

Automatic Data Locality Optimization through Self-Optimization

Rainer Buchty, Jie Tao, Wolfgang Karl

Universität Karlsruhe (TH), Institut für Technische Informatik, 76128 Karlsruhe, Germany
{buchty|tao|karl}@ira.uka.de

Abstract. Data locality optimization in parallel systems is a non-trivial task. This task is typically done by the programmer: based upon an exhaustive analysis of an application's run-time behavior, data access and distribution is re-modeled manually. Once the system, application, or just the input data set changes this effort has to be repeated.

Ideally, this task can be automated which requires introduction of Self-X qualities into the system. We developed an architecture concept for self-organizing parallel computer systems. This architecture is based on two main principles which are flexible monitoring to instantiate self-awareness, and adaptive components for all aspects of self-configuration. It is completed by a self-awareness mechanism, the autonomic planning. These Self-X properties pervade all system layers.

Based on this architecture concept, we implemented an autonomic data locality optimization system. With the achieved results presented in this paper we successfully demonstrated suitability and applicability of the architecture concept and were able to highlight the benefits of autonomic data locality optimization.

1 Introduction and Motivation

Past prognoses always saw Moore's Law as the ultimate barrier for future system development. However, in modern systems the complexity has risen to an amount where not physical constraints but the complexity itself turns into a problem. While it is possible to build such complex systems, maintenance and especially application optimization are increasingly difficult to handle by humans.

Because of growing system complexity combined with increasing usability and reliability requirements, such tasks should be fulfilled automatically, i.e. the systems with self-organizing characteristics and capabilities are mandatory. In general, such systems require introspection to acquire system-wide state-information to tune for desired performance, energy consumption, reliability, security, or other metrics. In a typical self-organizing system, a monitor probe will acquire state, and then a steering component will adapt the system – either dynamically at runtime, or statically as in profile directed feedback techniques for future executions.

This requires flexible and powerful monitoring resources on all system layers. It is vital to avoid restriction to some few monitoring points as present in current systems: by extending monitoring to all system layers, it is possible to exploit emergence effects contributing to improved self-awareness.

Based on this data, a more suitable (i.e. optimized) system configuration can be determined. This requires presence of adaptive components on all system layers which then supply necessary reconfiguration capabilities. Amount of reconfiguration is determined by analysis of monitoring data and evaluation against given rules or problem specifications, so-called objective functions.

Objective functions describe the wanted, optimal system behavior. Because of their often contradictory nature, weighing these objective functions is a non-trivial task, and therefore different approaches and strategies exist to achieve what is considered the “optimal” solution. This weighing process is based on application type, field of use, and additional system conditions. Further complexity is added from the fact that objective functions are not necessarily one-dimensional but can also be multi-dimensional, aggregate functions.

We developed an architecture concept, ASoCS, which specifically supports design and programming of self-optimizing parallel and distributed computer systems. As a case study for using this architecture concept, we implemented an automatic data locality optimization based on the ASoCS concept. Data locality optimization is a well-known problem within the field of parallel programming: based upon an exhaustive analysis of an application’s run-time behavior, data access and distribution is re-modeled manually. Once the system, application, or just the input data set changes this effort has to be repeated. With our exemplary implementation we were able to demonstrate suitability and applicability of this concept and highlight the benefits of autonomic data locality optimization.

This paper is organized as follows: in Section 2 we present related approaches regarding the introduction of autonomous features, so-called Self-X capabilities such as Self-Awareness or Self-Optimization, for system and application design. We then introduce the ASoCS framework in Section 3 followed by an exhaustive discussion of our exemplary implementation of an autonomic data locality optimizer based on ASoCS principles in Section 4. The paper concludes with Section 5.

2 Related Work

Data locality optimization requires sophisticated system introspection by using various monitoring techniques. That way data access patterns and frequencies can be detected, which allow better use of local memory and cache resources.

The ASoCS concept was specifically designed to aid such introspection by a powerful monitoring concept explained in Section 3. Monitoring is able to not only gather data from various system layers, but also capable of processing, interpretation, and evaluation against given objective functions already in place without necessarily requiring a central post-processing instance.

In adaptive systems, this data serves to compute a possible, more suitable system status resulting in a system reconfiguration affecting all system layers. This optimization process ideally runs automatically without human interaction, i.e. the system turns into what is typically called an *autonomic system*.

Certain aspects of the above problems have been targeted in current and previous research activities and industrial initiatives. Within this section activities involving automatic optimization are discussed.

Initiatives targeting Autonomic Computing IBM's *Autonomic Computing* (AC) initiative [15] targets automated system administration without need for human interaction. The initiative covers performance, energy efficiency, reliability, and security [34, 35]. To achieve this goal, a system is partitioned into several components: *Tivoli Monitoring* [17] monitors the most important system resources. Collected data is evaluated against an optimizing objective function; this is done by the *Business Workload Manager* (BWLM).

Within their AC initiative, IBM already implemented self-repairing and query-optimizing components for their DB2 relational database. The query optimizer is part of *IBM's Learning Optimizer* (LEO) and was developed for the *Smart Managing and Resource Tuning* (SMART) database technology project. Additional servers, based on the *eLiza* project [16], are able to automatically manage computing and memory resources. Another part of this project is *Enterprise Workload Management* (eWLM), which among other features will enable performance self-optimization.

Similar initiatives exist, such as Sun Microsystems' *Network Virtualization Strategy* (N1) for dynamic allocation of network resources [28], or Hewlett-Packard's *Planetary Computing* project [4]. The latter is an architecture enabling supercomputing centers to automatically reconfigure software infrastructure, and assign needed memory and server resources.

In addition to the already mentioned industrial projects, numerous academic projects exist such as UC Texas' *EDGE architecture* [9] or the adaptive software from Michigan State University [10].

Common to all approaches is their limited scope resulting from the addressed scenario, automated system administration. Only certain applications, like database optimization or data storage, are addressed. Furthermore those approaches are usually targeting higher system levels and leave out low-level optimization on hardware level.

Feedback systems Feedback loops are e.g. used in real-time systems for industrial and automotive control: input data provided by sensors is collected and evaluated. Based on evaluation result, appropriate control takes place. Such feedback loops are prerequisites for self-organizing systems.

Current research interest is typically put on adjusting computing power and energy efficiency. Here, for instance, IPC count is used as a measure to adjust issue width and number of functional units [7]. Another approach is using miss ratio, IPC count, and jump ratio for detection of execution phases with intent to reconfigure caches and TLBs to optimally match phase requirements [3].

Another project, described in [23], targets the memory subsystem by measuring how much time elapses between consecutive accesses to an energy-aware memory. Based hereon, appropriate power mode and most energy-saving page allocation are determined.

Similar to monitoring systems, again a multitude of solutions exists, each targeting a special area. A generic and flexible architecture could be used as an universal framework for all types of feedback systems. In addition, feedback systems as described above greatly benefit from a system-wide monitoring infrastructure providing supplemental data.

Monitoring Feedback systems as introduced in the previous section require techniques to gather and process system parameters to be able to create a certain sense of self-awareness. These parameters can be collected on various system levels such as lowest hardware level, driver level, OS level, or application level.

On lowest hardware level, performance counters offer some rudimentary monitoring support. They are typically used to profile an application and investigate possible application optimizations such as enhanced data layout in memory to improve cache use. Modern processor architectures offer so-called event counter registers [1, 8, 19, 20, 30, 18, 29]. Number and use of these registers are dependent on the individual architecture: counter registers are either bound to certain events or can be more or less freely assigned [29, 19]. The majority of existing tools is based on these counter registers.

Counter-based methods typically suffer from basic limitations [37], and do not allow differentiation between events being triggered by speculative and non-speculative execution. False counts from speculation are addressed with the precise event-based sampling (PEBS) of Intel's Pentium 4 architecture [20, 36]. Similar, but less complex methods are implemented in the IBM's Power architecture [30, 18].

For architectures without such hardware support, monitoring can be achieved using plain software. For example, `gprof` [11] uses compiler-generated function prologues to log information such as called address and number of calls. To collect statistical information and enable mapping of execution times to functions, `gprof` periodically parses the program counter.

It is also possible to embed monitoring routines on driver level as demonstrated by the Myrinet-based Shrimp Cluster [24]. A combined hardware/software method was used on the SMiLE monitor for SCI networks [14].

Monitoring APIs as described in [32], [2], or [25, 26], decouple monitoring devices from post-processing software by offering an abstract programming interface rather than directly accessing the monitoring hardware.

The drawbacks of the above examples can be subsumed as follows: not only are existing monitoring infrastructures in hardware inherently fixed, they are typically very application-specific. In addition, no standardized, generic API exists to work with monitoring infrastructures. So far, several approaches for monitoring APIs exist, However, these are typically bound to certain application such as e.g. monitoring parallel systems [25–27].

An approach into this direction was taken within the APART project [13] and the associated EP-Cache project [12]. These, however, only target monitoring and analysis in parallel and distributed systems. With respect to self-organizing systems, still no uniform, application-independent, and standardized monitoring API exists including reporting existing monitoring resources, permitting access to these resources, and – regarding self-organization – enabling reconfiguration.

3 The ASoCS Architecture Concept

The data locality optimization (DLO) implementation closely follows the ASoCS architecture concept. This concept was previously presented in [6], therefore in this section we give a quick overview over our architecture as required to understand how the exemplary DLO implementation makes use of the concept.

3.1 Architecture Details

Integral part of the proposed architecture concept is a novel monitoring infrastructure. As previously explained, monitoring must take place on all system layers. It furthermore must be flexible to adhere to demands of the planning stage which computes necessary system reconfiguration according to monitoring data. To achieve this, monitoring will not be a monolithic part of respective system layers. Instead, it will be split into monitoring capsules and monitoring modules.

Monitoring capsules are embedded into all system layers and provide appropriate interfacing required to dock or plug monitoring modules into the respective layer. Monitoring modules represent the monitoring functionality consisting of the sensory part (data pickup) and defined pre-processing capabilities. Splitting the monitor resources into capsule (interface) and module (functionality) enables exchange of monitoring modules, thus the monitoring infrastructure itself can be reconfigured depending on the planning stage's needs. Monitoring modules will be stored in a repository from where they can be retrieved and docked into the appropriate monitoring capsule.

A dedicated API will serve as an abstraction layer and enable access to monitoring capsules and adaptive components by other system services such as the adaptive planning stage. This stage decides which monitoring modules are loaded into their respective capsules.

Data collected by the monitoring infrastructure will be buffered in a local performance repository. This data can be merged with other local data from additional system nodes resulting in a global performance manager enabling scalability of the entire performance repository system. A dedicated query interface provides access to all local performance repositories and the monitoring infrastructure.

This query interface is used by the adaptive planning stage which evaluates collected data against objective functions to determine appropriate system reconfiguration. This evaluation is based on certain metrics quantizing system parameters and objective functions; when adapting this architecture to distinct application, it is necessary to investigate and evaluate existing metrics for use in adaptive planning and to eventually develop novel metrics aiding in the evaluation process where needed.

The overall architecture concept basically enhances and refines a common control loop scheme: a system is split into a 5-tier hierarchical scheme: the bottom layer is formed by the system hardware, with the (Real-time) Operating System (OS) including hardware drivers on top. The OS is assisted by libraries which in term are required by the compiler to finally create the desired application. Depending on its type, an application might influence only some or all levels. On each hierarchy level monitoring capsules exist. These capsules can be loaded with monitoring modules stored in a mon-

itoring repository. Data collected and preprocessed by monitoring modules are stored in a local performance repository.

The architecture concept explicitly addresses parallel and distributed systems: local repositories of all system nodes can be retrieved by a global repository manager (GRM). This GRM is virtually centered between the local repositories of all nodes. Repositories are accessed by the adaptive planning stage (APS) through a dedicated query interface. APS then evaluates this data and determines required reconfiguration. To enable such reconfiguration, each hierarchy level contains adaptive components which are instrumented by APS.

In Section 2 several projects and initiatives were presented which more or less exhaustively target such closed control loop systems. So far, none of them addresses a uniform architecture for self-organizing or organic architectures, but rather focuses on certain applications or system aspects. Contrary to these, our concept is application- and system-independent, and addresses all system layers rather than specific system parameters. It is furthermore applicable to single nodes as well as parallel and distributed systems.

When applying the architecture concept, methods for data evaluation and computation of system reconfiguration based on monitoring data, objective functions, and additional data such as amount of possible reconfiguration must be investigated and developed with respect to the given application scenario. This also includes evaluation of existing metrics and eventually developing of novel metrics to be able to quantize requirements and reconfiguration efforts as required by the target application scenario.

4 Prototypical Implementation with Data Locality as Initial Objective Function

In order to evaluate the ASoCS concept, we built a prototypical architecture and developed several components for a feedback loop with respect to data locality optimization on NUMA architectures. The reason for choosing NUMA locality as the initial objective function lies in the fact that we have been doing research work in the area of shared memory programming on top of NUMA architectures [33].

A typical NUMA (Non-Uniform Memory Access) machine is comprised of several commodity PCs or workstations connected through modern interconnection technologies. On such a machine, the main memory is distributed over the system, but globally organized into a shared virtual memory accessible from all processor nodes. Due to the different property of local and remote memory accesses, however, references targeting a remote memory can take up to two orders of magnitude longer than local accesses. As a consequence, unoptimized applications often suffer from poor data locality and the resulting high memory access latencies. As the first step towards self-organizing computer systems, we tackle this locality issue on NUMA machines.

4.1 Structure of the Feedback Loop

Figure 1 depicts how we map this problem onto the general framework of ASoCS. First of all, such data locality optimization needs a set of system parameters such as memory

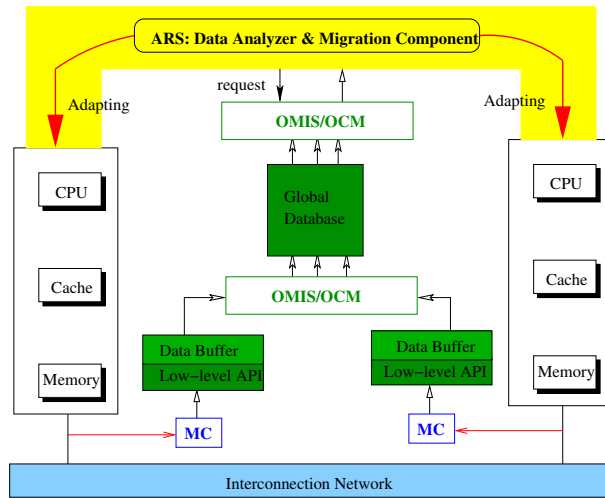


Fig. 1. Memory Locality Adapting on NUMA Systems.

access distribution and remote access characteristics. we assume Monitoring Capsules (MC) integrated in the NUMA network interface capable of observing all remote memory transaction. On top of these MCs, we use low-level APIs for preprocessing the original monitoring data. For each MC, a Data Buffer is maintained for storing the local monitoring information. From there, data is aggregated and combined into the Global Database. This work is done by the OMIS/OCM [40] monitoring interface. Besides, OMIS/OCM also provides a Query Interface that delivers the memory locality information to higher levels. Finally, an Adaptive Planning and an Adaptive Component are needed for self-tuning. The former is implemented with a Data Analyzer that automatically analyzes the runtime interconnection traffic and detects access hot spots, while the latter is achieved with a Migration Component that transparently modifies the data layout via moving data to its dominating node.

Monitoring Capsule: The design of this component is based on the SMiLE hardware monitor [22] which has been developed for observing the interconnection traffic on the SMiLE (Shared Memory in a LAN Environment) cluster connected via the Scalable Coherent Interface (SCI), a low-latency, high-bandwidth interconnection technology. For acquiring comprehensive data about the inter-node communication, we designed two analysis modules for monitoring the memory transaction: static and dynamic. The former allows to explicitly program the hardware for event triggering and action processing on special memory regions of interest, while the latter is based on histogram-driven monitoring, in which all memory transactions through the local interconnect bus are monitored in order to provide fine-grain monitoring statistics across the complete application’s working set.

A hardware Monitoring Capsule is also designed to hold these analysis modules, which can be dynamically loaded into the capsule at the runtime. The capsule implements two interfaces: a link interface and a PCI interface. The former is used to snoop a local bus, which connects a single node to the actual interconnection fabric, and extract

information from the transactions delivered over this bus, while the latter, an interface between the PCI bus and the monitor, offers direct access to the host node enabling the users or system software to configure the hardware monitor and read the gathered monitoring data.

Low-level API and the Data Buffer: In order to avoid delivering fully detailed monitoring data which is not essential for further use, we implemented a library of functions for data processing. This PAPI-like [5] standard API contains a set of routines capable of generating statistical information in the form of e.g. memory access histograms that show the number of accesses from all processor nodes to the whole working set at different granularity. These histograms can be used to detect communication bottlenecks where required data is mainly acquired from a remote processor node. For locally storing this monitoring data a Data Buffer is maintained on each processor. The buffer is organized as a histogram chain in order to enable fast searching of the needed information.

Data Aggregation and Querying Interface: We use the OMIS/OCM [39] monitoring system to combine monitoring data from all nodes. OMIS (On-line Monitoring Interface Specification) is a specification of an interface between a programmable on-line monitoring system for distributed computing and the tools that reside on top of it. It offers two interfaces: one for the interaction with different tools and the other for the interaction with the program and runtime system layers. OCM is an OMIS Compliant Monitoring system adhering to OMIS. It has been implemented for a series of loosely coupled environments including clusters and NoWs and has initially been designed for message passing tools. It is structured into a core and several extensions. For this work we extended OCM for providing services with respect to the access of data delivered by the Monitoring Capsules. Further, we extended OCM with a high-level Query Interface that provides the memory locality information to higher levels.

Adaptive Planning and Component: The remainder components include a Data Analyzer (Adaptive Planning) and a Migration Component (Adaptive Component). The former is used to analyze the monitoring data and determine whether to move a data page to another processor node. Based on the Querying Interface, the Data Analyzer is capable of accessing the memory access histograms created by the low-level API. It then compares the number of accesses to a data page from all processor nodes. If the accesses performed by a remote node exceed a predefined threshold, it is decided to move this page to the remote node. This decision is then delivered to the Migration Component, which uses system calls to move the data from the original location to the node that more requires it. This kind of adapting is performed periodically either at specified time or by synchronization points, and is held during the whole execution of the applications.

A critical issue with this approach is the migration algorithm used to make the migration decision. Commonly used page migration mechanisms are based on competitive algorithms, which migrate a page if the difference between the number of local references and the number of remote references concerning one node exceeds a predefined threshold. A similar one, called U-Mig, is also proposed within this work. As the local accesses can not be acquired by the monitors, the migration decision is based on the references performed by all remote nodes on the page under consideration. If the

difference between the number of the remote accesses from the dominant node and the average remote accesses performed on the page exceeds a threshold, it is decided to move the page to the dominant remote node.

Using this algorithm, however, a correct decision can be made only after a large amount of references have been issued, resulting in late migrations and thereby a loss of performance. Therefore, We implemented several novel migration algorithms, which 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.

An example is the so-called *W-Mig* scheme that uses the number of relative references to decide the location of a page. The number of relative memory accesses to page P from node N is 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 i th 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. Besides that, the neighborhood is restricted to the pages located on the same node of page P , since only these pages see the same remote nodes and hence their monitoring information includes the accesses from all P 's remote nodes. For any page located on another node, while a remote node seen by P is a local node, no access information from this node can be acquired.

Application	Description	Working set size
FT	Fast Fourier Transformations	$64 \times 64 \times 64$
LU	LU-decomposition for dense matrices	$32 \times 32 \times 32$
MG	Multigrid solver	$32 \times 32 \times 32$
CG	Grid computation and communication	1400
RADIX	Integer radix sort	262144 keys
OCEAN	Simulation of large scale ocean movements	130×130
WATER	Evaluation of water molecule systems	343 molecules
SOR	Successive Over Relaxation	1024×1024
Gauss	Gaussian elimination	512×512

Table 1. Description of benchmark applications

To determine the location of a page, the numbers of 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. The advantage of this algorithm 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, based on the higher number of accesses, a migration decision can be made earlier.

Summary: Overall, we implemented a closed feedback loop for adapting the data distribution on NUMA systems. At the same time, we also established the basis framework of the proposed architecture. Most components within this framework can be applied to build other feedback loops with slight extension. For example, we are currently working on an adaptation disk for improving the cache performance. Monitoring data is acquired from performance counters; the established databases and Query Interface are directly applied; the existing Adaptive Planning component is slightly extended; and a new Adaptive Component is under development.

4.2 Experimental Results

The prototypical implementation of the proposed architecture has been verified with standard applications. In this subsection, we discuss the achieved results.

Experimental Setup: Since the hardware Monitor Capsules are not yet available, we created a simulation environment based on SIMT [38]. SIMT is a multiprocessor simulator modeling the parallel execution of shared memory applications on NUMA machines. As it aims at research work on the memory system, SIMT contains mainly mechanisms for simulating the complete memory hierarchy in detail. This includes a flexible cache simulator which models caches of arbitrary levels and various cache coherence protocols, a DSM simulator which models the management of distributed shared memories and a set of data allocation schemes, and a network mechanism modeling the interconnection traffic. For this experiment, we extended SIMT to model the Data Analyzer and the Migration Component. We also implemented the Monitoring Capsule within SIMT, including all interfaces for dealing with memory references provided by SIMT and configuration information from the user.

Benchmark Applications: The established prototype was evaluated with several OpenMP applications (FT, LU, MG, CG) from the NAS parallel benchmark suite [21], a few codes (RADIX, OCEAN, WATER) from the SPLASH-2 benchmark suite [41], and two self-coded kernels (SOR, GAUSS). A short description and the used working set size of these applications are shown in Table 1.

Adaptation Effect: First, we compared the parallel execution time with three versions: transparent (original), manual optimized, and self-tuned. The optimized version is achieved by manually specifying data allocations in the source code, based on the access pattern of applications presented by an existing data visualizer. This visualizer [31] presents the access pattern of applications in understandable graphical views, providing guidance towards an improved memory locality.

Figure 2 shows the experimental results on a 32-node NUMA system with a local access latency of 150 CPU cycles and a remote access latency of 1500 cycles. Overall, both optimized and self-tuned versions perform better than the transparent execution, with the manual optimization generally better than self-adaptation. The best performance was achieved with the manually optimized version of SOR (a small code used for iteratively solving partial differential equations), where a performance gain of factor 1.89 has been observed. This can be explained by the fact that manual optimization

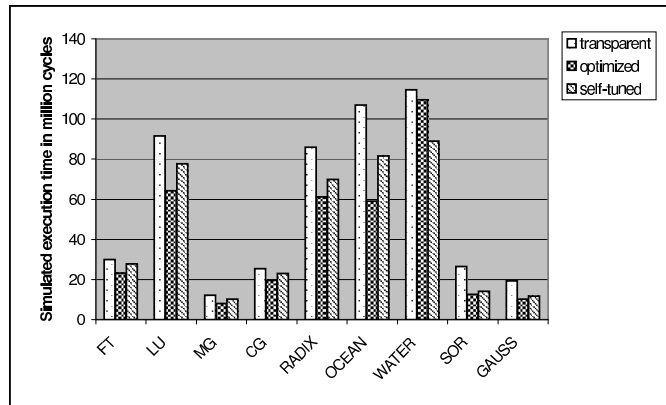


Fig. 2. Simulated execution time (in million CPU cycles) of tested applications in different versions

introduces an initial correct data layout and results in no runtime overhead. However, WATER is an exclusion, with which self-tuning outperforms manual optimization. This is caused by the dynamic access pattern of WATER, which renders that static optimization, which places data on fixed nodes, does not suit for the changing access behavior where data is alternately accessed by several processors.

Comparison of Migration Algorithms: In order to evaluate the novel migration schemes, we simulated all applications with different migration algorithms enabled. Besides U-Mig and W-Mig described in the previous section, we also simulated an L-Mig algorithm in order to examine the impact of information about local references and to evaluate the other schemes. L-Mig assumes the awareness of local accesses which can be acquired by the simulation system. In this case, the number of the dominant accesses will not be compared with the average accesses as it is the case of U-Mig, but with the local references.

The result is summarized in Figure 3. The y-axis gives the improvement of each migration version to the transparent version. This is calculated via dividing the execution time with transparent version by the execution time with a kind of migration enabled.

Examining U-Mig and W-Mig it can be observed that, as expected, W-Mig performs better in case of CG, OCEAN, WATER, and GAUSS. For others, both algorithms behave similarly. The gain in performance with W-Mig is caused by more migrations which were shown by the number of migrations provided by the simulation system. It was also found that these migrations 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.

Comparing U-Mig and W-Mig with L-Mig, it can be noted that the distance between the results of migration with or without local access information is insignificant. In some cases, like for RADIX, W-Mig is even better. According to the migration behavior shown by the simulator, both U-Mig and W-Mig rarely migrate a page mainly accessed by the local node to a remote node, even though the information about local references is not available. Hence, they introduce comparable performance to those algorithms which have knowledge about local accesses.

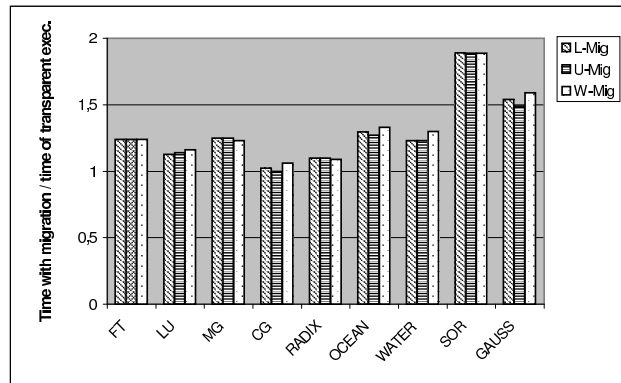


Fig. 3. Comparison between different migration algorithms

Threshold Adapting: As mentioned, we use a threshold to determine whether a page should be migrated. However, it is difficult to choose an adequate threshold for all applications. Depending on the access pattern of individual application, this threshold can be large or small in order to make correct migrations without causing performance loss. In this case, we deploy an adaptive threshold which can be dynamically adjusted according to the runtime execution behavior: if excessive migrations are performed, the threshold is lowered; if excessive remote accesses are observed, the threshold is increased.

In order to evaluate this approach, we tested the execution behavior of several applications with different thresholds. For comparison, we selected other three constant thresholds which are individually defined as 1.5, 2, and 3 factors of the average references performed on a page. Figure 4 presents the experimental results and illustrates the simulated execution time versus the threshold.

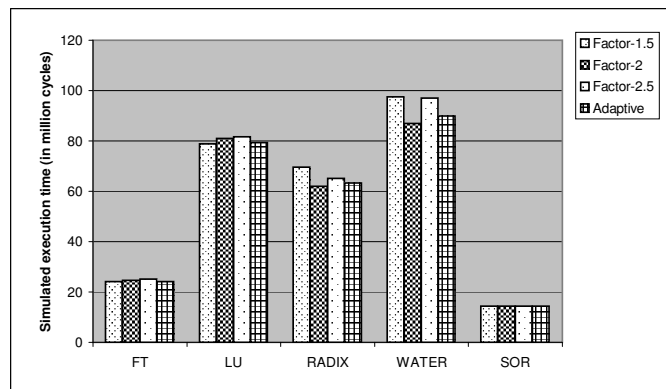


Fig. 4. Execution time of applications with adaptation using different thresholds.

Examining the constant thresholds, it can be seen that applications behave differently, with FT and LU presenting a better performance by factor 1.5, WATER and RADIX a better performance by factor 2, and SOR showing no significant change with varying factors. This result means that no constant threshold is optimal with respect to various applications.

For the adaptive threshold, however, it can be seen that all applications show a good performance, either the same with or only slightly worse than the best performance acquired by the optimal threshold factor for an individual application. This again proves the necessity of self-adaptation.

5 Conclusion

In this paper we presented an autonomic method of data locality optimization in parallel and distributed systems. This implementation was modeled upon our ASoCS architecture concept, proving the general applicability of that concept.

The need for automated optimization was discussed in the Section 1 where it was shown that future systems must employ self-optimization capabilities to overcome current limitations resulting from increased system complexity.

Section 2 presented an overview over related work addressing autonomic systems and how these are used to achieve self-optimization in their specific field of use. It was furthermore noted, that – different to our ASoCS concept – these existing concepts typically either address only certain aspects of autonomic computing or are inherently limited to dedicated applications and application scenarios.

In Section 3 we presented an overview over the ASoCS concept. The applicability of this concept was demonstrated by a prototypical implementation targeting data locality on NUMA systems described in 4. It was shown, how the locality optimizer was modeled after the ASoCS concept, and simulation results for various benchmark applications were presented to compare static (manual) optimization and adaptive (automatic) optimization.

The presented results are promising and show, although manual optimization is able to achieve better results in some cases, that on average adaptive optimization is already on par with manual optimization. With the help of presented and additional results we expect to further improve the adaptive optimization process.

References

1. AMD. AMD Athlon processor, x86 Code Optimization Guide. 2002.
2. J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.-T.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct 1997.
3. R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the International Symposium on Microarchitecture*, Dec 2000.
4. Jamie Beckett. Scaling IT for the Planet: Creating the worldwide computing utility. <http://www.hp1.hp.com/news/2001/oct-dec/planetary.html>.

5. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.
6. Rainer Buchty, Georg Acher, Jürgen Jeitner, Wolfgang Karl, Jie Tao, and Carsten Trinitis. ASoCS: An Architecture Concept for Self-optimizing Parallel and Distributed Computer Systems. *PARS Newsletter*, 22:108–117, Dec 2005. ISSN 0177-0454.
7. E. Chi, M. Salem, I. Bahar, and R. Weiss. Combining software and hardware monitoring for improved power and performance tuning. In *Boston Area Architecture Workshop (BARC) 2003*, Jun 2003.
8. Compaq Computer. Alpha 21264 Microprocessor Hardware Reference Manual.
9. Doug Burger et al. Scaling to the End of Silicon with EDGE Architectures. In *IEEE Computer*, pages 44–55, Jul 2004.
10. Philip K. McKinley et al. Composing Adaptive Software. In *IEEE Computer*, pages 55–65, Jul 2004.
11. J. Fenlason and R. Stallman. GNU gprof: The GNU Profiler. 1997.
12. Michael Gerndt and Wolfgang Karl. EP-Cache. February 2005.
<http://wwwbode.cs.tum.edu/~gerndt/home/Research/EP-Cache/-EPcache.htm>.
13. APART IST Working Group. Automatic Performance Analysis: Real Tools. 2004.
<http://www.kfa-juelich.de/zam/RD/coop/apart/>.
14. Robert Hockauf, Wolfgang Karl, Markus Leberecht, Michael Oberhuber, and Michael Wagner. Exploiting Spatial and Temporal Locality of Accesses: A New Hardware-Based Monitoring Approach for DSM Systems. In David Pritchard and Jeff Reeve, editors, *Euro-Par'98 Parallel Processing, 4th International Euro-Par Conference, Southampton, UK, September 1-4, 1998 Proceedings*, volume 1470 of *Lecture Notes in Computer Science*, Berlin, September 1998. Springer Verlag.
15. Paul Horn. IBM's Perspective on the State of Information Technology.
<http://www.research.ibm.com/autonomic/manifesto>.
16. IBM. Autonomic Computing Web Page.
<http://www.ibm.com/servers/autonomic/>.
17. IBM. Tivoli Software Web Page.
<http://www.ibm.com/software/tivoli/>.
18. IBM. PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual. 1999.
19. Intel. Intel Itanium Architecture Software Developer's Manual. 2000.
20. Intel. Intel Architecture Software Developer's Manual Volume 3: System programming Guide. 2002.
21. H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.
22. W. Karl, M. Leberecht, and 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*, volume 1734 of *Lecture Notes in Computer Science*, chapter 24, pages 417–432. Springer-Verlag, Berlin, 1999.
23. A.R. Lebeck, X. Fan, H Zeng, and C.S. Ellis. Power-aware page allocation. In *Proceedings of ASPLOS IX*, Nov 2000.
24. C. Liao, M. Martonosi, and D.W. Clark. Performance monitoring in a myrinet-connected shrimp cluster. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics'98)*, Aug 1998.
25. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS – On-line Monitoring Interface Specification (Version 2.0)*. Shaker Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.

26. Thomas Ludwig and Roland Wismüller. OMIS 2.0 — A Universal Interface for Monitoring Systems. In M. Bubak, J. Dongarra, and J. Wasniewski, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*, pages 267–276, November 1997.
27. Thomas Ludwig, Roland Wismüller, and Arndt Bode. Interoperable Tools based on OMIS. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*, page 155, Welches, OR, USA, August 1998. ACM Press. <http://www.acm.org/pubs/citations/proceedings/metrics/-281035/pl155-ludwig/>.
28. Scott McNealy, Greg Papadopoulos, and Jonathan Schwartz. N1: Revolutionary IT Architecture for Business. <http://www.sun.com/software/solutions/n1>.
29. Sun Microsystems. Ultra-SPARC Iii User's Manual. 1997.
30. Motorola. MPC7450 RISC Microprocessor Family User's Manual. 2001.
31. T. Mu, J. Tao, M. Schulz, and S. A. McKee. Interactive Locality Optimization on NUMA Architectures. In *Proceedings of the ACM Symposium on Software Visualization*, San Diego, USA, June 2003.
32. P.J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP User Group Conference*, Jun 1999.
33. M. Schulz, J. Tao, C. Trinitis, and 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)*, pages 247–254, Berlin, Germany, May 2002.
34. InstallShield Software. Web Page. <http://www.installshield.com/>.
35. ZeroG Software. Web Page. <http://www.zerog.com/>.
36. B. Sprunt. Pentium 4 performance-monitoring features. In *IEEE Micro*, pages 72–82, Jul/Aug 2002.
37. B. Sprunt. The basics of performance-monitoring hardware. In *IEEE Micro*, pages 64–71, Jul/Aug 2002.
38. Jie Tao, Martin Schulz, and Wolfgang Karl. A Simulation Tool for Evaluating Shared Memory Systems. In *Proceedings of the 36th Annual Simulation Symposium*, Hyatt Orlando, Florida, April 2003. To be appear.
39. R. Wismüller. Interoperability Support in the Distributed Monitoring System OCM. In *Proceedings 3rd International Conference on Parallel Processing and Applied Mathematics - PPAM'99*, pages 77–91, Kazimierz Dolny, Poland, September 1999.
40. R. Wismuller, J. Trinitis, and T. Ludwig. OCM - A Monitoring System for Interoperable Tools. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*, page 149, Welches, OR, USA, Aug 1998. ACM Press.
41. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.