

In the previous section we have shown that in case of unbalanced memory allocation maps the cache-aware refinement step of N-MASS improves performance over maximum-local. In this section we look in detail at the performance of the N-MASS and maximum-local policies in case of unbalanced memory allocation maps, and extend our measurements to the 8-process workloads. Figures 11 and 12 show the performance for each of the programs in the various workload sets (WL1 to WL11). In many cases the maximum-local mapping scheme performs well, and the final refinement step of N-MASS brings only small benefits. However, in the case of WL1, WL7, WL8, WL9, and WL11 individual programs of the workloads experience up to 10% less performance degradation with N-MASS than with maximum-local. Performance degradations relative to default scheduling are also reduced from 12% to at most 3%.

4.5 Estimating the NUMA penalty

Figures 11 and 12 also show the effect of estimating the NUMA penalty through linear regression. For this evaluation, we do not use the profile-based information about the NUMA-penalty of programs (as this number may be difficult to obtain in current multi-core systems). Instead, we estimate the NUMA penalty based on the MPKI rate, fitting a simple linear model onto the data shown in Figure 7. Before fitting the model we remove the outlier `mcf` (data point "1" on Figure 7) as well as all non–memory-bound programs (programs with a MPKI smaller than 1). The resulting model's slope intercept and slope are 0.015 and 1.05, respectively.

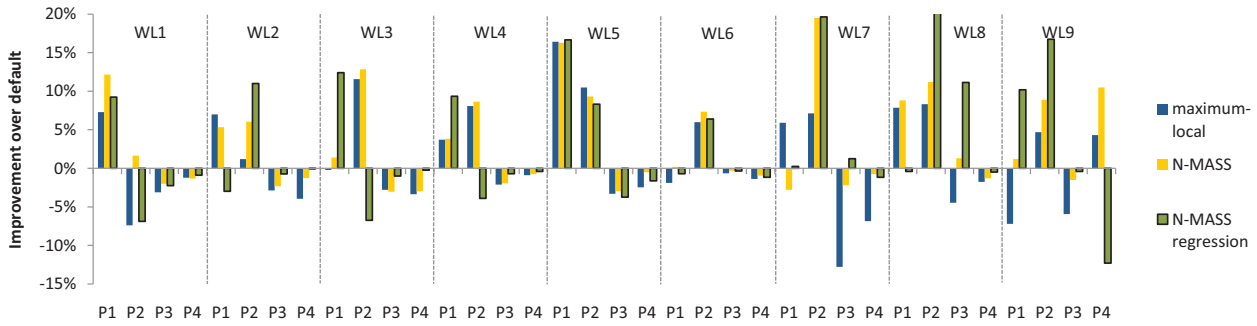


Figure 11: Performance improvement of 4-process workloads.

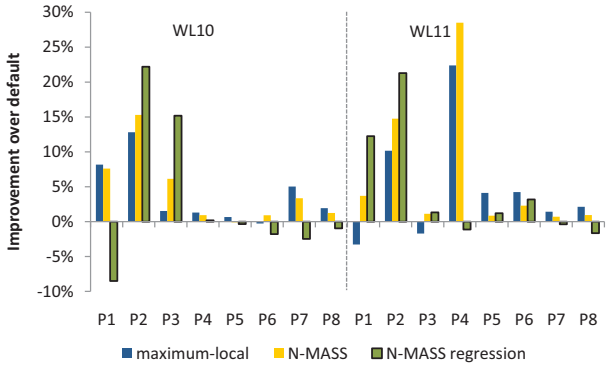


Figure 12: Performance improvement of 8-process workloads.

For the measurements the memory of all processes of the multiprogrammed workloads was allocated on the same processor, Processor 0 (allocation maps 0000 and 00000000).

Regression-based N-MASS also improves performance over the maximum-local scheme, but in the case of four workloads it results in a performance degradation of individual programs of more than 5% relative to the default scheduler (but on average not more degradation than the maximum-local scheme). The degradation is due to the imprecision of the MPKI-based estimates of the NUMA penalty. On-line techniques to estimate the NUMA penalty are difficult to construct with the PMU of current CPU models. We hope that future PMUs will provide events that can be used to estimate NUMA penalty better. If the NUMA penalty cannot be obtained directly, the MPKI offers a reasonable approximation.

5. Discussion

In summary, if the memory allocation in the system is balanced, then maximum-local scheduling provides large performance benefits. If the memory allocation setup of the system unbalanced, the mapping given by the maximum-local scheme needs adjustment, otherwise it causes performance degradation even relative to default scheduling.

In cases with unbalanced memory allocation, the refinement step of N-MASS can re-map processes onto a different LLC to reduce cache contention. When memory allocation in the system is balanced, maximum-local mapping is performed, and cache contention within a processor is minimized as in an SMP context, using existing approaches [12, 29]. Our scheme is orthogonal to these schemes: the cache-aware step of N-MASS kicks in only if the memory allocation map in the system is unbalanced, and additional cross-processor cache balancing is required.

Memory migration is an alternative technique to improve data locality in NUMA systems. We limit the discussion to process scheduling because of two issues: (1) it is difficult to estimate the cost of memory migration, and (2) memory migration is not always possible because there is not always enough free memory available on the destination processor. In these cases the process scheduler is the only part of the system software that can optimize performance.

We do not consider multithreaded programs with a shared address space. For these programs sharing caches can be beneficial, therefore finding a tradeoff between data locality and cache contention is difficult. To limit the number of cases that must be evaluated, we restrict memory allocation of a process to a single processor. OSs provide information about the distribution of pages in the system. This information can be used to determine a program's preferred home processor if the program's memory is scattered around.

6. Related work

Memory system analysis Molka et al. [20] analyze the memory system of an Intel Nehalem-based machine. They use sophisticated synthetic benchmarks to determine the bandwidth and latency of memory accesses to different levels of the memory hierarchy. Hackenberg et al. [8] compare the memory system of different NUMA architectures using these synthetic benchmarks. Majo et al. [16] use synthetic benchmarks to evaluate the fairness of bandwidth sharing of the Intel Nehalem. Here we focus on more realistic programs and also consider caching effects. Blagodurov et al. [3] describe the sources of performance degradation that cause slowdowns to programs co-executing on NUMA systems (the remote latency and interconnect degradation). The NUMA penalty used in this paper quantifies the slowdown that a single program experiences due to both factors.

Shared resource contention Chandra et al. [4] use analytical models to predict the inter-thread cache contention of co-executing programs. Jiang et al. [11] prove that the complexity of optimal co-scheduling on chip multiprocessor systems is NP-complete. Mars et al. [18] describe a system that characterizes resource contention on runtime. Zhuravlev et al. [29] compare the accuracy of different models used to characterize the interference of co-executing programs. They find that the MPKI is reasonably accurate.

There are several methods to mitigate shared resource contention. Qureshi et al. [23] partition caches between concurrently executing processes. Tam et al. [26] identify the size of cache partitions on runtime. Mars et al. [19] halt low priority processes when contention is detected. Herdrich et al. [9] analyze the effectiveness of frequency scaling and clock modulation to reduce shared resource contention. Awasthi et al. [1] show that data migration and adaptive memory allocation can be used to reduce memory controller overhead in systems with multiple memory controllers (such

as NUMAs). OS process scheduling is also well suited for reducing contention on shared caches, as described by Fedorova et al. [6].

Process scheduling for contention avoidance Fedorova et al. present an OS scheduling algorithm that reduces the performance degradation of programs co-executed on multicore systems [7]. Banikazemi et al. [2] describe a cache model for a process scheduler that estimates the performance impact of program-to-core mapping in multicore systems. The process scheduler mechanism described by Knauerhase et al. [12] and Zhuravlev et al. [29] is most closely related to the N-MASS scheme presented in this work. The schemes presented by both groups schedule processes so that each LLC must handle approximately equal memory pressure. These approaches were evaluated on SMPs with uniform memory access times. We show that cache balancing algorithms do not work well in NUMA systems if the memory allocation setup of the system is not considered.

Performance-asymmetric multicore architectures Recent research proposed performance-asymmetric multicore processors (AMPs). In contrast to AMPs the cores of a NUMA system have the same performance, but the memory system is asymmetric, and programs have different performance on remote execution. Li et al. present an OS scheduler for AMPs [15]. They evaluate their system also on NUMA systems, but they do not account for cache contention. Saez et al. [24] and Koufaty et al. [13] independently describe a scheduler for AMPS based on the efficiency specialization principle. Their schedulers implement a strategy similar to the maximum-local policy presented in this paper, but their system targets performance asymmetry instead of memory system asymmetry (compute-bound processes are scheduled onto high performance cores with larger priority than memory-bound processes).

7. Conclusions

We have shown that operating system scheduling fails to obtain good performance in NUMA-multicores if it does not consider the structure of the memory system, and the allocation of physical memory in the system. If memory allocation in a NUMA-multicore system is balanced (the cumulative memory demand of processes homed on each processor in the system is approximately the same), then it is beneficial to simply map processes onto the architecture so that data locality is favored, and avoiding cache contention does not bring any benefits. Nonetheless, when the memory allocation in the system is unbalanced (the sum of the memory demands of processes homed on each processor in the system is different), then mapping processes so that data locality is maximized can lead to severe cache contention. In these cases refining the maximum-local mapping so that cache contention is reduced improves performance, even with the cost of some processes executing remotely. The N-MASS scheme described in this paper successfully combines memory management and process scheduling to better exploit the potential of NUMA-multicore processors.

Acknowledgments

We thank Albert Noll, Michael Pradel, Oliver Trachsel, Faheem Ullah and the anonymous referees for their helpful comments.

References

[1] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *PACT'10*.

[2] M. Banikazemi, D. Poff, and B. Abali. PAM: a novel performance/power aware meta-scheduler for multi-core systems. In *SC'08*.

[3] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 2010.

[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA'05*.

[5] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 2008.

[6] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *ATEC'05*.

[7] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT'07*.

[8] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *MICRO 42*, 2009.

[9] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based QoS techniques for cache/memory in CMP platforms. In *ICS'09*.

[10] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, January 2011.

[11] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT'08*.

[12] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 2008.

[13] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *EuroSys'10*.

[14] H. Li, H. L. Sudarsan, M. Stumm, and K. C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *ICPP'93*.

[15] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC'07*.

[16] Z. Majo and T. R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *SYSTOR'11*.

[17] J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccNUMA systems. In *PPoPP'06*.

[18] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross core interference through contention synthesis. In *HiPEAC'11*.

[19] J. Mars, N. Vachharajani, M. L. Soffa, and R. Hundt. Contention aware execution: Online contention detection and response. In *CGO'10*.

[20] D. Molka, D. Hackenberg, R. Schne, and M. S. Müller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *PACT'09*.

[21] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS'09*.

[22] T. Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *OOPSLA'09*.

[23] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39*, 2006.

[24] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore processors. In *EuroSys'10*.

[25] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *SC'10*.

[26] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *ASPLOS '09*.

[27] M. M. Tikir and J. K. Hollingsworth. Hardware monitors for dynamic page migration. *Journal of Parallel and Distributed Computing*, 2008.

[28] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *ASPLOS'96*.

[29] S. Zhuralev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS'10*.