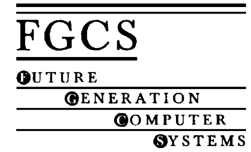
ELSEVIER

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Future Generation Computer Systems 19 (2003) 761–776



www.elsevier.com/locate/future

ARS: an adaptive runtime system for locality optimization

Jie Tao*, Martin Schulz, Wolfgang Karl

LRR-TUM, Institut für Informatik, Technische Universität München, 80290 München, Germany

Abstract

Shared memory programs running on Non-Uniform Memory Access (NUMA) machines usually face inherent performance problems stemming from excessive remote memory accesses. A solution, called the Adaptive Runtime System (ARS), is presented in this paper. ARS is designed to adjust the data distribution at runtime through automatic page migrations. It uses memory access histograms gathered by hardware monitors to find access hot spots and, based on this detection, to dynamically and transparently modify the data layout. In this way, incorrectly allocated data can be moved to the most appropriate node and hence data locality can be improved. Simulations show that this allows to achieve a performance gain of as high as 40%.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Page migration; Data locality optimization; Shared memory programming on NUMA; Hardware monitor

1. Introduction

Due to their excellent price–performance ratio, clusters built from commodity nodes have become increasingly popular as platforms for parallel processing. Among them, clusters of standard PCs interconnected with high-speed System Area Networks (SANs) are especially attractive and have been widely established. At the same time, the developments in interconnection technologies also formed the basis for the rise of Non-Uniform Memory Access (NUMA) architectures, i.e. systems with physically distributed memories, but with a global address space allowing an efficient but non-uniform access to any memory location in the system. These kinds of systems, especially when offered as non-cache coherent NUMA for loosely coupled commodity architectures, can easily be implemented in a straightforward manner without major hardware efforts. They form a favorable architectural tradeoff by combining the scalability and cost-effectiveness of standard clusters with a shared memory support close to symmetric multiprocessors.

The non-uniform memory access characteristic, however, introduces a distinction between local and remote memory causing different memory access latencies. In systems with such characteristics, a remote memory access can take up to one or two orders of magnitude longer than a local one. For the programmer, this difference is generally indistinguishable as shared memory programming models work on the assumption of a single uniform global address space. This situation can lead to extensive remote memory accesses, especially with rising numbers of nodes, and hence to a higher percentage of remote memory accesses in the overall system. Therefore, many shared memory applications initially do not achieve a good parallel speedup when running on NUMA-like architectures.

Manual optimizations [11,12,18,21] with respect to data placement can improve data locality, but they cannot solve this problem completely since these methods are not suitable for applications with dynamically

* Corresponding author.
E-mail addresses: tao@in.tum.de (J. Tao), schulzm@in.tum.de (M. Schulz), karlw@in.tum.de (W. Karl).

0167-739X/02/$ – see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0167-739X(02)00183-8

changing access patterns. Therefore, also a dynamic approach for locality optimization needs to be added which is capable of significantly reducing remote data accesses via an automatic runtime data redistribution.

Such an approach, called the Adaptive Runtime System (ARS), is presented in this paper. ARS is intended to adjust the data distribution at runtime during the execution of an application. It uses memory access histograms as the basis for its analysis of access patterns, and based on this analysis, dynamically and transparently modifies the location of data. Access hot spots and communication bottlenecks can be corrected in this way resulting at the end in a better runtime data layout and a performance gain.

The memory access histograms deployed by ARS for determining migrations are collected by hardware monitors. This hardware-based monitoring facility is designed to snoop all network transactions on a node, hence enabling a non-intrusive monitoring and an acquisition of comprehensive performance data which allows to explore novel migration algorithms with a larger decision base.

Based on these capabilities, several page migration algorithms, called *Out-U*, *Out-W*, and *In-W*, are investigated within ARS. These algorithms vary in their base, i.e. the amount and type of monitoring information, for making migration decisions. The *Out* algorithms make their migration decisions about whether to move a local page to a remote node according to the monitoring of outgoing memory traffic initiated by the local node. They implement hence a kind of *push-migration*. Correspondingly, the *In* algorithm, which implements a *pull-migration*, decides whether to bring a remote page to the local node depending on the monitoring of incoming memory traffic from remote nodes. On the other hand, the *U* algorithm establishes its analysis on information from memory accesses performed on a single page, while the *W* algorithms base their analysis on memory references to multiple shared pages, in a way that the accesses to a set of pages are combined using a weighted distribution.

First experimental results suggest that the *W* algorithms are in most cases better than the corresponding *U* algorithms due to their increased decision base leading to earlier and more correct migrations. Comparing the *Out* and the *In* algorithms, it depends on the individual application and its memory access behavior whether one algorithm is more efficient than the other. While the *In* algorithm behaves better in some cases, the *Out* algorithm introduces higher performance improvement in some other cases.

Besides these novel migration algorithms, ARS provides a graphical user interface which has been developed to visualize the actual data migration and page movement performed in the system. This allows a study of the migration behavior and an improvement of migration algorithms. The ARS GUI complements a previously developed tool called the Data Layout Visualizer (DLV) [21] which is used to present an application's memory access behavior in a human-readable and easy-to-use way, enabling the understanding of an application's access pattern as well as the location of memory access bottlenecks and communication hot spots. This can be used to explicitly optimize the source code with respect to data locality. These two approaches, i.e. static optimization supported by DLV and dynamic migration performed by ARS, complement each other and together form a general approach for improving data locality of applications with either static or dynamic access patterns.

The remainder of this article is structured as follows. Section 2 briefly outlines a few previous approaches for improving data locality using data migrations. Section 3 introduces the target system in combination with the hardware monitor providing performance data. This is followed by a detailed discussion of the ARS approach in Section 4, including the framework, the proposed migration algorithms, and the graphical user interface. In Section 5, first experimental results are presented with a focus on the selection of the migration criteria and the comparison of the migration algorithms. The paper is rounded up with some concluding remarks in Section 6.

2. Related work

Data locality on NUMA machines has been addressed over the last years. A significant amount of approaches has therefore been proposed for improving data locality through reducing accesses to remote memories. In addition to those schemes focusing on static data-reordering based on compilers [3,6,11,18], several approaches based on page migration have been implemented. This section briefly describes a few of

such approaches. In addition, a few approaches for thread migration have to be mentioned, as their techniques can in principle be applied to page migration as well.

Verghese et al. [22] study the performance improvement on CC-NUMA systems with OS supported dynamic migration and replication. This kind of page migration is based on the information about full-cache misses collected via instrumentation in the OS. Hot pages, i.e. pages on which a large number of misses occur, are migrated if referenced primarily by one process or replicated if referenced by many processes. Results of their experiments show a performance increase of up to 29% for some workloads. This approach, however, relies on software instrumentation and hence introduces high overheads.

Nikolopoulos et al. [13] present two algorithms for moving each virtual memory page to the node that performs the most references. The purpose of the page movement is to minimize the maximum latency due to remote memory accesses. One algorithm works with iterative parallel programs and is based on the assumption that the page reference pattern of one iteration will be repeated throughout the execution of the complete program. The other algorithm checks periodically for hot memory areas and migrates pages with excessive remote references. Both algorithms assume compiler support for identifying hot memory areas, i.e. memory areas which are likely to concentrate excessive remote accesses. Performance evaluations on an SGI Origin2000 show a significant improvement in throughput. However, this approach is based on the limited static analysis done by the compiler and hence is incapable of adjusting to runtime behavior. The information for making migration decision is therefore incomplete and potentially inaccurate.

Amber [2] is a multithreaded object-based system for distributed computations on networks of workstations. In Amber data is not moved (or replicated) to the location of the accessing thread, but rather, a thread that invokes an operation on a remote object is moved to the node where the object resides. This kind of migration is done explicitly by programmers using mobility primitives. In this approach, the programmer is burdened with the task of controlling the location of objects, which again requires extensive knowledge.

The RAHM (Remote Access Histories Mechanism) [8,16] is a technique that uses remote access histories for thread migration in order to improve the locality of memory references in distributed shared memory systems. The goal of the thread migration is maximal locality. Therefore, only those threads are selected to migrate, whose migration is expected to minimize the number of remote memory references for both the source and destination host. The information supporting thread selections are remote-reference histories collected from a statistical component in the underlying DSM system.

MCRL [5] is a multithread, distributed shared memory system that implements a dynamic choice between data and computation migration. MCRL data objects (programmer-defined regions) are managed using a home-based sequentially consistent protocol. When a processor accesses a region that is not cached locally, MCRL contacts the region's home node. The home node then decides between data or computation migration. This decision is made depending on two policies: the "static" policy always migrates computation for remote writes and data for remote reads, while the "repeat" policy always migrates computation for remote writes and dynamically chooses data migration or computation migration for remote reads. These policies can improve performance if the computation accesses a single large data region, but imposes unnecessary overhead if small regions are accessed by only one processor.

In the Olden [1] project, compiler support is used to statically determine whether to migrate computation or data. In this approach, the programmer is required to give the compiler indirect knowledge about the data layout. Such a strategy works well for applications with a predictable data structure layout and predictable access patterns. However, a static decision between computation and data migration does not work for applications with unpredictable or dynamically changing data access patterns.

Overall, these previously proposed approaches have shown that dynamic migrations are capable of introducing significant performance gains. However, all of them make the migration decisions depending on the memory access histograms gathered by software with support of the operating system, the compiler, or other memory management mechanisms. This information has to be either inaccurate, incomplete, or associated with a high probe overhead. In order to avoid this problem, the ARS approach establishes its migration

decision on information gathered by hardware monitors with only a minimal probe overhead and without the involvement of compilers and the necessity to intrude the operating system. Besides this, ARS explores algorithms which use a larger decision base enabling a comprehensive analysis and earlier migrations.

In addition, unlike all previous approaches, ARS deploys graphical views to show the actual data movement and the migration flow of virtual pages. This allows a better understanding of a program's memory access behavior and an improvement of migration mechanisms. In combination with the Data Layout Visualizer [21], the user is capable of understanding the runtime behavior of the application and to spot and correct performance bottlenecks or misbehaviors of the automatic adaptation heuristics.

3. Target system

ARS is currently established on top of the NUMA characterized SMiLE-like PC clusters. SMiLE stands for *Shared Memory in a LAN-like Environment* [15] and is a project broadly investigating in SCI-based cluster computing.[1] SCI (Scalable Coherent Interface) [4] is an IEEE-standardized [7] interconnection technology with extremely low latency (<2 μs) and very high bandwidth (>300MB/s for modern PC architectures). In addition, SCI provides a single physical address space across processor nodes in hardware, supporting the establishment of a distributed shared memory. This forms the base for hybrid DSM systems. Such a system, called the SCI Virtual Memory (SCI-VM) [10], has been developed within SMiLE and serves as the base for a software framework HAMSTER (Hybrid-dsm-based Adaptive and Modular Shared memory archiTEctuRe) [14]. This framework enables the establishment of arbitrary shared memory programming models on top of a single core. Based on HAMSTER, existing shared memory applications can easily be ported to and executed on the SMiLE-like PC clusters.

Like it is the case on any of the other NUMA machine, shared memory programs initially do not show a high speedup on the SMiLE system due to excessive remote references. In order to understand the shared memory traffic, a hardware monitor [9] has been designed which is capable of observing the interconnection fabric with minimized probe overhead. Based on this monitoring device, an integrated monitoring infrastructure has been built with the goal of optimizing the data locality of NUMA-based shared memory applications.

As shown in Fig. 1, the monitoring infrastructure is composed of three modules: the execution environment forming the data acquisition system, the tool middleware responsible for creating a global view of the acquired information and for interoperability issues, and the shared memory tools capable of steering the execution of parallel programs.

The execution environment itself (right in Fig. 1) is clearly layered with the SMiLE components described above including the hardware, i.e. the cluster, the SCI adapter card, and the hardware monitor, as well as the software infrastructure, i.e. the drivers and APIs, the SCI-VM, HAMSTER, and the programming models. Combined, they contribute information about memory accesses which is gathered by the hardware monitor and information about, e.g. address mapping and synchronizations which is provided by the software layers.

As the main component supplying performance data, the SMiLE hardware monitor has been developed to trace shared memory transactions on SCI-based clusters. The need for a hardware monitoring facility stems from the fact that shared memory traffic by default is of implicit nature and performed at runtime through transparently issued load and store operations to remote data locations. In addition, shared memory communication is very fine-grained (normally at word level). This renders code instrumentation recording each global memory operation infeasible since it would slow down the execution significantly and thereby distort the final monitoring to a point where it is unusable for an accurate performance analysis. The only viable alternative is therefore to deploy a hardware monitoring facility for observing the actual link traffic of the NUMA interconnection network. Only this guarantees fine-grained information about the actually inferred communication with a minimal influence onto the actual execution behavior.

The SMiLE hardware monitor is designed to be attached to an internal link on current PCI–SCI bridges, the so-called B-Link. This link connects the SCI link

[1] More information at http://smile.in.tum.de/.

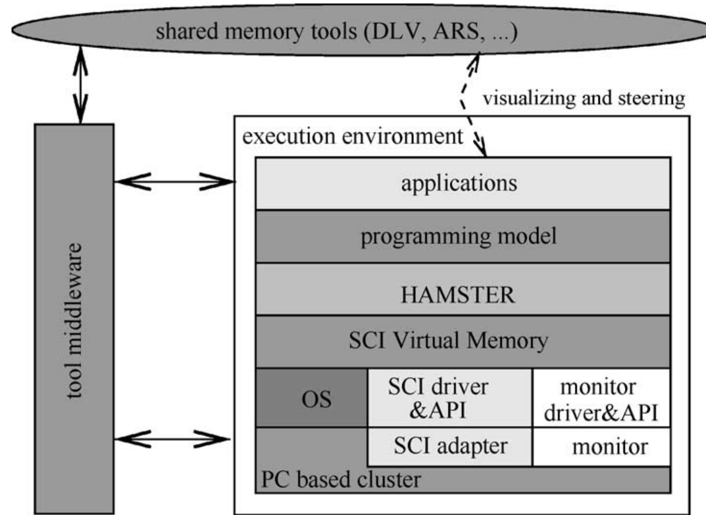


Fig. 1. The SMiLE monitoring infrastructure.

chip to the PCI side of the adapter card and is hence traversed by all SCI transactions intended for or originating from the local node. Due to the bus-like implementation of the B-Link, these transactions can be snooped without influencing or even changing the target system and can then be transparently recorded and preprocessed by the SMiLE hardware monitor. As the monitoring data is low-level and fine-grained, a software infrastructure including a driver and a C-API has been developed to further process the monitoring information. The results are so-called *memory access histograms* which show the numbers of memory accesses across the complete virtual address space of an application's working set separated with respect to target node IDs. These histograms form the base for any locality optimizations.

Locality optimization is a complex and difficult task requiring the support of performance tools. Currently, two shared memory tools have been implemented including the DLV and the ARS. DLV [21] visualizes the memory access behavior and pattern allowing an easy detection of communication bottlenecks. In addition, DLV projects virtual addresses into the data structure enabling an explicit optimization of the source code with respect to data placement.

However, it is found that DLV is not adequate for applications with irregular access patterns. For these cases, ARS, an Adaptive Runtime System, has been implemented in order to handle the dynamic behavior of applications. Together, DLV and ARS form a general solution for locality issues on NUMA architectures, where DLV enables an initial data placement and ARS allows a runtime adjustment.

4. The ARS approach

Shared memory programs running on NUMA machines suffer from the memory access latency induced by excessive remote memory references. While the performance of some applications can be improved by manually optimizing the source code with respect to data placement, others that exhibit dynamically changing access patterns can only be tuned by runtime redistribution of data or computation. ARS implements such a mechanism that migrates shared data during the execution of a program.

4.1. Framework

ARS is capable of analyzing the monitoring information, finding the communication hot spots, determining the optimal location of shared data, and initiating appropriate page migrations. For this purpose, both a decision-making mechanism and a migration mechanism are implemented. The decision-making mechanism reads data from the hardware monitor. The monitoring data is not directly transferred to

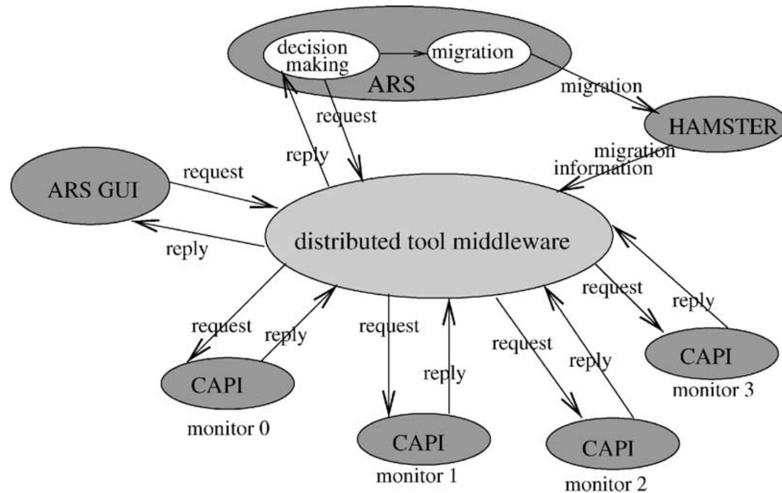


Fig. 2. Data transfer between ARS, DLV, tool middleware, and the hardware monitors.

ARS, but delivered via a distributed tool middleware [19] shown in Fig. 1. This enables the integration of arbitrary tools with the system execution environment.

The ARS decision-making mechanism periodically checks for migrations. It calculates the memory accesses and makes its decision according to the chosen migration algorithms. The migration decisions are then delivered to the migration mechanism, which informs the memory management module of HAMSTER to perform the corresponding changes in the SCI-VM. In addition, migration information is transferred to the ARS GUI via the tool middleware for the visualization of the runtime migrations.

Fig. 2 shows the communications between these components. The decision-making module of ARS issues service requests to the tool middleware in order to acquire the monitoring information, while the latter interacts with the concerned hardware monitor(s) and replies the ARS requests with monitoring data. The decision-making module of ARS analyzes the memory access behavior and detects the incorrectly allocated pages. In case a migration decision is made, the migration mechanism as well as HAMSTER is informed. The latter performs the modification in the shared virtual memory and delivers the migration information as service replies further to the GUI of ARS enabling the visualization of the migration behavior.

Migration algorithms play a central role in developing a migration system. Commonly used page migration mechanisms [17,22] are based on competitive algorithms which migrate a page if the difference between the number of local references and the number of remote references from one node exceeds a predefined threshold. This scheme is easy to implement, hence a similar one, called *Out-U*, is also applied within ARS as a baseline for comparison.

In addition to *Out-U*, two new page migration algorithms are proposed. These novel algorithms, called *Out-W* and *In-W*, respectively, deploy not only the memory accesses on the relevant page but also access information about pages spatially neighboring this page. Since they take the spatial locality of memory accesses into account and use a larger decision base, these algorithms are likely to perform more accurate and timely page migrations. The main difference between *Out-W* and *In-W* is that they base their migration decisions on different monitoring information: *Out-W* only uses the outgoing memory traffic initiated by the local node, while *In-W* is based on the incoming traffic from all remote nodes.

As the hardware monitor does not observe local memory accesses, migration decisions made by these algorithms are based on the accesses performed by remote nodes. This possibly causes Ping-Pongs, in which a page is migrated to a remote node and later moved back again to the local node. However, first

experimental results have shown that ARS is capable of avoiding high Ping-Pongs by using an adequate threshold.

4.1.1. The Out-U algorithm

Out-U makes decisions whether to move a page from the local node to a remote node and can hence be classified as *push-migration*. The decision is based on the references performed to the single page from all remote nodes. If the difference between the accesses from the dominant note and the average accesses exceeds a given threshold, it is decided to move this page to this dominant remote node.

The main challenge connected with this approach is the tradeoff between earlier and correct migrations. Using this algorithm, a correct migration decision can be made only after a large amount of references have been issued, resulting in late migrations and thereby a loss in performance. On the other hand, if a decision is made based on only a small amount of references, many incorrect migrations may be caused. In order to solve this problem, another two algorithms are proposed which base their migration decisions on references performed on multiple pages and therefore are able to make a migration decision earlier, but still correct.

4.1.2. The Out-W algorithm

Out-W uses the number of relative references in order to decide the location of a page. The number of relative memory accesses to page P from node N is thereby calculated as the sum of weighted references from the same node to the pages spatially neighboring page P using the following formula:

$$R_{PN} = \sum_{i=0}^{n} W_i C_i$$

In this formula, W_i is a weight representing the importance of the ith page to page P and C_i is the number of references to page i, while n is the number of pages located on node N. The weight is assigned according to the distance of a page to page P, whereby a closer page is assigned with a higher weight due to the spatial locality of memory accesses. Concretely, page P has a weight of 1, page $P-1$ and $P+1$ a weight of $(1-1\times(2/n))$, page $P-2$ and $P+2$ a weight of $(1-2\times(2/n))$, and so on. Besides that, the neighborhood is restricted to the pages located on the same node of page P. This avoids the overhead of transferring the monitoring information to other nodes, by using only the monitoring information provided by the local hardware monitor.

In order to determine the location of a page, the relative references from all remote nodes are compared. If the difference between the number of relative accesses from the dominant node and the average relative accesses exceeds a threshold, it is decided to move the page to the dominant remote node. Here, the same threshold as *Out-U* is used.

The advantage of *Out-W* over *Out-U* comes from the fact that theoretically spatially neighboring pages have similar access behavior due to the spatial locality of memory accesses. This means that if a node predominately accesses a page, it is also likely to access its neighboring pages in the same way. Therefore, the behavior of neighboring pages can be used to determine the location of this page. The benefit is that, due to the higher number of accesses, a migration decision can be reached earlier since the sum of the memory accesses (even weighted) performed by a node on a page and on its neighboring pages can be several times the accesses to the single page. As the *Out-W* algorithm calculates the relative accesses of a node on a page, which is a sum of weighted memory accesses from this node to the page and to its neighboring pages, the value for comparing used by *Out-W* (difference between the dominant relative accesses and the average relative accesses) is greater than the one used by *Out-U* (difference between the dominant accesses and the average accesses).

As the migration decision is made by comparing this value with the same threshold, *Out-W* has the tendency to migrate a page earlier than *Out-U* due to its bigger comparison value gained by aggregating the information of neighboring pages. This has the potential to result in a higher gain in performance if the decision is correct. The first experimental results, which will be presented in the next section, have shown the correctness of migrations performed by the *Out-W* algorithm.

4.1.3. The In-W algorithm

While *Out-W* determines whether to move a local page to a remote node, i.e. performs a *push-migration*, *In-W* decides whether to migrate a remote page to

the local node, i.e. *pull-migration*. For this purpose, it uses the monitoring information about the incoming memory traffic from remote nodes to determine the frequency of a remote page being accessed by the local node.

To make the migration decisions, *In-W* calculates the number of relative references to all remote pages accessed by the local node. It uses the same formula as the *Out-W* algorithm, but involves accessed remote pages (not local pages) into the calculation. If the difference between the number of relative accesses on a remote page and the average relative accesses performed on all remote pages exceeds a threshold, this remote page is brought to the local node.

While *Out-W* relies only on local pages, *In-W* allows to integrate consecutive remote pages. Hence, *In-W* can potentially be more accurate than *Out-W* for some data allocation schemes, like *Round-robin* which allocates shared data cyclically over all nodes on the system. Consider a four-node system using *Round-robin*, for example. According to this data allocation scheme, pages $0, 4, 8, \dots, 4i+0, \dots$ are placed on node 0, pages $1, 5, 9, \dots, 4i+1, \dots$ are placed on node 1, pages $2, 6, 10, \dots, 4i+2, \dots$ are placed on node 2, and pages $3, 7, 11, \dots, 4i+3, \dots$ are placed on node 3 (see the upper graph in Fig. 3). When determining the optimal location for page 100 which is located on node 0, for instance, *Out-W* takes pages $\dots, 94, 96, 104, 108, \dots$ as its neighbors, which are located on the same node as page 100. However, it is clear that these pages are not directly neighboring page 100 and those with a longer distance potentially have a different access behavior. In contrast, *In-W* uses the incoming traffic from all remote nodes. Again for page 100, *In-W* decides whether to bring it from node 0 to node 1, 2, or 3. For node 1, for example, it uses pages $\dots, 98, 99, 102, 103, \dots$ (see the second column of the lower graph in Fig. 3) to make the decision. As it handles all direct neighbors except those located on the local node, *In-W* has the potential to better use the spatial locality of the memory accesses.

The *In-W* algorithm, however, is more expensive. All nodes, except the one on which a page is located, are possibly the new locations for a page. Hence, each node performs the decision process for one page introducing more overhead than *Out-W*.

node 0	node 1	node 2	node 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
....	....	....	....
88	89	90	91
92	93	94	95
96	97	98	99
100	101	102	103
104	105	106	107
108	109	110	111
112	113	114	115
....	....	....	....

Data allocation on a 4 node system (information for OutW)

node 0	node 1	node 2	node 3
1	0	0	0
2	2	1	1
3	3	3	2
5	4	4	4
6	6	5	5
7	7	7	6
9	8	8	8
10	10	9	9
...	...	...	...
98	98	99	98
99	99	100	100
101	100	101	101
102	102	103	102
103	103	104	104
...	...	...	...

Incoming traffic seen by each node (information for InW)

Fig. 3. Data allocation using Round-robin and incoming traffic seen by processor nodes.

4.2. Graphical user interface

In order to allow an understanding of the migration behavior and an improvement of the migration algorithms, a graphical user interface has been developed. This GUI provides several representations to show the actual data migrations, page movements, and data locations. Fig. 4 illustrates three sample displays. The *runtime migration* (middle) presents the actual page movements using items composed of circles and arrows, with the top circle representing the source node, the bottom circle representing the destination node, and the label next to the arrow, which stands for the direction of the migration, showing the number of the migrated page. Items are dynamically added to the window whenever a migration occurs.

The *page show* (left in Fig. 4) illustrates all page movement during the entire execution. The most left rectangle stands for the initial location of a page and

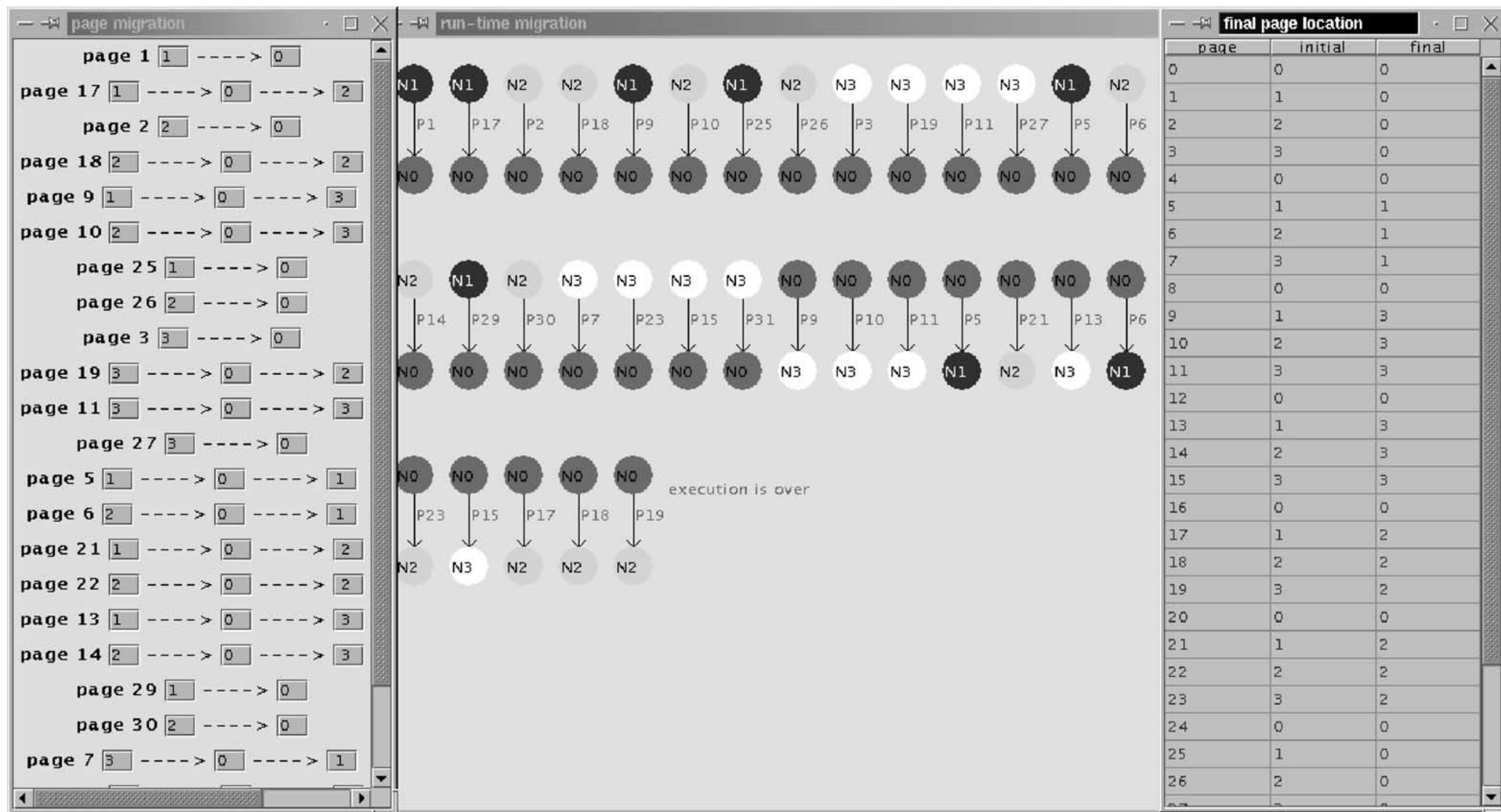


Fig. 4. ARS GUI display windows.

the most right stands for its final location, while the rectangle(s) in the middle are the nodes where the page has ever resided in the case of repeated migrations. This view is especially helpful for detecting Ping-Pong scenarios. An example is page 5 which is initially allocated on node 1, migrated to node 0, and finally migrated back again to node 1. This enables the evaluation and improvement of the data migration algorithms.

The last window *page location* (right in Fig. 4) shows the initial and final location of all shared pages. This window can be used to understand the parallel nature and the data structure of applications.

5. Experimental results

In order to evaluate the migration algorithms described above and the ARS approach itself, a large number of experiments have been done using standard benchmark applications. These experiments focus on three issues:

1. Which threshold is suitable?
2. Is there a single "best" migration algorithm?
3. What is the impact for not having local access information?

5.1. Benchmark applications

The benchmark applications are mostly chosen from the SPLASH-2 Benchmark suite [23], except the Successive Over Relaxation (SOR) code, a self-coded numerical kernel used to iteratively solve partial differential equations. Table 1 shows a brief description of these applications and their simulated working set sizes.

Table 1
Applications and their workload sizes

Applications	Description	Working set size
FFT	Fast Fourier transformation	2×14 data points
LU	LU decomposition for dense matrices	28×128 matrix
RADIX	Integer radix sort	262 144 keys
WATER	Evaluation of water molecule systems	343 molecules
SOR	Successive over relaxation	200×200 grid

FFT performs a fast Fourier transformation with a regular communication pattern between processor nodes. LU performs an LU decomposition for dense matrices, where the matrix is divided into blocks and assigned to different processors for parallel computation. RADIX implements an integer radix sort and relies on an all-to-all communication. SOR is a numerical kernel that is used to iteratively solve partial differential equations. Its main working set is a large dense matrix which is split into blocks of equal size during the parallelization process. Each processor computes one block and requires only the data assigned to the neighboring processors for exchanging boundary rows.

Unlike the above applications in which data is regularly accessed by processors, WATER shows a different access pattern. WATER is an *N*-body molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. It uses a predictor–corrector method to integrate the motion of the water molecules over time. Both the calculation process and the helper process access the data, but in an irregular way. Hence, this program requires a dynamic processing with respect to data allocation.

5.2. Evaluation platform

As the hardware monitor is still under development, an event-driven multiprocessor simulator, called SIMT, has been developed serving as an evaluation platform. SIMT [20] was originally designed to simulate the SMiLE hardware monitor and to provide the exact monitoring information when a hardware monitor is not available. Besides this initial task, it has been consequently continued to the point that it allows the simulation of other NUMA machines and a transparent transformation between the simulation platform and the real hardware. It therefore forms a general tool for system design and performance evaluation.

SIMT comprises a front-end which simulates the parallel execution of a program and a backend which simulates the target architecture. The front-end is a memory reference generator capturing events like memory references and synchronization events. The backend consists of functionality simulating the handling of these events on real hardware. As SIMT focuses on the simulation of the complete memory hierarchy, the main components of the backend are a

flexible cache simulator with several cache coherence protocols, a shared memory simulator with various memory management policies, a network mechanism modeling the data transfer across the NUMA fabric, and a monitor simulator modeling the SMiLE hardware monitor. In addition, at the end of each simulation, SIMT provides performance data such as the monitoring output and information about the cache behavior.

5.3. Selection of the threshold

The first step for the performance evaluation is to find an appropriate threshold for the migration algorithms. This threshold is defined as a factor (referred to as threshold factor in the following) of the average accesses on the page under consideration. In order to study the impact of threshold factors, all benchmark applications were executed using different factors. Figs. 5 and 6 present the results.

Fig. 5 shows the simulated execution time versus the factor used in the threshold. It can be seen that for all applications the execution time is the same with a factor of over 3. This stems from the fact that memory accesses from any node cannot exceed three times of the average accesses on a four-node system. Hence no migration is performed. For other cases, a factor of 2 behaves well with WATER and RADIX. For LU, FFT, and SOR there is no significant change when the factor varies from 1 to 3.

The better performance at a factor of 2 with WATER and RADIX is caused by correct migrations which is illustrated in Fig. 6. This figure presents the migration behavior of WATER on the left side and RADIX on the right side. The y-axis shows the simulated execution time (line graph), the number of migrations (dark bars), and the performed Ping-Pongs (light bars). The x-axis shows the factors used in the thresholds. For WATER it can be seen that 22 Ping-Pongs over the total 98 migrations are caused when using factors 1 and 1.5, and no Ping-Pong occurs with factors 2 and 2.5. For factor 2.5, however, since many pages which are predominantly accessed by remote nodes are not migrated, the execution is slower than that achieved with a factor of 2. For RADIX, 48 Ping-Pongs and a total of 201 migrations are performed with factor 1, 22 Ping-Pongs among 154 migrations are caused at factors 1.5, 1.7, and 1.9, and no Ping-Pong can be seen for factors 2–2.5. Therefore, factors 2–2.5 outperform the other cases, where factors 2 and 2.1 introduce the best performance due to their sufficient and correct migrations.

According to the above observation, a factor of 2 is chosen for the following evaluation of the migration algorithms. This threshold, even if specific for the benchmark applications, can be potentially used

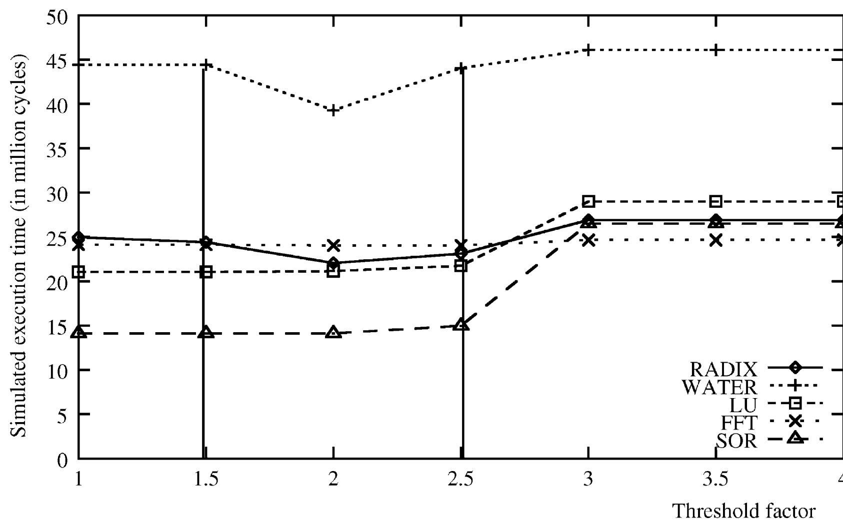


Fig. 5. Simulation time using different threshold factors.

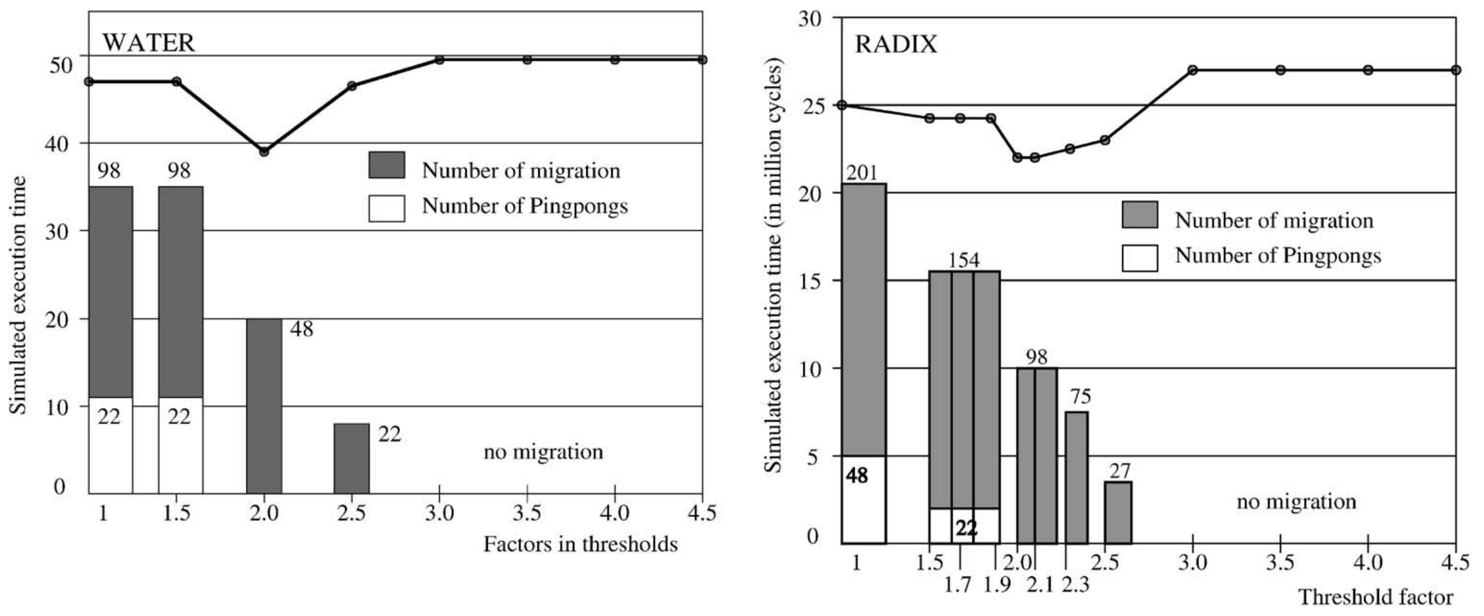


Fig. 6. Migration behavior of WATER (left) and RADIX (right).

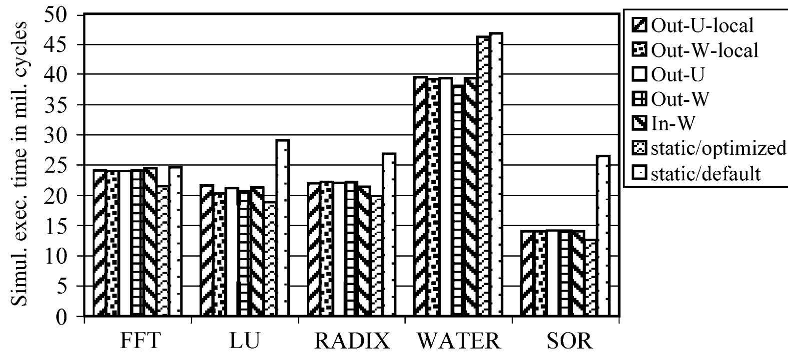


Fig. 7. Simulation time for different programs using Round-robin.

for other applications as well since it is chosen on the analysis of a large number of programs. In the next line of this research work, a flexible threshold will be considered with the ability of being automatically modified during the run of an application depending on the changes of the application's access pattern.

5.4. Evaluation of the migration algorithms

In order to evaluate the migration algorithms, the benchmark applications are executed on a four-node system using the simulated evaluation platform SIMT. Since SIMT, in contrast to the final hardware monitor, is capable of providing information about local references, for both *Out-U* and *Out-W*, additional algorithms are implemented. These algorithms, called *Out-U-local* and *Out-W-local* correspondingly, exploit local access information in order to examine the relevance of information about local references. Within them, the number of the dominant accesses are not compared with the average accesses as it is the case for *Out-U* and *Out-W*, but with the local references.

Fig. 7 illustrates the experimental results by applying the various migration algorithms. In all cases, the *Round-robin* scheme is used as the default allocation policy to initially distribute data. In addition, simulation results of an unoptimized as well as a manually, but statically optimized run are included for comparison. This kind of optimization is done within the source code by explicitly placing pages on the nodes which most frequently access them.

Examining the migration versions and the transparent default version, it can be seen that all programs run faster after enabling migration, independent of the deployed migration algorithm. The best performance improvement is gained by the SOR code, where a speedup as high as 1.88 is achieved. This stems from SOR's regular access behavior, where each page is accessed by only one node. When comparing the migration result with the manually optimized code, it can be observed that in most cases the optimized version is better. This can be explained by the fact that manual optimization enables an initial optimal data placement, timely and without overhead. However, the WATER code behaves differently: the migration version performs better than the optimized version. This is caused by the dynamically changing access pattern of WATER, where the same data is alternately accessed by different nodes in different phases. Due to its ability of moving data onto the accessed node in each phase, the ARS approach performs well for applications with such access patterns.

Comparing the individual migration schemes, it can be noted that the distance between the results of migration with or without local access information is insignificant. The information shown in Table 2 can give an explanation for this behavior. This table presents the number of total migrations, multiple migrations (a page is moved to a node and then to another node), incorrect migrations (a page is accessed primarily by the local node, but migrated to a remote node), and Ping-Pongs. The numbers of incorrect migrations in this table show that the *Out-U* and *Out-W* algorithms rarely migrate a page mainly accessed by the local node to a remote node, even though the information about local references is not available. Also, only one Ping-Pong is performed for all applications.

Table 2
Migration behavior (mig: total number of migration; p-p: Ping-Pong; mul: multiple migration; err: incorrect migration)

Application	*Out-U-local*				*Out-W-local*				*Out-U*				*Out-W*				*In-W*			
	mig	p-p	mul	err	mig	p-p	mul	err	mig	p-p	mul	err	mig	p-p	mul	err	mig	p-p	mul	err
FFT	41	0	0	0	89	0	0	0	69	0	0	0	27	0	0	0	96	0	0	3
LU	28	0	3	0	30	0	3	0	16	0	1	1	20	0	0	1	30	0	3	0
RADIX	81	0	0	0	191	0	0	0	98	0	1	1	98	0	0	0	106	0	0	0
WATER	26	0	0	0	89	0	29	0	48	0	1	0	119	0	43	0	33	1	1	0
SOR	24	0	0	0	24	0	0	0	25	0	0	1	27	0	0	2	26	0	0	2

Both indicate that the chosen threshold is adequate. In addition, Table 2 explains the abnormal behavior of WATER. It can be seen that many multiple migrations are performed, identifying that pages are alternatively accessed by more nodes. A static optimization placing a page on a fixed node is therefore not suitable and hence the migration result is better.

For the *U*- and the *W*-algorithm, Fig. 7 shows that, as expected, *Out-W* outperforms *Out-U* in case of LU and WATER. For FFT, RADIX, and SOR both algorithms behave similarly. The gain in performance by *Out-W* for LU and WATER is caused by more migrations, which can be seen in Table 2. The table also shows that these additional migrations are correct. In addition, these migrations have been analyzed using the GUI of ARS and it is found that they are performed in the earlier phase of the program's execution. Programs thereby benefit from the local references that would be remote if no migration was performed, despite the overhead introduced by the migrations. For *In-W*, however, the result is not as expected. In principle, *In-W* should be better than *Out-W* since it should be able to better use the spatial locality of the memory accesses. However, only the SOR code exhibits a gain. This is probably caused by the fact that *In-W* migrates a page from the local node to one of the remote nodes and the page is moved to the first node performing the check in case the migration threshold is exceeded. In this case, a page can be fixed on a node that does not perform the maximal accesses. For the SOR code, basically all pages are accessed by only one node. Hence all migrations, if performed, are correct as initiated by *In-W*.

In summary, the results of these first experiments show that a significant improvement has been achieved by the ARS approach. More importantly, it has been found that migrations based only on remote accesses do not introduce significant Ping-Pongs. This means that similar performance gain can be expected when applying the ARS approach on top of hardware monitors. As for the migration algorithms, it has been observed that the *W* algorithms generally behave better than the *U* algorithm, while the *In* algorithm does not show its expected benefit on a general basis, but can improve the performance for some applications.

6. Conclusions

High memory access locality is essential for good performance in NUMA-based environments. This is caused by the sometimes extreme differences in access latencies between local and remote memory modules. In addition to static optimization mechanisms and tools, it is also beneficial to provide dynamic and adaptive mechanisms. These can work without user interaction and require no prior knowledge of the application or even code modifications. In addition, they are also applicable to dynamic or irregular applications in which static optimizations fail.

In this work, a runtime system capable of performing such dynamic locality adaptations is presented. This system, called ARS, uses memory access histograms to analyze the memory access behavior and correct the access hot spots at runtime during the execution of an application. First experiments show that ARS is capable of significantly improving performance.

Unlike similar systems, ARS deploys hardware monitors to gather performance data and hence

performs data migrations with very low overhead. In addition, the hardware monitor supplies complete and accurate information, allowing ARS to investigate in migration algorithms capable of achieving timely and correct migrations. As an important feature, ARS also includes a graphical user interface which is capable of reporting the dynamic runtime behavior of the application back to the user in an on-line fashion. This gives the user, together with the DLV, a data layout visualization tool, a deep insight into the memory access patterns of the application and thereby enables further optimizations.

References

[1] M.C. Carlisle, A. Rogers, Software caching and computation migration in olden, in: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95, Santa Barbara, CA, July 1995, pp. 29–38.

[2] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, R.J. Littlefield, The amber system: parallel programming on a network of multiprocessors, in: Proceedings of the 12th ACM Symposium on Operating System Principles, December 1989, pp. 147–158.

[3] E.D. Granston, H.A.G. Wijshoff, Managing pages in shared virtual memory systems: getting the compiler into the game, in: Proceedings of ACM 1993 International Conference on Supercomputing, Tokyo, Japan, July 1993, pp. 11–20.

[4] H. Hellwagner, A. Reinefeld (Eds.), SCI: Scalable Coherent Interface: Architecture and Software for High-Performance Computer Clusters, Lecture Notes in Computer Science, vol. 1734, Springer, Berlin, 1999.

[5] W.C. Hsieh, M.F. Kaashoek, W.E. Weihl, Dynamic computation migration in DSM system, in: Proceedings of the Supercomputing'96, ACM Press and IEEE Computer Society Press, Pittsburgh, November 1996.

[6] Y.C. Hu, A. Cox, W. Zwaenepoel, Improving fine-grained irregular shared-memory benchmarks by data reordering, in: Proceedings of the SC2000 on High Performance Networking and Computing, Dallas, TX, USA, November 2000, pp. 61–74.

[7] IEEE Computer Society, IEEE Std 1596–1992: IEEE Standard for Scalable Coherent Interface, The Institute of Electrical and Electronics Engineers Inc., New York, NY, USA, August 1993.

[8] A. Itzkovitz, A. Schuster, L. Shalev, Thread migration and its applications in distributed shared memory systems, J. Syst. Softw. 42 (1) (1998) 71–87.

[9] W. Karl, M. Leberecht, M. Oberhuber, SCI monitoring hardware and software: supporting performance evaluation and debugging, in: SCI Scalable Coherent Interface Architecture and Software for High-Performance Compute Clusters, Lecture Notes in Computer Science, vol. 1734, Springer, Berlin, 1999, Chapter 24, pp. 417–432.

[10] W. Karl, M. Schulz, Hybrid-DSM: an efficient alternative to pure software DSM systems on NUMA architectures, in: Proceedings of the Second International Workshop on Software DSM (held together with ICS 2000), May 2000. http://www.cs.rutgers.edu/∼wsdsm00/.

[11] A. Krishnamurthy, K. Yelick, Analyses and optimization for shared space programs, J. Parallel Distrib. Comput. 38 (2) (1996) 130–144.

[12] A.G. Navarro, E.L. Zapata, An automatic iteration/data distribution method based on access descriptors for DSMM, in: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing (LCPC'99), San Diego, La Jolla, CA, USA, August 1999.

[13] D.S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta, E. Ayguade, User-level dynamic page migration for multiprogrammed shared-memory multiprocessors, in: Proceedings of the 29th International Conference on Parallel Processing, Toronto, Canada, August 2000, pp. 95–103.

[14] M. Schulz, Efficient deployment of shared memory models on clusters of PCs using the SMiLEing HAMSTER approach, in: A. Goscinski, H. Ip, W. Jia, W. Zhou (Eds.), Proceedings of the Fourth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), World Scientific Publishing, December 2000, pp. 2–14.

[15] M. Schulz, J. Tao, C. Trinitis, W. Karl, SMiLE: an integrated, multi-paradigm software infrastructure for SCI-based clusters, in: Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid), Berlin, Germany, May 2002.

[16] A. Schuster, L. Shalev, Using remote access histories for thread scheduling in distributed memory system, Technical Report LPCR-9701, Computer Science Department, Technion, Haifa, Israel, January 1997.

[17] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, J. Hennessy, Flexible use of memory for replication/migration in cache-coherent DSM multiprocessors, in: Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98), June 1998, pp. 342–356.

[18] S. Tandri, T.S. Abdelrahman, Automatic partitioning of data and computations on scalable shared memory multiprocessors, in: Proceedings of the 1997 International Conference on Parallel Processing (ICPP'97), Washington, Brussels, Tokyo, August 1997, pp. 64–73.

[19] J. Tao, W. Karl, A tool environment for efficient execution of shared memory programs on NUMA systems, in: Proceedings of the Fourth International Workshop on Advanced Parallel Processing Technologies (APPT'01), Ilmenau, Germany, September 2001, pp. 156–165.

[20] J. Tao, W. Karl, M. Schulz, Using simulation to understand the data layout of programs, in: Proceedings of the IASTED International Conference on Applied Simulation and Modelling (ASM 2001), Marbella, Spain, September 2001, pp. 349–354.

[21] J. Tao, W. Karl, M. Schulz, Visualizing the memory access behavior of shared memory applications on NUMA architectures, in: Proceedings of the 2001 International Conference on Computational Science (ICCS), Lecture Notes in Computer Science, vol. 2074, San Francisco, CA, USA, May 2001, pp. 861–870.
[22] B. Verghese, S. Devine, A. Gupta, M. Rosenblum, OS support for improving data locality on CC-NUMA compute servers, Technical Report CSL-TR-96-688, Computer System Laboratory, Stanford University, February 1996.
[23] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 1995, pp. 24–36.

Jie Tao received her Master's degree in computer science from Jilin University of China in 1989. Then she worked at the same university first as an assistant and afterwards as an associate professor. In 1998 she came to Germany with the support of a German foundation. Currently she is pursuing her PhD at the Lehrstuhl für Rechnertechnik und Rechnerorganisation of the Technische Universität München and working on the project SMiLE and EP-CACHE. Her research interests include shared memory programming, scalable coherent interface (SCI), performance tools, monitoring, cluster computing, and data locality optimization.

Martin Schulz received his Master's degree at the University of Illinois at Urbana-Champaign in 1997 under the supervision of Dr. A. Chien and his doctorate degree at the Technische Universität München in 2001 under Prof. Dr. A. Bode. Since then he is working at Prof. Bode's Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR) as a postdoctoral assistant. For the last 5 years, he has been actively involved in the SMiLE project which broadly investigates cluster computing based on the scalable coherent interface (SCI) and includes both extensive hardware and software developments. Within this project, he was responsible for the development of HAMSTER, a flexible framework for shared memory programming on top of SCI-based clusters. Further interests include all aspects of coherency and consistency, optimizations of the memory subsystem, and parallel I/O, as well as real-world application studies.

Wolfgang Karl received his doctorate degree at the Technische Universität München in 1992. Currently he is a senior researcher at the Lehrstuhl für Rechnertechnik und Rechnerorganisation at the same university. His research interests include computer architecture, microprocessors, computer and systems design, parallel and distributed systems, distributed shared memory (DSM), scalable coherent interface (SCI), fault tolerance, cache architectures, and reconfigurable computing.