International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 4, Number 1, pages 189–206, January 2014

NUMA Computing with Hardware and Software Co-Support on Configurable Emulated Shared Memory Architectures

Martti Forsell
Platform Architectures Team
VTT Technical Research Centre of Finland
Box 1100, FI-90571 Oulu, Finland
Martti.Forsell@VTT.Fi

Erik Hansson, Christoph Kessler
Computer Science Department
Linköping University
S-58183 Linkoping, Sweden
Erik.Hansson@LIU.Se, Christoph.Kessler@LIU.Se

and

Jari-Matti Mäkelä, Ville Leppänen
Department of Information Technology
University of Turku
Joukahaisenkatu 3-5, FI-20014 Turku, Finland
jmjmak@UTU.Fi, Ville.Leppanen@UTU.Fi

Received: July 27, 2013
Revised: October 27, 2013
Accepted: November 29, 2013
Communicated by Akihiro Fujiwara

Abstract

The emulated shared memory (ESM) architectures are good candidates for future general purpose parallel computers due to their ability to provide an easy-to-use explicitly parallel synchronous model of computation to programmers as well as avoid most performance bottlenecks present in current multicore architectures. In order to achieve full performance the applications must, however, have enough thread-level parallelism (TLP). To solve this problem, in our earlier work we have introduced a class of configurable emulated shared memory (CESM) machines that provides a special non-uniform memory access (NUMA) mode for situations where TLP is limited or for direct compatibility for legacy code sequential computing and NUMA mechanism. Unfortunately the earlier proposed CESM architecture does not integrate the different modes of the architecture well together e.g. by leaving the memories for different modes isolated and therefore the programming interface is non-integrated. In this paper we propose a number of hardware and software techniques to support NUMA computing in CESM architectures in a seamless way. The hardware techniques include three different NUMA shared memory access mechanisms and the software ones provide a mechanism to integrate and optimize NUMA computation into the standard parallel random access machine (PRAM) operation of the CESM.

The hardware techniques are evaluated on our REPLICA CESM architecture and compared to an ideal CESM machine making use of the proposed software techniques.

Keywords: parallel computing, models of computation, programming model, shared memory emulation, NUMA, PRAM

1 Introduction

Among the architectural approaches for parallel and multicore computing making use of memory – let it be distributed either on-chip or among a number of chips – there are very few that support simple programmability and performance scalability with respect to sequential computing [21]. This is because most approaches define asynchronous execution of computational threads and do not support efficient hiding of the new kind of memory reference/ intercommunication latency – delay caused by routing the references/messages to their targets and if necessary replies back. This prevents programmers from using simple parallel algorithms with a clear notion of the state of computation and therefore makes programming complex and many parallel algorithms weakly scalable. A notable exception is so called *emulated shared memory* (ESM) machine [23, 18, 2], which provides a synchronous programming model mimicking the *parallel random access machine* (PRAM) model of computation [12] and hides the latency of the distributed memory system by employing the high-throughput computing scheme, i.e. executing other threads while a thread is referring to the memory.

In order to have full performance, an ESM machine needs to have applications with enough *thread-level parallelism* (TLP). This poses a problem related to functionalities with low TLP and a compatibility problem with existing sequential and *non-uniform memory access* (NUMA) programs. In our earlier work we have proposed a *configurable emulated shared memory* (CESM) machine to solve this problem by allowing a number of threads to be bunched together to mimic native NUMA operation so that the overhead introduced by plain ESM architectures can be eliminated [8, 11]. The original PRAM-NUMA model of computation [9, 11] defines separate networks and memory systems for the different modes of the machine, which is impractical from the point of view of writing unified programs making use of both modes. In order to simplify programming, we have proposed unifying the modes by embedding the NUMA system into the PRAM system so that there is no need for dedicated NUMA network nor dedicated NUMA memories [10]. That work, however, left hardware details open and did not provide a clean way to integrate the usage of these two modes into a single program with intercommunication support.

In this paper we propose a number of hardware and software techniques to support the NUMA computing in CESM architectures in a seamless way. The hardware techniques include three different NUMA shared memory access mechanisms and the software ones provide mechanisms to integrate and optimize NUMA computation into the standard PRAM operation of the CESM. The hardware techniques are evaluated on our REPLICA CESM architecture and compared to an ideal CESM machine making use of the proposed software techniques.

The rest of the paper is organized so that in Section 2 we describe the class of CESM architectures in which these techniques are relevant, in Section 3 we propose a number of NUMA realizations with different memory organizations, in Section 4 we consider programming of CESM architectures making use of the realizations of Section 3, realizations are evaluated and compared to each other and also to the ideal CESM machine in Section 5, and finally, in Section 6 we give our conclusions.

2 Configurable emulated shared memory architectures

In order to solve the performance and programmability problems of current parallel/multicore machines, the concept of shared memory emulation has been introduced and a number of academic

[0]This work was funded by the REPLICA project of VTT. This paper is an extended version of an APDCM'13 paper.

architectures has been proposed [14, 2, 24, 10]. In this section we will discuss the idea of shared memory emulation and adding NUMA support for it to support configurability in those architectures.

2.1 Idea of shared memory emulation

The main problems of current architectures are that synchronization (of asynchronous execution) takes hundreds of clock cycles and that cache-based latency hiding scales weakly. These apply especially to current symmetric multiprocessor (SMP) and NUMA architectures. Emerging general purpose graphics processors (GP-GPU) make use the throughput computing scheme to relieve the latency problem.

To solve these problems in an unified way one would need a fast/low-overhead synchronization mechanism and scalable latency hiding/tolerance mechanism. The standard way to solve these problems, known as *shared memory emulation*, is to use wave synchronization to provide lock-step synchronous execution and to employ multithreading with a pipelined memory system along with a machine consisting of P processors (each T_p-threaded) connected to M memory modules via a high-bandwidth network of diameter ϕ_{net}.

The idea of wave synchronization is to separate references belonging to consecutive steps of execution during which each thread of the processor executes a single instruction. This is done by sending a set of synchronization references between the steps. Routing of synchronization references happens in an elastic wave-like manner so that synchronicity is maintained [18, 2].

Latency hiding using excessive multithreading is based on overlapping relatively long latency memory references in a pipelined memory system so that when a thread refers to a memory location, other threads are executed e.g. in an interleaved manner until the reply is received [2]. Hot spots and congestion are minimized by using randomized hashing of memory locations [1].

As these two methods are combined the synchronization cost is dropped so that the overhead of lock-step synchronicity in execution time becomes $1/T_p$ and the latency will be hidden with a high probability assuming $T_p \geq 2\phi_{net}$ [23] like the current single core architectures are emulating the model of sequential computation efficiently with a high probability.

2.2 Adding NUMA support

The utilization fraction U_p of a P-processor ESM system with T_p threads per processor as the function of software parallelism T_b (the number of threads in execution at the current moment of time) is

$$U_p = \frac{min\{T_b, P \times T_p\}}{P \times T_p}$$

We can easily see that if the software parallelism T_b is low U_p gets weak. Most current multicore architectures do not have this problem since they are not using multithreading ($T_p = 1$) for latency hiding but coherent caches to exploit access locality in programs (where available) or just try to tolerate natural latency defined by the distance of memory access making memory access non-uniform [17, 22].

In our earlier work we have introduced the idea of adding the plain NUMA model support for the ESM machine by allowing a set of threads in a processor core to join together as a single *bunch* that is sharing a single register set and executing instructions consecutively even inside a step instead of executing the same instruction for each thread [8]. This is done by reorganizing the thread storaging mechanism, adding indirection to the register set addressing so that threads can use the register storages of other threads (see Figure 1 left), merging the ESM pipeline with the standard NUMA pipeline (see Figure 1 right), and providing non-uniform locality-aware access to memory modules. The obtained class of architectures is called configurable emulated shared memory (CESM). By setting the threads of the processors of a CESM machine to the NUMA mode the utilization fraction becomes the same as for the current architectures and the problem of low U_p with low parallelism is eliminated.

We have also introduced the PRAM-NUMA model of computation capturing the details of this solution in more theoretical way [9]. It consists of T processors grouped as P groups of T_p processors,

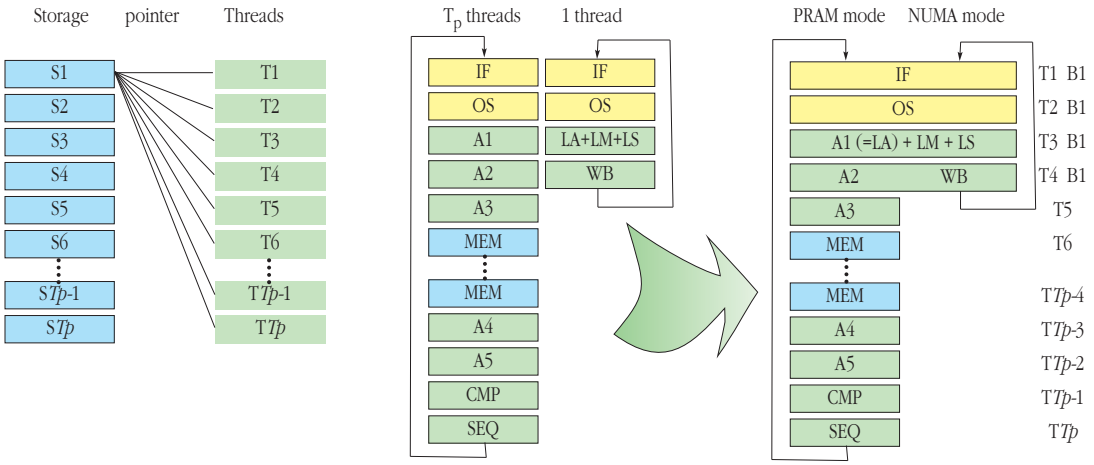

Figure 1: Indirecting the register storages of threads T1...TT_p to use a single register storage S1 in ESM (left) and merging a T_p-threaded ESM pipeline and standard 4-stage NUMA pipeline to obtain CESM architecture (right) with an unified pipeline. $A_n = ALU_n$, B1=single bunch, CMP=compare unit stage, IF=instruction fetch stage, MEM=memory unit stage, OS=operand select stage, Tn=thread n, WB=write back stage.

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 2).

3 NUMA realization alternatives

The original PRAM-NUMA model of computation [9] defines separate networks and memory systems for the different modes of the machine, which is impractical from the point of view of writing unified programs making use of both modes. In order to simplify hardware implementation and programming, we have proposed unifying the modes by embedding the NUMA system into the PRAM system so that there is no need for a dedicated NUMA network while dedicated NUMA memories are retained as local memory modules [10]. This solution does, however, not define a simple way to unify memory allocation and execution control mechanisms in the different modes of the processor and leaves low-level hardware organization open. Our new idea is to use the PRAM shared memory system to implement storage also for shared NUMA variables while the private PRAM variables are moved to the local memories to reduce the load of the network. Technically there are three obvious ways to implement shared memory load accesses for the NUMA mode – freeze processor, freeze bunch, and load with explicit receive. In the following we describe the main principles of these alternatives.

- *Freeze processor* (FP) The whole processor core freezes until the reply arrives. As a consequence, the rest of the threads let them be PRAM threads or other NUMA bunches on the same core will also be halted. This resembles adding wait states until the memory reply is received in a standard pipelined processor.

- *Freeze bunch* (FB) The currently executed bunch freezes but the rest of the threads and bunches continue execution. This is implemented by re-executing the tail of the load instruction during the execution slots of the bunch until the reply is received from the memory system. This resembles freezing the current instruction and all the dependent instructions in a standard superscalar processor.

- *Load with explicit receive* (LER) The original load instructions are divided into two new memory instructions – load and receive. The new load instruction sends the shared memory ref-

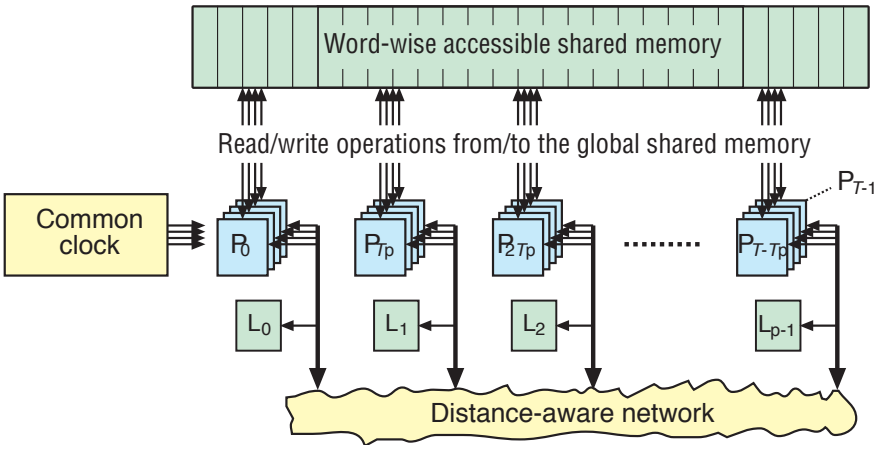

Figure 2: The PRAM-NUMA model of computation. The word-wise accessible shared memory and links to it are related to the PRAM mode while local memories and distance-aware network is related to the NUMA mode. (P=processor, L=local memory, T_p=number of processors per group, T=total number of threads.)

erence on its way to the memory system. The reply is fetched by the receive instruction. If the reply has not arrived at the time of executing the receive, the current bunch freezes until the reply is received, just like in the freeze bunch alternative. With a help of explicit receive it is possible schedule the load and receive separately or even to overlap multiple shared load operations unlike in the other alternatives while the shared store operations are automatically overlapped in all alternatives. A practical upper bound for a number of overlapping references is the number of threads per the NUMA bunch divided by two (assuming both send and receive instructions are executed in the same functional unit) and the upper bound for the latency between a load and corresponding receive is a single step. The former is potentially bounded also by the number of available registers.

Since these solutions are using the PRAM memory system, the memory wait logic is taking care that the replies are received within the current step of execution. In the case of contention in the network, it can happen that the thread slot initiating the load is trying to leave the memory waiting stages causing the whole processor core to freeze until the reply is received. The synchronization wave is ultimately guaranteeing that all the references made during a step of execution have completed before the references of the next step are applied.

4 Programming considerations

In our earlier work we have proposed a parallel application development scheme for ESM machines consisting of a strong model of computation, a C-like TLP programming language *e* [5] or *REPLICA* [20], ILP-TLP optimization algorithm, and application development flow [7]. The scheme allows programmers to write applications with a help of supporting theory of parallel algorithms [13, 14]. Orchestration of threads is arranged in standard PRAM-style by providing thread identifier, number of threads, and synchronous and asynchronous variants of C-style constructs. Since the NUMA mode of the CESM machine does not execute threads synchronously nor provide access distance independent latency hiding like the PRAM model, the scheme does not directly support NUMA programming of CESM machines. In the following we describe how NUMA mode programming can be closely unified to the ESM programming scheme, give some programming examples, explain what kind of support the programming scheme requires, as well as consider compilation and optimization issues for PRAM-NUMA.

4.1 Orchestrating parallelism in the NUMA mode

The CESM machine boots up in the PRAM mode, supports switching groups of threads to the NUMA mode and back to PRAM mode, and allows for multiple simultaneous PRAM and NUMA executions. This suggests that the control of execution mode could be implemented as standard blocks at the programming language level. For this we propose using two control constructs that switch the execution to NUMA mode and back. The first construct is

numa(s)

that bunches all the threads of current thread group in all participating processor into NUMA bunches and executes statement s. After that the mode is switched back to PRAM. Sequential portions of computation can be supported with a construct

sequential(s)

that bunches all the threads of current thread group in the participating processor having the lowest Id into a single NUMA bunch and executes statement s while the other threads are waiting. This mechanism would just allow switching to NUMA mode and back to PRAM mode but prevents more complex schemes including nesting modes (e.g. switching to NUMA mode, splitting to subgroups to execute them in the PRAM mode). Since there can be multiple PRAM thread groups executing in parallel, it would be possible to set up multiple NUMA bunches that run in parallel with the remaining PRAM groups if any.

In order to provide similar programming interfaces for the both modes, we propose using the PRAM thread orchestration mechanisms also in the NUMA mode where applicable. Controlling individual bunches could e.g. be done with the same *thread identifier* and *number of threads* variables and synchronization could happen with the same barrier construct. Since the cost of synchronization is high in the NUMA mode, we do provide only the standard asynchronous control constructs for NUMA and leave synchronous ones limited to the PRAM mode only.

In the original ESM development scheme passing data between the different parts of the program happens with standard programming language mechanisms, e.g. global variables, stack frames and pointers. To support easy data exchange between the modes we propose reusing the REPLICA memory model consisting of unified shared memory and private memory making use of local memory modules also for the NUMA mode (see Figure 3 for the resulting memory organization map as a portion of code executed in the PRAM mode calls a NUMA mode portion in a T-threaded CESM). This means that exchange between NUMA bunches happens via the shared memory system only and exchange between the modes happens via the shared memory and bunch leaders private subspace. Then a programmer can directly use the shared variables defined in the PRAM mode but without the notion of locality and refer also to the single set of private variables that belong to the bunch leader. Switching back to the PRAM mode passes the modifications done to shared variables and bunch leader's private variables. All the variables declared inside the NUMA portion are accessible in the PRAM mode before or after the portion.

4.2 Programming example

Consider the computational problem of determining the prefix sums of an array of N integers. Let us denote that array with symbol *source*. The standard (sub-optimal) way to compute the prefix sums in parallel is the known as the recursive doubling logarithmic algorithm, where each element of the array gets added by the element 2^i positions left on iteration i, $i = 0...\log(N)-1$ (see Figure 4). Since the number of processors P in the NUMA mode can be smaller than N, one needs to compute each iteration by an N/P-iteration inner loops. The algorithm requires that data is accessed truly parallel manner in each outer loop iteration meaning that a temporary array *temp* has to be used to store intermediate results during inner loop execution and the obtained data must be copied back to the original array. Figure 5 shows the implementation of this algorithm in the e language.

The data is initialized in the PRAM mode and the logarithmic prefix computation is done inside the *numa*(s) construct. The execution time of this algorithm is $O(N\log(N)/P)$ assuming $P<N$ otherwise it is $O(N\log(N))$.

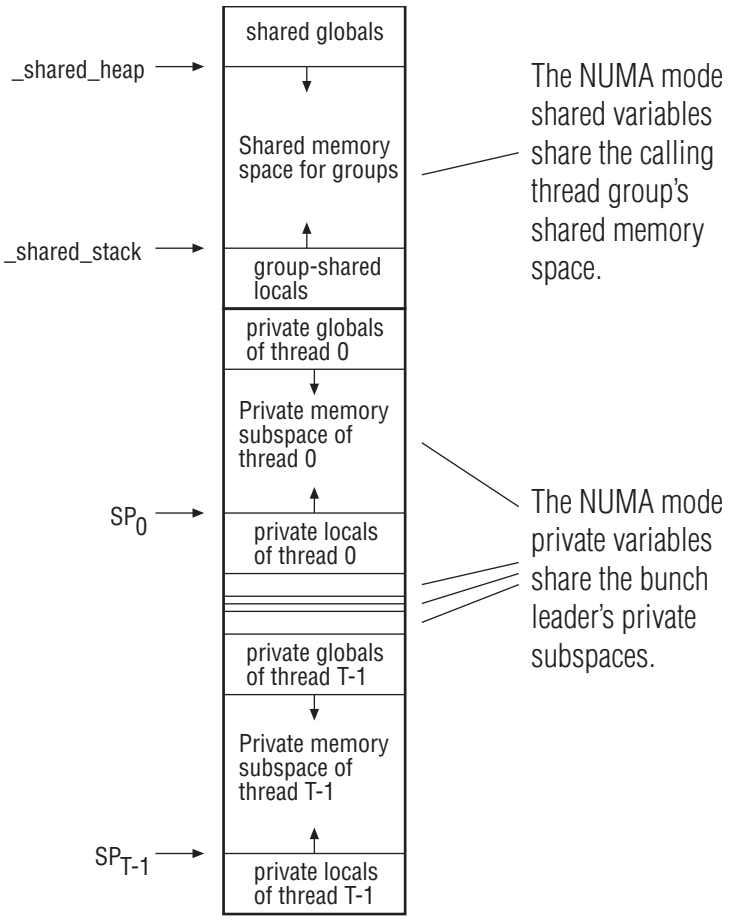

Figure 3: The memory organization of a T-threaded CESM machine making use of the NUMA mode.

4.3 Supporting the NUMA mode operation

The above NUMA programming scheme requires implementation of the numa construct and barrier synchronization routine on a top of the ESM programming scheme.

In order to switch from the PRAM mode to the NUMA mode, the current state of the PRAM thread group is stored into block-local variables and restored from these variables as the execution of NUMA block ends. There is also need to allocate room for a synchronization variable from the shared stack and compute the new *_thread_id* and *_number_of_threads* variables reflecting the bunch id and number of bunches in the NUMA block. Just before entering to the statement s the PRAM group is switched to the NUMA mode. After the execution of the statement, the bunches are deformed and the state of the PRAM group is restored from the local variables. The code for the numa construct switching CESM to the NUMA mode, executing the statement s and switching back to the PRAM mode is shown in Figure 6.

Synchronizing the bunches in the NUMA mode is done with a NUMA-specific library routine *_RTL_SYNCHRONIZE_NUMA* that first checks if the previous synchronization is still going on and waits for that if necessary. Then the synchronization variable is decremented by one for each arriving bunch. To manage the situation in which two to or more bunches do this simultaneously we use REPLICA processor's fast multiprefix operations that work partially also in the NUMA mode. The arrived bunches wait until the synchronization variable reaches zero meaning that all the bunches have arrived and continue after that. In order to manage asynchrony of bunches in exiting the routine and reinitializing the synchronization variable the threads decrement the synchronization

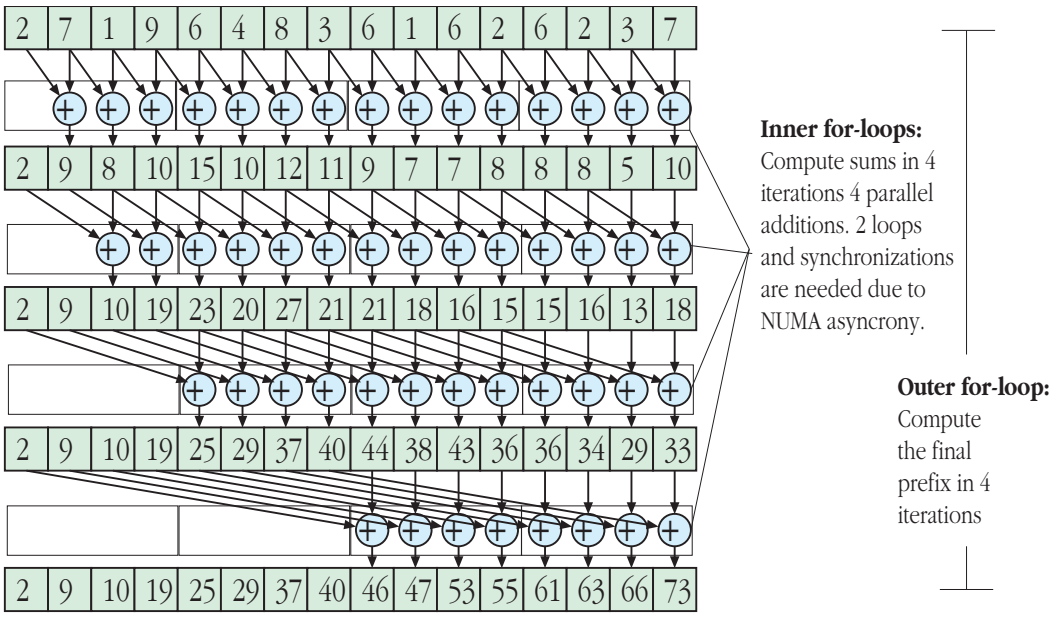

Figure 4: The logarithmic prefix algorithm in the case of N=16, P=4.

variable once more and the last bunch leaving the routine does the initialization. The REPLICA assembler code for NUMA-specific barrier synchronization is shown in Figure 7.

4.4 Compiler support

We support the REPLICA hardware with a tailored compiler tool chain. The code generation part is based on the LLVM compiler framework [16]. Earlier versions of our REPLICA compiler only supported to generate code for PRAM mode [19, 15]. For the previous architectures, [8, 11], the so called e-compiler was developed and used [6, 4]. To a large extent it can be used for the current REPLICA architecture as well, and we use it for some of the evaluation benchmarks in Section 5, since we try to support both compilers under a transition period.

In this paper we present the first native REPLICA compiler version that supports both PRAM and NUMA compilation. The earlier PRAM compiler, [15], can generate code for different configurations of the REPLICA processor, e.g. for different numbers of functional units (ALUs, MUs etc.) placed in a chain etc. but has no NUMA support.

The first step in the code generation phase, both for PRAM and NUMA mode, is to generate code for a minimal configuration. In the PRAM case it is then translated and optimized for larger configurations. For NUMA mode we still use this initial minimal configuration as a starting point, but instead of optimizing for a more functional units etc., our scheduling algorithm enforces the stricter scheduling constraints such as no chaining of functional units. One example is that a compare instruction and a depending branch instruction can not be scheduled in the same VLIW instruction. Another is that, after a branch instruction two NOPs are needed. These two examples do not apply in PRAM mode.

To change from PRAM mode to NUMA mode a special instruction, `JOIN`, is used. To go back to PRAM mode `SPLIT` is used. The compiler detects the basic blocks that are encapsulated between the `JOIN` and `SPLIT` instructions and marks them to be NUMA basic blocks. If a `JOIN` or `SPLIT` instruction is found inside a basic block, the block is divided into two new ones.

For the case of *load with explicit receive* (LER), the REPLICA compiler can in NUMA mode translate a load instruction to a new load instruction together with a receive instruction. This desired behavior can be specified with a special compiler flag. It is important to mention that the LER is only for shared memory while private memory is handled with traditional load instructions.

```
#define size N
int source_[size]; // Allocate data array from the shared memory
int temp_[size]; // Allocate space for a shared temporary array
int main()
{
    int i;
    int j;
    for (i=_thread_id; i<size; i+=_number_of_threads)
        source_[i] = i; // Initialize source_ in parallel in the PRAM mode
    numa(
        for (i=1; i<size; i<<=1)
        {
            for (j=_thread_id; j<size; j+=_number_of_threads)
                if (j-i>=0) temp_[j]=source_[j] + source_[j-i];
            synchronize;
            for (j=_thread_id; j<size; j+=_number_of_threads)
                source_[j]=temp_[j];
            synchronize;
        }
        synchronize;
    );
}
```

Figure 5: Non-optimized NUMA prefix computation in the e language in which symbol "_" at the end of the variable name declares it shared.

To distinguish between loading shared and private memory the compiler recursively analyzes if an address computation uses shared memory or not inside a NUMA block. Of course, there are rare cases where it is not possible to do it statically, then it is up to the programmer to manually adapt the code.

4.5 NUMA optimizations

In order to provide decent performance in the NUMA mode there exists a number of optimizations familiar from all NUMA systems, e.g. synchronization minimization, locality maximization, as well as some that are specific to CESM architectures, e.g. scheduling of receive instructions and overlapping two or more shared memory loads.

Since synchronizations (e.g. with the barrier algorithm) take a long time to execute in the NUMA mode compared to a single instruction execution, performance increases if the computation is reorganized so that the number of synchronizations is minimized. A typical way to do this is to divide the data at hand into blocks and process blocks inside processors so that synchronizations are not used. Unfortunately this is not simple and not even always possible. For the prefix example described above there exists a way to do synchronization minimization by blocking although there are a relatively large number of synchronizations in the unoptimized algorithm. For this we divide the shared array to P blocks, compute prefix sums of blocks with a sequential algorithm in P parallel bunches, compute the prefix of the block sums in the bunch 0 with a sequential algorithm, and offset the blocks with the obtained prefixes of the blocks with a sequential algorithm executed in parallel in all bunches (see Figure 8).

Locality optimization for the NUMA mode means, in the prefix sums example, dividing the shared data array *source* into P private (local) parts that are processed in the same way. This maximizes the locality since the most data (except computing the prefix sum of the block sums and distributing the results of it) is then processed locally and the only place in which shared access is needed is computing prefixes of block sums. The resulting program is shown in Figure 9.

```
#define numa( s ) {
    int _old_thread_id = _thread_id;
    int _old_number_of_threads = _number_of_threads;
    int _old_group_id = _group_id;
    int _old_shared_stack = _shared_stack;
    int _processor_id = _thread_id/_threads_per_processor;
    shared_stack -= 4;
    _group_id = _shared_stack;
    _update_region_numa;
    join_marker;
    {
        s
    }
    split_instruction;
    write_back(32,_private_space_start);
    _thread_id = _old_thread_id;
    _number_of_threads = _old_number_of_threads;
    _group_id = _old_group_id;
    _shared_stack = _old_shared_stack;
}
```

Figure 6: Implementation of the **numa** construct. The *_update_region_numa* routine computes the new values for *_thread_id* and *_number_of_threads* variables and switches execution to the NUMA mode. The *split_instruction* routine switches execution back to the PRAM mode.

For suitable algorithms executed in LER-enabled CESM it is possible to schedule the receive instructions so that their distance to corresponding load instructions is maximized or made long enough for partially hiding the latencies of loads. The result is even better if two or more loads are overlapped e.g. with software pipelining. Figure 10 shows the main loop of the block benchmark in the unoptimized case, after applying maximization of distance between corresponding load and receive instructions, and after overlapping consecutive shared loads.

For more detailed information on quantitative performance effects given by these optimizations, see the evaluation in the next section.

5 Evaluation

In order to evaluate the performance, difficulty of programming, and complexity of the hardware and software techniques proposed in Sections 3 and 4, we applied them to the REPLICA *chip multiprocessor* (CMP) framework being developed at VTT.

5.1 Performance

We measured the execution time of 6 micro benchmarks (see Table 1) in 12 configurations of REPLICA making use of the techniques, the standard ESM (the PRAM mode of REPLICA), CESM (REPLICA making use of the fastest available mode) and corresponding PRAM assuming idealized memory system (see Table 2). Note that the benchmarks *block*, *prefix* and *rand* are variable sized (problem size=T_{total}) while *barrier*, *numa* and *edge* are fixed sized.

The benchmarks were written in the e language, compiled with the e compiler *ec* [4] applying options -ilp, -o2, synchronization minimization and simulated on our CMP tool *IPSMSim* [3]. To determine the effect local memory versus shared memory the NUMA mode tests were done with and without locality optimization maximizing the locality of memory references by using local memory where possible.

```
_RTL_SYNCHRONIZE_NUMA
        ADD0    R29,O0    OP0     -4          WB1     A0                  ; Save registers
        ADD0    R29,O0    OP0     -8          ST0     R2,R1     WB1       A0
        ADD0    O0,R32    OP0     __group_id  ST0     R3,R1     WB1       A0
        LD0     R1        WB1     M0                                      ; R1 <-- pointer to shared _group_id
L10_RTL20
        LD0     R1        WB2     M0                                      ; R2 <- The synchronization variable _group_id
        SLE0    R2,O0     OP0     0           ; If the synchronization variable <= 0 then the previous barrier is still goin on
        NOP0
        NOP0
        BNEZ    O1        OP1     L10_RTL20                               ; Wait until it is reinitialized
        MSUB0   O0,R1     OP0     1           ; Decrement the synchronization variable of the group
L20_RTL20
        LD0     R1        WB2     M0          ; R2 <-- The synchronization variable _group_id_
        SGT0    R2,O0     OP0     0           ; Test if all threads of the group have arrived
        NOP0
        NOP0
        BNEZ    O1        OP1     L20_RTL20   ; If not, continue waiting
        MPSUB0  O0,R1     OP0     1           WB2     M0          ; Decrement the synch variable, R2 <- previous value
        ADD0    O0,R32    OP0     __number_of_threads             WB3     A0          ; R3 <-- number of threads
        LD0     R3        WB3     M0
        ADD0    R2,R3     WB2     A0          ; R2 <-- number of threads + previous value of the synchronization variable
        SNE0    R2,O0     OP0     1           ; If the sum <> 1 then do not reinitialize the synchronization variable
        NOP0
        NOP0
        BNEZ    O1        OP1     L30_RTL20
        ST0     R3,R1                         ; Reinitialize the synchronization var
L30_RTL20
        ADD0    R29,O0    OP0     -4          WB1     A0                  ; Restore registers
        ADD0    R29,O0    OP0     -8          LD0     R1        WB1       A0        WB2     M0
        NOP0
        NOP0
        LD0     R1        WB3     M0          JMP     R31                 ; Return
```

Figure 7: The NUMA-specific barrier synchronization routine in REPLICA assembler.

The results of measurements are shown in Figures 11 – 12. From the results we can make the following observations:

- The LER alternative is the fastest shared memory NUMA technique by a small margin while FP is the slowest. The difference is significant especially in locality optimized prefix and rand benchmarks This is because freezing the whole processor unbalances the timing of the synchronization wave leading to relatively long delays.
- As expected the PRAM mode is often much faster than the NUMA mode but in the *barrier* and strictly sequential *rand* benchmarks all NUMA executions are much faster than the PRAM mode. This is because the ESM can not execute sequential code efficiently and because the number of threads in the PRAM mode is much higher than in the NUMA mode.
- CESM can benefit from the NUMA mode but the cost of switching the machine to the NUMA mode for fast computation and back is substantial ruling fine-grained NUMA exploitation impractical, see e.g. results for *numa*.
- Locality optimizations speed up execution. If there is enough computation per element, like in the *edge* benchmark, this applies even if data is originally in the shared memory. In that case data needs to be divided into local blocks, processed locally and copied back to the shared memory. Sometimes moving data between the memories can neglect the speedup as seen in the rand benchmark executed with the FP alternative.
- The LER alternative provides the programmer with an option to overlap also load operations speeding up operation significantly compared to non-overlapped operation. This opens up interesting optimization possibilities for programmers and compilers.

```
#define size N
volatile int source_[size]; // Allocate from the shared memory
int main()
{ int i, blocksize, start, stop, prev;
    numa(
        blocksize=size/procs;
        start = _thread_id * thrds;
        stop = start + blocksize - 1;
        for (i=start; i<=stop; i++) // Initialize blocks in parallel
            source_[i]=i; // with a sequential algorithm
        synchronize;
        for (i=start+1; i<=stop; i++) // Determine prefixes of
            source_[i]+=source_[i-1]; // blocks in parallel
        synchronize;
        if (_thread_id==0) // Prefix for block sums sequentially
        { for (i=start+thrds+thrds-1; i<procs*thrds; i+=thrds)
            source_[i]+=source_[i-thrds]; }
        synchronize;
        prev = start - 1;
        if ( prev>=0 ) // Add results of prefix sum
        { for (i=start; i<stop; i++) // of block sums to blocks
            source_[i]+=source_[prev]; }
        synchronize;
    );
}
```

Figure 8: Synchronization optimized NUMA prefix computation.

In order to figure out the potential performance of the LER-specific software pipelining optimizations, we applied them to the block benchmark. We measured the execution time of the shared memory block in the baseline configuration (BASE), where no optimization is done but the receive instruction is placed just after each shared load instruction, after maximizing the distance between load and receive instructions but not overlapping two or more loads (MAX-DIST), after overlapping each shared memory load with the next one and then maximizing the distance between load and corresponding receive (OVERLAP-1). For comparison purposes we measured the execution time of local memory versions of the block benchmark and the baseline version assuming ideal shared memory. The programs were compiled with the REPLICA compiler and the software pipelining optimizations (MAX-DIST and OVERLAP-1) were done by hand. The results are shown in Figure 13.

We can do the following observations from these results:

- For this simple but very widely used block copy functionality, maximizing the distance between load and corresponding receive increases the performance by 46%-60% and overlapping consecutive loads increases the performance by 89%-124%.

- Overlapping the consecutive loads drops the execution times very close to that of the localized algorithm and ideal shared memory emphasizing the potential of this optimization.

The cost of applying MAX-DIST and OVERLAP-1 techniques was two and four extra registers, respectively. This indicates that overlapping a high number of shared memory loads is not possible without special hardware support.

```
#define size N
volatile int final_[size]; // Allocate from the shared memory
volatile int sums_[procs]; // Allocate from the shared memory
int main()
{ int i, blocksize, start, stop, prev;
    numa(
        int source[thrds]; // Allocate from the local memory
        int offset;
        blocksize=size/procs; // Divide into blocks
        start = _thread_id * thrds;
        stop = start + blocksize - 1;
        for (i=0; i<blocksize; i++) // Initialize blocks in parallel
            source[i]=i+start; // with a sequential algorithm
        synchronize;
        for (i=1; i<blocksize; i++) // Determine prefixes of blocks
            source[i]+=source[i-1]; // with a sequential algorithm
        sums_[_thread_id]=source[blocksize-1];
        synchronize;
        if (_thread_id==0) // Prefix for block
            for (i=1; i<procs; i++) // sums sequentially
                sums_[i]+=sums_[i-1]; // in a single processor
        synchronize;
        if (_thread_id>0)
        { offset=sums_[_thread_id-1];
            for (i=1; i<blocksize; i++) // Add results of prefix
                source[i]+=offset; } // sum of block sums to blocks
        synchronize;
        for (i=0; i<blocksize; i++)
            final_[i+start]=source[i];
    );
}
```

Figure 9: Locality and synchronization optimized NUMA prefix computation.

5.2 Code size and programmability

In order to roughly characterize the difficulty of programming, we determined the size of code of all benchmarks. The results are shown in Figure 14. We can make the following observations:

- The size of the code is higher for the NUMA execution than it is for the PRAM execution. The difference is biggest in applications making use of frequent exchange of data, e.g. *prefix*, generating a lot of synchronizations in asynchronous NUMA execution. The synchronization optimization increases this difference due to application of the blocking technique.

- Optimizing the code with locality optimization increases the code size in all benchmarks except for barrier and numa that do not make use of user specified statements or contain inter-thread dependencies.

6 Conclusions

We have proposed a number of hardware and software techniques to support NUMA computing in CESM architectures in a seamless way. The hardware techniques include different NUMA shared memory access mechanisms and the software ones provide a way to integrate NUMA computation

NON-OPTIMIZED-VERSION

```
_BB5_2
    OP0 _source_    WB8 O0
    OP0 2           SHL0 R1,O0      WB7 A0
    OP0 _target_    WB9 O0
    ADD0 R8,R7      WB8 A0
    ADD0 R2,R1      WB1 A0
    ADD0 R9,R7      WB7 A0
    LD0 R8          WB8 M0
    WB8 M0          REC0 R8
    ST0 R8,R7
    SLT R1,R6
    OP0 _BB5_2      BNEZ O0
    NOP0
    NOP0
```

DISTANCE-MAXIMIZED-VERSION

```
; Prefix
    OP0 _source_    WB8 O0
    OP0 2           SHL0 R1,O0      WB7 A0
    OP0 _target_    WB9 O0
    ADD0 R8,R7      WB8 A0
    ADD0 R2,R1      WB1 A0
    ADD0 R9,R7      WB7 A0
    LD0 R8          WB10 M0         WB11 R7  ; First load
    SLT R1,R6
; Main body
_BB5_2
    OP0 _source_    WB8 O0
    OP0 2           SHL0 R1,O0      WB7 A0
    OP0 _target_    WB9 O0
    ADD0 R8,R7      WB8 A0
    ADD0 R2,R1      WB1 A0
    ADD0 R9,R7      WB7 A0
    WB10 M0         REC0 R10  ; From the previous iteration
    ST0 R10,R11               ; From the previous iteration
    LD0 R8          WB10 M0         WB11 R7
    SLT R1,R6
    OP0 _BB5_2      BNEZ O0
    NOP0
    NOP0
; Postfix
    WB10 M0         REC0 R10        ; Final receive
    ST0 R10,R11                     ; Final store
```

OVERLAPPED-VERSION

```
; Prefix 1
    OP0 _source_    WB8 O0
    OP0 2           SHL0 R1,O0      WB7 A0
    OP0 _target_    WB9 O0
    ADD0 R8,R7      WB8 A0
    ADD0 R2,R1      WB1 A0
    ADD0 R9,R7      WB7 A0
    LD0 R8          WB10 M0  WB11 R7    ; First load odd
    SLT R1,R6
; Prefix 2
    OP0 _source_    WB8 O0
    OP0 2           SHL0 R1,O0      WB7 A0
    OP0 _target_    WB9 O0
    ADD0 R8,R7      WB8 A0
    ADD0 R2,R1      WB1 A0
    ADD0 R9,R7      WB7 A0   WB12  R10  WB13  R11
    LD0 R8          WB10 M0      WB11 R7 ; First load even
    SLT R1,R6
; Main body
_BB5_2
    OP0 _source_    WB8 O0
    OP0 2           SHL0 R1,O0      WB7 A0
    OP0 _target_    WB9 O0
    ADD0 R8,R7      WB8 A0
    ADD0 R2,R1      WB1 A0
    ADD0 R9,R7      WB7 A0
    WB12 M0         REC0 R12      ; From the previous iteration
    ST0 R12,R13 WB12 R10 WB13 R11 ; From the previous iteration
    LD0 R8          WB10 M0         WB11 R7
    SLT R1,R6
    OP0 _BB5_2      BNEZ O0
    NOP0
    NOP0
; Postfix 1
    WB12 M0         REC0 R12        ; Final receive odd
    ST0 R12,R13                     ; Final store odd
; Postfix 2
    WB10 M0         REC0 R10        ; Final receive even
    ST0 R10,R11                     ; Final store even
```

Figure 10: The main loop of the block benchmark without optimizations, after maximizing distance between receives and corresponding loads, and after overlapping consecutive loads.

into the standard PRAM operation of CESM and to optimize NUMA mode performance with standard synchronization and locality optimizations. According to the evaluation making use of our REPLICA CMP framework, the proposed solutions can be used to provide relatively unified programming scheme for the CESM architecture making use of both the PRAM and NUMA modes. As expected the PRAM mode is faster in most cases but there are a few clear exceptions, e.g. strictly sequential code in which NUMA performs better. Since the number of threads in the NUMA mode is much smaller than in the PRAM mode the barrier synchronizations in the NUMA mode are much faster. In order to achieve good NUMA performance applying synchronization and locality optimizations are crucial especially if the problem requires frequent synchronization or involves data exchange between the computational threads. From the evaluated hardware techniques the one splitting shared loads to schedulable send and receive instructions (the LER technique) provides the best performance especially if overlapping of consecutive loads, e.g. with software pipelining, is applied. This does however not mean that the NUMA mode would be typically better for algorithms requiring tightly synchronous execution since the implied wave synchronization of the PRAM mode

Benchmark	Description
block	Move a block of T_{total} integers in memory. Measures the overall throughput of the memory system.
barrier	Synchronize the threads in the current group of threads (dependent of the mode). Measures the latency of synchronization.
numa	Switch the current thread group from the PRAM mode to NUMA mode and back. Measures the latency of switching between the modes.
edge	Detect edges of an RGB image of 640x30 pixels. For locality optimization divide the shared array evenly to P blocks, move them to local memory for processing and copy the results back to the original array. Measures the the performance for stencil-access patterns.
prefix	Calculate the prefix of T_{total} integers. Measures performance for reductions.
rand	Calculate the sequence of T_{total} random numbers using the linear congruential technique. Measures the performance of sequential computation.

Table 1: Benchmarks used in the evaluation.

	Symbol	FP-P	FB-P	LER-P	ESM-P	CESM-P	PRAM-P
#processors	P_{total}	P	P	P	P	P	P
#threads	T_{total}	P	P	P	PT_p	PT_p	PT_p
#registers/thread	R_t	R	R	R	R	R	R
#FUs	F_{total}	3	3	3	F	F	F
FU organization	F_{org}	parallel	parallel	parallel	chained	chained	chained
Memory scheme	M_{scheme}	NUMA	NUMA	NUMA	PRAM	PRAM-NUMA	ideal PRAM
Network diameter	ϕ_{net}	$2P^{0.5}$	$2P^{0.5}$	$2P^{0.5}$	$2P^{0.5}$	$2P^{0.5}$	1

Table 2: Configurations used in the evaluation (P=number of processors 4, 16 in the evaluation).

eliminates the most barriers and multithreading provides latency hiding that is not available in the NUMA mode. The code size for the NUMA mode is higher than that for the PRAM mode. The difference is most substantial for benchmarks making use of frequent exchange of data between the computational threads. This points out that while NUMA mode can solve a number of performance problems related to workloads with low parallelism, this happens with the cost of programmability.

Our future work includes extending the current REPLICA language and compiler tool-chain with all NUMA-related constructs, investigating possible combined PRAM-NUMA algorithms and optimization possibilities for NUMA mode execution such as loop unrolling/software pipelining and load instruction overlapping/scheduling.

References

[1] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM Journal on Computing*, 23(4):738 – 761, August 1994.

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

[3] M. Forsell. Advanced simulation environment for shared memory network-on-chips. In *Proceedings of the 20th IEEE NORCHIP Conference*, pages 31–36, Copenhagen, Denmark, November 2002.

[4] M. Forsell. Compiling thread-level parallel programs with a C-compiler. In *Proceedings of the IV Jornadas sobre Programacion y Lenguajes*, pages 215–226, Malaga, Spain, November 2004.

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

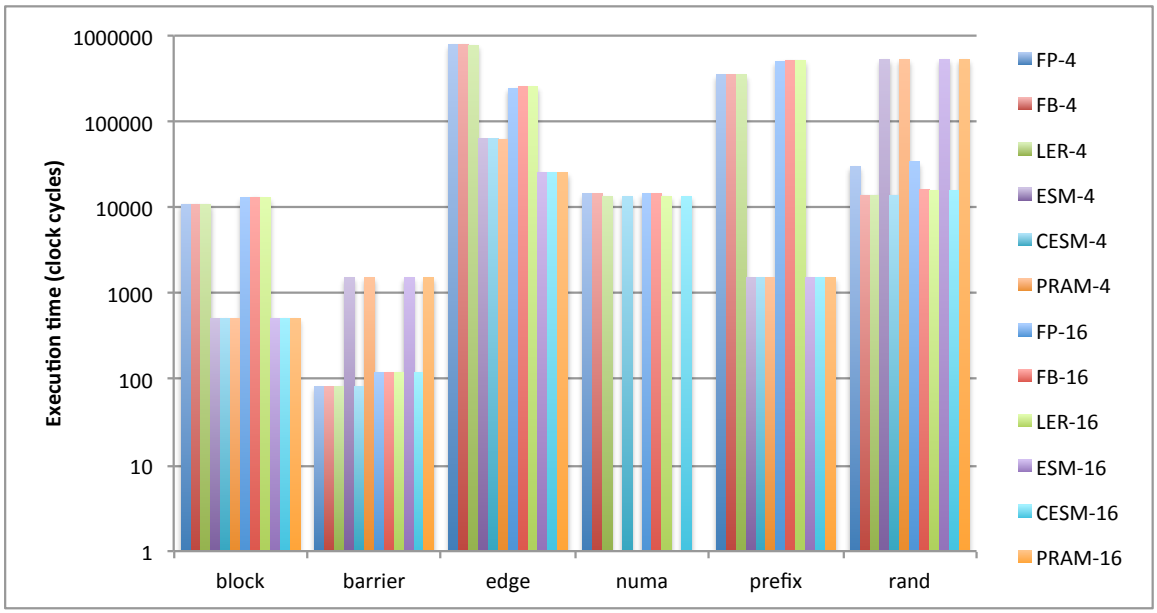

Figure 11: Execution time without NUMA optimizations.

[6] M. Forsell. Ec - a compiler for the e-language. In *Proceedings of the 2004 International Symposium on System-on-Chip*, pages 157–160, 2004.

[7] M. Forsell. Parallel Application Development Scheme for General Purpose NOCs. In *Proceedings of the 2005 ECTI International Conference (ECTI-CON), Pattaya, Thailand*, pages 819–822, 2005.

[8] M. Forsell. Configurable Emulated Shared Memory Architecture for General Purpose MP-SoCs and NoC Regions. In *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip, May 10-13, 2009, San Diego, USA*, pages 163–172, 2009.

[9] M. Forsell. A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads. In *Proceedings of the 12th Workshop on Advances in Parallel and distributed Computational Models (in conjunction with the 24th IEEE International Parallel and Distributed Processing Symposium, IPDPS'10), April 19, 2010, Atlanta, USA*, pages 1–8, 2010.

[10] M. Forsell. *TOTAL ECLIPSE – An Efficient Architectural Realization of the Parallel Random Access Machine*, pages 39–64. IN-TECH, Vienna, 2010. Editor: Alberto Ros.

[11] M. Forsell. A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads. *International Journal of Networking and Computing*, 1(1):21–35, 2011.

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

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

[14] J. Keller, C. Kessler, and J. Träff. *Practical PRAM Programming*. Wiley, New York, 2001.

[15] Martin Kessler, Erik Hansson, Daniel Åkesson, and Christoph Kessler. Exploiting instruction level parallelism for REPLICA - a configurable VLIW architecture with chained functional units. In *Proceedings of PDPTA'12: Volume II*, pages 275–281, 2012.

[16] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

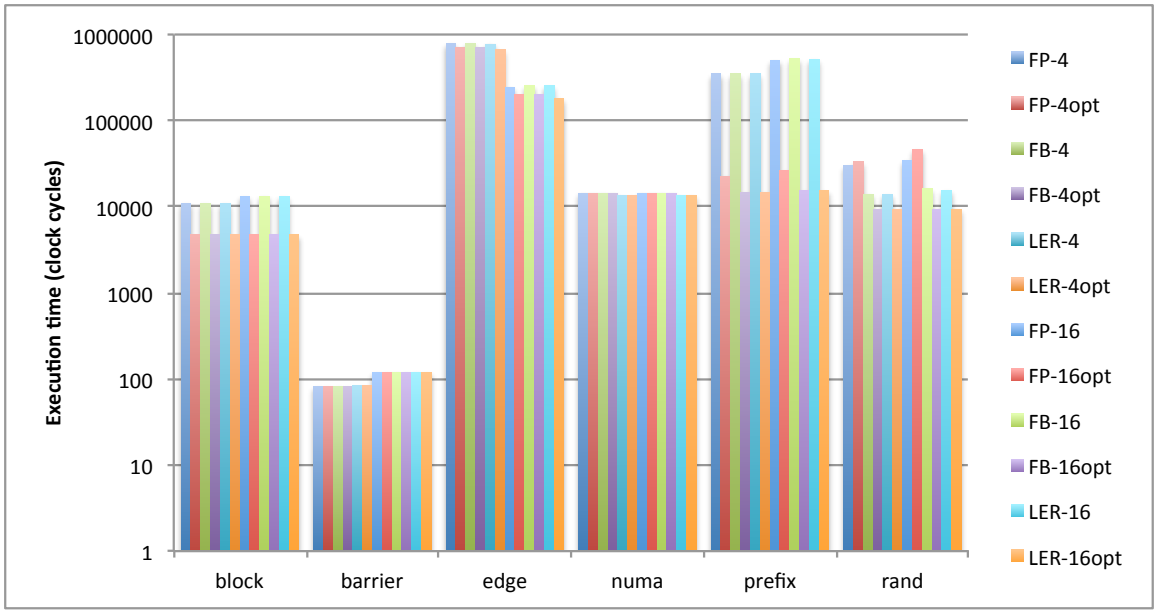

Figure 12: Execution time without optimizations and with locality and synchronizations optimizations (opt).

[17] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[18] V. Leppänen. *Studies on the realization of PRAM, Dissertation 3*. Turku Centre for Computer Science, University of Turku, 1996.

[19] J. Mäkelä, E. Hansson, D. Åkesson, M. Forsell, C. Kessler, and V. Leppänen. Design of the language REPLICA for hybrid PRAM-NUMA many-core architectures. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 697–704, 2012.

[20] J. Mäkelä, E. Hansson, and M. Forsell. REPLICA language specification, 2013. In preparation.

[21] D. Patterson. The Trouble With Multicore. *IEEE Spectrum*, 47(7):28–32, 2010.

[22] S. Fuller R. Swan and D. Siewiorek. Cm* – A Modular Multiprocessor. In *Procedings of NCC*, pages 645–655, 1977.

[23] A. Ranade. How to Emulate Shared Memory. *Journal of Computer and System Sciences*, 42:307–326, 1991.

[24] U. Vishkin. Towards Realizing a PRAM-on-Chip Vision, 2007. Workshop on Highly Parallel Processing on a Chip (HPPC), August 28, Rennes, France (see http://www.hppcworkshop.org/HPPc07/talks.html).

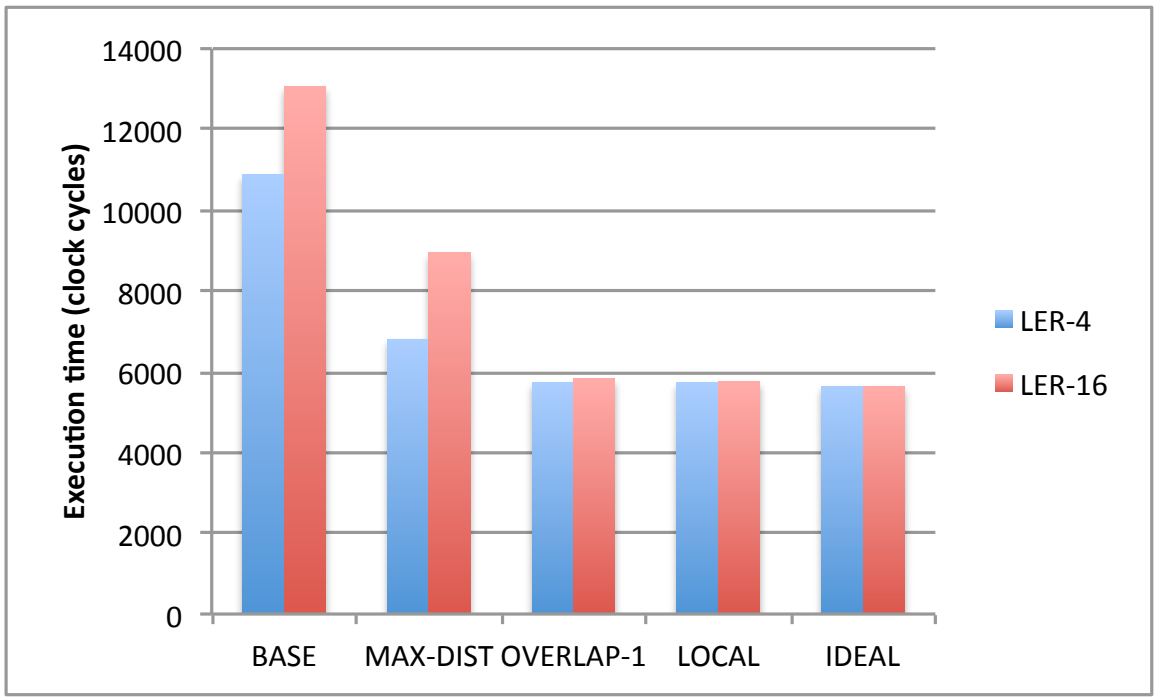

Figure 13: Execution time of the block benchmark with LER optimizations compared ideal shared memory and local memory executions.

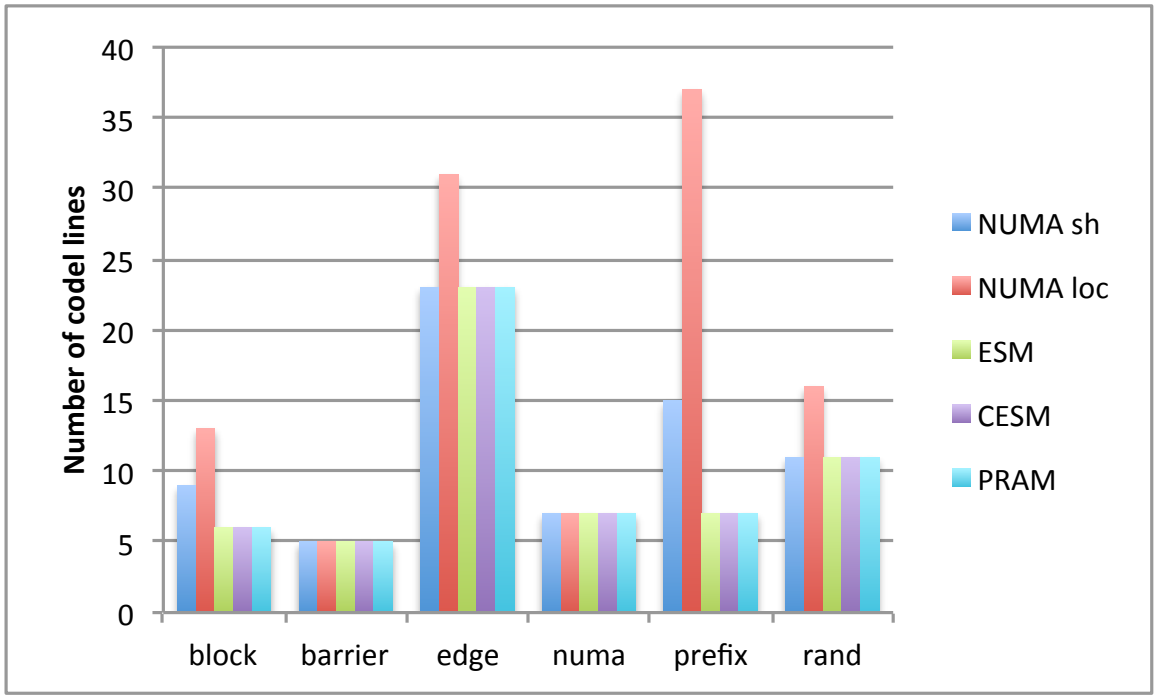

Figure 14: Length of the measured code segment of the benchmarks in e source code lines.