Supporting Concurrent Memory Access in TCF-aware Processor Architectures

Martti Forsell, Jussi Roivainen
Computing Platforms
VTT
{Martti.Forsell|Jussi.Roivainen}@VTT.Fi

Ville Leppänen
Information Technology
University of Turku
Ville.Leppanen@UTU.Fi

Jesper Larsson Träff
Faculty of Informatics
Vienna University of Technology
traff@par.tuwien.ac.at

***Abstract*—The Thick Control Flow (TCF) model packs together self-similar computations to simplify parallel programming and to eliminate redundant usage of associated software and hardware resources. While there are processor architectures supporting native execution of programs written for the model, none of them support concurrent memory access that can speed up execution of many algorithms by a logarithmic factor. In this paper, we propose an architectural solution implementing concurrent memory access for TCF-aware processors. The solution is based on bounded size step caches and two-phase structure of the TCF-aware processors. Step caches 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 2-phase structure reduces some concurrent accesses to a frontend operation followed by broadcast in the spreading network. According to our evaluation, a concurrent memory access-aware B-backend unit TCF processor executes certain algorithms up to B times faster than the baseline TCF processor.**

***Keywords*—parallel computing; processor architecture; programming model; TCF; concurrent memory access**

I. Introduction

The standard programming model of current multicore processors provides a set of computational threads that execute individual code asynchronously and independently of each other. In the case of interdependencies, threads need to communicate with each other using messages or shared memory. The lack of synchrony implies that execution of threads needs to be orchestrated by explicit synchronization constructs, e.g., barriers, locks and atomic operations, to guarantee that critical operations are carried out in the right order. Since the cost of using these constructs in terms of time and hardware resources is high, it is clear that many sophisticated and efficient primitives of parallel computation are ruled 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 most computational problems contain a lot of parallelism, the number of needed threads can be much higher than that supported by current and future hardware. While software based threading systems support more threads than the hardware, in practice, the number of software threads is 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 programmer is forced to emulate high parallelism with the few available hardware threads by using blocking, looping or repetition. This kind of architecture-driven ambiguity of the state of computation and lack of native parallel computing capabilities tend to make programming error-prone, complex and cause the processor hardware to do redundant computation, e.g., by repeating base address computations and allocating registers containing replicated values for the threads. A promising way to eliminate these model-related problems is the *thick control flow* (TCF) programming model [1,2] packing user adjustable number of threads following the same control flow into a single entity and guaranteeing synchronous execution independently of the number of computational elements. The TPA chip multiprocessor architecture [3] 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 [4].

Supporting concurrent memory access as a primitive of parallel computation is meaningful only for the class of architectures that supports synchronous execution of threads, e.g., so-called *Emulated Shared Memory* (ESM) architectures that use multithreading to hide the (distributed) shared memory system access and provide low-cost synchronization mechanisms [5,6]. The convention for deciding which of the threads performing a concurrent write succeeds, is traditionally either ARBITRARY or PRIORITY, by which the thread with the lowest identifier succeeds [7].

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

- **Combining networks.** ESM machines that utilize light-weight interleaved multithreading along with low-cost synchronization to emulate an ideal shared memory [5]. The main idea is to reduce the needed bandwidth by combining the references targeted to the same location in the network. Requires sorting of memory requests prior to injection that decreases the speed of this solution for all memory accesses.
- **Streamlined combining networks.** This operates like [5] but manages to reduce the number of routing phases from six used in [5] to five and reducing the number of memory modules [6]. Unfortunately, also this requires the same sorting phase as the non-streamlined combining networks.
- **Active memories.** Implements limited and partial concurrent memory access [8]. This supports concurrent memory access for a limited number of special memory locations. Compared to the previous attempts, this solution, however, eliminates the need for sorting.

978-1-5386-2844-7/17/$31.00 ©2017 IEEE

- **Step caches.** This solution implements full concurrent memory access for all memory locations [9]. The idea is to filter out everything except the first reference to each location per step and thus reduce memory traffic. The limited associativity of the step cache requires 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 can be used to provide concurrent memory access for TCF-aware 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 fibers 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.

In this paper, we propose an architectural solution implementing concurrent memory access for TCF-aware processors. The solution is based on bounded size step caches and two-phase structure of the TCF-aware processors. Step caches 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 2-phase structure reduces some concurrent accesses to a frontend operation followed by a broadcast in the spreading network. According to our evaluation, a concurrent memory access-aware B-backend unit TCF processor executes certain algorithms up to B times faster than the baseline TCF processor. In Section 2, we describe the TCF model and TPA architecture, Section 3 proposes support for concurrent memory access in TCF-aware architectures, Section 4 evaluates the proposed solutions with simulations in TPA, and Section 5 draws conclusions and outlines future work.

II. TCF Model and TPA Architecture

The *Thick Control Flow* model joins threads following the same control flow into entities called *thick control flows* (TCF) [1, 2] (see 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 guaranteed to complete before the current step starts and their results are 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 (see Figure 1).These challenges are addressed by the *Thick Control Flow Processor Architecture* (TPA) [3]. It uses a two-level structure with frontend and backend units. Fibers are executed dynamically on the backend pipeline. The backend units have a special replicated register block scheme connected to an external overflow mechanism. Standard ESM implementation techniques are used for streamlined shared memory multiprocessor execution. 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 (see Figure 2). The frontend units are connected to a *non-uniform memory access* (NUMA) style memory system for low-latency access to locality-aware memory and the backend units are connected to distributed shared memory system modules via a small number of parallel mesh networks.

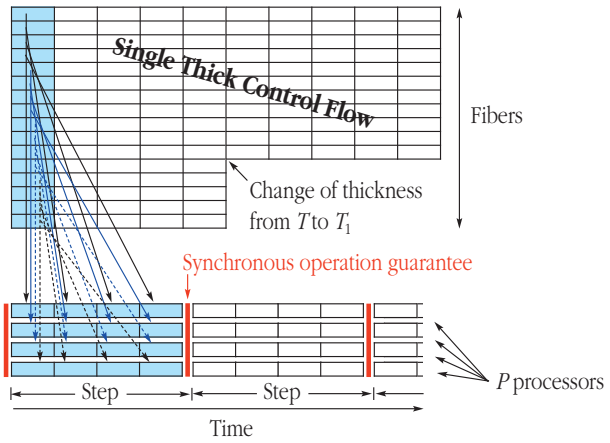

Figure 1. A thick control flow, change of thickness from 16 to 11 and execution as steps. A step of execution in a TCF-aware processor. The machinery assignings T=16 fibers for execution in P=4 processors and guarantees synchronicity between the steps.

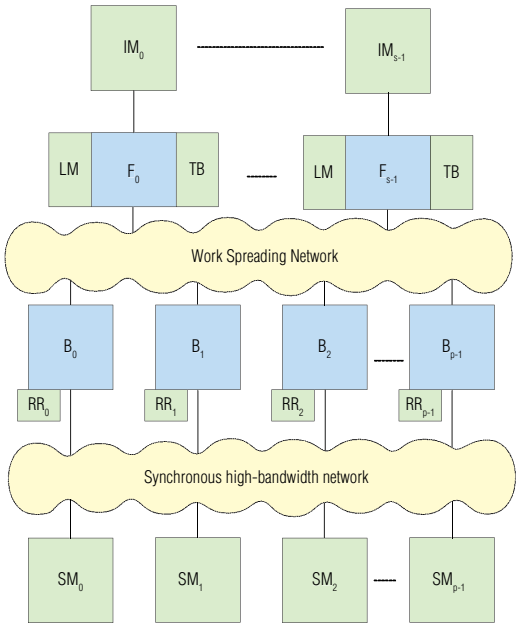

Figure 2. The overall structure of TPA (IM=instruction memory, F=processor frontend, LM=local memory, TB=TCF buffer, B=processor backend unit, RR=replicated register block, SM=shared memory module). The external memory system is not shown.

Execution of a TPA instruction happens in three frontend and three backend phases:

For each active frontend **do**
F1. Select the next TCF from the TCF buffer if requested.
F2. Fetch an instruction pointed by the PC of the TCF.
F3. Execute the subinstructions in the functional units specified by the instruction. If the instruction contains a backend part, select operands and send them along with the instruction to the backends assigned to the frontend via the work spreading network. Ask for a TCF switch if requested by the subinstruction.

For each backend **do**
B1. If the backend is not occupied, fetch the next TCF from the spreading network and determine the fibers to be executed in the backend.
B2. Generate the fibers of the TCF to be pipelined according to the assignment determined in the phase B1.
B3. For each fiber:
B3.1 Select the operands from the received frontend data and replicated register block.
B3.2 Execute the backend subinstructions.
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.

III. Concurrent Memory Access

Depending on the targeted concurrent memory access model and architectural realization of TCF-model, there are alternative schemes: For example, if only a single PRIORITY mode concurrent memory access per a TCF is allowed, the frontend can easily and cost-efficiently take care of the concurrent access assuming the frontend has access to the shared memory system of the backends. On the other hand, if there is a high number of ARBITRARY mode accesses in parallel, it is preferable for the backend units to take care of them. Architecturally speaking, the key challenge of concurrent memory access in a TCF-aware processor is to avoid explosion of resource usage as storage needed by the existing solutions relying on step cache and reply buffer entry per thread would need to scale up without bounds [8, 9] (see Figure 3/left). Our solution is to exploit the possibility to change the thicknesses of the TCFs, apply bounded size step caches and reply buffers as well as the mechanism returning data from backends to the frontend to support thick concurrent memory access operations.

We first look at the cases in which all the fibers of a TCF are participating to the same concurrent memory access, or only one concurrent access per TCF is allowed. In the case of read, we can use the frontend to fetch the value from the shared memory so that it is available to all the backends via the work spreading network (see Figure 3/right). In its turn, a concurrent write can be handled by just setting the thickness to one for a single instruction and performing just a single write. This applies to both ARBITRARY and PRIORITY modes. Alternatively, if a copy of the value is in the frontend, it can take care of the write. There are three obvious ways to implement this kind of a *frontend access technique* to the shared memory system: a dedicated frontend port to the shared memory, sharing of the memory port with a backend unit, and commanding 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 of the TPA [3]. For this paper, we use the backend variant since a dedicated port needs potentially substantial modifications to the interconnection network and its topology. The second possibility is quite close to the third one but requires a separate FIFO with explicit synchronization messages for orchestrating the access to the steps of execution. There are two optimizations within the backend variant: 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.

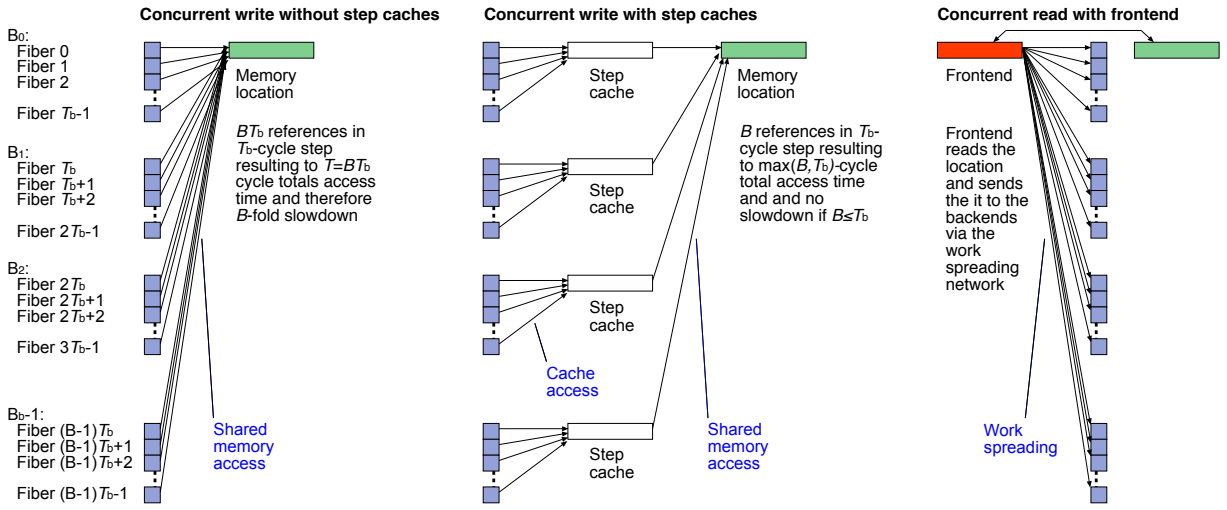

Figure 3. Concurrent in the baseline system (with no step caches), concurrent write with step caches and concurrent read with a the fronend.

The case of having multiple concurrent reads per TCF is more challenging since we cannot make requests for reading them with the frontend based on compile time knowledge 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 a help of bounded size step cache (see Figure 3/center). Let us call this the *backend access technique*. Since the thickness can be arbitrarily high, it is possible that the cache line or reply buffer entry containing the accessed value gets overwritten while there are references to the same location still coming by the remaining fibers. In that case, a new step cache entry allocation is made when the first new reference is executed. If the reply buffer entry for a certain concurrent access is in danger of being overwritten, then the step cache line referring to that needs to be updated with a new reply buffer entry early enough to prevent overwriting. The case of multiple concurrent writes operates similarly, but there are no reply buffer allocation that would retire along with the step cache line update.

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 close to T/B which is the minimum execution time of a step of execution. Otherwise the number of fibers participating a concurrent access will slow down the execution time of the corresponding step.

IV. Evaluation

To show that concurrent memory access-aware TCF processors meet the expectations, we evaluated the performance with the proposed technique and discuss implementation issues.

A. Performance

In order to determine the performance of the proposed technique, we measured the execution time of six kernel benchmarks representing different usage schemes of concurrent memory access (see Table 1). We used different problem size N and/or different data set in the baseline TCF-aware architecture (denoted TCF baseline), in a TCF architecture including the techniques of Section III (denoted TCF CRCW). For reference purposes, we also measured the performance of configurable ESM REPLICA architecture [10] (denoted CESM) with the same benchmarks. REPLICA applies the fixed threading scheme (T_p threads per processor) and employing the step cache technique representing the best known concurrent memory access-aware architecture with the same set of benchmarks. For the summary of the measured architectures, see Table 2.

The benchmarks were executed in clock accurate simulators modeling the architecture down to low-level details [3]. 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 inter-thread/fiber synchronization. The amount of memory was set large enough for holding all the data needed in all the tests.

The results of the simulations are shown as execution times in clock cycles and speedup w.r.t. baseline TCF. We show results also as a function of problem size N except for the mread and mwrite benchmarks where the performance is shown as a function of references per concurrent access (see Figure 4). 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 if we ignore the synchronization overhead of $(N/B+1)/(N/B)$, 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. 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 intact as we go from exclusive memory access gradually to fully concurrent access.
- The proposed technique is faster than CESM with step caches in all benchmarks mainly because while it can embed the looping costs needed to map the software threads to actual hardware threads, this is not possible for loop initializations, which

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*) [13]

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

Processor	CESM [10] (The best known CRCW)	TCF baseline [3] (Baseline)	TCF CRCW (This proposal)
Processing units	16 NUMA/16 Parallel	1 frontend/16 backend	1 frontend/16 backend
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	-/-/-	128/4-way set associative/random
Interconnect	4x4 mesh	4x4 mesh	4x4 mesh

Table 2. Tested processors.

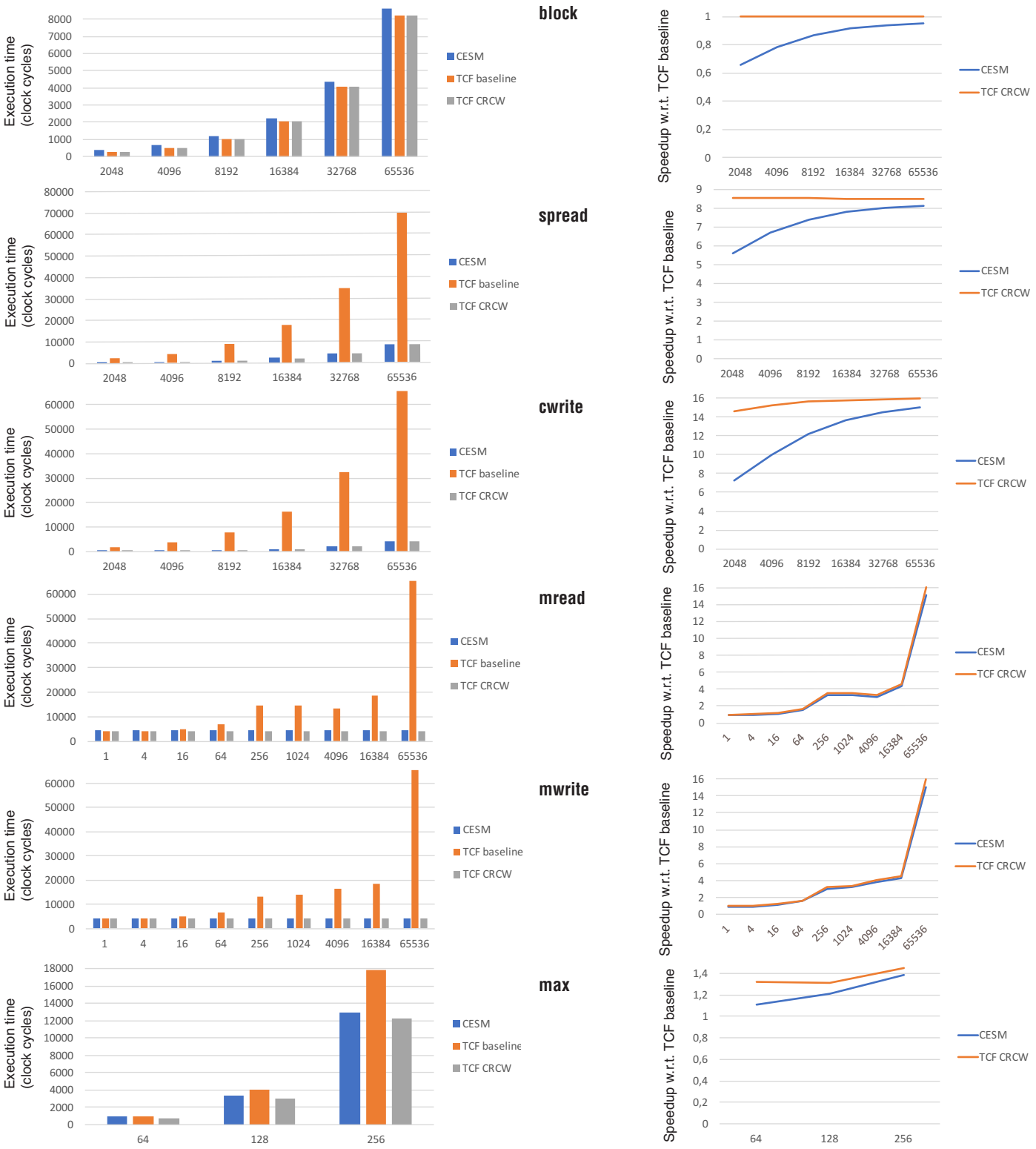

Figure 4. The execution time in the tested processors and speedup with respect to baseline TPA as the function of problem size N for the benchmark programs. In the mread and mwrite benchmarks, the x-axsis denotes the references per concurrent access.

take at least T_p cycles in CESM, where T_p is the number of threads per processor.

In order to see the practical difference between the fronend and backend access techniques, we implemented the spread benchmark also with the frontend. For that, we measured the execution time as a function of N and compared it to the backend access technique (see Figure 5). The results indicate speedups ranging from 55% to 98% and we conclude that for this kind of functionalities, the frontend technique can speed up functionalities containing single concurrent reads considerably.

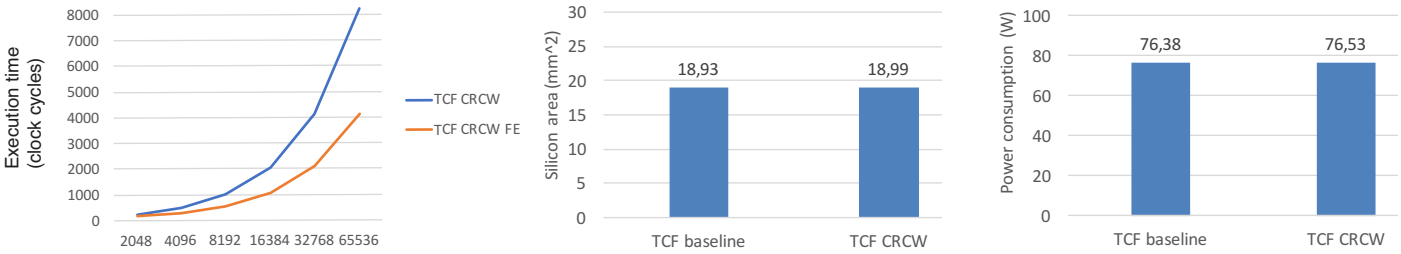

Figure 5. The differences in execution time of the spread benchmark implemented with the backend only access technique and frontend access technique (left), the silicon area (middle) and power consumption estimates (right) for 11 nm silicon process.

The limitations of the step cache technique resemble those of caching in general. While step caches do not suffer from coherence problem, 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.

B. Implementation considerations

Implementing the proposed technique in TPA does not require major modifications to the baseline architecure. One needs to add a bounded size step cache that works with the existing bounded size reply buffer-aware allocation policy for each backend, return channel mechanism and frontend access to the shared memory that can be also used for many other purposes.

We estimated the complexity the proposed technique by using our detailed analytical *performance-silicon area-power consumption* (PAP) model that breaks the processor to components down to gate estimates, adopts the parallel signal propagation model on silicon [11], and uses estimates given by ITRS roadmaps for realistic values of certain silicon process parameters [12]. According to it a 64-bit, 16-backend TPA with 10 backend functional units running at 1.5 GHz and containing 72.9 MB on-chip memory, implemented with an 11 nm silicon technology would occupy 18.93 mm^2 and consume 76.38 W without step caches. Adding a generous 1024-line step cache to each backend would only increase the silicon area by 0.29% and power consumption by 0.20% (see Figure 5). Note that with these parameters the memory system is estimated to occupy 85.1% of the total chip area and consume 66.0% of the overall power.

V. Conclusions

We have described an architectural solution to realize concurrent memory access for TCF-aware processors. The solution is based on bounded size step caches and a 2-phase structure of the TCF-aware processors. Step caches capture and hold the references made during the current step of execution that are independent by the definition TCF execution and therefore avoid coherence problems. If there is only a single 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 that the baseline TPA. Employing the frontend in the case of single 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.

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 multi(prefix)operations on TCF-aware architectures.

Acknowledgment

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

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

[7] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, *Proc. 10th ACM STOC*, New York, 1978, 114-118.

[8] M. Forsell, Realising constant time parallel algorithms with active memory modules, *International Journal of Electronic Business* **3**, 3-4 (2005), 255-263.

[9] 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.

[10] M. Forsell, J. Roivainen and V. Leppänen, Prototyping the MBTAC processor for the REPLICA CMP, *Proc. 16th Workshop on Advances in Parallel and Distributed Computational Models (APDCM'14)*, May 19, 2014, Phoenix, USA.

[11] 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.

[12] International Technology Roadmap for Semiconductors (ITRS), Semiconductor Industry Association (SIA); http://www.itrs.net.

[13] Y. Shiloach and U. Vishkin, Finding the Maximum, Merging and Sorting in a Parallel Computation Model, Journal of Algorithms 2, (1981), 88-102.