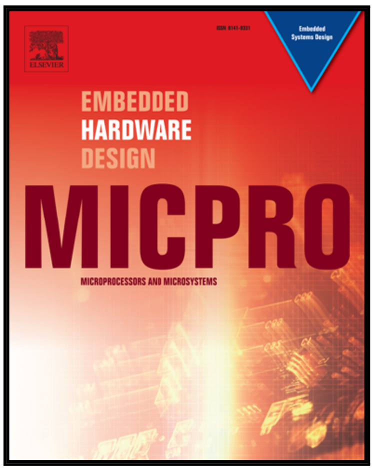


Supporting Concurrent Memory Access in TCF Processor Architectures

Martti Forsell , Jussi Roivainen , Ville Leppänen ,
Jesper Larsson Träff

PII: S0141-9331(18)30089-9
DOI: https://doi.org/10.1016/j.micpro.2018.09.013
Reference: MICPRO 2751

To appear in: *Microprocessors and Microsystems*

Received date: 19 February 2018
Revised date: 10 July 2018
Accepted date: 21 September 2018

Supporting Concurrent Memory Access in TCF Processor Architectures

Martti Forsella,*, Jussi Roivainena, Ville Leppänenb, Jesper Larsson Träffc

a*Efficient Computing and Communications, VTT, Finland*
b*Department of Future Technologies, University of Turku, Finland*
c*Faculty of Informatics,Vienna University of Technology, Austria*

a b s t r a c t

The Thick Control Flow (TCF) model simplifies parallel programming by bundling computations with the same control flow into single flows of variable thickness, and has the prospect of alleviating redundant usage of software and hardware resources. While architectures that can support the TCF model have been proposed, current proposals cannot support concurrent memory accesses that can both simplify programming and speed up many parallel algorithms by a logarithmic factor. In this paper, we extend current TCF architectures to efficiently support concurrent read as well as write memory accesses. The solution is based on bounded size step-caches, and exploit the two-part, hybrid, frontend-backend structure of current TCF processors, and synchronization properties of the TCF model itself. According to our simulation-based evaluation, a concurrent memory access TCF processor with *B* backends can execute algorithms with substantial concurrent memory accesses up to *B* times faster than a baseline TCF processor not supporting concurrent memory access. The hardware overhead of the solution is estimated to be modest. We include parallel program code to illustrate the gains by supporting concurrent memory accesses.

A R T I C L E I N F O

Article history:
Received 19 February 2018
Revised 10 July 2018
Accepted 21 September 2018
Available online ~~?? October 2018~~

Keywords:
Parallel computing
Processor architecture
Programming model
TCF
Concurrent memory access

* Corresponding author.
E-mail address: Martti.Forsell@VTT.Fi (M. Forsell)

1. Introduction

Standard programming models of current shared memory multicore processors expose sets of computational threads that execute individual code asynchronously [1, 2]. While this makes execution of independent threads straight-forward, data dependencies between threads make it necessary to explicitly synchronize them by constructs, like barriers, locks and atomic operations to guarantee that critical operations are carried out in the right order. Since the cost of applying these constructs in terms of execution time and usage of hardware resources is high in most current CPU and GPU architectures, many sophisticated and efficient paradigms for fine-grained parallel computing [3, 4, 5] are deemed impractical. These include *concurrent memory access* that lets a number of processors read and write a memory location concurrently, and execution of fine-grained parallel algorithms in general. Furthermore, since many computational problems contain abundant parallelism, the number of ideally needed threads is much higher than one or two per processor core as supported by current ARM and Intel hardware. While software based threading supports more threads than the hardware, in practice, the number of threads is often limited to few hundreds per processor core. Consequently, if there are frequent interthread dependencies, the performance of the system can degrade significantly if the thread count exceeds the number of hardware threads. To compensate this, a CPU programmer is forced to emulate high parallelism with the few available hardware threads by using looping or repetition. In GPUs, there are much more resources for data parallel execution but one needs to very carefully place data elements to different memories in order to have sufficient performance. This mismatch between architecture and application parallelism tends to make programming complex, error-prone and can cause the processor hardware to do redundant computation. Repeating base address computations and allocating registers containing replicated values for the threads are among the most notable examples of these redundancies. A promising way to eliminate these model-related problems is the *thick control flow* (TCF) programming model [6, 7] that packs user adjustable numbers of threads following the same control flow into a single entity and guarantees synchronous execution independently of the number of computational elements. The *Thick Control Flow Processor Architecture* (TPA) chip multiprocessor [8] can execute programs making use of the TCF model natively. While the architecture succeeds in supporting the unbounded parallelism of the model, it cannot support concurrent memory access that can speed up many algorithms by a logarithmic factor [9].

Supporting concurrent memory access as a primitive of parallel computation is meaningful only for the class of architectures that supports synchronous execution of threads. Most notable members of that class are so-called *Emulated Shared Memory* (ESM) architectures [10, 11] that implement an idealized synchronous shared memory model [12] via hardware emulation. For that, they use multithreading to hide the (distributed) shared memory system access and provide a low-cost synchronization mechanism for separating the steps of computational threads. While concurrent reads are well-defined operations in synchronous architectures, there are number of ways to handle concurrent writes. The convention for deciding which of the threads performing a concurrent write succeeds is traditionally COMMON, ARBITRARY or PRIORITY, by which the thread with the lowest identifier succeeds [12].

Previous attempts to support concurrent memory access and multioperations in ESM architectures include:

• **Combining networks.** This is used in early ESM architectures that utilize light-weight interleaved multithreading along with low-cost synchronization to emulate an ideal shared memory [10]. The main idea is to reduce the needed bandwidth by combining the references targeted to the same location in the network. When loading data from memory, network nodes need to store the addresses for recreating all the replies for the processors that initiated them. This requires allocating storage for the pending references in the nodes. The combining networks technique also requires sorting of memory requests prior to injection, which decreases the speed of this solution for all memory accesses.
• **Streamlined combining networks.** This operates like the combining networks technique but can reduce the number of routing phases from six used in the original technique to five, and reduce the number of memory modules [11]. Unfortunately, also this requires the same sorting phase as the non-streamlined combining networks [11].
• **Active memories.** This technique implements limited and partial concurrent memory access for a limited number of special memory locations [13]. Due to the limitation, programming involves managing those limited, special locations, which can slow down computation. Compared to the previous attempts, this solution, however, eliminates the need for sorting.
• **Step caches.** This solution implements full concurrent memory access for all memory locations [14]. It is based on filtering out all but the first reference to each location per step with the help of caches, thus reducing memory traffic. Practical associativity limitations require resending a reference if it has been wiped out from the step cache due to set overflow. Also this solution eliminates the need for sorting prior to injection of references to the network.

Except for the active memories, none of these architectures can be used to provide concurrent memory access for TCF processors. This is because they rely on fixed size buffers in which the size is proportional to the number of threads per processor, while for TCF processing the number of computational elements per processor is not bounded. The active memory solution could work but it provides only a very limited amount of active memory locations for each memory module and therefore it cannot be considered a general solution to the concurrent memory access problem. Finally, the practical execution time of many algorithms has turned out to be much lower for the step cache technique than for active memories [15].

The main contribution of this paper1 is a solution for implementing concurrent memory access in TCF processors. In particular, we

• introduce two architectural techniques significantly reducing the number of memory system references in the case of concurrent memory accesses. These rely on bounded size step caches and the two-part, hybrid, frontend-backend structure of current TCF processors. The former capture and hold the references made during the on-going step of an execution that are independent by the definition of TCF execution and therefore avoid coherence problems. The latter allows for reduction of single-location concurrent reads to a frontend operation followed by a broadcast in the spreading network of TCF processors;
• evaluate the performance gains of the techniques by simulating execution of benchmark programs in our modified TPA architecture. The measurements indicate that a concurrent memory access-aware, *B*-backend unit TCF processor executes certain algorithms up to *B* times faster than the baseline TCF processor with the same configuration;
• complement the execution time evaluations with simulations on different step cache sizes and associativities to help in configuring TCF processors for efficient concurrent memory access;
• give estimates of the silicon area and power consumption overheads of implementing the techniques with a state of the art silicon technology based on analytical modeling and publicly available information on silicon processes and their scaling. The results indicate modest hardware overheads;
• explain how concurrent memory accesses can be used in practical parallel programs and show how a constant time find-maximum algorithm can be implemented in a TCF processor supporting concurrent memory access.

In Section 2, we describe the TCF model and TPA architecture, Section 3 proposes support for concurrent memory access in TCF architectures, Section 4 evaluates the proposed solutions with simulations in TPA, and Section 5 draws conclusions and outlines future work.

2. TCF Model and TPA Architecture

The work presented in this paper is built on the TCF model and the first processor architecture implementing it, called TPA. In this section, we give a short introduction to the model and explain how the architecture works and what its main components are.

The *Thick Control Flow* model bundles threads following the same control flow into entities called *thick control flows* (TCF) [6, 7] as shown in Figure 1. The components of a TCF are called *fibers* to distinguish them from threads having their own control. The number of fibers in a TCF is called the *thickness* of the flow. The model allows TCFs to change their thickness during execution with no upper bound. Execution of a TCF happens in *steps* during which each fiber executes an instruction in parallel with the other fibers in the TCF. The model guarantees synchronous operation of steps so that the shared memory references generated by the previous step are completed before the current step starts and their results will be available for all the fibers during the current step, but no updates in the current step are visible to other fibers in the step.

The unbounded parallelism of TCFs is a challenge to processor design due to fixed hardware resources. The key technique is to assign an arbitrarily large set of fibers to processing elements for cost-efficient synchronous execution of computational steps as shown in Figure 1.These challenges are addressed by the *Thick Control Flow Processor Architecture* (TPA) [8]. It uses a two-part, hybrid, frontend-backend structure. The frontends fetch and execute VLIW instructions taking care of the duties common to all fibers of the TCFs and the backends perform fiber-wise processing for TCFs and operands sent by the corresponding frontends. The backend units have a special *replicated register* block scheme since no on-chip register block has room to keep data for unbounded numbers of fibers. The idea is to store fiber-wise registers to on-chip replicated register blocks as long as there is room for them. In the case of an overflow, least recently referred registers are sent out to external memory and fetched back before they are needed again. Standard ESM implementation techniques are used for efficient shared memory multiprocessor execution [10, 11, 16]. These include multifibering (a TCF-aware variant of multithreading) to hide the latency of shared memory accesses, low-cost wave-based synchronization, and low-level parallelism exploitation to maximize utilization of the functional units. A TPA processor consists of *F* frontend units (or cores) and *B* backend units, $F \leq B$, connected together via a work spreading network/return channel as shown in Figure 2. The frontend units are connected to a low latency *non-uniform memory access* (NUMA) style memory system aimed for keeping the program code and data common to all fibers of TCFs. The backend units are connected to distributed shared memory system modules via a small number of parallel mesh networks for high-throughput access to TCF-parallel data. TCF buffers keep the data of TCFs including a set of general purpose registers, program counter (PC), TCF identifier and thickness while TCFs are not being executed.

TPA instructions are executed in three front-end phases and three backend phases:

For each active frontend **do**
F1. Select the next TCF from the TCF buffer if requested by the previous instruction.
F2. Fetch a VLIW instruction pointed by the PC of the current TCF from the NUMA memory system.

F3. Execute the subinstructions in the functional units specified by the VLIW instruction. Memory subinstructions are typically targeted to the NUMA memory system. If the instruction contains a backend part, select operands and send them along with the part to the backends assigned to the frontend via the work spreading network. Store the data of current TCF to the TCF buffer and switch to the next TCF if requested by the corresponding subinstruction.

For each backend **do**
B1. If the backend is not executing the previous instruction any more, fetch the next instruction from the spreading network and determine the fibers to be executed in the backend. Otherwise continue executing the previous instruction.
B2. Generate the fibers of the TCF to be pipelined according to the assignment determined in B1.
B3. **For each** fiber **do**:
B3.1 Select the operands from the received frontend data and replicated register block.
B3.2 Execute the backend subinstructions. Memory subinstructions are targeted to the shared memory system.
B3.3 Write back the replicated register block and send the optional reply data back to the frontend via the return channel built into the spreading network.

After all active TCFs of a frontend have been in execution for a single instruction, TPA issues a special synchronization TCF of thickness one per backend that sends and receives a synchronization to/from the shared memory system. For this we rely on the wave synchronization scheme used in [10].

3. Implementing Concurrent Memory Access

General requirements for supporting efficient concurrent memory access in a multiprocessor system include keeping instructions executed synchronously and matching the number of references per time unit and per memory location to the limited bandwidth of the memory modules. These are needed to guarantee that concurrent memory access instructions are executed simultaneously, execution of them is not overlapping with other memory access instructions, and to avoid hot spots in the shared memory traffic. The former requirement is easy to meet by the construction at hand since TCF execution is synchronous by definition and existing architectures support it [8]. The main techniques to fulfill the latter requirement include parallel reduction and broadcasting. Our aim is to implement these via step caches and frontend-backend communication mechanisms of TCF processors.

3.1. Step caching

Step caches are associative buffers holding copies of the latest references to the memory system [14]. Unlike ordinary caches, there are no coherency issues in step caches since the lifetime of cache entries range just until the end of the current step. Step caches have been successfully applied to reduce the memory traffic in the case of ESM architectures featuring a fixed threading scheme [15]. Applying step caches directly to TCF processing featuring unbounded number of fibers raises, however, problems since the existing solutions allocate a step cache and reply functionality resources per thread [15]. Our idea is to extend step caches by synchronizing them with the logic that takes care of receiving replies from the shared memory and providing an overflow mechanism for the situations in which the step cache capacity is exceeded due to excessive number of references. The resulting TCF-aware step cache unit consists of a bounded number of step cache lines containing an InUse bit, an InMemory bit, an address tag and a payload data representing references to the target locations split to Tag, Index and Offset as shown in Figure 3. Like for ESM architectures [14], step caches in the TPA are placed close to the backend pipeline so that no slowdown occurs even if accesses miss the cache (due to the latency hiding).

3.2. Frontend backend communication mechanisms

A TCF processor needs mechanisms for sending information from the frontends to backends and vice versa. In TPA this is taken care for by the work spreading network and a return channel. The *spreading network* is a network that attaches a frontend of TCF processor to the backends. The main purpose of it is to pass operation codes for the backends' functional units along with selected data/operands. Topologically the spreading network resembles a binary tree (see Figure 3). Since the TPA employs a multimesh-based interconnection network between the backends and shared memory modules [8], one would be tempted to embed the spreading network into a mesh as proposed, e.g., in [17], and achieve possible network design synergies. In this work, however, we use a straight-forward tree-shaped spreading network for simplicity and leave possible embedding to our future work. The *return channel* is a network that takes care of sending backend data to the frontend that is controlling the backend unit. Since there are typically multiple backend units controlled by a single frontend, the return channel passes multiple data values or alternatively does a reduction to obtain a single value. Topologically, the shape of the return network is the same as that of in the spreading network but it operates to the opposite direction (see Figure 3). In this work we use a simple, non-combining mechanism for sending single backend values to the frontend.

3.3. Concurrent access via step caching and broadcasting/reduction

We first have a look at the typical concurrent memory access patterns, and usage of the two-part, hybrid, frontend-backend structure of current TCF processors to implement them. Depending on the targeted concurrent memory access convention and architectural realization of the TCF-model, there are alternative schemes: For example, if only a single-location PRIORITY concurrent memory write per TCF is allowed, the frontend can easily and cost-efficiently take care of the concurrent access, assuming that the frontend has access to the shared memory system of the backends. On the other hand, if there is a high number of ARBITRARY writes in parallel, it may be preferable for the backend units to take care of them. Without the techniques proposed in the previous subsections, a single-location concurrent access in the latter scheme forces all fibers of all B backends to send their references to a single ported memory causing sequentialization and thus B-fold slowdown (see Figure 4/left). Our solution for efficient concurrent memory access in TCF processors is to exploit the possibility to change the thicknesses of the TCFs, and apply bounded size step caches in synchrony with the functionality receiving replies of reads. For simple concurrent access patterns, we use the spreading network for broadcasting values from the frontend to the backends and the return channel for reducing the data from backends to the frontend to support thick concurrent memory access operations and their reduction to narrow ones.

We start from the cases in which all the fibers of a TCF are participating in the same single-location concurrent memory access, or only one concurrent access per TCF is allowed. In the case of read, we can use the *frontend read technique* in which frontends fetch the value from the shared memory and make it available to all backends via the work spreading network and backend-wise replication of data (see Figure 4/right). In its turn, a concurrent write can be handled by just setting the thickness to one for a single instruction and performing the single write. This *reduced backend write technique* applies to COMMON, ARBITRARY and PRIORITY conventions. Alternatively, the frontend can take care of the write in the case it has a copy of the value. There are three obvious ways to implement this kind of *frontend write technique* shared memory system: (1) a dedicated frontend shared memory port, (2) sharing the memory port with a backend unit, and (3) instructing one of the backends to perform the access and in the case of read, sending the received value to the frontend via the return channel mechanism. Here we use the backend variant (3), since a dedicated port variant (1) needs potentially substantial modifications to the interconnection network and its topology. The shared port variant (2) is quite close to the backend access variant (3), but requires a separate FIFO with explicit synchronization messages for binding the access to the right step of execution. There are two optimizations within the backend access variant (3): The frontend makes a request to execute the shared memory reference in the backend that has the shortest route to the memory module containing the memory location, or the backend passes the reply to the frontend as soon as it arrives. These techniques save time by exploiting the locality of the shared memory or bypassing the rest of the pipeline.

The case of multiple concurrent reads per TCF is more challenging since a single frontend can typically process only a single read at the time, and because there may not be register space for holding the necessary amount of values in the frontend. Our approach is to use backends for accessing the data and consider this just as a specific memory pattern that is accelerated with the help of bounded size step cache (see Figure 4/center). In this *backend access technique*, the backends access the step caches prior to the memory operation for all fibers. In the case of a step cache miss, the operation is executed normally and a new entry for the target address is created in the step cache. In the case of a step cache hit, the target address has been accessed already allowing one to cancel the memory operation in the cases of write and load the data from the step cache in the case of read. Since the thickness can be arbitrarily high, it is possible that the cache line containing the accessed value gets overwritten while there are references to the same location still coming from the remaining fibers. In that case, a new step cache entry allocation is made when the first new reference is executed, and the cache is synchronized with the logic for receiving memory replies. The case of multiple concurrent writes works similarly, but there is no need for involving the logic for receiving replies.

The *efficiency principle* of a B-backend unit TCF-processor executing a TCF with thickness T containing multiple simultaneous concurrent accesses is that the system works efficiently as long as the accesses per location are limited to T/B which is the minimum execution time of a step. Otherwise the number of fibers participating a concurrent access will slow down the execution time of the corresponding step.

4. Evaluation

To show that concurrent memory access-aware TCF processors meet the expectations, we evaluate the performance of TPA supporting the techniques proposed in the previous section and discuss implementation issues. Programming examples are given to showcase how concurrent memory accesses can be exploited in practice.

4.1. Performance

In order to determine the performance of the proposed technique, we measure the execution time of six kernel benchmarks representing different usage schemes of concurrent memory access (see Table 1). We use different problem size N and/or different data set in the baseline TPA architecture (denoted TCF baseline) and a TPA architecture including the techniques of Section 3 (denoted TCF CRCW). For reference purposes, we also measure the performance of configurable ESM REPLICA architecture [16] (denoted CESM) with the same set of benchmarks. CESM applies the fixed threading scheme (T_p threads per processor) and employs the step cache technique introduced in [14]. It represents the best known concurrent memory access-aware architecture. The measured architectures are summarized in Table 2.

The benchmarks were executed in clock accurate simulators modeling the TPA and CESM architectures down to low-level details [8]. To eliminate the effect of compilers and make comparisons fair, all the benchmarks were written in assembler. We optimized the benchmarks by hand for the backend access technique so that the performance is limited only by the memory bandwidth and thread/fiber synchronization. The amount of on-chip memory was set large enough for holding all data needed in all the tests.

First, we made runs for **block**, **spread** and **cwrite** benchmarks for N values ranging from 2048 to 65536. For **mread** and **mwrite**, tests were made for fixed problem size N=65536 for 1..65536 fibers participating to each concurrent accesses (65536...1 simultaneous concurrent accesses). For the **max** benchmark the case size N ranged from 64 to 256 since part of the computation involves N2 fibers. The results of the backend access technique simulations are shown as execution times in clock cycles and speedup with respect to the baseline TCF (see Figure 5). From these results we can make the following observations:

• The proposed step cache technique speeds up the concurrent accesses of the benchmarks exceptionally well so that the performance becomes optimal with respect to the available memory bandwidth independently of N. At the same time, the measured synchronization overhead achieves the theoretical minimum of $1/(N/B)$, where N is the problem size and B is the number of backend units. In practice, the overhead drops from 0.78% down to 0.024% as N grows from 2048 to 65536.
• Compared to the baseline TCF solution, the proposed new solution gives speedup approaching B per concurrent memory access especially for large problem sizes. This is because in the baseline, B backend units are simultaneously trying to access a single-ported memory module containing the target location.
• In the case of multiple concurrent memory accesses the **mread** and **mwrite** benchmarks interestingly show how the performance changes in the baseline but stays constant in the proposed new solution as we go from exclusive memory access gradually to fully concurrent access. The best speedup approaching B is achieved in the case where all fibers are accessing the same location.

• The proposed technique gives slightly better results than CESM with step caches in all benchmarks. This difference is mainly caused by the looping needed to map the software threads to actual hardware threads in CESM. In particular, loop initializations take at least T_p cycles in CESM, where T_p is the number of threads per processor, and have to be executed in the frontends of the TCF architectures and therefore typically overlap with the backend execution.

In order to evaluate the practical difference between the frontend and backend access techniques, we implemented the **spread** and **cwrite** benchmarks also with the frontend read and write techniques. For **spread**, we measured the execution time as a function of N and compared it to the backend read technique (see Figure 6). The results indicate speedups ranging from 55% to 98% over the backend access technique. We conclude that for this kind of functionalities, the frontend read technique can speed up single-location concurrent reads considerably. For **cwrite**, we implemented reduction to a single-location write with both frontend write and reduced backend techniques and compare them to the backend access technique. As expected, the results are excellent for reduced backend access technique showing speedups approaching to $T/2B$, where T is the thickness of the computation with respect to the non-reduced writing (see Figure 6). The frontend write technique also performs well compared to the plain backend access technique but shows over 20 times longer execution times than the reduced backend technique. The actual maximum speedup of these write results may not, however, be as large (2049 and 74, respectively) as these measurements are indicating, since external memory system delays are not taken into account.

To figure out the effect of step cache size and associativity on the concurrent memory access performance we executed **spread**, **cwrite**, **mread** and **mwrite** benchmarks in the configurations featuring sizes from 1 to 1024 (per backend) and associativity from 1 to 4 and measured both execution time and step cache hit rate. For all these benchmarks, use of step caches dropped execution time down to the optimum. This is caused by the TPAs ability to process memory references efficiently even without step caches. The performance drops in the case of step caches only if the number of references per shared memory module exceeds substantially the even distribution of T/B references per module. By changing the memory pattern of the **mread** benchmark we, however, found some extreme cases with very low step cache size and associativity 1 in which step caches were not able to speed up concurrent accesses (see Figure 7).

Another aspect of implementing concurrent memory access with the help of step caches is that reduction of memory traffic saves energy. We leave detailed energy estimates of concurrent memory access to future work but have a look at step cache hit rates in the measurements with the backend access technique, while the silicon area and power consumption of using TPA with step caches is estimated in the following subsection. Figure 8 shows the step cache hit rate and thus the reduction of memory traffic for benchmarks **block**, **spread**, **cwrite**, **mread** and **mwrite** N=65536 as well as **max** for N=256. We observe that for the concurrent memory access benchmarks, the step cache hit rate is high and therefore the reduction of traffic substantial.

The limitations of the step cache technique are similar to those of caching in general. While step caches do not suffer from coherence problems, periodic access patterns interfering with the replacement policy can cause a high number of cache misses and therefore invalidate the gains of the proposed concurrent memory access technique. On the other hand, our measurements show that the speedups of the proposed technique can be achieved even with small step caches. Since the coverage of this evaluation in terms of different kinds of algorithms is limited, further studies are needed to draw more general conclusions.

4.2. Implementation considerations

Implementing the proposed technique in TPA does not require major modifications to the baseline architecture. One needs to have a bounded size step cache that works with the existing bounded size reply buffer allocation policy for each backend, the return channel mechanism, and frontend access to the shared memory. Note that these additions can be also used for many other purposes, such as implementing multioperations. In this work we assume that the return channel mechanism is already included as explained in Section 2 and that we use the backend variant (3) for the frontend access leaving the step cache only major modification (see Figure 9). In the performance evaluation of Section 4.1, we used 1024-line step cache in most of our measurements. In a 16-backend TPA this would translate to 256 KB additional cache memory.

In order to roughly estimate the relative cost of this addition in terms of silicon area and power consumption, we modeled a 64-bit, 16-backend TPA baseline with 10 backend functional units running at 1.5 GHz and containing 72.9 MB on-chip memory with and without step caches using our 100-parameter analytical performance-silicon area-power consumption model [18]. For that we broke the processor description down to adders, multipliers, shifters, logic instruction blocks, multiplexers, registers, memory cells, switches and interconnection wires and calculated the overall number of logic gates, memory cells and interconnection links. Based on these input values as well as dimension, area and power information of transistors, memory cells and wires [19], the model calculates a silicon area and power consumption estimates of a possible implementation. Taking the topology of the main interconnection network into account and assuming that the resulting chip would be square, we can calculate the length of main interconnect wires. An estimate of maximum clock frequency is then calculated with help of a parallel signal propagation model of [20]. We use the parameters of a 65nm high performance process that are the latest publicly available and scale to the 11nm process using the scaling factors presented in more recent ITRS reports and in [21]. Ignoring the local wiring costs, the model predicts that the silicon area and power consumption costs of adding the step caches utilized by the proposed technique is just a few promilles.

4.3. Programming examples

Concurrent memory access should methodologically work as a natural addition to current machine and high-level languages. We will show with two programming examples that this goal can be achieved on TPA implementing the proposed techniques. As a presentation language we use a C-like parallel language in which **#** denotes the number of threads/thickness, **$** denotes thread/fiber identifier and **_** in the end of variable name denotes a shared variable. Program execution starts with a group of **#** threads or a TCF with thickness **#**. Statements are executed one by one in parallel. We assume that in the non-TCF architecture execution is asynchronous and requires explicit barrier synchronizations, denoted with **synchronize** to guarantee correct execution. In the TCF architecture this is not necessary due to the intrinsic synchronicity of TCFs.

Let us first consider the **spread** benchmark that spreads the value of first element to rest of the elements in an *N*-element array **a_** of integers. In an ordinary architecture without support for concurrent memory accesses we would need to replicate the first element with a single processor to two values. Then these two copies would be replicated with two processors. These exponential increase is repeated until the whole array is filled with the desired value. The following logarithmic spread algorithm implements this functionality:

```
int i, j, a_[N];          // i and j are thread-private variables, a_ shared array
for (i=1; i<N; i<<=1)
{    for (j=$+i; j<N; j+=#)              // Process # elements in parallel
          if (j-i>=0) a_[j]=a_[j-i];    // (and proceed to larger j values)
     synchronize;   }
```

Since *N* is typically greater than **#** in most machines, the algorithm copies at most **#** elements in parallel in the inner **for** loop. Employing concurrent memory access, the same functionality can be written

```
int i, a_[N];
for (j=$; j<N; j+=#)  // Process # elements in parallel
     a_[j]=a_[0];
synchronize;              // This can be dropped if N divides evenly by #
```

Since it is possible to set the thickness of the computation to match *N* in a TCF architecture, the inner loop can be completely eliminated. The example with exclusive memory access now becomes

```
int i, a_[N];             // i is a frontend variable, a_ shared array
#N;                       // Set the thickness:=N
for (i=1; i<N; i<<=1)
     if ($-i>=0) a_[$]=a_[$-i]; // Process the whole array in parallel
```

and with concurrent memory access it is just

```
int a_[N];
#N: a_[$]=a_[0];          // Process the whole N-element array in parallel
```

respectively. As we saw from the simulations, the performance of the latter is poor due to the *B* backend units referring to the same memory location placed in a single ported memory module unless we have the proposed concurrent memory access support. Employing the backend access technique requires no changes to the code above but the frontend would change the code to

```
int a0, a_[N];
a0=a_[0];                 // do the access with the frontend
a_[$]=a0;                 // Spread to the whole N-element array in parallel
```

Consider the **max** benchmark finding the maximum of an *N*-element input array **src** by comparing each element to all other [22]. For the ESM processors it can be implemented with three loops (see Figure 10). The first loop initializes the **src** array with pseudo randomly ordered numbers with # parallel threads. The second loop initializes the **dst** array by writing **1** to all elements to mark that any of them can be potentially the maximum. The third loop does the actual computation by doing *N2* comparisons in total and writing **0** to the corresponding **dst** element if there is a compared value that is greater than the element. The algorithm initiates both *N* concurrent read accesses and *N* concurrent write accesses.

For the TCF processors this happens by starting execution with the default TCF, setting the thickness to *N*, initializing the *N*-element output array **dst** to **true** and then comparing each element to all other in parallel and writing **false** to the corresponding **dst** element if the compared element is greater than it. For that, the algorithm sets thickness to *N2* prior to the comparison. Since the **if**-statement would require fiber-wise splitting the TCF into two sub TCFs according to the input data values, we use the temporary value array **tmp** to avoid creating those TCFs. After the comparison the maximum value of the array **dst** contain **true** while all other are **false**. Figure 10 shows implementations of max in TCF-aware high-level language and assembler. Each line represents a VLIW instruction composed of one or more subinstructions. The subinstructions marked in blue are executed in the backends. Note that even though there are 256 initializations and 65536 pairwise comparisons for only 16 backend units, there is no need to perform looping or repeating, but that the thickness of the computation is set to 256 for initialization and then to 65536 for comparisons. The versions for the baseline TCF and the concurrent memory access-aware versions are identical except for the execution time.

5. Conclusions

We have described an architectural solution to realize concurrent memory access for TCF processors. The solution is based on bounded size step caches and a two-part, hybrid, frontend-backend structure of current TCF processors. Step caches capture and hold the references made during the on-going step of an execution. Since operations within a step are independent by the definition of TCF execution, there are no coherence problems. If there is only a single-location concurrent operation per TCF, the active frontend unit can perform TCF-level accesses cost-efficiently. According to the evaluation, the concurrent memory access-aware *B*-backend unit TPA executes certain algorithms up to *B* times faster than the similarly configured baseline TPA. Employing the frontend in the case of single-location concurrent memory access can further speed up execution. The cost of the proposed technique in silicon area and power consumption is estimated to be very small. Our programming examples demonstrate that concurrent memory access can be integrated as a natural pattern of parallel programming in both high-level and assembler languages.

In the future we aim to further study and develop TCF-aware computing hardware and methodology. This includes an FPGA proof of concept implementation and studying the possibility to realize multiprefix operations on TCF architectures.

Acknowledgment

This work was funded by the Academy of Finland grant 289773.

References

[1] D. Culler and J. Singh, *Parallel Computer Architecture—A Hardware/ Software Approach*, Morgan Kaufmann Publishers Inc., San Fransisco, 1999.
[2] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5th Edition, Morgan Kaufmann Publishers Inc., Palo Alto, 2011.
[3] J. Jaja, Introduction to Parallel Algorithms, Addison-Wesley, Reading, 1992.
[4] U. Vishkin, Using Simple Abstraction to Reinvent Computing for Parallelism, *Communications of the ACM* **54**, 1 (January 2011), 75-85.
[5] F. Ghanim, R. Barua and U. Vishkin, Easy PRAM-based High-performance Parallel Programming with ICE, *IEEE Transactions on Parallel and Distributed Systems* **29**, 2 (February 2018).
[6] M. Forsell and V. Leppänen, An Extended PRAM-NUMA Model of Computation for TCF Programming, *International Journal of Networking and Computing* **3**, 1 (2013), 98-115.
[7] J-M. Mäkelä, M. Forsell and V. Leppänen, Towards a Language Framework for Thick Control Flows, *Proc. of the High Level Programming Models and Supporting Environments (HIPS'17)*, May 29, 2017, Orlando, FL, USA.
[8] M. Forsell, J. Roivainen and V. Leppänen, Outline of a Thick Control Flow Architecture, *Proc. 5th Workshop on Parallel Programming Models Special Edition on Task Parallelism*, October 26-28, 2016, Marina del Rey Marriott, Los Angeles, USA.
[9] R. Karp and V. Ramachandran, Parallel Algorithms for Shared Memory Machines, *In J. van Leeuwen editor, Handbook of Theoretical Computer Science (**Vol A**)*, MIT Press, Cambridge, MA, USA, 1990, 869 - 941.
[10] A. Ranade. How to Emulate Shared Memory. *Journal of Computer and System Sciences* **42**, (1991) 307–326.
[11] J. Keller, C. Keßler, and J. Träff, *Practical PRAM Programming*, Wiley, New York, 2001.
[12] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, *Proc. 10th ACM STOC*, New York, 1978, 114-118.
[13] M. Forsell, Realising constant time parallel algorithms with active memory modules, *International Journal of Electronic Business* **3**, 3-4 (2005), 255-263.
[14] M. Forsell, Step Caches—a Novel Approach to Concurrent Memory Access on Shared Memory MP-SOCs, *Proc. 23th IEEE NORCHIP Conference*, November 21-22, 2005, Oulu, Finland, 74-77.
[15] M. Forsell, Realizing Multioperations for Step Cached MP- SOCs, *Proc. of the International Symposium on System-on-Chip 2006 (SOC'06)*, November 14-16, 2006, Tampere, Finland, 77-82.
[16] M. Forsell, J. Roivainen and V. Leppänen, REPLICA MBTAC - Multithreaded Dual Mode Processor, *Journal of Supercomputing* **74**, 5 (2018), 1911-1933.
[17] S-K. Lee and H-A. Choi, Embedding of Complete Binary Trees intlo Meshes with Row-Column Routing, *IEEE Transactions on Parallel and Distributed Systems* **7**, 5 (May 1996), 493-497.
[18] M. Forsell, On the performance and cost of some PRAM models on CMP hardware, *International Journal of Foundations of Computer Science* **21**, 3 (2010), 387-404.
[19] International Technology Roadmap for Semiconductors (ITRS), Semiconductor Industry Association (SIA); http://www.itrs.net.
[20] 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.
[21] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam and D. Burger, Power Challenges May End the Multicore Era, *Communications of the ACM* **56**, 2 (February 2013), 93-102.
[22] Y. Shiloach and U. Vishkin, Finding the Maximum, Merging and Sorting in a Parallel Computation Model, *Journal of Algorithms* **2**, (1981), 88-102.

Table 1
Test programs for the proposed TCF-aware architecture.

Benchmark	**Description**
block	A parallel program that copies an array of 2048..65536 integers into another in the shared memory (*tests exclusive parallel memory access*)
spread	A parallel program that spreads the value of first element to rest of the elements of an array of 2048..65536 integers (*tests a single concurrent read access*)
cwrite	A parallel program that performs concurrent write of a set of 2048..65536 values (*tests a single concurrent write accesses*)
mread	A parallel program that performs 1..65536 concurrent reads from an array of 65536 values(*tests multiple concurrent read accesses per TCF*)
mwrite	A parallel program that performs 1..65536 concurrent writes to an array of 65536 values (*tests multiple concurrent write accesses per TCF*)
max	A parallel program that finds a maximum of an array of 64..256 integers by comparing all elements to each other (*tests* multiple *concurrent accesses*) [19]

Table 2
Tested processors.

Processor	**CESM [13] (The best existing result)**	**TCF baseline [5] (Baseline)**	**TCF CRCW (This proposal)**
Processing units	16 NUMA/16 Parallel	1 frontend/16 backend	1 frontend/16 backend
Fibers/threads per processor core	128	Unbounded	Unbounded
TCFs per frontend	-	128	128
Number of functional units	3 NUMA/9 Parallel	5 frontend/9 backend	5 frontend/9 backend
Step cache size/type/replacement policy	128/4-way set associative/random	-/-/-	0...1024/1..4-way set associative/random
Interconnect	4x4 mesh	4x4 mesh	4x4 mesh

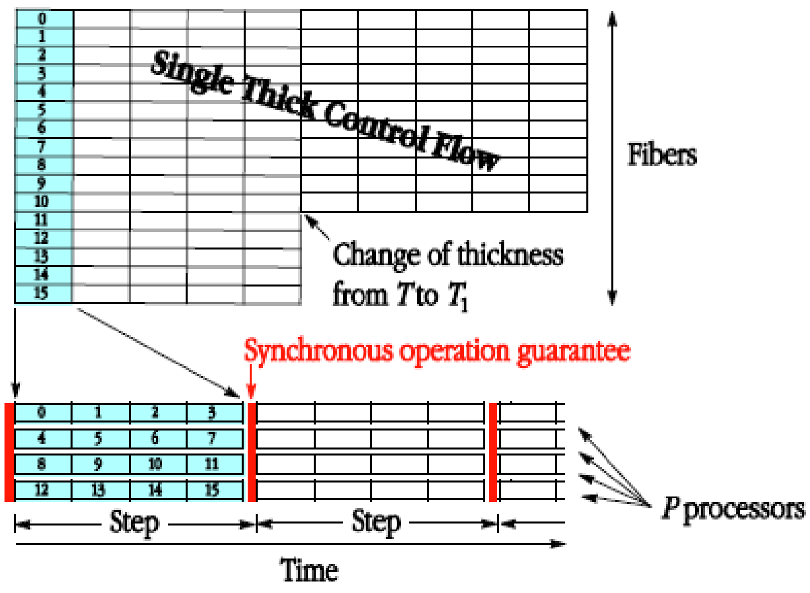

Single Thick Control Flow
Fibers
Change of thickness from T to T_1
Synchronous operation guarantee
P processors
Step
Step
Time

Fig. 1. A thick control flow, change of thickness from 16 to 11 and its execution as steps. Boxes represent computations. A step of execution in a TCF-aware processor. The machinery assignings T=16 fibers for execution in P=4 processors and guarantees synchronicity between the steps.

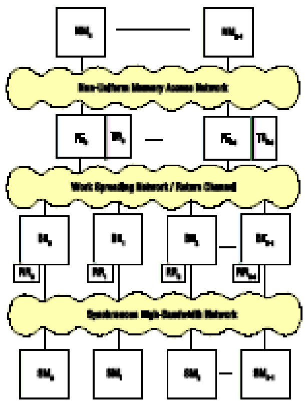
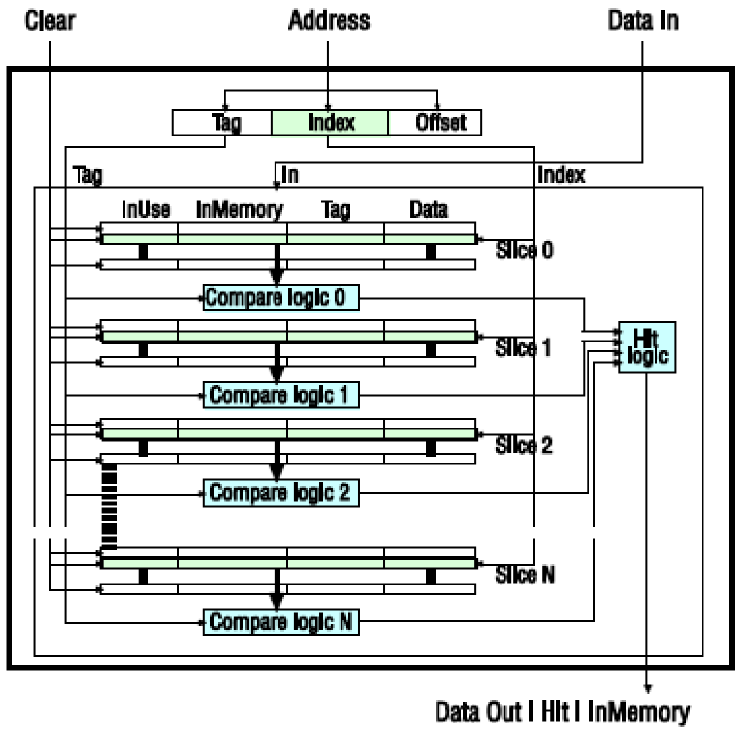

Non-Uniform Memory Access Network
Work Spreading Network / Return Channel
Synchronous High-Bandwidth Network

Fig. 2. The overall structure of TPA (FE=processor frontend, NM=NUMA memory module, TB=TCF buffer, BE=processor backend unit, RR=replicated register block, SM=shared memory module). For simplicity, the external memory system and topologies of the networks are not shown.


Clear
Address
Data In
Tag
Index
Offset
Tag
In
Index
InUse
InMemory
Tag
Data
Slice 0
Compare logic 0
Slice 1
Hit logic
Compare logic 1
Slice 2
Compare logic 2
Slice N
Compare logic N
Data Out | Hit | InMemory

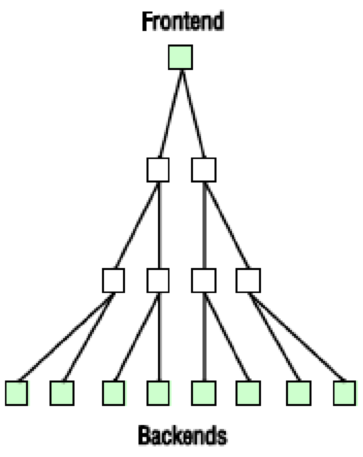


Fig. 3. Simplified block diagrams of a TCF-aware *N*-way set associative step cache (top) and spreading network/return channel (bottom).

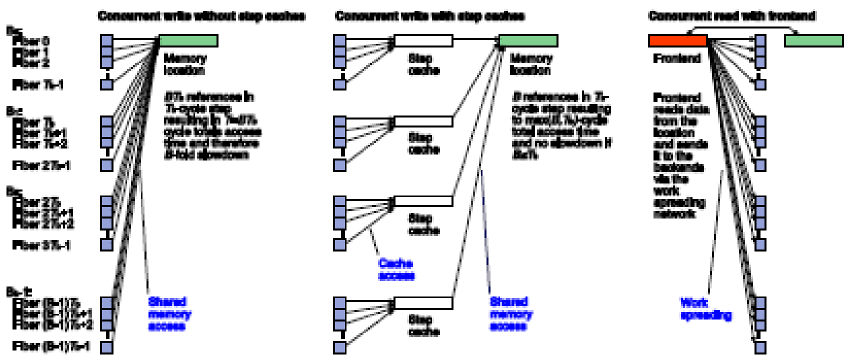

Concurrent write without step caches
B0:
Fiber 0
Fiber 1
Fiber 2
Fiber Tb-1
B1:
Fiber Tb
Fiber Tb+1
Fiber Tb+2
Fiber 2Tb-1
B2:
Fiber 2Tb
Fiber 2Tb+1
Fiber 2Tb+2
Fiber 3Tb-1
Bb-1:
Fiber (B-1)Tb
Fiber (B-1)Tb+1
Fiber (B-1)Tb+2
Fiber (B-1)Tb-1
Memory location
BTb references in Tb-cycle step resulting in Tb=BTb cycle total access time and therefore B-fold slowdown
Shared memory access
Concurrent write with step caches
Step cache
Memory location
B references in Tb-cycle step resulting to max(B, Tb)-cycle total access time and no slowdown if B≤Tb
Cache access
Shared memory access
Concurrent read with frontend
Frontend
Frontend reads data from the location and sends it to the backends via the work spreading network
Work spreading

Fig. 4. Concurrent in the baseline system (with no step caches), concurrent write with step caches and concurrent read with the fronend.

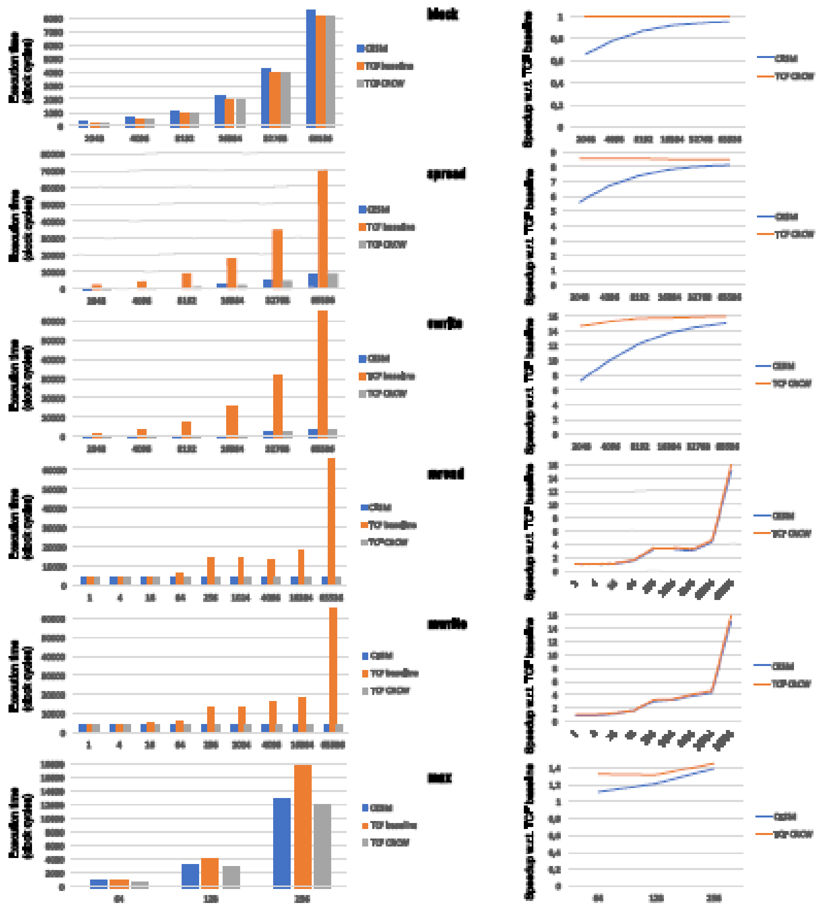


Fig. 5. The execution time of benchmarks implemented using the backend access technique in the tested processors and speedup with respect to baseline TCF as the function of problem size *N*. In the mread and mwrite benchmarks, the x-axsis denotes the number references per concurrent access.

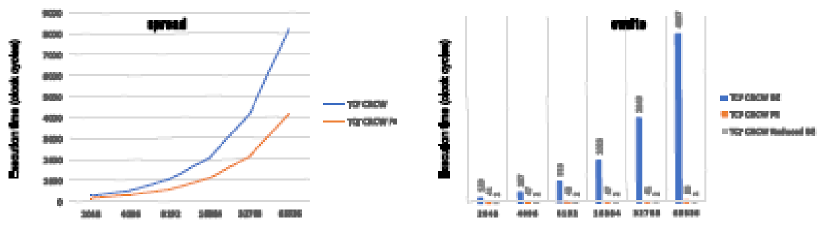


Fig. 6. The differences in execution time of the spread and cwrite benchmarks implemented with the backend access technique and frontend techniques.

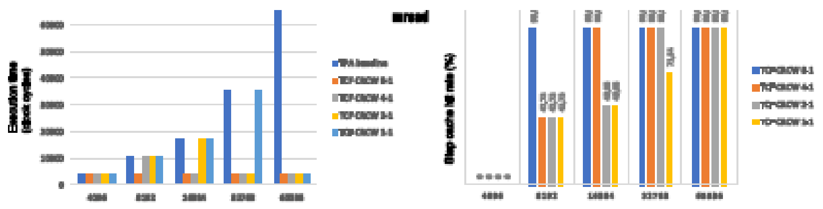


Fig. 7. The execution time and step cache hit rate in the proposed processor for mread with different step cache sizes and associativity 1 (direct mapping) for the mread benchmark.

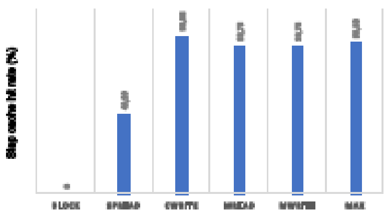


Fig. 8. The step cache hit rate for benchmarks *N*=65536 (for max *N*=256).

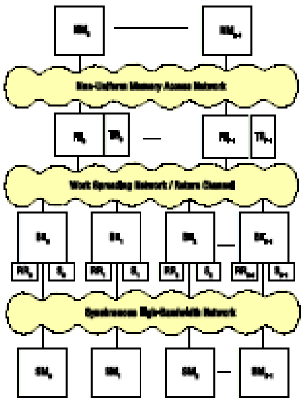
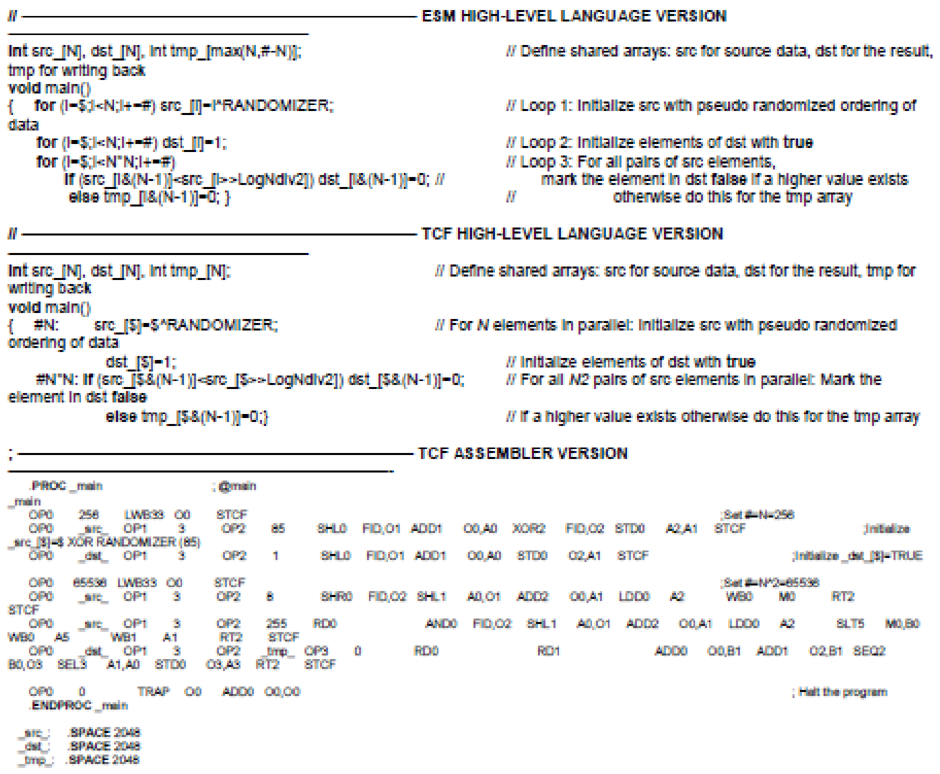

Non-Uniform Memory Access Network
Work Spreading Network / Return Channel
Synchronous High-Bandwidth Network

Fig. 9. The overall structure of TPA implementing the proposed concurrent memory access technique (FE=processor frontend, NM=NUMA memory module, TB=TCF buffer, BE=processor backend unit, RR=replicated register block, S=step cache, SM=shared memory module).



```
// ──────────────────────────────────── ESM HIGH-LEVEL LANGUAGE VERSION
──────────────────────────
int src_[N], dst_[N], int tmp_[max(N,#-N)];                // Define shared arrays: src for source data, dst for the result,
tmp for writing back
void main()
{    for (i=$;i<N;i+=#) src_[i]=i*RANDOMIZER;               // Loop 1: Initialize src with pseudo randomized ordering of
data
     for (i=$;i<N;i+=#) dst_[i]=1;                         // Loop 2: Initialize elements of dst with true
     for (i=$;i<N*N;i+=#)                                  // Loop 3: For all pairs of src elements,
         if (src_[i&(N-1)]<src_[i>>LogNdiv2]) dst_[i&(N-1)]=0; //     mark the element in dst false if a higher value exists
          else tmp_[i&(N-1)]=0; }                          //              otherwise do this for the tmp array

// ──────────────────────────────────── TCF HIGH-LEVEL LANGUAGE VERSION
──────────────────────────
int src_[N], dst_[N], int tmp_[N];                    // Define shared arrays: src for source data, dst for the result, tmp for
writing back
void main()
{    #N:       src_[$]=$^RANDOMIZER;                  // For N elements in parallel: Initialize src with pseudo randomized
ordering of data
               dst_[$]=1;                                  // Initialize elements of dst with true
     #N*N: if (src_[$&(N-1)]<src_[$>>LogNdiv2]) dst_[$&(N-1)]=0;  // For all N2 pairs of src elements in parallel: Mark the
element in dst false
               else tmp_[$&(N-1)]=0;}                      // If a higher value exists otherwise do this for the tmp array

; ──────────────────────────────────── TCF ASSEMBLER VERSION
──────────────────────────────────────────
   .PROC _main                    ; @main
_main
   OP0   256    LWB33  O0    STCF                                                                ;Set #=N=256
   OP0   _src_  OP1    3     OP2   85     SHL0  FID,O1  ADD1  O0,A0  XOR2  FID,O2  STD0  A2,A1  STCF        ;Initialize
_src_[$]=$ XOR RANDOMIZER (85)
   OP0   _dst_  OP1    3     OP2   1      SHL0  FID,O1  ADD1  O0,A0  STD0  O2,A1  STCF                ;Initialize _dst_[$]=TRUE

   OP0   65536  LWB33  O0    STCF                                                                ;Set #=N^2=65536
   OP0   _src_  OP1    3     OP2   8      SHR0  FID,O2  SHL1  A0,O1  ADD2  O0,A1  LDD0  A2   WB0   M0    RT2
STCF
   OP0   _src_  OP1    3     OP2   255    RD0          AND0  FID,O2  SHL1  A0,O1  ADD2  O0,A1  LDD0  A2   SLT5  M0,B0
WB0   A5        WB1    A1    RT2   STCF
   OP0   _dst_  OP1    3     OP2   _tmp_  OP3   0      RD0          RD1          ADD0  O0,B1  ADD1  O2,B1  SEQ2
B0,O3  SEL3  A1,A0  STD0  O3,A3  RT2   STCF

   OP0   0           TRAP  O0    ADD0  O0,O0                                                     ; Halt the program
   .ENDPROC _main

  _src_:  .SPACE 2048
  _dst_:  .SPACE 2048
  _tmp_:  .SPACE 2048
```

Fig. 10. Implementation of the max benchmark in high-level ESM, high-level TCF-aware language and TCF assembler. Blue color indicates execution in backends.

Jesper Larsson Träff is professor for Parallel Computing at TU Wien (Vienna University of Technology) since 2011. From 2010 to 2011 he was guest professor for Scientific Computing at the University of Vienna. From 1998 until 2010 he was working at the NEC Laboratories Europe in Sankt Augustin, Germany on efficient implementations of MPI for NEC vector supercomputers; this work led to a doctorate (Dr. Scient.) from the University of Copenhagen in 2009. From 1995 to 1998 he spent four years as PostDoc/Research Associate in the Algorithms Group of the Max-Planck Institute for Computer Science in Saarbrücken, and the Efficient Algorithms Group at the Technical University of Munich. He received an M.Sc.. in computer science in 1989, and, after two interim years at the industrial research center
ECRC in Munich, a Ph.D. in 1995, both from the University of Copenhagen.

Jussi Roivainen received M.Sc. in Electrical Engineering from University of Oulu, Finland in 1999. He is a Senior Scientist at VTT, Efficient Computation and Communications, Oulu, Finland. His research interests include digital logic implementations of processors and baseband.

Martti Forsell received M.Sc., Ph.Lic., and Ph.D. degrees in computer science from University of Joensuu, Finland in 1991, 1994, and 1997, respectively. He has acted as a lecturer, researcher, and acting professor in the Department of Computer Science, University of Joensuu. Currently he is a Principal Scientist of processor architecture and parallel computing at VTT, Oulu, Finland, as well as an Adjunct Professor of computer architecture in the Faculty of Information Technology and Electrical Engineering at the University of Oulu.

Ville Leppänen is a professor in software engineering and software security at the University of Turku, Finland. He has over 200 international conference and journal publications. His research interests are related broadly to software engineering and parallelism, ranging from software engineering methodologies, practices, and tools to security and quality issues, and to programming languages, parallelism, and architectural design topics. Currently Leppänen serves as vice head of department and leader of 7 research and development projects.