THE PERFORMANCE IMPLICATIONS OF LOCALITY INFORMATION USAGE IN SHARED-MEMORY MULTIPROCESSORS

FRANK BELLOSA AND MARTIN STECKERMEIER

Department of Computer Science IV, University of Erlangen-Nürnberg, Martensstraße. 1, 91058 Erlangen, Germany

Large caches used in scalable shared-memory architectures can avoid high memory access time only if data is referenced within the address scope of the cache. Consequently, locality is the key issue in multiprocessor performance. While CPU utilization still determines scheduling decisions of contemporary schedulers, we propose novel scheduling policies based on locality information derived from cache miss counters. A locality-conscious scheduler can reduce the costs for reloading the cache after each context switch. Thus, the potential benefit of using locality information increases with the frequency of scheduling decisions.
Lightweight threads have become a common abstraction in the field of programming languages and operating systems. User-level schedulers make frequent context switches affordable and therefore draw most profit from the usage of locality information if the lifetime of cachelines exceeds scheduling cycles. This paper examines the performance implications of locality information usage in thread scheduling algorithms for scalable shared-memory multiprocessors. A prototype implementation shows that a locality-conscious scheduler outperforms approaches ignoring locality information.

1 Introduction

Cache-coherent multiprocessors with **n**on **u**niform **m**emory **a**ccess (**NUMA** architectures) have become quite attractive as compute servers for parallel applications in the field of scientific computing. They combine scalability and the shared-memory programming model, relieving the application designer of data distribution and coherency maintenance. But locality of reference, load balancing and scheduling are still of crucial importance.
One goal of software development is a high degree of locality of reference from the system up to the application level. Even if application designers develop code with high locality, the impact of caches is reduced when scheduling policies ignore locality information. Disregarding locality, the scheduler initiates switches into uncached process contexts. The consequences are cache and TLB misses for the processor in question and cache line invalidations in caches of other processors.

NUMA architectures like KSR2 or Convex SPP already provide locality information gathered by special processor monitors or by the cache coherence hardware. The latest processor generations - e.g. HPPA 8000, MIPS R10000 or Ultra SPARC - include a monitoring unit. A processor monitor can count events like read/write cache misses and processor stall cycles due to load and store operations. This information is usually only used for off-line profiling. But why should we ignore this information for optimizing the behavior of multithreaded applications at runtime?

The parallelism expressed by “UNIX-like” heavy-weight processes and shared-memory segments is coarse-grained and too inefficient for general purpose parallel programming, because all operations on processes like creation, deletion and context switching invoke complex kernel activities and imply costs associated with cache and TLB misses due to address space changes.
Contemporary operating systems offer middle-weight kernel-level threads decoupling address space and execution entities. Multiple kernel threads mapped to multiple processors can speed up a parallel application. However, kernel threads only offer a medium-grained programming model, because thread management inside the kernel (e.g. creating, blocking and resuming threads) implies expensive maintenance operations on kernel structures.
By moving thread management and synchronization to user level, the cost of thread management operations can be drastically reduced to one order of magnitude more than a procedure call [1]. In general, light-weight user-level threads, managed by a runtime library, are executed by kernel threads, which in turn are mapped on the available physical processors by the kernel. Problems with this two-level scheduling arise from the interference of scheduling policies on different levels of control without any coordination or communication. Solutions to these problems are discussed in [2][10][11].
Efficient user-level threads, which make frequent context switches affordable, are predestined for fine-grained parallel applications. This type of application draws most profit from the use of locality information if the lifetime of cachelines exceeds scheduling cycles.

In this paper we propose a non-preemptive user-level threads package with an application interface to inform the runtime system about memory regions repeatedly used. These hints are used to trigger prefetch operations at each scheduling decision to hide memory latency. We outline scheduling policies using locality information and present results from a prototype implementation on a Convex SPP 1000/XA. This machine can be characterized as a cache-coherent NUMA multiprocessor.

The rest of the paper is organized as follows. Section 2 describes the architecture of the **E**rlangen **Li**ghtweight **T**hread **E**nvironment (**ELiTE**), a scheduling architecture for cache-coherent NUMA multiprocessors developed and implemented at the University of Erlangen. Several affinity policies are evaluated in Section 3. Finally, we conclude in Section 4.

2 Erlangen Lightweight Thread Environment (ELiTE)

Cache affinity scheduling for bus-based multiprocessors has been investigated in detail [14][18] because cache architectures became more and more dominant. The most valuable resource of these architectures is the computing power of the processor and the bandwidth of the bus system. Thus, those scheduling policies focus on a high processor utilization while reducing bus contention [1][19]. The decisions within this type of scheduling are based on CPU utilization and information about the processor where a specific thread was most recently executed. State timing information from each process is additionally used e.g. in SGI's IRIX operating system [3].
The focus of thread scheduling must change when we look at scalable shared-memory architectures with non-uniform memory access. Our approach to memory-conscious scheduling goes beyond the use of information about timing and execution location by using cache miss information for each level of the memory hierarchy.

Modern superscalar RISC-based processors are able to perform multiple operations per clock cycle while simultaneously performing a load/store operation to the processor cache. A multiprocessor system can only take advantage of this immense computing power if the processors can be supplied with data in time. The bandwidth of interconnection networks is no longer a bottleneck for today's scalable parallel processors (e.g. the Scalable Coherent Interface (SCI) of the Convex SPP has a bandwidth of 2.8 GBytes/s). But switches as well as affordable dynamic memory cause a latency of about a hundred nanoseconds, while processor cycles need only a few nanoseconds. The consequence of this discrepancy is that scheduling policies for NUMA architectures have to satisfy three essential design goals:

(1) **Distributed Scheduling:** Data structures of the scheduler (run queues, synchronization objects and pools for reusable memory regions) are distributed. There is not any global structure with the potential risk of contention.

(2) **Locality Scheduling:** Threads are assigned to the processor which is close to the data accessed by the thread. This policy aims to reduce processor waiting time due to cache misses. Fairness among threads of the same application is not necessary in most applications, as each optimally used processor cycle helps to increase throughput.

(3) **Latency Hiding:** Prefetch operations cause overlapping of computation and communication.

Contemporary threads packages are developed for use on shared-memory multiprocessors with a modest number of processors. They have design goals which cannot be applied to scalable NUMA multiprocessors with a high number of processors. Therefore, novel scheduling architectures have to be designed. After presenting the architecture of the Convex SPP, a cache-coherent NUMA multiprocessor, we describe the architecture and implementation details of the ELiTE runtime system.

2.1 Architecture of the CONVEX SPP

The Convex Exemplar Architecture [6] implemented in the Convex SPP multiprocessor is a representative of cache-coherent NUMA architectures. A symmetric multiprocessor called hypernode is the building block of the SPP architecture. Multiple hypernodes share a low-latency interconnect responsible for memory-address-based cache coherency. Each hypernode consists of two to eight HPPA 7100 processors, each having 1 MB direct mapped instruction and data cache with a cache line size of 32 bytes. The processors on a single hypernode can access up to two GBytes of main memory over a non-blocking crossbar switch. The memory of remote hypernodes can be accessed via the interconnect. To reduce network traffic, part of the memory is configured as a network cache with 64-byte cachelines.
There are non-blocking prefetch operations to concurrently fetch data regions from a remote node into the local network cache. These operations can be used to overlap computation and network traffic in order to hide latency.
Performance-relevant events can be recorded by a performance monitor attached to each CPU. The performance and event monitor registers cache misses satisfied by the local or a remote hypernode and the time the processor waits for a cache miss to be served. Several timers of various resolutions are available to produce high resolution time stamps. There is also a system-wide clock with a precision of 1µs. The operating system is a MACH 3.0 microkernel with a HP/UX-compatible Unix server on top. It provides the system call interface from Hewlett-Packard's Unix and an additional system interface to create and control kernel threads.

2.2 Scheduling Architecture of ELiTE

The overhead associated with lightweight processes goes beyond the cost of thread management, because of memory transfers between the various levels of the memory hierarchy. We present a scheduling architecture sketched in [5], refined and implemented in [15].

The following architectural features characterize the ELiTE runtime system:

– Division of thread control block (TCB) and stack allocation (see Fig. 1.):
  Each processor manages its own pool of free TCBs ① and stacks ③. If a new thread is created, the creating processor allocates and initializes a free TCB. After initialization the TCB is enqueued in a startqueue ②. A processor with an empty runqueue ④ fetches a TCB from a startqueue and can run the thread after allocating and initializing a stack. By separating TCB and stack allocation, memory objects of a thread which have to be modified, will be al-

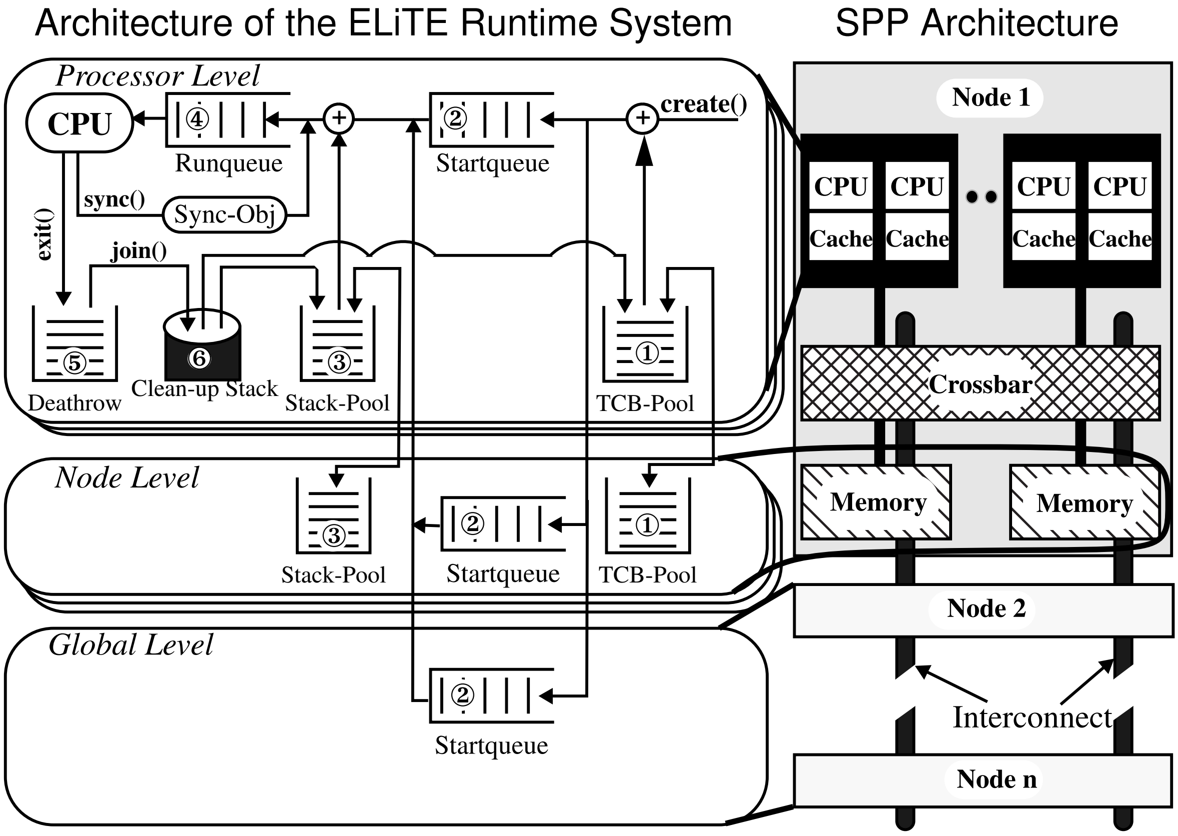

FIG. 1. Scheduling Architecture ≈ Memory Architecture

located from memory pools managed and touched by the modifying processor. The consequence is a high cache reusage and a low cache miss rate.

- Pool hierarchy corresponds to memory hierarchy (see Fig. 1.):
  When the stack- and TCB-pools of the first level (processor level) are empty, new memory objects will be enqueued from the second level. Likewise, memory objects will be moved from the first level to the second level when an overflow occurs. If a pool on the second level is empty, new memory will be allocated from global memory. Therefore, global pools are not useful. The consequence of this strategy is high reusage of local memory while keeping memory allocations to a minimum.
- Local runqueues with load balancing:
  Each processor manages its own priority runqueue (see Fig. 1. ④). The priority of a thread depends on its affinity. Locality information about each thread gathered at runtime can be used to calculate an affinity value (see section 3). The scheduler prefers threads with high affinity. A processor with an empty runqueue, finding no threads in the startqueues, scans the runqueues of the processors in the same node and finally the runqueues of all other processors for runable threads. The advantages of local runqueues are high cache reusage, low data cache invalidation and minimal contention for queue locks.
- Local deathrow with local clean-up stack:
  When a thread exits, its context is stored in the deathrow (see Fig. 1. ⑤) of the processor. The processor executing the join() reads the exit status of the joined thread and pushes an entry on the clean-up stack (see Fig. 1. ⑥) of the processor, which executed the exit(). The processors periodically scan their local clean-up stacks and remove the contexts of joined threads from the deathrow and push memory objects (stacks and TCBs) into the local memory pools. Because processors executing a join never modify the deathrow and memory pools, cache invalidations can be avoided and memory locality will be preserved. Cache misses are reduced to a minimum, because the push onto the clean-up stack concerns only a single cacheline.

We measured the number of threads a single thread can fork and join with *n* CPUs starting and running the created threads. We compared an approach with central pools to another with distributed pools (see Fig. 2.). The results show that the fork-join-rate of the centralized approach drops when using 4-8 processors due to lock contention and dramatically drops when using more than 8 processors (more than one hypernode) due to allocation of stacks not cached on the local node. The distributed approach does not scale because of the limited fork rate of the single forking thread. But there is no severe performance degradation, because all stacks are allocated from local pools with encached memory. Furthermore, locking of central

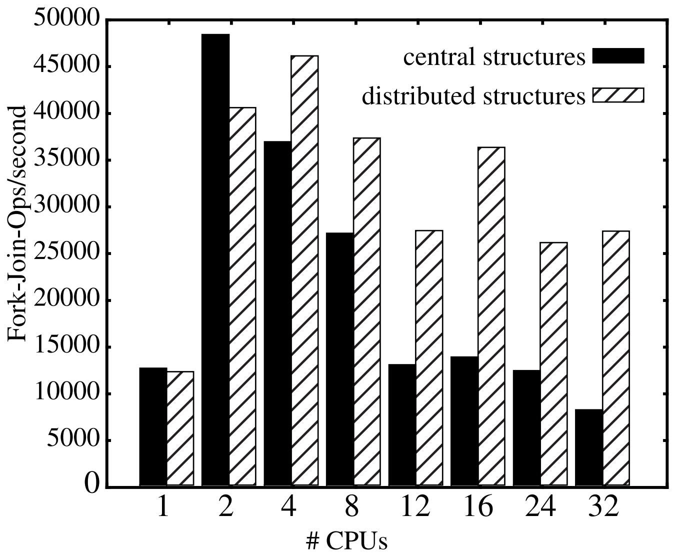

FIG. 2. Central vs. distributed hierarchical structures

structures (pools and queues) and remote memory access can be reduced to a minimum by the mechanisms of the local deathrow and the clean-up stack.

– Distributed synchronization objects with local wake-up stack:
  Unlike common UNIX sleep queues with hashed entries [12], the ELiTE runtime system binds blocked threads to synchronization objects (see Fig. 3. ②). If a process becomes unblocked, a reference to its TCB will be pushed on the wake-up stack (see Fig. 3. ③). Similar to deathrow management, the processors scan their local wake-up stacks periodically and enqueue unblocked threads in the local runqueue (see Fig. 3. ①).

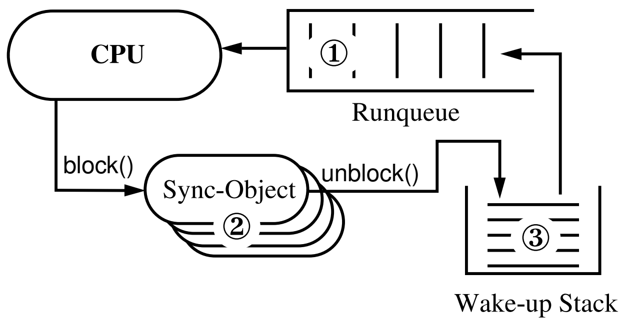

FIG. 3. Distributed Lock Structures

2.3 Implementation details

2.3.1 Fast context switch

A fast context switch free of race conditions is the basis of most synchronization mechanisms inside a runtime system. The preswitching model is used in ELiTE. After saving the state of the old thread, the stack of the new thread is used to enqueue the TCB of the old thread without the danger of a race condition. This mechanism assumes that the next thread is known and existent before the switch occurs and that the next thread already owns a stack, which makes lazy stack allocation difficult. Based on the QuickThreads package of the University of Washington [7], which provides the preswitch model for various processor architectures, we have ported QuickThreads to the HPPA-RISC processor architecture. On a CONVEX SPP using this type of processor, the following times for a context switch can be reported:

TABLE I
Number of Cycles required for a context switch

Operation	Clock Cycles
Context switch between threads with all data in cache	153
Context switch between threads on the same node	1122
Context switch between threads on different nodes	1805

The proportion of clock cycles for a context switch between threads having stored their state in one of the three levels of the memory hierarchy is equivalent to the proportion of bandwidth when accessing the respective memory level.

2.3.2 Fast synchronization

Lim and Agarwal [13] have investigated waiting algorithms for synchronization in large-scale multiprocessors. With increasing number of CPUs, the type of synchronization has a significant influence on the performance of fine-grained parallel applications. As proposed we use two-phase locking with a fixed number of spin cycles in the ELiTE runtime system.
The proposed two-phase waiting algorithm combines the advantage of polling and signaling. A thread blocks after a default polling interval. The polling threshold depends on the overhead of blocking.
We count the number of clock cycles a lock is held and calculate the average duration (in clock cycles) of the last 8 times each specific lock was acquired. If the average of lock-holding cycles exceeds a proposed value (default is 50% of the cycles for a context switch), we block at once. Otherwise we spin the default number of cycles. To reduce memory accesses while spinning we use exponential backoff. For details refer to [15].
We measure the peak performance for synchronization by starting 4096 micro-threads (8 kBytes stack and no workload) doing nothing but synchronizing. The total amount of memory is about 4096*8192 KBytes = 32 MBytes.
With central queues we observe a severe performance loss due to lock contention and data cache corruption as a consequence of non local memory accesses (see Fig. 4.)
Using a distributed approach, the time for a synchronization depends on the time to save/restore the stacks into the processor/network-caches and to access the synchronization objects. We reach the peak performance of about 100000 synchronizations/second per processor (1-8 processors in Fig. 4.), if all

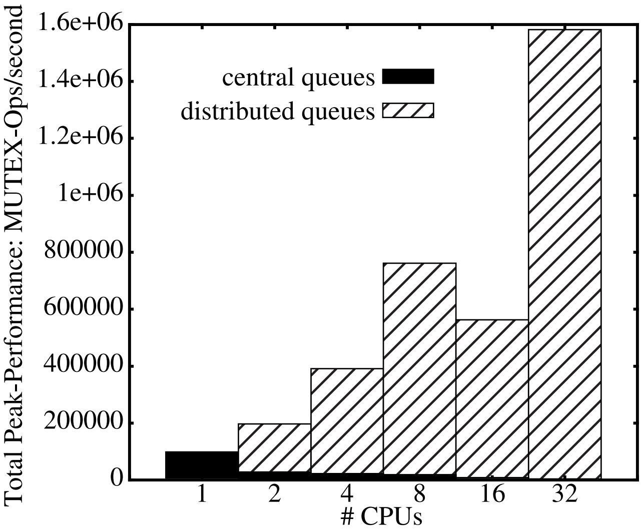

FIG. 4. Fast synchronization and context switch with 4096 threads

data can be held in the processor caches. If the stacks can be stored in the processor caches, but the synchronization objects have to be touched partly in multiple nodes, the synchronization performance deteriorates to about 50000 synchronizations/second per processor (32 processors in Fig. 4.). If the stacks do not fit into the processor-caches (16 processors), they are stored in the network caches (= local memory). This effect reduces the performance to about 40000 synchronizations/second per processor.

The scheduling approach presented in this paper considers the locality behavior of individual threads at runtime. This approach is possible if the hardware provides information about cache misses. An additional optimization is possible, if we look at the interaction of threads on the same memory regions, by comparing their working set. In [13] TLB information is used to find cooperating processes at the kernel level. For fast interaction a user-level readable TLB would be necessary, accessible as fast as the processor cache and with a fine resolution in the range of 1 kByte. This could be a feature of new processor generations. The knowledge of synchronization events is an alternative way to identify cooperating threads. This approach will be discussed in section 3.3.

2.3.3 Application Interface

Contemporary NUMA architectures like Convex SPP or KSR1/2 have non-blocking prefetch operations in their instruction set to concurrently fetch data regions from a remote node into the local network cache, overlapping computation and network traffic and thus hiding latency. If thread-specific data can be stored in a single block, a pointer to this block and its length can be stored in the thread control block. If there is an interface to the scheduler, a currently running thread can ask the runtime system to prefetch the data of the thread which will run in the near future. This idea was motivated by implementations of adaptive numerical methods [4][12], where thousands of threads, each corresponding to a point of an adaptive grid, resume the threads representing the grid points in the neighborhood after calculating the local grid point before they suspend themselves. This numerical method, called *active threads strategy* can only run with high efficiency on NUMA architectures if all thread-specific data is resident in the cache before the context switch occurs.

3 Affinity Scheduling

The performance of a computer is considerably influenced by the fast supply of data to the available processing units.The processor may suffer reduced performance due to memory latency and contention unless data essential for operation is cached in fast memory. Affinity scheduling tries to prefer processes with a high amount of cached data in order to increase throughput.

Besides information about processor number and time behavior, we use information about data locality in our scheduling architecture. Locality information about each process/thread like the cache miss rate, the processor stall time, and the processor of last execution can be used to calculate an affinity value. A prerequisite is a computer architecture providing information about cache misses and CPU latencies due to memory access. NUMA architectures like Convex SPP and KSR1/2 provide this information. The latest processor architectures like HPPA 8000, MIPS R10000 or Ultra SPARC gather this information on chip.

In the next subsections we describe several affinity strategies and the prospect of the proposed technique.

3.1 Scheduling Strategies

We have designed and implemented a user-level runtime system, offering the possibility to easily import new affinity strategies. The strategies examined are listed in Tab. II:

TABLE II
Scheduling Strategies

Scheduling Strategy	Basis of Decision	Policy
No Affinity	Processor location	LIFO
Virtual Time	Sequence numbers	Most recently run
Minimum Misses	Cache misses	Thread with minimal # of cache misses
Reload Transient	Cache misses	Minimal reload transient based on Markov chains

- Using *virtual time* stamps, each thread is assigned a sequence number after its run. This strategy does not need cache miss information, and can be used on every type of hardware. Threads with the highest time stamp given by the same processor will be preferred. If a fast global time source with high resolution is included in the hardware, more precise timing strategies can be used [3].

- The *Minimum Misses* strategy compares the number of cache misses during the last run. The thread with the lowest number of cache misses is preferred. This strategy favors threads that block frequently, since they have shorter runs and few cache misses.
- The reload transient model [14] is more complex, but offers some potential. Similar to Thiebaut and Stone[16], we refer to the working set of a thread that is present in the cache as its footprint in the cache. Our model based on Markov chains enables us to calculate the expected footprint size as well as the number of cache misses a restarted thread will experience.
  It uses the following assumptions:
  - the cache is direct mapped and virtually indexed
  - accesses to the cache are independent and uniformly distributed

  Fig. 5. shows the Markov chain for a running thread. The probability that a cache miss increases the thread's cache state from C cachelines to $C+1$ cache lines is $(N-C)/N$, the possibility that it remains constant is C/N for a cache with N cachelines.

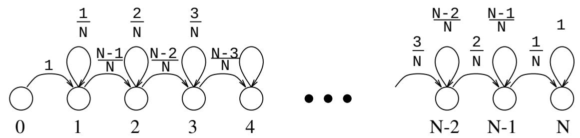

FIG. 5. Markov chain for running threads.

This Markov chain can be represented by its transition matrix M, with each element $m_{i,j}$ as the probability that one cache miss increases the cache state from i to j cache lines:

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & \frac{1}{N} & \frac{N-1}{N} & 0 & \dots & 0 & 0 \\ 0 & 0 & \frac{2}{N} & \frac{N-2}{N} & \dots & 0 & 0 \\ & & & \dots & & & \\ 0 & & & & \dots & \frac{N-1}{N} & \frac{1}{N} \\ 0 & & & & \dots & 0 & 1 \end{bmatrix} \qquad (1)$$

Raising this matrix to the n-th power leads to the probabilities $m^n_{i,j}$, for a transition from i cachelines to j after n cache misses:

$$m^n_{i,j} = \begin{cases} 0 & : \quad j < i \\ \dfrac{(N-i)!(-1)^{j+i}}{(N-j)!(j-i)!N^n} \displaystyle\sum_{k=0}^{j-i} \binom{j-i}{k} (k+i)^n (-1)^k & : \quad i \le j \le i+n \\ 0 & : \quad i+n < j \end{cases} \qquad (2)$$

Using these probabilities we calculated the expected number of valid cachelines for a thread that started with i cachelines and produced n cache misses:

$$E^n_i = \sum_{j=0}^{N} j m^n_{i,j} = \dots = N - (N-i)\left(\frac{N-1}{N}\right)^n \qquad (3)$$

An equivalent model is used for threads that are waiting in some queue. They loose cachelines due to cache misses caused by other running threads.

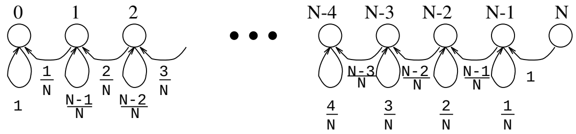

FIG. 6. Markov chain for waiting threads.

Similar to the approach above we evaluated the expected number of remaining cachelines for a thread that stopped with a footprint size of i cachelines and n cache misses which have been produced by other threads in the meantime:

$$E^n_i = i\left(\frac{N-1}{N}\right)^n \qquad (4)$$

By applying both formulas (3) and (4) successively, starting with an empty footprint and using the expected footprint size as start value for the next iteration, we are not only able to predict at any moment the footprint size, but we can also estimate the number of cache misses this thread will produce when it is restarted. If the thread T_i stopped with an expected footprint of $E(T_i)$ and $n_i(t)$ cache misses have happened in the meantime, the reload transient can be estimated by

$$R(i, n_i(t)) = \log_K\left(\frac{N - E(T_i)}{N - K^{n_i(t)}E(T_i)}\right) \qquad (5)$$

with the abbreviation $K = (N-1)/N$.

The simplest form to use this reload transient is to assign the highest priority to threads with the lowest reload transient, i.e:

$$P_i(t) > P_j(t) \leftrightarrow R(i, n_i(t)) < R(j, n_j(t)) \qquad (6)$$

This strategy prefers threads that have most of their working set resident in the processor's caches and will therefore produce few cache misses. As fewer cache misses are caused by this thread than by any others, all other threads get less disturbed and will create fewer cache misses themselves.

Although thread priorities based on this kind of reload transient are time dependent, the order in the runqueue is constant. Therefore, it is sufficient to calculate the relation between the threads in the runqueue once a thread is placed on it. Using this property and lookup tables instead of real-time calculations, a highly efficient strategy can be implemented.

3.2 Interpretation of Measurements

We have implemented and tested the proposed strategies as part of the ELiTE architecture on a Convex SPP 1000/XA in a range from 1 to 32 processors. The test environment includes synthetic tests with artificial workloads as well as real-world numerical kernels like Gaussian elimination, LU-decomposition and adaptive solvers on irregular grids [4][12]. All measurements show that a fine-grained parallelization does not inherently imply low performance. On the contrary, a fine-grained numerically efficient algorithm outperforms most conventional methods, because fine-grained parallelism implies a high data locality in most cases. This locality can be used by a sophisticated scheduler to achieve good overall performance even if we register some overhead for complex strategies. But the impact of scheduling in a fine-grained environment affects performance much more than with a coarse-grained approach. The runtime of a fine-grained LU-decomposition can vary by a factor of ten, depending on the scheduling architecture.

With a synthetic workload of 1024 threads, each snooping through a working set of 64 kBytes between synchronizations, we can get a good impression of affinity strategies (see Fig. 7.). We measure how many synchronizations can be executed in one second with different numbers of processors. This example resembles parallel numerical applications with regular execution order like iterative solvers on block-structured grids.

Applications with irregular execution order make high demands on the scheduling policy. We measure the number of smoothing operations per second on an unstructured grid executed by a full-adaptive iterative solver [12]. To demonstrate the influence of the scheduling strategy on the performance of applications, we compare the smoothing rate of the proposed affinity strategies with the *No Affinity* strategy. A relative smoothing performance of 2 means, that the adaptive solver executes twice the number of smoothing operation under the affinity scheduler compared to an execution under the *No Affinity* scheduler (see Fig. 8.)

- *No Affinity* will be outperformed in general by all strategies using locality information
- The simple *Minimal Misses* strategy performs quite well in cases with homogeneous execution behavior. As this strategy favors threads that block frequently, anomalous behavior is possible if some threads acquire locks during the polling interval whereas other threads block.
- The *Virtual time* approach only uses timing behavior and the processor number of the last run. It never shows anomalous behavior and performs very efficiently with moderate processor numbers (1-16 processors) because it offers the minimal overhead compared to the other strategies. It should be the policy of choice for UMA architectures not offering cache miss information (see [3]).
- The *Reload Transient* strategy shows the best performance in multinode architectures with non-uniform memory access. The overhead of gathering cache miss information and computing the expected working sets is only justified when memory latency really strikes. This is the case on all contemporary and future scalable parallel processors like Convex SPP, KSR 1/2, SUN Ultra MPP, Sequent NUMA-Q or multiprocessors coupled with SCI-Hardware (SCI = Scalable Coherent Interface).

The hardware of the Convex SPP 1000 used for our implementation cannot distinguish between processor- and network-cache misses, but these two types of cache miss differ by a factor of 4 in their penalty. Coming SPP generations offer information about both types of cache faults.This permits a much better estimation of the *Reload Transient*.

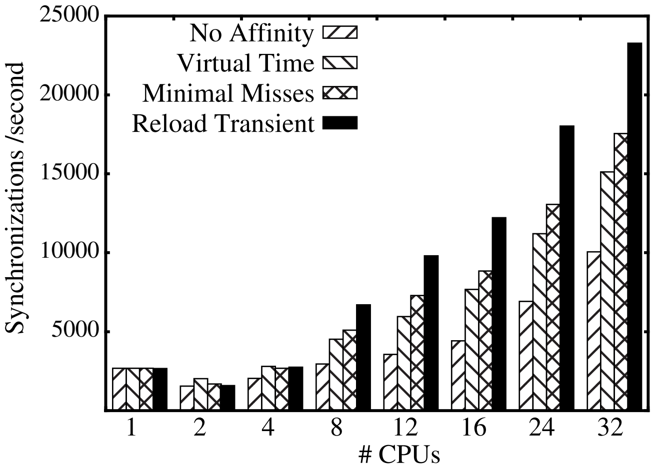

FIG. 7. Performance of applications with regular execution order

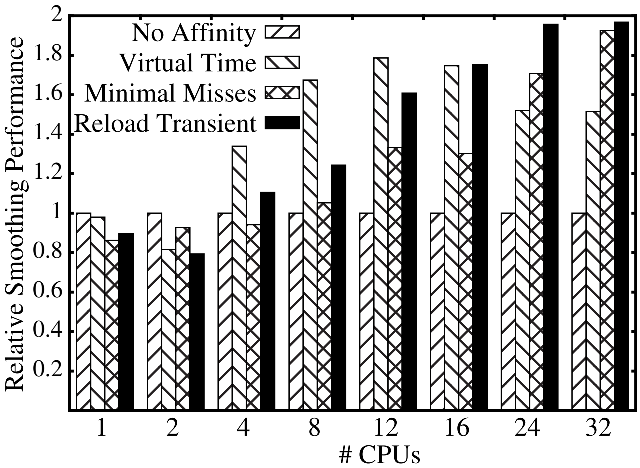

FIG. 8. Performance of adaptive applications with irregular execution order

3.3 Towards Optimal Affinity Scheduling

Scheduling policies should use all available information to calculate the best possible process mapping. For optimal process scheduling, knowledge about synchronizing threads is necessary. Cooperating threads can be identified by synchronization events and collocated at the same processor, whereas a compromise between collocation and load balance has to be made. To decide an ideal point defined by the location and timing of execution, we define gravitational and repulsive forces between threads by the frequency and extent of information sharing.

The forces influence affinity in a metric space, so that threads migrate near to their (sub)optimal operation point. This affinity model resembles the computational field model for migrating objects proposed in [17].

A thread *A* running on processor *X* with a great cache affinity exerts a gravitational force on thread *B* running on processor *Y* with just a small cache affinity. Thread *A* exerts the force by increasing the affinity of thread *B* to processor *X* each time *A* synchronizes with *B*. The value used for increasing the affinity depends on the size of the memory region shared by the two threads (see Fig. 9.). In the ELiTE runtime system we use the number of cache misses occurring during the modification of the shared-memory region as the basis of the affinity adjustment.

Cooperating threads with a high compute load and rare synchronization events exert repulsive forces on each other (see Fig. 9.). Threads showing this behavior can be recognized by a high number of cache misses and a long time-frame between blocking events. If these threads run on the same processor, they decrease the affinity of their synchronization partner. Threads with a low affinity can be caught by an idle processor.

Threads with low affinity sharing a memory region which is large relative to the private working set exert aggregative forces on each other (see Fig. 9.). Before unblocking, the affinity values of a synchronization partner will be modified such that the partner will run up on the same processor as the unblocking thread. The aggregative force is influenced by the number of cache misses on shared and thread-private memory regions.

The proposed strategy has the potential to come closer to the optimum operation point of all threads concerning location and timing of execution. By introducing gravitational and aggregative forces, the number of cache misses of cooperating threads can be reduced, while compute-intensive threads with low synchronization rate exerting repulsive forces can be distributed very easily.

The hardware of the Convex SPP 1000 cannot distinguish between processor- and network-cache misses. Furthermore the costs for reading the cache miss counters in this architecture are equivalent to a user-level context switch. Therefore, we could not get any publishable results of our implementation. Coming processor generations with fast readable on-chip miss counters will allow reasonable measurements using this promising scheduling strategy.

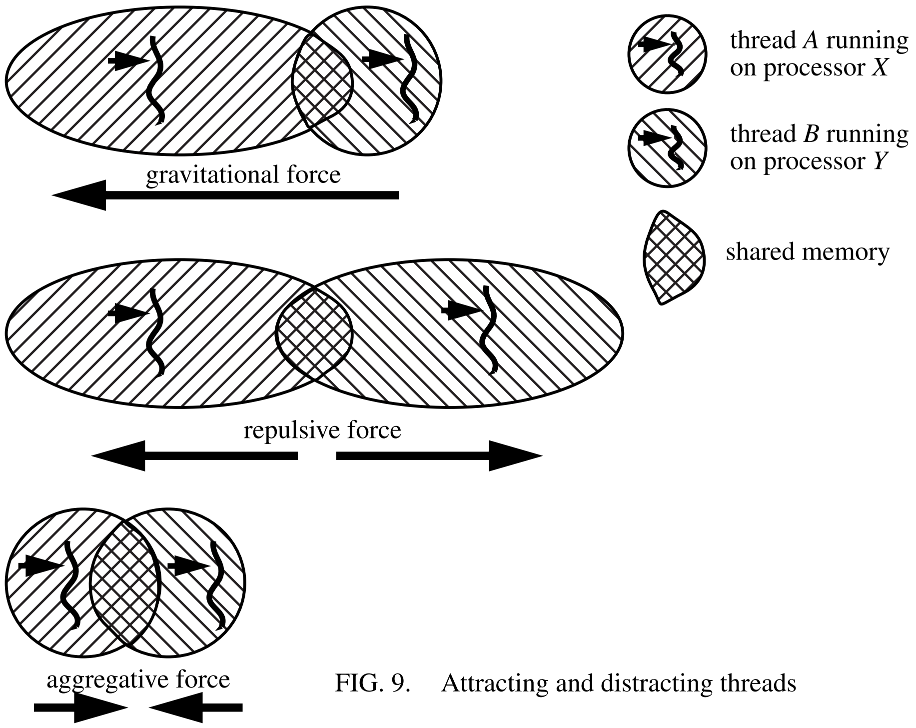

FIG. 9. Attracting and distracting threads

4 Summary and Conclusions

Algorithmic optimizations of the application and scheduling mechanisms for the management of parallelism considerably influence the processor performance. The applications' designer cannot be relieved of algorithmic considerations concerning memory locality. However, a scheduling strategy that employs memory-conscious thread management makes a fine-grained architecture-independent programming style possible. As maximum throughput is the goal of our efforts, we have presented the architecture of the Erlangen Lightweight Thread Environment (ELiTE). The focus of this scheduling architecture lies on the reduction of cache misses. Distributed data structures like those proposed in the ELiTE architecture are an absolute necessity. Scheduling strategies using locality information improve cache locality and therefore throughput. Strategies based on Markov chains offer the best process reordering in scalable architectures with non-uniform memory access despite their algorithmic overhead.

The trade-off between scheduling overhead and performance gain due to better locality will favor complex strategies using cache miss information particularly in architectures with high memory latency and large caches. Consequently, the proposed scheduling techniques can be used from the desktop to the supercomputer. Further research will be dedicated to novel strategies using all available locality information.

5 References

1. T. Anderson; E. Lazowska; H. Levy: The Performance Implication of Thread Management Alternatives for Shared-Memory Multiprocessors, *ACM Trans. on Computers*, 38(12), Dec. 1989, pp. 1631-1644
2. T. E. Anderson; et al.: Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1), Feb. 1992, pp. 53-79
3. J. Barton, N. Bitar - SGI Corp., A Scalable Multi-Discipline, Multiple-Processor Scheduling Framework for IRIX", In *Proc. of IPPS'95 Workshop of Job Scheduling Strategies for Parallel Processing*, LNCS 949, Apr. 1995
4. F. Bellosa: Implementierung adaptiver Verfahren auf komplexen Geometrien mit leichtgewichtigen Prozessen, University Erlangen-Nürnberg, IMMD IV, TR-I4-94-07, Oct. 1994
5. F. Bellosa: Memory-Conscious Scheduling and Processor Allocation on NUMA Architectures, University Erlangen-Nürnberg, IMMD IV, TR-I4-95-06, May 1995
6. Convex: Exemplar SPP1000/1200 Architecture, Convex Press, May 1995
7. D. Keppel: Tools and Techniques for Building Fast Portable Threads Packages, University of Washington, TR UWCSE 93-05-06, May 1993
10. C. Koppe: Sleeping Threads: A Kernel Mechanism for Support of Efficient User-Level Threads, In *Proc. of International Conference of Parallel and Distributed Computing and Systems PDCS'95*, Washington D.C., Oct. 95
11. T. J. LeBlanc; et al.: First-Class User-Level Threads, Operating Systems Review, 25(5), Oct. 1991, pp. 110-121
12. S. Leffler: The Design and Implementation of the 4.3BSD UNIX Operating System, Addison-Wesley, 1990
13. B. Lim; A. Agarwal: Waiting Algorithms for Synchronizations in Large-Scale Multiprocessors., *ACM Transactions on Computer Systems*, 11(1), Aug. 1993, pp. 253-297
12. Ulrich Rüde: On the multilevel adaptive iterative method, *SIAM Journal on Scientific and Statistical Computing*, Vol. 15, 1994
13. P. Sabalvarro, W. Weihl: Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors, In *Proc. of IPPS'95 Workshop of Job Scheduling Strategies for Parallel Processing*, LNCS 949, April 1995
14. S. Squillante; E. D. Lazowska: Using Processor Cache Affinity in Shared-Memory Multiprocessor, Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2),Feb. 1993, pp.131-143
15. M. Steckermeier: Using Locality Information in User-Level Scheduling, University Erlangen-Nürnberg, IMMD IV, TR-I4-95-14, Dec. 1995
16. D. Thiebaut, H. Stone: Footprints in the Cache, *ACM Transactions on Computer Systems*, 5(4), Nov. 1987, pp. 305-329
17. M. Tokoro: Computational Field Model: Toward a New Computation Model/Methodology for Open Distributed Environment, Sony Computer Science Laboratory Inc., SCSL-TR-90-006, June 1990
18. J. Torellas, A. Tucker, A. Gupta: Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors, *Journal of Parallel and Distributed Computing*, Vol. 24, 1995, pp. 139-151
19. A. Tucker: Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors, Phd Thesis, Department of Computer Science, Stanford University, CS-TN-94-4, Dec. 1993

Frank Bellosa received the diploma in computer science from the Friedrich-Alexander University Erlangen-Nürnberg in 1993. Currently he is research assistant at the operating systems department of the University of Erlangen. His research focuses on operating system support for parallel programs, parallel data structures and algorithms, and locality maintenance in shared-memory multiprocessors.

Martin Steckermeier received his diploma in computer science from the Friedrich-Alexander University Erlangen-Nürnberg in 1995 and has recently started to work as a research assistant at the department for operating systems of the University of Erlangen.