Memory Latency Reduction with Fine-grain Migrating Threads in NUMA Shared-memory Multiprocessors

Mikhail Dorojevets
Dept. of Electrical and Computer Engineering
SUNY Stony Brook, NY 11794 USA
midor@ece.sunysb.edu

Dmitri Strukov
Dept. of Electrical and Computer Engineering
SUNY Stony Brook, NY 11794 USA
strukov@ece.sunysb.edu

Abstract

In order to fully realize the potential performance benefits of large-scale NUMA shared memory multiprocessors, efficient techniques to reduce/tolerate long memory access latencies in such systems are to be developed. This paper discusses the concept, software and hardware support for memory latency reduction through fine-grain non-transparent thread migration, referred to as mobile multithreading, in the proposed scalable NUMA shared-memory architecture. The performance evaluation results for the conjugate gradient NAS benchmark demonstrate that the proposed fine-grain thread migration combined with data prefetching can be effectively used to reduce memory latency and switch traffic in NUMA shared-memory multiprocessors with a large non-uniformity memory access ratio.

Keywords

Multithreading, Latency Tolerance, Thread Migration

1. Introduction

Recently, large-scale tightly-coupled non-uniform memory access (NUMA) shared-memory multiprocessors have become considered to be a promising architecture to deliver petaflops-level computing power. However, in order to fully realize the potential performance benefits of these architectures, efficient techniques to reduce/tolerate long memory access and synchronization latencies in such systems are to be developed.

Several techniques, such as coherent caches and data prefetching, have been implemented and/or proposed to address the memory latency problem in NUMA multiprocessors. While some commercial systems, such as SGI Origin [1], demonstrate a viability of this CC-NUMA approach, it is still an open issue whether their efficiency, complexity, and scalability can meet the requirements of high sustained performance on real-world applications often exhibiting poor data locality. Coherent caches and data prefetching can be characterized as data migration, which combines transfer of data from remote memory towards the processor that accesses them and replication of the data in the processor's cache(s).

Another technique to hide memory latency is multithreading, which was implemented in both commercial and experimental multithreaded computers, such as Cray MTA [2], MARS-M [3], Alewife [4], the SUN MAJC [5], Intel P4 Xeon [6]. Multithreading implemented in its traditional forms, such as synchronous thread interleaving, switch-on-miss/on-use, and blocked-on-miss scheduling policies, does not improve the performance of a single thread, however. While combining of data migration techniques with multithreading is one way to address the issue of a single-thread performance, another possible technique of latency reduction in NUMA systems is thread migration, in which a thread moves towards the remote memory module where the accessed data are located. Examples of this approach in the context of software distributed shared memory (DSM) systems include computation migration [7], Ariadne [8], Millipede [9], Active threads [10], and other schemes [11], which all suffer from the huge overhead of their thread migration implemented in software on the top of the existing architectures, not well suited for the fine-grain thread migration model.

In contrast to many other researchers in the thread migration field, we started our work from a clean sheet, being released from the necessity to rely upon any existing architecture and system software.

The preliminary architectural work has been done during the HTMT petaflops project in 1997-2000 [12], when the first version of the SPELL-1 multithreaded architecture (without thread migration) has been developed [13]. The problem of hiding memory latency is extremely challenging in exotic designs, such as the HTMT petaflops computer, where the quest for ultimate performance requires a use of ultra-high speed superconductor SPELL processors, while almost all memory is to be built with silicon CMOS SRAM and DRAM technologies. The solution proposed for this system was to hide memory latency through preemptive data and code migration (called percolation) from silicon memory toward superconductor memory local to SPELL processors. The percolation is proposed to be done by processors-in memory (PIMs) under control of the runtime system. To many people, including the authors

of the paper, the reliance upon the runtime system with its high overhead and data-flow like computational paradigm in the large-scale HTMT DSM system has more disadvantages than advantages.

The objective of this paper is to present the results of our evaluation study of a new scalable NUMA shared-memory architecture with thread migration, referred to as mobile multithreading, which was developed in the framework of our current project on a tightly-coupled petaflops level computing system. This concept requires neither the runtime system nor PIMs involvement, while extending the basic multithreaded SPELL architecture with new adaptive latency reduction and tolerance mechanisms that could be implemented in any technology. The results of our evaluation study on several benchmark programs indicate that the proposed architecture provides significant performance improvements for the programs studied. The rest of the paper is organized as follows. Section 2 discusses main features of our mobile multithreaded (MMT) architecture with fine-grain migrating threads. Section 3 presents the performance evaluation study. Finally, we conclude this paper in Section 4.

2. Fine-grain Locality-guided Thread Migration in the MMT Architecture

The goal of the proposed MMT NUMA architecture is to reduce the latency of remote memory load and atomic swap operations issued by a thread through the non-transparent migration of the thread to the processor local to the accessed remote memory module. We do not apply migration to remote store operations because their latency is outside of the critical path of computation in the architecture with mobile multithreading and without coherent caches.

2.1 Granularity of Thread Migration

To make thread migration effective as a latency tolerance technique, the granularity of threads in the MMT architecture is made very fine. Logically, each thread can be considered as an execution path within a user's program, starting with the thread's initial instruction address. In order to be able to execute, a thread must be created and assigned to one of the thread hardware contexts available in each multithreaded processor (MTP). This is done in hardware either by performing a thread create operation in another, master, thread running in the same MTP, or by a special interrupt signal.

When created, each thread gets its unique place of origin (the original MTP and the original context id), written in a special thread control register. This place of origin will be used later by each migrating thread to report about its termination to its master thread. Master threads (one per MTP, always with a zero context id) are not permitted to migrate because it would make impossible the correct execution of merge/quit operations by migrating (slave) threads.

Each hardware context includes: a set of private general-purpose registers (a register frame) in the register file, its thread status register, thread id, program counter (PC), four condition registers to hold 4-bit Boolean values. Other special registers include temporary data registers to implement some complex arithmetic operations (e.g., fused multiply-add/subtract, divide, square root), and shared and non-shared control registers for thread creation, synchronization and migration across MTPs.

During migration of a thread from one hardware context to another hardware context in some remote MTP, only the state of the thread's live register variables, the thread's "continuation" address, and some thread control information including the place of origin, have to be transferred to the remote hardware context. Because the locations of the registers to be migrated are known before execution, it is easy for the compiler/user to create a live register vector mask (LRVM) for each migrating point, one bit per register in a hardware context. Each hardware context has its LRVM register, which has to be loaded to enable migration. We will explain below how this mask can be used by the hardware to decrease the timing overhead of the thread migration.

2.2 Implementation of Thread Migration

The mobile multithreading is a mixed software/hardware approach. The compiler/user must decide for which memory access points in the program thread migration could be beneficial and generate the migration code consisting of the enabling and relocation code sections for those points.

The enabling code starts with loading of the thread's LRVM value into the context's migration control register. Also, for each migration point, the compiler uses special form of remote load/swap instructions. At the runtime, besides executing these operations as "normal" load/swap operations, the hardware checks which memory module these operations access. Only when/if it is remote, the local MTP hardware generates a migration request for a free context to the remote MTP associated with the accessed memory module. If there is a free hardware context in the remote MTP, the context will be reserved by hardware, and its id returned to the thread-requestor. If there are no free contexts at the remote MTP, the negative acknowledgment will be sent back to the thread–requestor by the remote MTP hardware. In both cases, the reply to the migration request will be written in the requestor's migration condition code register and made accessible for a use in the thread's program.

In the relocation code, the thread's continuation address and the contents of registers with the thread's live variables must be sent to the remote context by special local-to-remote register move (MVL2R) instructions. Before finishing the migration procedure by performing the instruction to release the current context, the

relocation code must execute the memory barrier operation to guarantee that all memory load/swap/store operations issued by the current thread have been completed.

The thread migrated to the remote context can start its execution as early as its continuation address loaded into PC, even before receiving all its migrated register variables. The execution of the migrated thread on the remote context will be stalled if/when the accessed register is not received yet. To help hardware with the detection of such migration-caused data hazards, a request for a hardware context includes the thread's LRVM value. If the request is granted by the remote MTP, its hardware writes this LRVM into the hazard detection table (HDT) for the allocated context. Once the value migrated to a register is received, the remote MTP clears the corresponding bit in HDT, allowing the register to be used by the thread's instructions.

3. Simulation Environment

3.1 Baseline System Architecture

The focus of our study is a NUMA system consisting of multiprocessor/memory nodes connected by a packet-switched network (Figure 1). Each node is a NUMA shared-memory multiprocessor consisting of up to 16 multithreaded processors each with its local memory (LM) connected by a crossbar switch and a broadcasting global barrier (GBAR) interconnect. The latter is used to signal the execution of so-called *global barrier* operations by each MTP. In this paper, we will present the results of our evaluation study of the technique of fine-grain mobile multithreading in the context of a single multiprocessor node.

We assume that each multiprocessor node with distributed memory can have a very large non-uniformity memory access ratio (the access time to remote memory divided by the access time to local memory). In this paper, we will study a node configuration with the ratio ranging from ~4:1 to ~20:1.

3.2 Multithreaded Processor Microarchitecture

Besides providing support for thread migration, the 64-bit MMT ISA has a RISC core of control, load/store with word addressing, integer and floating-point arithmetic operations including fused FP multiply-add/subtract, atomic read-modify memory swap operation, thread control and synchronization operations such as create/quit a thread, merge threads, local and global barriers. The latter operations are used to implement a weak consistency model [14]. This model allows multiple memory accesses to be pipelined, enforcing memory consistency only at synchronization points identified by the programmer or compiler.

In order to make the MTP design implementable with new aggressive, non-silicon technologies, we decided to keep the design complexity low by choosing:

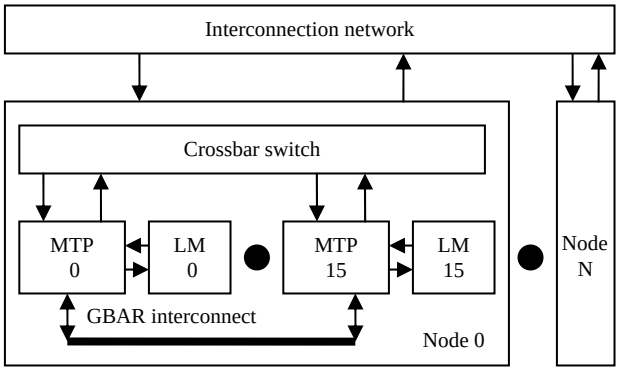

Figure 1. Baseline system structure.

- a simple RISC-type instruction set with less than 100 32-bit instructions of four formats;
- in-order single-instruction issue from each thread and out-of-order completion;
- no branch prediction, no register renaming.

The baseline multithreaded processor considered in our study has 16 hardware contexts, each with its own fetch/decode/control unit used for fetching and decoding instructions from a thread running in the context. MTP supports partitioning of its unified (integer/floating-point) register file into variable-size register frames assigned to threads. There are two register-addressing modes for each instruction: relative to the thread's register frame base (with register offsets) and absolute. HDT with full/empty bits associated with each data register is used to determine when contents of these registers are valid. The instruction issue from any thread is stalled by the dispatcher each time when the thread's operation accesses a non-valid register until the register contents is valid. Multiple threads running simultaneously within the same MTP/node can use a single copy of the program code.

Only four of the sixteen available thread contexts are allowed to issue an operation to the MTP datapath shared by all threads in each cycle. The datapath includes two integer units, two floating-point units, one load/store unit, two branch units, an instruction cache, and interface units communicating with the local memory and the crossbar switch. There is also a shared data cache. However, in this study we will not use it at all in order to evaluate the latency reduction through thread migration without any help from the hardware-controlled data cache. Table 1 gives the other critical parameters of the simulated MTP architecture, whose values are relatively close to the parameters of modern high-performance microprocessors.

3.3 Simulation Tools

The performance evaluation has been carried out with our own cycle-accurate parameterized model of an MMT system. It correctly simulates structural and data hazards occurring due to sharing of the MTP datapath by simultaneously running threads, as well as the runtime

Table 1. Baseline parameters.

Parameter	Value
Instruction Cache	2KB, direct mapped, with 32B lines, 4 sectors per line
a. latency	2 cycles
b. bandwidth	4 read + 1 write ops per cycle
Register file	16x16 64-bit integer/floating-point regs
a. fetch latency	2 cycles
b. bandwidth	8 read + 4 write-back ops per cycle
Local memory	2 MB
a. latency	6 cycles
b. bandwidth	2 accesses/cycle (one local, one remote)
Crossbar switch / node	16x16 for requests and 16x16 for replies
a. one-way latency	8-56 cycles (simulated range)
b. bandwidth	1 request + 1 reply per cycle per MTP
Instruction decode	2 cycles in 16 Decode/Issue Units
Instruction issue	1 cycle (max. 4 ops per cycle per MTP)
Branch address calc.	2 cycles in two Branch Units
Data address calc.	3 cycles in the Load/Store Unit
Integer Add/Subtract	3 cycles
FP Add/Subtract	7 cycles
Integer/FP Multiply	9 cycles
Integer/FP Divide	21 cycles (non-pipelined)

effects of the chosen sizes of operation & operand queues (including those used for receiving requests and replies from other processors/memory over the crossbar switch).

The simulator consists of more than 42K lines of code (mostly written in C, with TCL also used for GUI and script support). In this paper, we will show the basic steps necessary to write a mobile multithreaded program and present the performance evaluation results for only one of several benchmarks we simulated on the MMT architecture, namely the conjugate gradient (CG) Class S (Sample) from the NAS benchmark suite [15].

3.4 Parallelization with mobile threads

We started with the implementation of four CG versions: non-mobile sequential, non-mobile multithreaded, mobile sequential, and mobile multithreaded. Then, we added data prefetching to both of the multithreaded versions to get additional performance by fully utilizing all available registers in each thread's register frame. The number of the registers available for prefetching limits the depth of prefetch to one for the non-mobile and two for the mobile versions. There are 245 and 248 registers used in the non-mobile and mobile code with data prefetching, respectively.

To solve a linear system of equations iteratively, the conjugate gradient algorithm extensively uses a sparse matrix-vector product. The sparse matrix is stored in a compressed way by rows, which causes irregular and unpredictable memory access patterns at the runtime. Together with the $O(n^2)$-complexity of the sparse matrix-vector product, this explains why the program spends 85-90 percent of time on the execution of the product. Because of this, we will focus further on that part of the program, skipping the details of parallelization of other parts of the multithreaded versions. Below (Figure 2) is the sequential sparse matrix-vector product algorithm.

```
for (j=0, j<n, j++)  {
        sum = 0.0;
        for  (k = rowstr[j]; k < rowstr[j+1]; k++)
             sum = sum + a[k] * p[colidx[k]];
         q[j] = sum;
}
```

Figure 2. Sequential sparse matrix-vector product algorithm.

Here *n* is a vector length, *a* is a real array of length *nz* with nonzero elements (*nz* = 78148), *colidx* is an integer array of the same length *nz*, which contains the corresponding column indices, auxiliary integer array *rowstr* of length *n*+1, which points to the first element of each row, and arrays *q* and *p*, both of length *n*.

The outer iterations on the rows are parallel, so they can be assigned to parallel threads. In the beginning of the program, the master thread calculates the initial and final indices of array *rowstr* and passes them to each thread created later.

For the sequential CG algorithms, data are distributed statically in order to maximize their locality. As for the multithreaded versions, at first, we considered and rejected the naïve, non-interleaved, data distribution because the allocation of small arrays, like *p*, completely in one memory module would create "access bottleneck" for those memory modules and seriously hurt performance. Instead, we used block interleaving to spread data across memory modules. The MMT architecture provides hardware support for flexible memory interleaving through special interleaving control registers, which the user/compiler can write to specify both the size of interleaved data blocks (chunks) and the memory modules to be involved in interleaving. The interleaving factor for the non-mobile versions is 8 64-bit words, the minimum possible block size, which is proved to be the best experimentally.

In the base CG algorithm, locality of data is low. The majority of the memory access operations inside the inner loop are indirect, so their memory addresses cannot be calculated in advance. Mobile multithreading increases data locality by allowing threads to migrate to these indirectly-accessed data. In the algorithm, arrays *a* and *colidx* are accessed with the same index. These two arrays are aligned with respect to each other and interleaved. As a result of this, almost two thirds of the read memory accesses from the thread migrated to the location of these arrays become local. The store operations are not on the critical path of the execution because the data calculated in each (macro) step of the whole CG algorithm will not be used until the next step. The time spent on barriers, when the program waits for all store operations to be completed, is negligible because it is done only once at the end of each macro step. The basic structure of the mobile multithreaded sparse matrix-vector product algorithm is shown in Figure 3.

Table 2. Statistics for the CG class S NAS benchmark (for the 32-cycle one-way crossbar switch latency).

Program	Label	Exec. time (M cycles)	Inst. count (M)	Inst/cycle (max. 4)	Aver. mem. access time (cycles)	Switch traffic (M bytes)
Sequential	$SEQ1_S$	3,391.52	3,238.7	0.0066	44.15	918.6
Multithreaded w/o gbar	$T256_S$	30.46	3,031.1	0.6889	66.87	1,496.8
Multithreaded w. gbar	$T256_SB$	28.63	2,991.6	0.7244	67.10	1,480.7
Multithreaded w. gbar 1x prefetch	$T256_SP_1B$	18.16	3,894.0	1.4525	68.16	1,479.2
Sequential mobile	$SEQ1_M$	3,139.71	3,027.6	0.0067	25.36	476.5
Multithreaded mobile w/o gbar	$T224_M16_S$	24.37	3,139.9	0.8891	30.13	620.0
Multithreaded mobile w. gbar	$T224_M16_SB$	21.38	3,099.1	1.0014	29.79	606.5
Multithreaded mobile w. gbar 2x prefetch	$T224_M16_SP_2B$	11.74	3,995.9	2.3017	30.31	620.0

```
initialize thread context's migration control register;
for (j=start_of_chunk, j<end_of_chunk, j++) {
      sum = 0.0;
      end_of_k = rowstr[j+1];
      /* inner loop */
      for (k = rowstr[j]; k < end_of_k ; k++) {
          sum = sum + (remote) a[k] * p[ (remote) colidx[k]];
          remote_received = status of request;
          if (remote_received) {
              start thread with inner loop in remote context;
              perform memory barrier operation;
              copy k, end_of_k, j, end_of_chunk, sum to remote context;
              release current (local) thread context;
          }
      }
      q[j] = sum;
}
```

Figure 3. Base sparse matrix-vector product algorithm with migrating threads.

The LVRM value written into the migration control register specifies the registers containing the *k, end_of_k, j, end_of_chunk, sum* variables which are to be copied in the process of migration. Remote load operations are used for accessing arrays *a* and *colidx*. If arrays *a* and *colidx* are local, no migration requests are generated. When a thread migrates to some remote context, it has its data local, and there will be no need for migration for a certain number of iterations. This number can be calculated from the size of the block used for interleaving. The final version of the algorithm has two different inner loops, one with the migration code, and another without it (the latter is executed by a thread after migration).

We will be using the following notations for different versions of the CG codes: SEQ (Sequential) or T (Multithreaded), followed by the number of threads with subscript S for static (non-mobile), and M for mobile threads. P means that prefetching is used with its subscript number denoting the depth of prefetch. B indicates the use of global barrier operations implemented in hardware.

3.5 Performance results

The performance evaluation results for all CG versions studied and the speedup for the best multithreaded versions are shown in Table 2 and Figure 4. With 16 master threads (one per MTP) used for synchronization purposes only, there could be up to 240 mobile threads in the code. However, in order to make thread migration effective, one hardware context must be always left free in each MTP at the start of the program(s).

Not surprisingly, the lack of such free contexts in the code with 240 mobile threads leads to the performance worse than that with 224 threads (Figure 4).

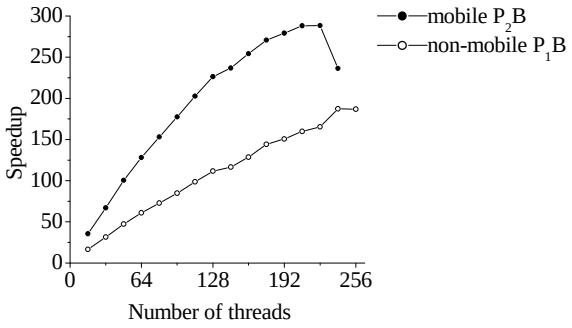

Figure 4. Speedup for the best multithreaded versions in relation to the best non-mobile sequential version.

With the fixed local memory latency of 6 cycles, we varied the switch latency to analyze how efficient mobile multithreading can be in handling the remote access latencies up to 20 times larger than the local one.

For all the CG cases simulated, the code with mobile threads and data prefetching achieves the best performance in terms of execution time (Table 3) and efficiency in terms of the speedup achieved (Figure 5).

Table 3. Execution times (in millions of cycles) for the different values of switch latency.

Program	One-way switch latency (cycles)						
	8	16	24	32	40	48	56
$SEQ1_S$	1,927.5	2,415.5	2,903.5	3,391.5	3,879.5	4,367.4	4,854.7
$T256_S$	14.9	19.5	24.9	30.5	36.1	41.7	47.3
$T256_SB$	14.3	18.4	23.4	28.6	33.9	39.2	44.5
$T256_SP_1B$	12.7	13.7	15.6	18.2	21.1	24.1	27.3
$SEQ1_M$	1,744.1	2,209.3	2,674.5	3,139.7	3,604.8	4,070.0	4,535.1
$T224_M16_S$	13.8	17.2	20.8	24.4	28.0	31.7	35.4
$T224_M16_SB$	12.4	15.3	18.3	21.4	24.6	27.7	30.9
$T224_M16_SP_2B$	10.7	10.9	11.1	11.7	12.7	14.3	16.0

The speedup curve shown in Figure 5 also gives a clear evidence of huge performance benefits of combining thread migration with program-controlled data prefetching in order to reduce/tolerate data access latency.

On average, both the average memory access time (Figure 6) and the switch traffic (Figure 7) observed in the mobile multithreaded programs are approximately 45-50 percent of the relevant time and traffic in the non-mobile code for the analyzed range of the switch latencies.

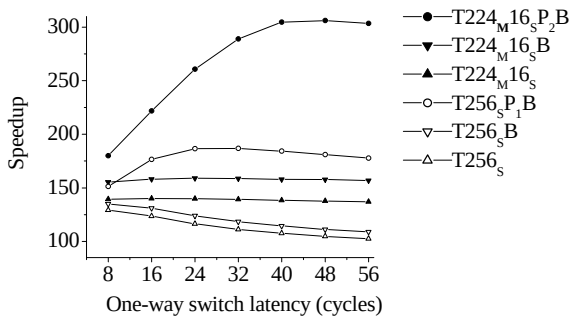

Figure 5. Speedup in relation to the best non-mobile sequential program versus switch latency.

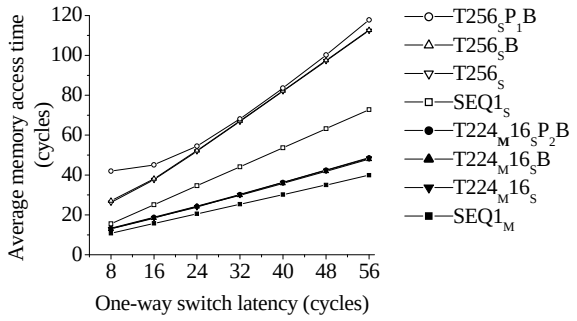

Figure 6. Average memory access time versus switch latency.

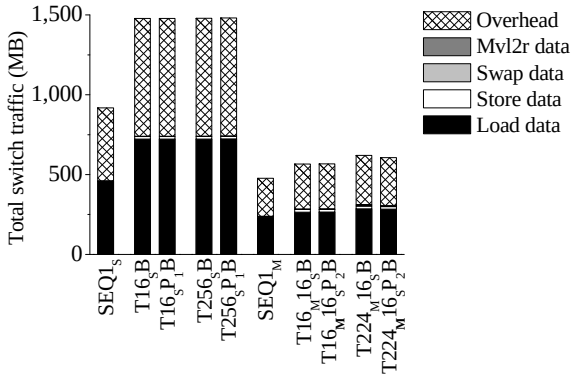

Figure 7. Total switch traffic.

4. Conclusions

The performance evaluation results for the CG NAS benchmark (as well as our results for other benchmarks not shown here due to a lack of space) demonstrate that fine-grain thread migration supported in the MMT architecture can be effectively used to reduce memory latency and switch traffic for the NUMA shared-memory multiprocessors considered in this paper. The sustained processor performance expressed as the average IPC in each MTP is more than 50% of the peak performance for the mobile multithreaded CG code with data prefetching.

Acknowledgments

This work is supported by DoD.

References

[1] J. Laudon and D. Lenoski, The SGI Origin: A ccNUMA highly scalable server, *Proc. 24th Int'l. Symp. on Computer Architecture*, Denver, USA, 1997, 241-251.

[2] R. Alverson *et al*, The Tera computer system, *Proc. the 1990 Int'l Conf. on Supercomputing*, Amsterdam, The Netherlands, 1990, 1-6.

[3] M. Dorojevets and P. Wolcott, The El'brus-3 and MARS-M: recent advances in Russian high-performance computing, *The J. of Supercomputing*, 6, 1992, 5-48.

[4] A. Agarwal, R. Bianchini, D. Chaiken *et al*, The MIT Alewife machine: architecture and performance, *Proc. 22nd Int'l Symp. on Computer Architecture*, Santa Margherita Ligure, Italy, 1995, 2-13.

[5] M. Tremblay, MAJC: Microprocessor architecture for Java computing, HotChips '99, Palo Alto, USA, 1999.

[6] A. Cataldo, Intel looks to bridge gap in multithreading CPU landscape, *EE Times*, Oct. 16, 2001.

[7] W.C. Hsieh, Dynamic Computation migration in distributed shared memory systems, PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1995. Available as MIT/LCS/TR-665.

[8] E. Mascarenhas and V. Rego. Ariadne, Architecture of a Portable Threads system supporting thread migration, *Software- Practice and Experience*, 26(3), 1996, 327-357.

[9] A. Itzkovitz, A. Schuster, and L. Shalev, Thread migration and its applications in distributed shared memory systems, *The J. of Systems and Software*, 42(1), 1998, 71-87.

[10] B. Weissman, B. Gomes, J. Quittek, and M. Holtkamp, Efficient fine-grain thread migration with Active Threads, *Proc. 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing*, Orlando, USA, 1998, 410-414.

[11] K. Thitikamol, and P. Keleher, Thread migration and communication minimization in DSM systems, *Proc. of the IEEE*, 87(3), 1999, 487-497.

[12] G. Gao, K. Likharev, P. Messina, and T. Sterling, Hybrid technology multithreaded architecture, *Proc. 6th Symp. Frontiers of Massively Parallel Computation*, Annapolis, USA, 1996, 98-105.

[13] M. Dorojevets, COOL Multithreading in HTMT SPELL-1 processors, *Int'l. Journal on High Speed Electronics and Systems*, 10(1), 2000, 247-253.

[14] M. Dubois, C. Scheurich, and F. Briggs, Memory access buffering in multiprocessors, *Proc. 13th Int'l. Symp. on Computer Architecture*, Tokyo, Japan, 1986, 434-432.

[15] D. H. Bailey *et al*, The NAS Parallel Benchmarks, RNR Technical Report, 1994. Available as RNR-94-007.