

Improving Data Locality Using Dynamic Page Migration Based on Memory Access Histograms

Jie Tao, Martin Schulz, and Wolfgang Karl

LRR-TUM, Institut für Informatik,
Technische Universität München, 80290 München, Germany
E-mail: {tao,schulzm,karlw}@in.tum.de

Abstract. This Paper presents an approach which dynamically and transparently improves the data locality of memory references in Non-Uniform Memory Access (NUMA) characterized systems. The approach is based on run-time data redistribution via user-level page migration. It uses memory access histograms gathered by hardware monitors to make correct decisions related to the placement of shared data. First performance experiments on several applications show the potential for a significant gain in speedup. In addition, a graphical user interface has been developed showing the actual data movement thereby helping the user to understand the behavior of the application and to detect performance bottlenecks. This feature complements an already existing Data Layout Visualization tool for the observation of memory locality.

1 Introduction

Due to the excellent price-performance ratio, clusters built from commodity PCs or workstations have established themselves as reasonable alternatives in the area of parallel architectures. In addition, in combination with novel developments in interconnection technologies, they have managed to break into the domain of shared memory multiprocessors, an area which used to be dominated by tightly coupled systems with Uniform Memory Access (UMA) organization, as it is given with Symmetric Multiprocessors (SMPs). Especially, loosely coupled machines with Non-Uniform Memory Access (NUMA) characteristics are becoming increasingly popular because of their scalability and straightforward implementation.

NUMA systems, however, are burdened with an additional performance problem since any memory access to global memory can either be intended for local or remote memory modules with significantly different latency properties. 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 thereby to a higher percentage of remote memory access in the overall system. Manual optimizations with respect to data placement can improve data locality, but they can not solve this problem completely since this method is not capable of dealing with applications with dynamically changing access patterns. In this case, also a dynamic approach for the locality optimization needs to be chosen which is capable of significantly reducing remote data accesses via an automatic run-time data redistribution.

Such an approach, called the Adaptive Runtime System (ARS), is explored in this paper. ARS is intended to adjust the data distribution at run-time during the execution of an application. It uses memory access histograms, gathered at runtime by a hardware monitor, as the basis for its analysis of access patterns, and dynamically and transparently modifies the location of data. This improves the locality of memory accesses and thus results in better performance.

This work investigates three page migration algorithms which vary in their monitoring information and criteria for making migration decisions. In addition, ARS uses a graphical user interface to provide information about the actual data migrations. This GUI is connected to a Data Layout Visualizer (DLV) [8] which is used to present an application's memory access behavior in a human-readable and easy-to-use way, thus enabling the understanding of an application's access pattern as well as the location of memory access bottlenecks and communication hot spots. These two graphical representations from both the DLV and the ARS therefore complement each other and give the user a good overview of the behavior of the application in either a static or a dynamic scenario.

The remainder of this paper is structured as follows. Section 2 briefly outlines a few previous approaches for improving data locality via data migration. Section 3 discusses the ARS approach, including the framework, the proposed migration algorithms, and the graphical user interface. In Section 4, first experimental results are presented with a comparison of the migration policies. The paper is rounded up with some concluding remarks in Section 5.

2 Related Work

Data locality on NUMA machines has been addressed in many projects over the last years. Among the projects focusing on improving data locality, a few approaches based on page migration have been proposed; most of them, however, only target tightly coupled architectures.

Vergheze et.al. [9] study the improvements of performance on CC-NUMA systems, provided by OS supported dynamic migration and replication. This kind of page-migration is based on the information about full-cache misses collected via instrumenting the OS. Hot pages, i.e., pages to which a large number of misses are occurring, 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.

Nikolopoulos et.al. [5] present two algorithms for moving virtual memory pages to the nodes that reference them more frequently. The purpose of this page movement is the minimization of the worst case latency incurred in remote memory accesses. Their first algorithm works on 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 program. The second proposed algorithm checks periodically for hot memory areas and migrates the pages with excessive remote references. Both algorithms assume compiler support for identifying hot memory areas.

These proposed approaches and systems focus only on the migration algorithms and their implementations. In fact, it is of the same importance to understand the data movement and the behavior of the migration and to report this behavior back to the user in an appropriate way. This requirement, however, has to our knowledge not been followed by any current work.

3 The ARS Approach

ARS is motivated by the fact that shared memory programs running on NUMA machines suffer from 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 run-time redistribution of data or computation. ARS implements such a mechanism that migrates shared data during the execution of a program.

The previously proposed approaches make the migration decisions according to 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 based on information gathered by hardware monitors with only a minimal probe overhead and without the involvement of compilers, the user, and any system software.

3.1 Framework

Currently, the hardware monitor is designed for our NUMA characterized SMiLE PC clusters. SMiLE stands for *Shared Memory in a Lan-like Environment* and it is a project [3] broadly investigates in SCI-based cluster computing¹. SCI (Scalable Coherent Interface [1]) is an IEEE-standardized [2] interconnection technology with extremely low latency and very high bandwidth. In order to explore shared memory programming on top of this architecture, a software framework, called HAMSTER [4] (Hybrid-dsm based Adaptive and Modular Shared memory archiTEcture), is built within SMiLE enabling the establishment of arbitrary shared memory programming models on top of a single core.

In order to allow an efficient solution for monitoring in this environment, a hardware monitor has been developed. This is necessary because shared memory traffic by default is of implicit nature and performed at runtime through transparently issued loads and stores 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.

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 chip to the

¹ More information at <http://smile.in.tum.de/>

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 by the SMiLE hardware monitor. The result of monitoring are the so-called *memory access histograms* which show the number 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 of a data migration decision by ARS migration mechanisms.

As the hardware monitor is still under development, an event-driven multiprocessor simulator, called SIMT, has been developed within the SMiLE project. SIMT [7] was originally designed to simulate the SMiLE hardware monitor and to provide the exact monitoring information when a hardware monitor is not available. For this purpose, SIMT simulates not only the hardware monitor itself, but also the processor model, the shared memory, the programming interface and models, as well as the parallel execution of applications. Besides that, SIMT comprises functionality enabling a transparent transfer between the simulation platform and a real cluster. Currently, the ARS migration algorithms are implemented on top of SIMT.

3.2 Page Migration Algorithms

Commonly used page migration mechanisms [9, 6] 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; a similar one, called *Out-U*, is therefore also applied within ARS. Besides this, we propose two novel page migration algorithms, called *Out-W* and *In-W*, which use a larger decision base and are therefore likely to perform more accurate and timely page migrations. The main difference between them is that they base their migration decisions on different monitoring information: *Out-W* only looks at outgoing memory traffic initiated by the local node, while *In-W* is based on incoming traffic from remote nodes.

The *Out-U* Algorithm *Out-U* makes decisions whether to move a page from the local node to a remote node. The decision is based on the references performed to the single page from all remote nodes. If the difference between the biggest remote accesses and the average accesses exceeds a threshold, the page is decided to move to the remote node which accesses the page most frequently.

Using this algorithm, however, 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 of performance. On the other hand, if a decision is made only based on a small amount of references, many incorrect migrations may be caused. Therefore, we propose another two algorithms which base their migration decisions on references performed on many pages and therefore are able to make a migration decision earlier.

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 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 physical 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 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 . This avoids the overhead of transferring the monitoring information to other nodes, by using only the monitoring information provided by the local hardware monitor.

To determine the location of a page, the numbers of relative references from all remote nodes are compared. If the difference between the highest number and the average accesses exceeds a threshold, the page is decided to move to the remote node owning the biggest number. Here, the same threshold as in *Out-U* is used.

Examine the *Out-U* and the *Out-W* algorithm. 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 as well accesses the neighboring pages of this page in the same way. Hence, by using the aggregated monitoring data from a page and its neighboring pages, the information necessary to decide about a page migration can be acquired earlier. This should result in a greater gain in performance if the decision is correct. In order to investigate this, we have analyzed the memory access pattern of all SPLASH2-Benchmark applications [10] and a few other numerical kernels and found that for most applications most pages are frequently accessed only by a single node. This indicates that *Out-W* can make correct decisions because there is only one migration target. In addition, we have observed that for pages accessed by multiple nodes the accesses are not equally distributed. Rather one node normally accesses a page more frequently. Together with an adequate threshold, *Out-W* can therefore also make correct decisions for these pages.

The *In-W* Algorithm While *Out-W* determines whether to move a local page to a remote node, *In-W* decides whether to migrate a remote page to the local node. For this purpose, it uses the monitoring results of memory accesses performed by the local node to other 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 remote pages into the calculation. If the difference between a relative access and the average exceeds a threshold, the corresponding remote page is decided to be brought to the local node.

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. For these allocation schemes *Out-W* takes pages, which are not directly neighboring in virtual memory, since it only deals with local pages. *In-W*, however, handles all direct

neighbors except those located on the local node. It therefore has the potential to better use the spatial locality of the memory accesses. *In-W*, however, is more expensive. All nodes, except the one where a page is located, have the ability to make a decision about the location of this page. Hence, once a migration decision is made by a node, all other nodes must be informed. This increases the communication overhead and the complexity of the management.

3.3 Graphical User Interface

In order to show the run-time page movement, a graphical user interface has been developed and combined with the Data Layout Visualizer (DLV) [8], which has already been implemented within the SMiLE project. It is an on-line tool that provides a set of display windows to show the memory access histograms with different views, allowing programmers to understand the execution behavior of their applications. It also projects the memory addresses back to the data structures within the source code, enabling the optimization of applications resulting in a better data locality at run-time.

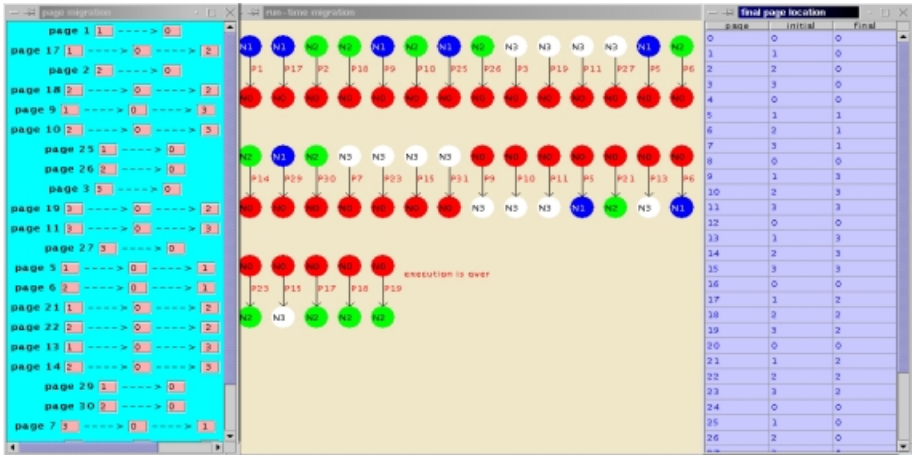


Fig. 1. ARS GUI Display Windows.

The ARS GUI provides several representations to show data migrations, page movements, and data locations. Figure 1 illustrates three sample displays. The *runtime migration* (middle) presents the actual page movements with source node on the top of an item, destination on the bottom, and page number next to the arrow which stands for the direction of moving. Items are dynamically added to the window according to the real time migration. The *page show* (left) illustrates the page movement during the complete execution. The most left rectangle stands for the initial location of an page and the most right stands for the final location of this page, while the rectangle(s) in the middle show intermediate nodes on which the page has resided for some time during the overall runtime of the application. Ping-pong scenarios, such as occurred with page

5, can easily be observed in this view. This enables the evaluation and improvement of the data migration policies. The last window *page location* (right) shows the initial and final location of all shared pages and thereby adds valuable information to the static view given in the DLV.

4 Validation

As mentioned in section 3, the SMiLE hardware monitor is still under development. In order to verify the ARS approach and to evaluate the migration algorithms, we have implemented the first version of ARS on top of SIMT [7] and simulated a number of applications on a 4-node system. These applications are mostly chosen from the SPLASH2-Benchmark suite [10] except the Successive Over Relaxation (SOR) code which is a self-coded numerical kernel used to iteratively solve partial differential equations. The simulated working set size is 2^{14} data points for FFT, a 128×128 matrix for LU, 262144 keys for RADIX, 343 molecules for WATER, and a 200×200 grid for SOR.

For all migration schemes we use a constant threshold of two times of the average references performed to a page. This value is chosen to ensure minimal number of Ping-Pongs. As mentioned in section 3.2, we have studied the access patterns of our benchmark applications and found that most pages are predominately accessed by a single node. Also, we have found that across all applications this dominant access is about two times more than the other accesses, and between the other accesses no disparity of two factors can be found. Therefore, using a factor of two in the threshold can guarantee that no remote access exceeds the threshold in the case of dominating local accesses (which are unobservable in the chosen monitoring approach), and on the other hand that migrations are performed in the case of dominating remote accesses. In the next line of this research work, a flexible threshold will be used with the ability of being automatically modified during the run of an application depending on the changing of the application's access pattern.

Since SIMT, in contrast to the final hardware monitor, is capable of providing information about local references, for both *Out-U* and *Out-W* an additional corresponding algorithm is implemented which exploits local access information, in order to examine the relevance of information about local references and to evaluate the ARS migration algorithms. Should the local information be known, the greatest access number will not be compared with the average accesses as it is the case for *Out-U* and *Out-W*, but with the local references.

Figure 2 illustrates the experimental result by performing all migration algorithms on various programs which use *round-robin* as the default allocation policy to initially distribute data. In addition, simulation results of an unoptimized and 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 migration, no matter which migration algorithm is deployed. 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,

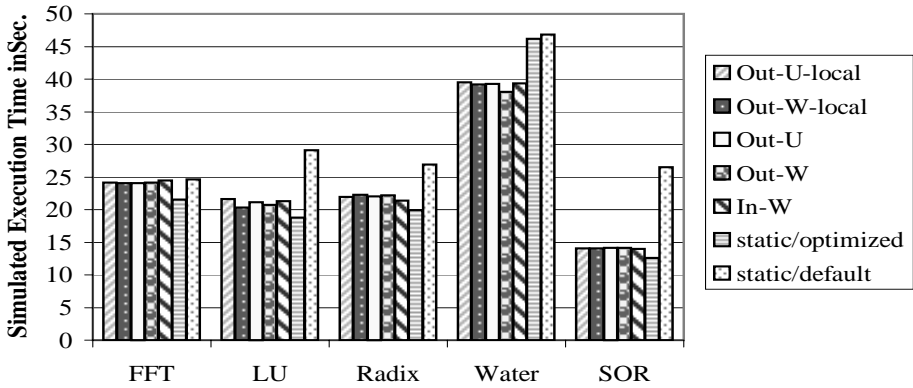


Fig. 2. Simulation time for different programs using round-robin.

where a 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 stems from the fact that manual optimization introduces an initial correct data placement, timely and without introducing any overheads. However, the WATER code behaves differently, where the migration version performs better than the optimization version. This is caused by the dynamically changing access pattern of WATER, which renders static optimization almost useless.

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

Table 1. Migration behavior (mig: total number of migration; p-p: Ping-Pong; mul: multiple migration; err: incorrect migration).

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. In some cases, like for RADIX, the migration without local information is even better. The information shown in Table 1 can give an explanation for this behavior. This Table presents the number of total migrations, multiple migrations², incorrect migrations³, and Ping-Pongs. The numbers of incorrect migrations in this table show that the *Out-U* and *Out-W* algorithms scarcely migrate a page mainly accessed by the local node

² A page is moved to a node and then to another node.

³ A page is accessed most frequently by the local node but migrated to a remote 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. Both indicate that the chosen threshold is adequate. In addition, table 1 also 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, Figure 2 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 speedup by *Out-W* for LU and WATER is caused by more migrations which can be seen in Table 1, where we can also observe that these additional migrations are correct. In addition, we have analyzed these migrations using the ARS GUI and found that they are performed in the earlier phase of the program's running. Programs thereby benefit, despite the overhead introduced by the migrations, from the local references that would be remote if no migration was performed. 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 using the *In-W* algorithm every node can decide whether to move a page to itself. When a page is accessed by multiple nodes, it can happen that the page is migrated and fixed to the first node, but not to the one accessing the page most frequently. For the SOR code, most pages are accessed only by one node, causing the migration not to rely on the node order. The *In-W* scheme behaviors therefore better.

In summary, the results of these first experiments show that a significant improvement has been achieved by ARS's migration approach. It is expected that similar results can be gained when running the applications on actual NUMA machines and with larger and more complex applications.

5 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.

This work presented a runtime system, called ARS, which is capable of performing such dynamic locality adaptations. It uses memory access histograms, gathered through a hardware monitor with low probe overhead, as input and evaluates this high-level information to perform adequate page migrations. First experiments using three different algorithms implemented within ARS show that ARS is capable of improving the performance significantly in all cases.

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 Visualiza-

tion tool, a deep insight into the memory access patterns of the application and thereby enables further optimizations.

References

1. H. Hellwagner and A. Reinefeld, editors. *SCI: Scalable Coherent Interface: Architecture and Software for High-Performance Computer Clusters*, volume 1734 of Lecture Notes in Computer Science. Springer Verlag, 1999.
2. IEEE Computer Society. *IEEE Std 1596–1992: IEEE Standard for Scalable Coherent Interface*. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, August 1993.
3. W. Karl, M. Leberrecht, and M. Schulz. Supporting Shared Memory and Message Passing on Clusters of PCs with a SMiLE. In A. Sivasubramaniam and M. Lauria, editors, *Proceedings of Workshop on Communication and Architectural Support for Network based Parallel Computing (CANPC) (held in conjunction with HPCA)*, volume 1602 of LNCS, pages 196–210, Berlin, 1999. Springer Verlag.
4. M. Schulz. Efficient deployment of shared memory models on clusters of PCs using the SMiLEing HAMSTER approach. In A. Goscinski, H. Ip, W. Jia, and W. Zhou, editors, *Proceedings of the 4th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 2–14. World Scientific Publishing, December 2000.
5. D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade. User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the 29th International Conference on Parallel Processing*, pages 95–103, Toronto, Canada, August 2000.
6. V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and 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)*, pages 342–356, June 1998.
7. J. Tao, W. Karl, and 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)*, pages 349–354, Marbella, Spain, September 2001.
8. J. Tao, W. Karl, and 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)*, volume 2074 of LNCS, pages 861–870, San Francisco, CA, USA, May 2001.
9. B. Verghese, S. Devine, A. Gupta, and 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.
10. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop 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.