International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 1, pages 21–35, January 2011

A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads

MARTTI FORSELL
Platform Architectures, VTT, Box 1100
Oulu, FI-90571, Finland

Received: June 30, 2010
Revised: October 27, 2010
Accepted: December 8, 2010
Communicated by Akihiro Fujiwara

Abstract

It is possible to implement the parallel random access machine (PRAM) on a chip multiprocessor (CMP) efficiently with an emulated shared memory (ESM) architecture to gain easy parallel programmability crucial to wider penetration of CMPs to general purpose computing. This implementation relies on exploitation of the slack of parallel applications to hide the latency of the memory system instead of caches, sufficient bisection bandwidth to guarantee high throughput, and hashing to avoid hot spots in intercommunication. Unfortunately this solution can not handle workloads with low thread-level parallelism (TLP) efficiently because then there is not enough parallel slackness available for hiding the latency. In this paper we show that integrating non-uniform memory access (NUMA) support to the PRAM implementation architecture can solve this problem and provide a natural way for migration of the legacy code written for a sequential or multi-core NUMA machine. The obtained PRAM-NUMA hybrid model is defined and architectural implementation of it is outlined on our ECLIPSE ESM CMP framework. A high-level programming language example is given.

Keywords: Parallel computing, Computational models, Thread-level parallelism, PRAM, NUMA

1 Introduction

The processor manufacturers aim to duplicate the number of *chip multiprocessors* (CMP) every second year to provide speedup now that the speed development of single-core processors has virtually halted [19][20]. Unfortunately, virtually all software has so far been written using the sequential computing paradigm and there is no obvious way to execute sequential programs on a CMP with a high utilization. This raises two questions to limelights of scientific and industrial research and development: How to program a parallel computers for general purpose functionality? Can an average programmer do it? Execution models used in current CMPs to address these issues include —*symmetric multiprocessors* (SMP), *non-uniform memory access* (NUMA), its cache coherent variants, e.g. *cache coherent non-uniform memory access* (CC-NUMA), and *message passing* (MP). SMP consists of a small number of identical processors with local caches that are connected to the main memory via a bus or crossbar so that access to the memory is equidistant. NUMA consists of multiple processors (with local memory banks) connected together via an intercommunication network so that non-local memory accesses have higher distance and traffic situation dependent latency than local accesses [32]. CC-NUMA has similar structure to NUMA but adds caches that are kept coherent also for remote accesses to partially hide the memory access latency [23]. MP consists of multiple processors with local memories communicating via message passing network [18][25]. These models are tedious to program because a programmer can not be sure about the exact state of computation unless he inserts costly barrier synchronizations and takes care of complex low-level communications in MP. Furthermore, the performance scalability of SMP and NUMA with respect to number of cores per chip is weak, while MP suffers from high software overheads due to suboptimal implementation of message passing primitives making it non-ideal for exploitation of fine-grained parallelism. In *vector computing* (VC) [17][33], multiple vector lanes process multiple vector elements in parallel under the control of a single instruction. It solves part of the problem by providing a synchronous model of computation but can not be efficiently applied to code containing control parallelism, heterogeneity, or non-vectorizable portions. Despite of a common belief the programmability problem has been partially solved in the 70 ’ s with the introduction of the *parallel random access machine* (PRAM) model of computation [15]. PRAM is a fine-grained step-synchronous shared memory model consisting of a set of processors connected to the same clock and shared memory. All operations including parallel memory accesses execute in unit time implying that execution is lock-step synchronous. The PRAM model provides a simple abstraction of a parallel computer that is easy to understand and program as a natural extension of the widely used sequential computational model. This is because of synchronicity of subtask execution, full control of both data and control operations, high enough abstraction of intercommunication in the form of uniform shared memory. At programming language level this makes possible to use shared variables and eliminates the need for lockings and atomic operations unless the program declares asynchronous tasks at high level in purpose [21][22]. Past attempts to realize a PRAM include the omega network-based NYU Ultracomputer [31], CEDAR [16] and IBM Research Parallel Processor Prototype (RP3) [28], butterfly-based Fluent machine [29], 3D torus-based Cray MTA supercomputer [2] and its successors MTA2 and XMT provided by Cray, multiport memory-based solution [4], and butterfly-based SB-PRAM [1, 22]. Unfortunately, these attempts have been mainly unsuccessful due to sub-optimal solutions, like non-scalable interconnect topologies, inefficient co-exploitation of multiple levels parallelism, sub-optimal shared memory emulation algorithms, and out-dated prototyping technologies. Recently, applying these ideas to architectures designed especially for PRAM implementation on a CMP with a help of the network-on-chip technology [3] have lead to two very promising research lines:

- Forsell et al. try to implement *multioperation concurrent read concurrent write* (MCRCW) PRAM with the sparse/multimesh-based ECLIPSE CMP architecture [5] [10] [11] [14].
- Vishkin et al. try to implement a PRAM-like machine with a bit more relaxed synchronicity than in the pure PRAM with the mesh of trees-based *XMT* CMP architecture [34][35].

Besides the actual CMP implementation architectures, both these research lines are addressing other necessary infrastructure, including a high-level parallel programming language [7][26], compiler for such language [8][26], optimizations for the compiler [6], and sample algorithms. Unfortunately these PRAM implementations can not handle workloads with low *thread-level parallelism* (TLP) efficiently because then there would not be enough parallel slackness available for hiding the latency. In our recent work we have proposed the *Configurable emulated shared memory* (CESM) architecture for addressing the single threaded workload performance problem by using bunching of threads [12]. However, that work does not support efficient execution of true NUMA employing multiple cooperative cores and non-local accesses nor presents the effects of NUMA access to the model of computation. In this paper we show that integrating full NUMA support to the PRAM implementation architecture can solve the low-TLP workload performance problem and provide a natural way for migration of the legacy code written for a sequential or multi-core NUMA machine without compromising the ability to execute medium and high TLP code efficiently. The obtained PRAM-NUMA hybrid model is defined and architectural implementation of it is outlined on our ECLIPSE CMP framework. A high-level programming language example is given. The rest of the paper is organized so that in Section 2 we describe the idea of the PRAM-NUMA model of computation, explain why it solves the low-TLP problem and discuss programming issues. In Section 3 we

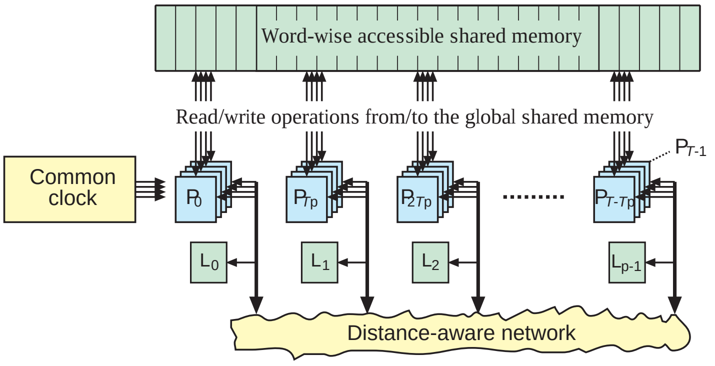


Figure 1: PRAM-NUMA model. $P_0..P_{T_p-1}$ are processors and $L_0..L_{p-1}$ are local memories.

outline the architectural implementation of a CMP using the PRAM-NUMA model. An evaluation of the architecture is given in Section 4. Finally, in Section 5 we give our conclusions.

2 PRAM-NUMA model of computation

In order to be able to exploit locality and address low parallelism driven problems of current PRAM implementations and to support simple migration of the sequential or multicore NUMA legacy code, we take a look at a new model providing full NUMA capabilities. Unlike PRAM, the model needs to be bounded and provided with the concept of distance metric, reflecting the relative distance of the processors in our physical 3-dimensional world.

2.1 Model

The PRAM-NUMA model of computation consists of T processors grouped as P T_p-processor groups, a word-wise accessible global shared memory, P local memory blocks, a metric defining distance between the processor groups and target memory blocks, and distance-aware interconnection network (see Figure 1). Each processor is attached to the shared memory like in PRAM and each processor group is attached to its own local but interconnected memory block like in NUMA. The interconnection network connects the local memory access paths of processor groups together and is distance-aware with respect to the metric in a sense that the latency of routing is proportional to the distance between the source processor and destination memory block. The bandwidth of a group of processors to the shared memory and local memory are the same. Each processor can be configured to either the PRAM mode or the NUMA mode. Along with configuration from the PRAM mode to the NUMA mode, one can set the state of the processor to point an arbitrary state within the group it belongs to. With this indirection of states, two or more processors belonging to a group can be configured to a NUMA *bunch* so that they execute a common instruction stream and share their state with each other, i.e. execute code like a single processor.

2.2 Solving the low-TLP problem

The only known PRAM implementation, *emulated shared memory machine* (ESM), uses parallel slackness, i.e. grouping the processors and lowering the execution speed of individual processors of the group, to hide the latency of the physically distributed memory system [30]. The PRAM-NUMA model solves the low-TLP execution problem by providing a possibility to configure two or more processors of a group to use a common state like they were a single processor. That way, it can perform NUMA access to the local memories and remote memories via the distance-aware NUMA network without suffering from the decreased execution speed of individual processors. In the case of a low TLP portion of code, a programmer can just set up T_p processors per group to run the portion as a NUMA bunch and gain more performance proportionally to the number of processors in the bunch. This applies both to sequential code portions written for a sequential machine and parallel code portions written for a multicore NUMA machine.

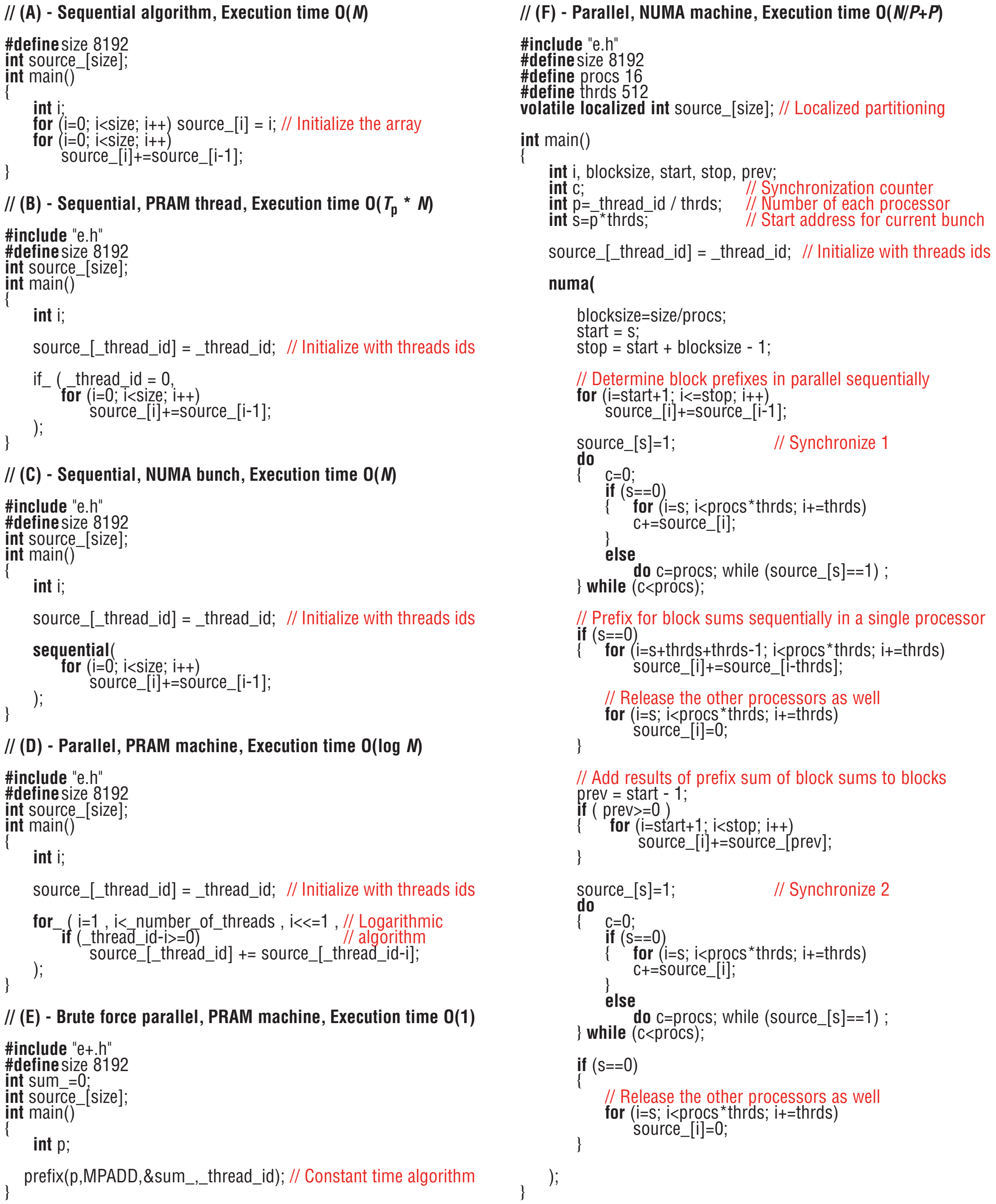
2.3 Using the hybrid model in programming

In our earlier work we have outlined a development methodology for ESM CMPs [9]. It consists of a strong model of computation, a c-like TLP programming language e, an multi-level parallelism optimization algorithm, and application development flow: The model integrates exploitation of *instruction-level parallelism* (ILP) seamlessly to TLP execution. The e-language supports explicit synchronous and asynchronous parallel programming with special primitives. The ILP-TLP optimization algorithm yields to high ILP utilization in TLP execution independently of the degree of inter-thread dependencies. The detailed application development flow allows a developer to write parallel applications with a help of supporting theory of parallel algorithms [21][22]. Finally, it allows one to apply parallel programming techniques from fully asynchronous coarse grained threads (processors in the PRAM-NUMA model terminology) down to synchronous threads interchanging information with the finest granularity. With this methodology, a PRAM-NUMA CMP can be programmed like any ESM machine for parallel enough functionality but it allows also efficient execution of low-TLP code. To support easy inclusion of sequential portions of code, we add the *sequential(s)* construct to the e-language. It bunches all the threads of processor 0, which arrive to the construct and in which the thread with id 0 is running, and puts the threads running in other processors waiting until the bunch has executed the statement *s* but does not effect on execution of threads not arriving to the construct. After that, it restores the synchronicity of all the threads with a fast barrier synchronization. To support full NUMA execution among multiple processors, we add also the *numa(s)* construct. It sets up a NUMA bunch in each processor (from which threads are arriving to the construct) to execute the statement s in parallel. The sizes of the bunches are determined processor-wisely by the number of threads arriving the construct. Other threads are not affected. Unfortunately, designing programs for this kind of asynchronous NUMA execution without any latency hiding mechanism is much more difficult than for the PRAM mode. In order to get good performance one must identify independent portions of code, partition data so that locality of data references is maximized, and orchestrate execution of asynchronous bunches with explicit synchronizations where necessary. Easy-to-use PRAM algorithms execute very inefficiently in the NUMA mode if no above optimizations are applied due to slow synchronization compared to the PRAM mode, and unfortunately, there is no algorithm to automatically do these optimizations for an arbitrary functionality. Migrating sequential legacy code to a P-processor T-threaded PRAM-NUMA CMP happens with joining all the threads of a single processor core to a bunch e.g. with the *sequential()* construct and executing the code on the bunch so that the full power of the core can be used for it. If the legacy code is written and optimized to an L-core NUMA machine and $L < T$, one can set up L bunches so that threads divide to all P processor cores as evenly as possible ($P < L$) or to as many processors as there is room for ($P >= L$), and execute the code with them. Otherwise one needs to port the code to T threads and execute the code in the PRAM mode with a help of fast synchronizations and other strong properties of the PRAM mode or better yet, rewrite the functionality as a full PRAM program and execute it in the PRAM mode to get the best performance out of it.

2.4 Programming example

Consider the computational problem of calculating a prefix sum for an array of N (8192) integers. Consider solving it in a PRAM-NUMA machine with P (16) T_p (512)-threaded processors (where the total number of threads is $T = T_pP$ (8192) for the PRAM mode and the number of processor for the NUMA mode is P (16)) sequentially and in parallel on both the PRAM and NUMA modes. A well-known sequential algorithm to solve this computational problem makes use of a running sum while iterating from the first to last element of the array (see Figure 2a). This algorithm can be executed on a single PRAM thread in the PRAM mode e.g. by assigning it to thread 0 and putting the rest of the threads to wait for synchronization while thread 0 does the computation (see Figure 2b for implementation in the e-language [7]). In the NUMA mode, this algorithm can be executed with a single bunch joining together all the threads of processor 0 (see Figure 2c for implementation in the e-language) with a help of a *sequential* construct. The execution time for both the PRAM mode and NUMA mode programs is O(N) but due to exploitation of parallel slackness in the PRAM mode, individual threads operate at the $1/T_p$ frequency of the processor. As a result, the PRAM mode program executes T_p times slower than the NUMA version if higher synchronization costs and memory overheads of the NUMA execution are not taken into account.

A fine-grained parallel PRAM algorithm for solving this problem can be formed e.g. so that N-1 pairs of adjacent integers are added together in parallel reducing the problem to summing N-2 partial sums. After that those N-2 partial sums are summed with a data element two positions to left to it and N-4 partial sums are obtained. These iterations are continued for $\log N$ rounds in total so that all the prefix sums have been computed (see Figure 2d). Note that one needs to synchronize the threads only in the end of the for_-loop because as the condition of the inner if-statement becomes false it will remain false until the exit condition is met. Thus, the unbalanced if-statement forms a miniature asynchronous programming area inside the synchronous for_-loop. If the ordering of the prefix computation does not matter and the underlying PRAM-NUMA machine supports the arbitrary ordered multiprefix operations [10], we can further speedup the computation with a brute force algorithm that produces the result in constant time O(1). This happens with just two machine instructions that are needed to execute the primitive *prefix*(p,MPADD,&sum_,_thread_id) computing an arbitrarily ordered multiprefix with a help of the shared variable sum_ to local variable p for each thread (see Figure 2e). The initialization is done implicitly by using the initial values _thread_id for each thread and the prefix sum is stored to local variable p for each thread. Note that constant time execution of a multiprefix does not violate the logarithmic lower bound of multiprefix computation, since the individual threads of the PRAM-NUMA machine are executing $T_p > \log P$ times slower than the clock of the machine for latency hiding reasons. In the NUMA mode, parallel execution is asynchronous and barrier synchronizations take far longer time than instruction execution. In order to solve this computational problem efficiently, processed data must be distributed so that it is close to processors referring to it (or local if possible), computation must be done sequentially in all processors (with all processor doing this concurrently), and the number of synchronizations and amount of data intercommunication should be minimized. Another problem arises from the fact that the number of processor cores, i.e. available threads, is smaller than in the PRAM mode implying that a processor should process more than one data element. Matching the number of data elements to the available processors so that the problem gets solved is not trivial, but still relatively simple, resulting to a blocking coarse-grained parallel algorithm: One needs to divide the data array into P blocks, distribute the blocks to the local memories of processors that are going to process them. Execution happens by computing the blockwise prefix sums locally on each processor, determining an offset for each block by computing the prefix sum of block sums in a single processor, distributing the obtained offsets back to the blocks, and adding the offset to the blockwise prefixes sequentially on each processor. This kind of an algorithm requires only two barriers and intercommunication happens only during synchronizations and offset computation. Figure 2f shows an implementation of the algorithm in the e-language. The execution time of this program is O($N/P+P$) plus remote memory access and synchronization delays, because local computations take O(N/P) time and prefix of block sums takes O(P) time. Note that if $N > PT_p$ and one needs an ordered prefix sum, one must use similar three phase algorithm executing now in time O(N/P+$\log N$) also for the PRAM mode but without the locality maximization and need to compute offsets sequentially. Finally, if the implemented NUMA functionality would have been more complex, far more complex programming structures, including movements of data so that it is close to the processors using it and a lot of full and partial barrier synchronizations and lockings, would have been needed for efficient NUMA processing.



```
// (A) - Sequential algorithm, Execution time O(N)

#define size 8192
int source_[size];
int main()
{
    int i;
    for (i=0; i<size; i++) source_[i] = i; // Initialize the array
    for (i=0; i<size; i++)
        source_[i]+=source_[i-1];
}
```

```
// (B) - Sequential, PRAM thread, Execution time O(Tp * N)

#include "e.h"
#define size 8192
int source_[size];
int main()
{
    int i;

    source_[_thread_id] = _thread_id;  // Initialize with threads ids

    if_ ( _thread_id = 0,
        for (i=0; i<size; i++)
            source_[i]+=source_[i-1];
    );
}
```

```
// (C) - Sequential, NUMA bunch, Execution time O(N)

#include "e.h"
#define size 8192
int source_[size];
int main()
{
    int i;

    source_[_thread_id] = _thread_id;  // Initialize with threads ids

    sequential(
        for (i=0; i<size; i++)
            source_[i]+=source_[i-1];
    );
}
```

```
// (D) - Parallel, PRAM machine, Execution time O(log N)

#include "e.h"
#define size 8192
int source_[size];
int main()
{
    int i;

    source_[_thread_id] = _thread_id;  // Initialize with threads ids

    for_( i=1 , i<_number_of_threads , i<<=1 , // Logarithmic
        if (_thread_id-i>=0)                  // algorithm
            source_[_thread_id] += source_[_thread_id-i];
    );
}
```

```
// (E) - Brute force parallel, PRAM machine, Execution time O(1)

#include "e+.h"
#define size 8192
int sum_=0;
int source_[size];
int main()
{
    int p;

   prefix(p,MPADD,&sum_,_thread_id); // Constant time algorithm
}
```

```
// (F) - Parallel, NUMA machine, Execution time O(N/P+P)

#include "e.h"
#define size 8192
#define procs 16
#define thrds 512
volatile localized int source_[size]; // Localized partitioning

int main()
{
    int i, blocksize, start, stop, prev;
    int c;                          // Synchronization counter
    int p=_thread_id / thrds;       // Number of each processor
    int s=p*thrds;                  // Start address for current bunch

    source_[_thread_id] = _thread_id;  // Initialize with threads ids

    numa(

        blocksize=size/procs;
        start = s;
        stop = start + blocksize - 1;

        // Determine block prefixes in parallel sequentially
        for (i=start+1; i<=stop; i++)
            source_[i]+=source_[i-1];

        source_[s]=1;                // Synchronize 1
        do
        {   c=0;
            if (s==0)
            {   for (i=s; i<procs*thrds; i+=thrds)
                c+=source_[i];
            }
            else
                do c=procs; while (source_[s]==1) ;
        } while (c<procs);

        // Prefix for block sums sequentially in a single processor
        if (s==0)
        {   for (i=s+thrds+thrds-1; i<procs*thrds; i+=thrds)
                source_[i]+=source_[i-thrds];

            // Release the other processors as well
            for (i=s; i<procs*thrds; i+=thrds)
                source_[i]=0;
        }

        // Add results of prefix sum of block sums to blocks
        prev = start - 1;
        if ( prev>=0 )
        {    for (i=start+1; i<stop; i++)
                 source_[i]+=source_[prev];
        }

        source_[s]=1;                // Synchronize 2
        do
        {   c=0;
            if (s==0)
            {   for (i=s; i<procs*thrds; i+=thrds)
                c+=source_[i];
            }
            else
                do c=procs; while (source_[s]==1) ;
        } while (c<procs);

        if (s==0)
        {
            // Release the other processors as well
            for (i=s; i<procs*thrds; i+=thrds)
                source_[i]=0;
        }

    );
}
```

Figure 2: Multiprefix sum of an array of 8192 integers as (a) sequential algorithm, (b) sequential PRAM program for a single thread, (c) sequential NUMA program for a single bunch, (d) parallel PRAM program for the whole machine, (e) parallel brute force PRAM program for the whole machine, and (f) parallel NUMA program for the whole machine.

3 Architectural implementation

The PRAM-NUMA model of computation can partially be implemented with the *configurable emulated shared memory machine* (CESM) architecture [12] but CESM threads can only refer to memories local to their processors in the NUMA mode. In order to remove this limitation we outline architectural solutions needed for full NUMA support on a top of CESM. We will also discuss on alternative architectural solutions for implementing the PRAM-NUMA model.

3.1 CESM architecture

The CESM architecture is a hybrid architecture implementing a PRAM model and partially supporting the NUMA model. A CESM CMP consists of P T_p-threaded (in total constituting $T = PT_p$ threads) F-functional unit *MultiBunched/Threaded Architecture with Chaining* (MBTAC) processor cores [14] connected to a distributed memory system (see Figure 3). The memory system has P dedicated instruction memory and local data memory modules, P T_p-line step caches and scratchpads attached to processors, P fast data memory modules with active memory units, and a high-bandwidth multimesh interconnection network. Step caches, scratchpads and active memory units are used to support concurrent memory access and multioperations in the PRAM mode [10]. A MBTAC processor features A ALUs, M memory units, compare unit, and sequencer organized as a chain for the PRAM mode and a single ALU, memory unit, and sequencer organized in parallel for the NUMA mode. In order to save in hardware costs and to provide as seamless configurability between the PRAM and NUMA models as possible, the execution pipelines for the modes are merged and share some units like fetcher, operand select, and the first ALU (see Figure 4). The effective length of the pipeline is T_p for the PRAM mode and 4 for the NUMA mode. The CESM architecture implements support for fast and synchronous switching between the PRAM and NUMA modes for groups of threads with dedicated machine language instructions JOIN and SPLIT [12]. Synchronous switching is necessary because a NUMA bunch may get switched back to the PRAM mode in the middle of a PRAM step.

The first realization of CESM is our TOTAL ECLIPSE architecture [14] aimed for universal general purpose CMP use. TOTAL ECLIPSE supports the arbitrary MCRCW PRAM model and provides limited support for NUMA execution in a form of processor-wise thread bunching with local memory access but relying on the PRAM mode for all non-local memory accesses. This makes execution of low-TLP functionalities as efficient as with standard sequential processors using the NUMA convention as long as NUMA bunches access only local data.

3.2 Support full NUMA

The high-bandwidth interconnection network used for PRAM emulation can also be used for NUMA operation. This is because the maximum number of memory system injections per clock cycle stays the same regardless of the mode and the synchronization wave technique used for the PRAM mode does not interfere or cause deadlocks with NUMA references. Instead the PRAM mode offers free synchronizations after the number of threads in the bunch instructions have been executed as long as referencing happens within a single step. Unfortunately, the virtual ILP optimized pipeline of the original MBTAC processor does not support a single injection point to the network if both PRAM mode and NUMA mode threads are executed concurrently (see Figure 4). Furthermore, concurrent local and remote accesses can not be supported with a normal operation speed of a single port memory. This suggests that a separate NUMA network, dual speed or dual ported memory modules, some amount of buffering capacity to handle dual injection conflicts, or revision of the pipeline structure is needed. For the PRAM-NUMA architecture proposed in this work, we selected to use fast memories supporting dual access per clock cycle since the current MCRCW support also requires fast memories. In the future, PRAM-NUMA-aware TOTAL ECLIPSE CMP variants, however, we will likely trade these fast memories to normal ones attached to fast module-level caches and therefore we also evaluate the slower buffer-based solution in which simultaneous accesses are queued. The dual injection point problem does delay non-local accesses if both PRAM mode threads

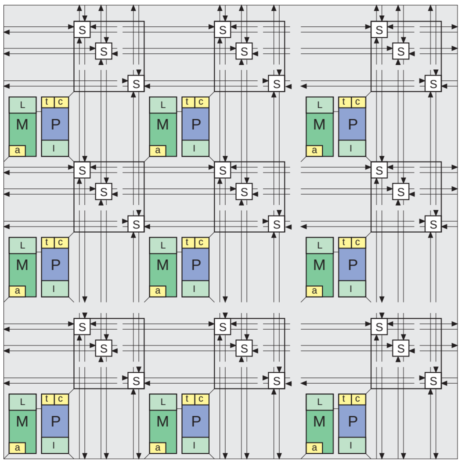


Figure 3: Block diagram of the CESM architecture (P=processor, M=shared data memory, L=local data memory, I=instruction memory, a=active memory unit, c=step cache, and t=scratchpad).

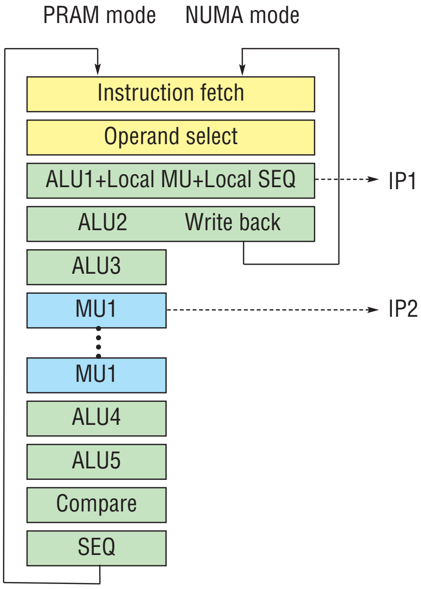


Figure 4: Block diagram of the MBTAC pipeline with 5 ALUs and a single memory unit (MU=Memory Unit, SEQ=Sequencer, IP=Injection Point).

and NUMA mode bunches are executed simultaneously. The throughput will, however, remain the same since the maximum number of injections per step stays the same assuming that the local memory access latency will be hidden with extra pipeline stages. In any case, it should be noted that need to use full NUMA support may not be as frequent and efficient as one might think at the first glance, since in the case of a sequential functionality it is possible to maximize the locality of computation by placing all the data needed in that computation into the local data memory of the processor group. Furthermore, remote accesses are often faster in the PRAM mode than in the NUMA mode due to missing latency hiding in the latter (a memory read instruction blocks the processor pipeline until the read reply is received) and slow synchronization. Finally, there is evidence that large amounts of parallelism is available for almost any kind of problems if a strong enough model of computation (e.g. PRAM) is implemented [24] [22] [36]. This leaves a seamless support for the legacy code the most important motivation for including the full NUMA support for future ESM CMPs.

3.3 Alternative solutions

It is possible to provide NUMA support to ESM machines also with more traditional techniques, e.g. with dedicated scalar units and a dedicated intercommunication network between them. This does not however support seamless configurability between PRAM mode and NUMA modes like the PRAM-NUMA architecture does but rather concurrent operation of the PRAM and NUMA hardware. As a result, programming such a machine could be tricky and some of the advantages gained with the simplicity of the plain PRAM model would be lost. An alternative approach for low-TLP code support in the CESM architecture would be to use global memory units and certain global memory locations to send and receive messages between the low-TLP functionality threads. While we predict that the plain CESM architecture with its high-throughput interconnection network would make the software overhead of a low-level message passing library implementation very low, potentially yielding to very efficient locality-aware execution and easy exploitation path for message passing legacy code, the details of such a solution are out of the scope of this paper.

4 Evaluation

In order to illustrate the improvements achievable with the proposed (dual-access) PRAM-NUMA architecture on a realistic CMP, we mapped parallel and sequential versions of seven parallel computational problems (see Table 1) to three CMP configurations. The configurations are E4, E16 and E64 having 4, 16 and 64 eleven-FU 512-threaded processors with 4-way set associative step caches connected to fast (single cycle access/cycle time SRAM) on-chip memory via a network utilizing switches with 16-element FIFOs (see Table 2). Three of the problems are fixed size and others depend on the number of threads in a processor core. For comparison purposes, the programs were mapped to both PRAM threads (processors in the model terminology) and NUMA bunches. We compiled, optimized (ecc -O2 -ilp -fast) and loaded the programs to the CMP configurations and executed them with our CMP simulator modified for the PRAM-NUMA model. The mapping of functionality in the PRAM mode was done in a straight-forward way assigning the elements of repetitive data structures like arrays to threads in a natural order, i.e. data element 0 is assigned to thread 0, data element 1 is assigned to thread 1 on so on. For the NUMA programs, we maximized the locality of processing and references by partitioning the data to local memories so that accessing remote modules is needed only in the cases of synchronizations and combining the results of the subtasks.

In order to determine the PRAM mode execution performance, we executed the parallel versions of the programs in the PRAM-NUMA CMPs in the PRAM mode. The results as execution time overhead with respect to PRAM with the same configuration but with an ideal memory system are shown in Figure 5a. We can observe that the PRAM mode execution speed of the PRAM-NUMA architecture is very close to that of ideal PRAM, mean overheads being only 0.8%, 1.7%, and 1.4% for E4, E16, and E64, respectively. These PRAM mode results are identical to the CESM architecture [12] since the PRAM-NUMA architecture is a direct extension of CESM and no

Table 1: Evaluated computational problems and features of their sequential and parallel implementations (E=execution time, M=size of the key string, N=size of the problem, P=number of processors, T=number of threads, W=work). Note that fft, mmul, and sort are fixed size problems, while others depend on T. In order to determine the performance of non-local accesses in the NUMA mode of the PRAM-NUMA architecture, 4-, 16- , and 64-thread versions of aprefix, max, spread, and sum are also used.

		SEQ			PAR		
Name	N	E	P	W	E	$P=W$	Explanation
aprefix	T	N	1	N	1	N	Determine a multiprefix of an array of N integers
fft	64	$N \log N$	1	$N \log N$	1	N^2	Perform a 64-point complex Fourier transform
max	T	N	1	N	1	N	Find the maximum of a table of N integers
mmul	16	N^3	1	N^3	1	N^3	Compute the product of two 16-element matrixes
sort	64	$N \log N$	1	$N \log N$	1	N^2	Sort a table of 64 integers
spread	T	N	1	N	1	N	Spread an integer to all N threads
sum	T	N	1	N	1	N	Compute the sum of an array of N integers

Table 2: Evaluated configurations (c=processor clock cycles). Notations En-p and En-n stand for En in the PRAM mode and En in the NUMA mode, respectively.

	Symbol	E4	E16	E64
Model of computation	M_{tlp}	PRAM-NUMA	PRAM-NUMA	PRAM-NUMA
ILP model PRAM mode	M_{ilp}	Chained VLIW	Chained VLIW	Chained VLIW
ILP model NUMA mode	M_{ilp}	VLIW	VLIW	VLIW
Processors	P	4	16	64
Threads per processor	T_p	512	512	512
Total number of threads	T	2048	8192	32768
Functional units (PRAM mode)	F_p	10	10	10
Functional units (NUMA mode)	F_n	3	3	3
On-chip shared data memory	M_{sd}	2 MB	8 MB	32 MB
On-chip local data memory	M_{ld}	2 MB	8 MB	32 MB
On-chip bank access time	A_b	1 c	1 c	1 c
On-chip bank cycle time	C_b	1 c	1 c	1 c
Length of FIFOs	Q	16	16	16
Step cache associativity	A_c	4	4	4

slowdown is happening while executing in the PRAM mode. The reason why the overhead is higher in E16 than in E64 may be random effects caused by the used memory hashing function, since the differences are small.

The NUMA mode performance was measured by executing the sequential versions of the programs in a single thread in the PRAM mode and in the same program in the NUMA mode with a single NUMA bunch unifying all the threads of a single processor. The results of these simulations as execution time are illustrated in Figure 5b. We see that the NUMA mode indeed provides radically better performance for sequential programs, but is not able to exploit virtual ILP up to degree possible in the PRAM mode. The mean speedups of using NUMA mode are 13200%, 13196%, and 13995% for E4, E16, and E64, respectively. This does not, however, mean that these NUMA bunches can solve these computational problems faster than the ESM architecture or this PRAM-NUMA architecture in the PRAM mode if parallel algorithms are used. Namely, the parallel versions are 1421%, 3111%, and 6889% faster than the best sequential ones for E4, E16, and E64, respectively. The speedup is not linear with respect to the number of processors here, since 3 out of 7 benchmarks are fixed size computational problems. These single-threaded NUMA mode results are identical to the CESM architecture since the PRAM-NUMA architecture is an extension of CESM and in these tests no remote accesses were required in the NUMA mode.

To illustrate seamless configurability between the NUMA and PRAM modes in the PRAM-NUMA architecture, we measured the NUMA mode execution time for the sort algorithm for a bunch with different number of threads ranging from 1 to 512 threads per bunch in the E4 configuration. The results are shown in Figure 5c. We can see linear performance increase as the number of threads per the bunch increases (taking the exponential thread scale into account).

We compared also the NUMA mode performance of PRAM-NUMA CMPs with a single bunch occupying all thread slots for each processor to that of the PRAM mode and single processor (NUMA bunch) performance. This was done by executing 4-, 16- and 64-threaded full NUMA, PRAM and single processor versions of aprefix, max, spread, and sum assuming appropriate blocking, hashing, and single module memory allocations, respectively (see Figure 5d and 5e). We can observe that the PRAM mode versions are 679%, 809%, and 1843% faster than the full NUMA versions and that the full NUMA versions are 291%, 974%, and 1683% faster than the single processor NUMA mode versions for E4, E16, and E64, respectively, showing good NUMA speedups. The buffer-based variant of PRAM-NUMA architecture turned out to be 29.3%, 23.7% and 6.7% slower than dual access version in the full NUMA test for E4, E16, and E64, respectively. Figure 5f shows the sizes of source code files in lines for PRAM, NUMA, and sequential versions serving as a rough measure of complexity of programming. We can observe that the PRAM versions are a bit shorter than sequential versions and the NUMA versions are more than twice the size of the PRAM versions. It should be noted that the length difference of actual algorithms is much higher than indicated by the total length of the source files since all versions contain similar headers and comments and measurement primitives although we used explicit synchronization algorithms for the NUMA mode.

Finally, we estimated the silicon area, power consumption, and maximum clock frequency figures for E4, E16, and E64 targeted to a high-performance 65 nm silicon process, although implementation of full NUMA does not require extra hardware except some extra ports and multiplexers over the CESM architecture. We assumed that each processor core has 1 MB memory that is divided evenly between global shared memory and local memory (see Table 2). The estimations are based on models presented [27], ITRS 2007, and careful counting of architectural elements broken down to gate counts applied e.g. in [11]. The wire delay model gives maximum clock frequency 1.29 GHz for all CMPs assuming minimum width global interconnect wiring. The area and power results are shown in Figure 6.

5 Conclusions

We have shown that integrating full NUMA support to a PRAM implementation architecture can solve the inefficiency problem of current PRAM realizations in the case of low-TLP functionalities at the cost of more difficult NUMA programming. We described the corresponding model of computa-

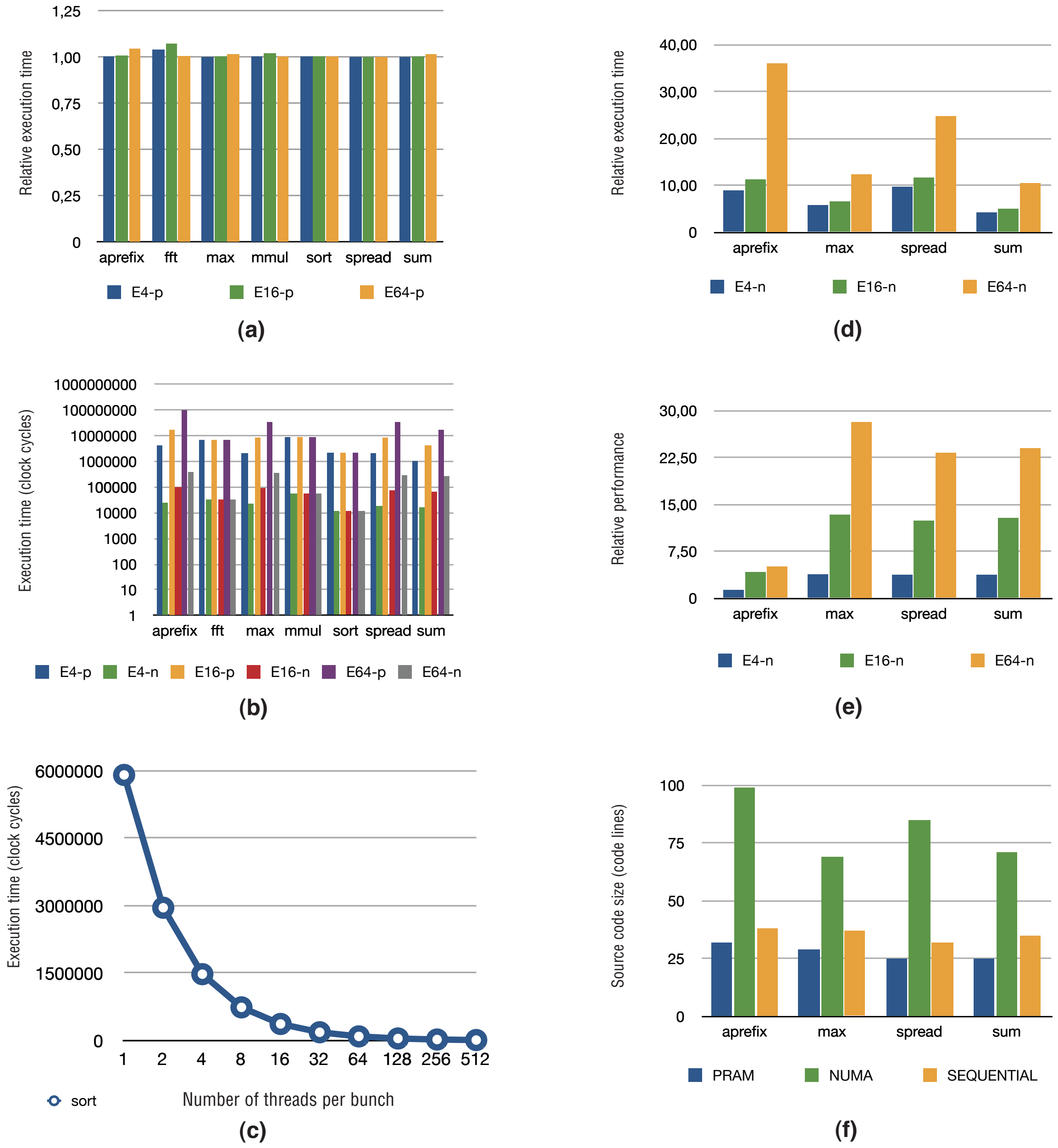


Figure 5: The relative execution time of PRAM-NUMA CMPs compared to ideal PRAMs with similar configuration (PRAM=1.0, shorter is better) (a), the execution time of sequential solutions of the computational problems on a single thread of a single CESM processor core and on a 512 thread NUMA bunch in a single MBTAC processor core (the time scale is logarithmic due to big differences in execution time) (b), execution time of as a function of number of threads in the bunch for E4 PRAM-NUMA configuration (c), the relative execution time of 4, 16, and 64 NUMA bunches with respect to PRAM mode threads (PRAM=1.0, shorter is better) (d), the relative performance time of 4, 16, and 64 NUMA bunches with respect to a single NUMA bunch in a single processor (single processor=1.0, longer is better) (e), source code size of the benchmarks in lines for PRAM, NUMA, and sequential versions (f).

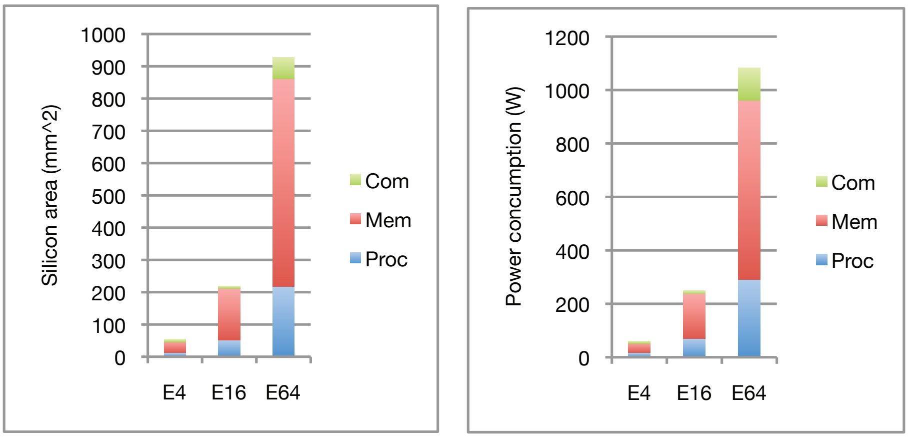


Figure 6: Silicon area and power consumption estimates for E4, E16, and E64 at 1.29 GHz on high-performance 65 nm technology (Com=communication network, Mem=memories, and Proc=processors).

tion, gave a programming example and outlined an architectural implementation for it with a help of our CESM architecture. According to the performance evaluation, the PRAM-NUMA architecture indeed provides good NUMA performance figures in low-TLP situations and extends the NUMA mode to multi-threaded workloads employing non-local memory accesses while not compromising the performance in medium and high-TLP cases. Surprisingly, the architectural implementation of the PRAM-NUMA model does not take virtually any more silicon or power than the CESM architecture making it a promising candidate for future general purpose parallel computers especially if low-TLP legacy code is expected to be executed efficiently along with new PRAM code. In our future work, we aim to realize the proposed hardware on FPGA and continue our studies on the optimal model for general purpose parallel computing and realization of it as a scalable CMP architecture.

6 Acknowledgment

This work was supported by the grants 107177 and 122426 of the Academy of Finland.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. Paul, D. Scheerer, On the Physical Design of PRAMs, Computer Journal 36, 8 (1993), 756-762.

[2] R. Alverson, D. Callahan, D. Cummings, B. Kolblenz, A. Porterfield, B. Smith, The Tera Computer System, Proceedings of the International Conference on Supercomputing, Association for Computing Machinery, New York, 1990, 1-6.

[3] L. Benini and G. De Micheli, Networks on chips: A new SoC paradigm,Computer, 35(1), 2002, pp. 70-78.

[4] M. Forsell, Are Multiport Memories Physically Feasible?, Computer Architecture News 22, 4 (September 1994), 47-54.

[5] M. Forsell, A Scalable High-Performance Computing Solution for Network on Chips, IEEE Micro 22, 5 (September-October 2002), 46-55.

[6] M. Forsell, Using Parallel Slackness for Extracting ILP from Sequential Threads, In the Proceedings of the SSGRR-2003s, International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet, July 28 - August 3, 2003, L ’ Aquila, Italy.

[7] M. Forsell, E – A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs, WSEAS Transactions on Computers 3, 3 (July 2004), 807-812.

[8] M.Forsell, Compiling Thread-Level Parallel Programs with a C-Compiler, In the Proceedings of the IV Jornadas sobre Programacion y Lenguajes (PROLE ’ 04), November 11-12, 2004, Malaga, Spain, 215-226.

[9] M. Forsell, Parallel Application Development Scheme for General Purpose NOCs, In the proceedings of the 2005 ECTI International Conference (ECTI-CON 2005), May 12-13, 2005, Pattaya, Thailand, 819-822.

[10] M. Forsell, Realizing Multioperations for Step Cached MP-SOCs, In the Proceedings of the International Symposium on System-on-Chip 2006 (SOC’ 06), November 14-16, 2006, Tampere, Finland.

[11] M. Forsell and J. Roivainen, Performance, Area and Power Trade-Offs in Mesh-based Emulated Shared Memory MP-SOC Architectures, submitted to Computing Frontiers 2008 conference.

[12] M. Forsell, Configurable Emulated Shared Memory Architecture for general purpose MP-SOCs and NOC regions, In the Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip, May 10-13, 2009, San Diego, USA, 163-172.

[13] M. Forsell, P. Hofstee, A. Jerraya, C. Jesshope, U. Vishkin and J. Träff, HPPC 2009 Panel: Are Many-Core Computer Vendors on Track?, Lecture Notes in Computer Science 6043, (2010), 9-15.

[14] M. Forsell, TOTAL ECLIPSE – An Efficient Architectural Realization of the Parallel Random Access Machine, In Parallel and Distributed Computing Edited by Alberto Ros, IN-TECH, Vienna, 2010, 39-64.

[15] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, Proceedings of 10th ACM STOC, Association for Computing Machinery, New York, 1978, 114-118.

[16] D. Gajski, D. Kuck, D. Lawrie and A. Sameh, CEDAR—A Large Scale Multiprocessor, Proceedings of International Conference on Parallel Processing,1983, 524-529.

[17] R. Hintz and D. Tate, Control data STAR-100 processor design, COMPCON, February 1972, 1-4.

[18] C. Hoare, Communicating Seguential Process, Prentice Hall, New York, 1985.

[19] Research at Intel From a Few Cores to Many: A Tera-scale Computing Research Overview, White Paper, Intel, 2006.

[20] International Technology Roadmap for Semiconductors, Semiconductor Industry Association, 2009; http://www.itrs.net.

[21] J. Jaja, Introduction to Parallel Algorithms, Addison-Wesley, Reading, 1992.

[22] J. Keller, C. Keßler, and J. Träff, Practical PRAM Programming, Wiley, New York, 2001.

[23] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta,J. Hennessy, M. Horowitz, and M. Lam, The Stanford Dash Multiprocessor, IEEE Computer 25, (March 1992), 63-79.

[24] L. Mak. Parallelism always helps. SIAM Journal of Computing 26, 1(February 1997) 153–172.

[25] Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, 1997 (http://www.mpi-forum.org).

[26] D. Naishlos, J. Nuzman, C-W. Tseng, and U. Vishkin, Towards a First Vertical Prototyping of an Extremely Fine-Grained Parallel Programming Approach, In Proc. 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA-01), July 2001.

[27] D. Pamunuwa, L-R. Zheng and H. Tenhunen, Maximizing Throughput Over Parallel Wire Structures in the Deep Submicrometer Regime, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 11, 2 (April 2003), 224-243.

[28] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E.A. Melton, V. A. Norton and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, Proceedings of International Conference on Parallel Processing (1985), 764-771.

[29] A. G. Ranade, S. N. Bhatt, S. L. Johnson, The Fluent Abstract Machine, Technical Report Series BA87-3, Thinking Machines Corporation, Bedford, 1987.

[30] A. Ranade, How to emulate shared memory, Journal of Computer and System Sciences 42, (1991), 307-326.

[31] J. T. Schwarz, Ultracomputers, ACM Transactions on Programming Languages and Systems 2, 4 (1980) 484-521.

[32] R. Swan, S. Fuller and D. Siewiorek, Cm*—A Modular Multiprocessor, In the Proceedings of NCC, 645-655, 1977.

[33] W. Watson, The TI ASC—A highly modular and flexible super computer architecture, Proceedings of the 1972 AFIPS Fall Joint Computer Conference, 221-228.

[34] U. Vishkin, S. Dascal, E. Berkovich and J. Nuzman, Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism (Extended Abstract), In the Proceedings of the SPAA ’ 88, 1998.

[35] U. Vishkin, Towards Realizing a PRAM-On-Chip Vision, Workshop on Highly Parallel Processing on a Chip (HPPC), August 28, 2007, Rennes, France (see http://www.hppc-workshop.org/HPPC07/talks.html).

[36] U. Vishkin, G. Caragea, A and B. Lee, Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform, In Handbook of Parallel Computing —Models, Algorithms and Applications (editors S. Rajasekaran and J. Reif), Chapman and Hall/CRC, Boca Raton, 2008, 5-1—5-60.