Microprocessors and Microsystems 98 (2023) 104807

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

# Realizing multioperations and multiprefixes in Thick Control Flow processors[☆]

Martti Forsell [a,*], Jussi Roivainen [a], Ville Leppänen [b], Jesper Larsson Träff [c]

[a] VTT, Finland
[b] Faculty of Technology, University of Turku, Finland
[c] Faculty of Informatics, TU Wien (Vienna University of Technology), Austria

ARTICLE INFO

*Keywords:*
Parallel computing
Multiprocessor architecture
Thick Control Flow
Multioperation
Multiprefix

ABSTRACT

Multioperations are primitives of parallel computation by which threads perform reductions, e.g., additions, on values provided by multiple threads into a single value in a constant number of steps. Multiprefixes resemble multioperations, but return to each participating thread a cumulative ordered reduction of all preceding values. Algorithmically, multioperations and multiprefixes can speed up parallel programs by a logarithmic factor over their single operation counterparts. In this paper, we introduce architectural techniques for realizing multioperations and multiprefixes in so-called Thick Control Flow (TCF) processors. A thick control flow is a computational construct that bundles homogeneous threads following the same control path into a data parallel entity. Our proposed processors optimized for executing TCFs feature a frontend-backend structure with low-latency processing of TCF-common computations and high-throughput execution of data parallel parts. Our solution relies on step caches and equally sized multioperation scratchpads, while on the memory side, we make use of active memory modules. The idea is to compute partial results in backend units to reduce the traffic to the referred shared memory location. The final multioperation result is then computed in the active memory unit of the target memory module. Multiprefixes use an additional phase where the final results are computed with a help of backend-wise prefixes. According to our evaluation, the proposed techniques indeed speed up certain $N$ data element algorithms by a $\log N$ factor with reasonable hardware costs.

## 1. Introduction

*Reductions* are patterns of parallel computation in which a set of values provided by a set of threads are reduced to a single value in some order (usually determined by a thread numbering or thread priority) by applying a (usually associative) binary operation. *Multioperations* are such reduction operations, where several reductions can be carried out concurrently by disjoint sets of threads. A trivial way to implement a reduction in parallel is by a tree-shaped computation in which the binary operation is repeatedly applied in parallel on pairs of non-reduced values until there is only one value left. With $N$ threads this takes $O(\log N)$ parallel time steps, and is therefore called a *logarithmic (time) reduction algorithm*. We also observe that the number of threads doing active computations reduce from $N/2$ down to 1 during the execution of the algorithm (see Fig. 1). Execution of this algorithm on current multicore processors with $P$ individual processor cores is inefficient since there is typically only a very small number of hardware threads available, and doing thread switching and synchronization by software twice per parallel iteration incurs a huge penalty compared to the execution time of the binary operation. A faster and more cost-efficient solution is to divide the $N$ values into $P$ blocks, perform block-wise reductions in parallel with all $P$ processor cores, and determine the final result using the logarithmic algorithm or even a sequential one if it turns out to be faster (see Fig. 1). A typical execution time for this *blocking reduction algorithm* is either $O(N/P + P)$ or $O(N/P + \log P)$, if the logarithmic algorithm is used to reduce the block results; but again the number of active processor cores changes along the run of the algorithm.

In this work we define *multioperation instructions* as hardware-assisted implementations of reductions. They apply multiple associative binary operations to obtain one or more reduction result values. The

[☆] This paper is a revised and significantly extended version of the paper "Implementation of Multioperations in Thick Control Flow Processors" presented at the Advances in Parallel and Distributed Computional Models workshop at the International Parallel and Distributed Processing Symposium, 2018 (Forsell et al., 2018) [1] with support for multiprefix operations, more detailed exposition of the proposed techniques, and additional, new benchmarks and measurements.

[*] Corresponding author.

*E-mail addresses:* Martti.Forsell@vtt.fi (M. Forsell), Jussi.Roivainen@vtt.fi (J. Roivainen), Ville.Leppanen@utu.fi (V. Leppänen), traff@par.tuwien.ac.at (J.L. Träff).

https://doi.org/10.1016/j.micpro.2023.104807
Received 7 January 2022; Received in revised form 15 October 2022; Accepted 18 February 2023
Available online 22 February 2023

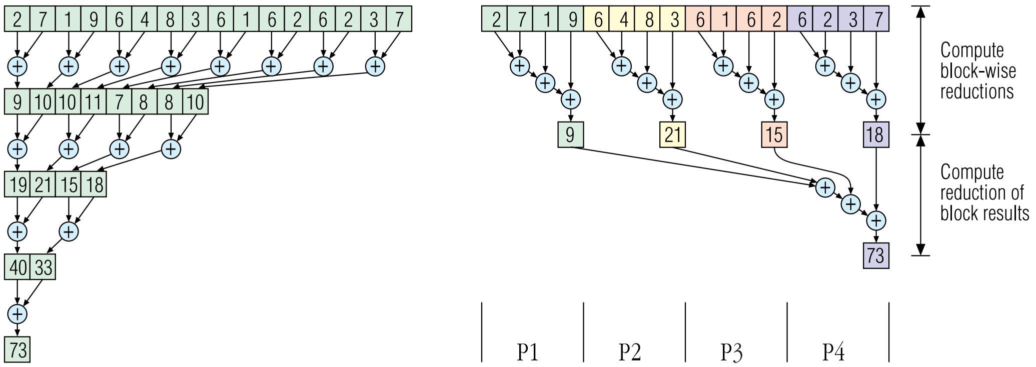


**Fig. 1.** The logarithmic algorithm (left) and blocking algorithm (right) implementing additive reduction of 16 values on a 4-core system.

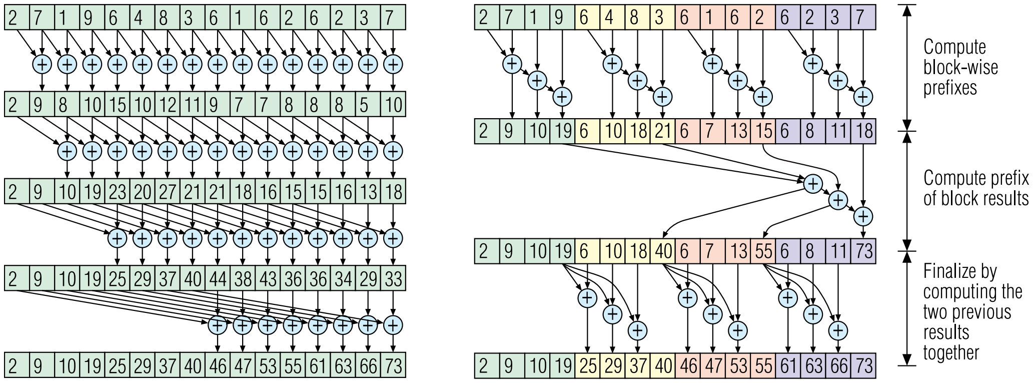


**Fig. 2.** The logarithmic algorithm (left) and blocking algorithm (right) implementing multiprefix addition of 16 values on a 4-core system.

values are defined by specified target addresses of the data that are associated with one or more sets of threads. The obtained result value(s) will reside in memory after the execution of a multioperation instruction while the operands can be read from registers or from memory depending on the variant of the multioperation instruction.

*Multiprefix* operations are similar to multioperations, but return a cumulative ordered reduction of all preceding elements for each participating thread. Like multioperations, multiprefix operations can also be implemented with both a logarithmic and a blocking algorithm (see Fig. 2). The former is an iterative algorithm, which at iteration $I$ applies the binary operation to the $N - 2^I$ rightmost elements with the first operands coming from element placed $2^I$ positions to the left. The execution time is $O(\log N)$ parallel time steps. The latter algorithm takes three phases since the results of the second phase need to be combined with the results of the first phase to obtain the final results [2]. A typical execution time for this *blocking multiprefix* algorithm is again either $O(2N/P + P)$ or $O(2N/P + \log P)$ (with the constant 2 included to emphasize the two block phases).

We define here *multiprefix instructions* as hardware-assisted implementations of multiprefixes. They apply multiple associative binary operations to obtain one or more multiprefixes (results). The input data can reside either in memory or in registers depending on the variant of the instruction. Likewise, the obtained result values will reside either in memory or in registers of the processor after execution of a multiprefix instruction depending on the variant of the multiprefix instruction.

Constant time multioperation and multiprefix instructions can increase the performance of a parallel machine by a logarithmic factor for classes of algorithms containing reductions and multiprefixes [3]. Thus, one would expect to see the same kind of performance boost in practical computers. Supporting multioperation and multiprefix instructions is, however, meaningful only for the class of architectures supporting synchronous execution of threads, e.g., so-called *Emulated Shared Memory* (ESM) architectures [4–10], that use multithreading to hide the (distributed) shared memory system access latency, and low-cost synchronization mechanisms to support lock-step synchronous execution. Previous attempts to support multioperations and multiprefixes in ESM architectures include:

**Combining networks.** These implement multioperations and multiprefixes for ESM machines that utilize light-weight interleaved multithreading along with low-cost synchronization to emulate an ideal shared memory [11]. The main idea is to combine references targeted to the same memory location when they meet in the intercommunication network. Unfortunately, this technique requires sorting of the memory requests prior to injection into the network which effectively doubles the latency for all memory accesses. In addition, a buffer for holding the combined load/multiprefix messages is needed for each interconnection network switch.

**Streamlined combining networks.** These implement multioperations and multiprefixes for ESM machines [8] and operate like combining networks. They reduce the number of routing phases from six used in [11] to five and in addition also reduce the number of memory modules. Unfortunately also this requires the same sorting phase as non-streamlined combining networks.

**Active memories.** These implement limited and partial concurrent memory access [12] and can support multioperations for a limited number of special memory locations. The key idea is to mimic the blocking algorithm by calculating the block-wise results within the processor cores in parallel and then determine the final result with a help of block results. Compared to the previous ideas, this solution eliminates the need for sorting and combining machinery in the network.

**Step caches and scratchpads.** These implement multioperations and multiprefixes for all memory locations in ESM machines [13]. The main algorithm is the same as with active memories; but special associative memories called step caches are used to recognize references belonging to the same operation and scratchpads are used to retain the block-wise results. The associativity of the step cache, however, limits the applicability of this solution. Resending of partial results to the target address is needed if it has been wiped out from the step cache due to a set overflow. Also this solution eliminates the need for sorting prior to injection of references to the network.

The performance of existing ESM architectures [8,9,14] even with these extensions is, however, not optimal for algorithms with low and non-matching parallelism. To address this problem and still retain the benefits of ESM for functionalities with sufficient parallelism, we have introduced the thick control flow programming model and processor architecture for emulating execution of workloads with arbitrary number of simplified thread-like computational elements. A *Thick Control Flow* (TCF) [15] is a computational abstraction that bundles any number of homogeneous threads following the same control path into a vector-like entity. The elements of a TCF are called *fibers* to distinguish them from ordinary threads having their own individual control. Efficient execution of TCF programs requires unique hardware, in which common parts of TCFs are assigned to frontend execution units providing low latency, and individual fibers are assigned to high-throughput parallel backend execution units [16]. Using a programming language supporting TCFs, an example of a parallel program calculating the logarithmic multiprefix sum shown in Fig. 2 with a single TCF can now be written into a compact form:

```
int i, A_[N];
for (i=1, #N-1; i<N; #-=i, i<<=1)
    A_[$+i] += A_[$];
```

The program declares a scalar loop variable **i** that is common to all fibers and an N-element shared array **A_**. The **for**-loop initializes **i** with value 1 and the thickness of the TCF (#) with N-1. During each iteration for each fiber in parallel, the fiber **$**, $\$ \in \{0, 1, .., \# - 1\}$, adds **A_**[$] to **A_**[$+i] in parallel and decreases the thickness by **i** and doubles **i**. The loop ends when the condition $i < N$ does not hold any more. There is no need to add synchronizations after parallel operations since the fibers of the TCF are executed synchronously.

According to our investigations [16–18], use of TCFs simplifies parallel programming, improves utilization of hardware resources, and greatly increases the performance over industry standard multicore processors if suitable processor, interconnect and memory system architectures are used. TCF processors turned to widely accelerate execution of parallel parts of the code, especially if there are dependencies between the parallel components of the executed program or shared memory access patterns are non-trivial. We expect therefore TCF processors to be important in application domains, such as personal computing, virtual and augmented reality, artificial intelligence, machine learning, high-performance computing, automated cars, telecom and defence. Unfortunately none of the above mentioned multioperation and multiprefix techniques support these operations for variable, unbounded numbers of threads needed in TCF processors. Fig. 3 illustrates a parallel program written to employ 21 threads. If we try to bundle homogeneous threads of the program following the same control path together to form TCFs, we end up identifying three bundles visualized in the top of the figure. Bundle 0 is shown in middle of the figure as a TCF where operations are homogeneous and common operations are executed only once for the whole TCF and the thickness of computation is altered to improve the utilization of execution units. Executing instructions related to the TCF on a processor with a single frontend unit and four backend units is shown at the bottom of the figure. Common operations are executed in the frontend unit 0 while individual operations are executed at backend units. For instructions involving multiple homogeneous operations, such as Instruction 3 marked with blue color, the individual operations numbered from 0 to 15 are distributed evenly among the backend units and executed there. The figure shows the default stacked assignment or mapping of operations to backends. Steps of execution are shown for both the frontend and backends.

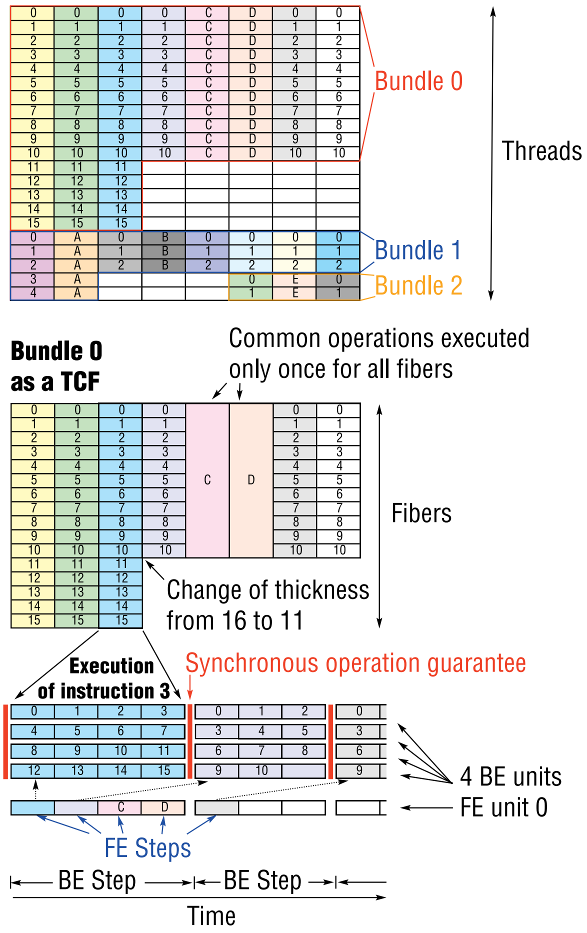

Bundle 0
Bundle 1
Bundle 2
Threads
Bundle 0 as a TCF
Common operations executed only once for all fibers
Fibers
Change of thickness from 16 to 11
Execution of instruction 3
Synchronous operation guarantee
4 BE units
FE unit 0
FE Steps
BE Step
BE Step
Time

**Fig. 3.** A parallel program written to employ 21 threads, bundling of threads for TCF formulation, a bundle as a TCF and execution of instructions of the TCF in a TCF processor with a single frontend (FE) units and four backend (BE) units as synchronous steps. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

In this paper, we propose architectural techniques for supporting multioperations and multiprefixes in Thick Control Flow (TCF) processors. On the processor side, our proposal relies on step caches and equal-sized multioperation scratchpads, while on the memory side, we make use of active memory modules. The idea is to compute partial results in the backend units to reduce the traffic to target memory location. The final multioperation result is then computed in the active memory unit of the target memory module, while multiprefixes need an additional phase where final results are determined with the help of backend-wise prefixes. According to the evaluation made with our TCF-aware processor equipped with multi(prefix)operation scratchpads and active memory units, it indeed executes certain $N$ data element algorithms $\log N$ times faster than the baseline processor. The cost of the realization is preliminarily evaluated.

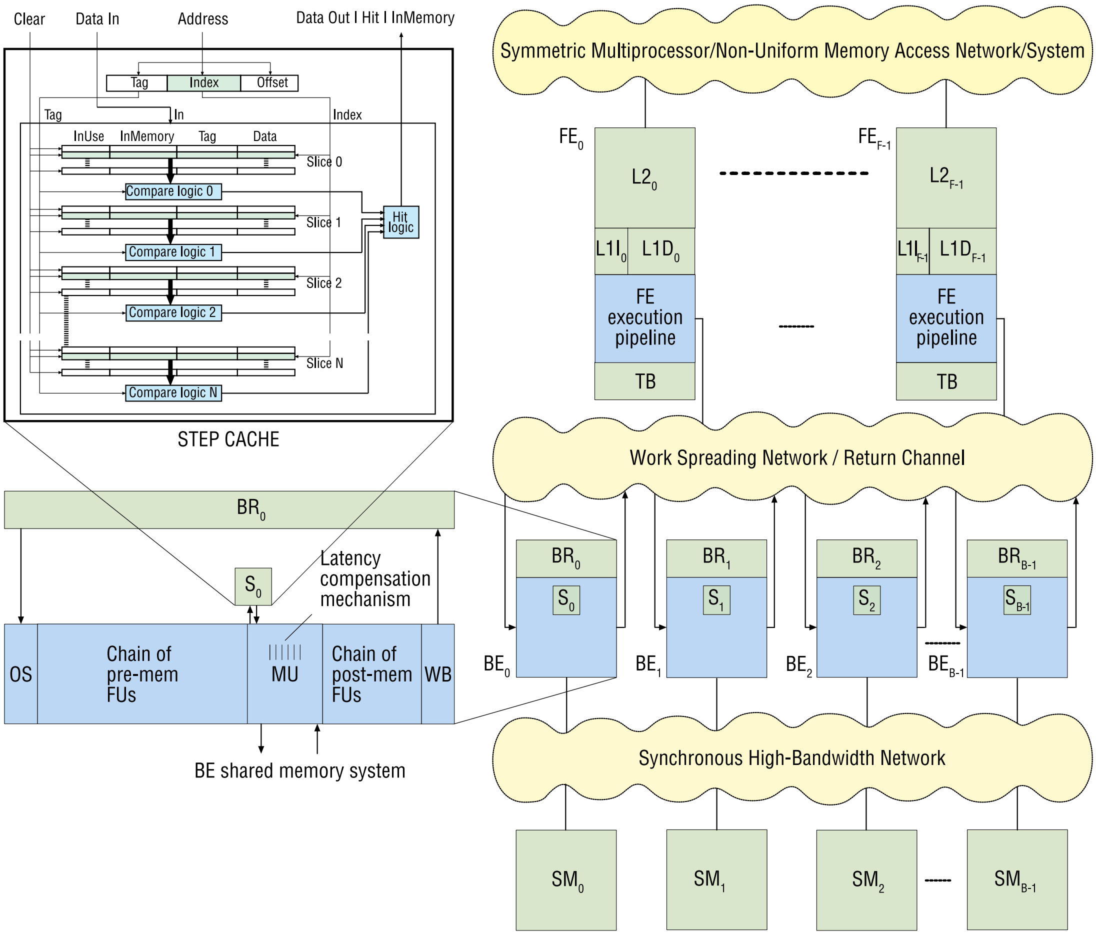


**Fig. 4.** The overall structure of the Thick Control Flow Processor Architecture with F FEs and B BEs (L1D = SMP/NUMA level 1 data cache, L1I = SMP/NUMA level 1 instruction cache, L2 = SMP/NUMA level 2 cache, FE = processor frontend unit, TB = TCF Buffer, BE = processor backend unit, OS = Operand Select, WB = Write Back, MU = Memory Unit, BR = Backend Register block, S = step cache and SM = Shared Memory module). The outer memory hierarchy, forwarding network and intermediate registers are not shown.

The rest of the paper is organized as follows: Section 2 describes TCF processors, Section 3 proposes support for parallel multioperations in TCF architectures, Section 4 does the same for multiprefixes, Section 5 evaluates the solution with simulations, and Section 6 draws conclusions and outlines future work.

## 2. TCF processors

According to our tests, TCFs execute poorly in current multicore processors [18] due to high threading and synchronization costs. Processing common parts of TCFs resembles executing threads in a multicore processor while handling individual fibers corresponds executing threads in an ESM processor. This motivates our frontend-backend organization, where common parts are processed in a frontend and individual fibers are computed in backends.

The *Thick Control Flow Processor Architecture* (TPA) [16] is our implementation of a TCF processor. It consists of *F* frontend (FE) units and *B* backend (BE) units connected together via a work spreading network (see Fig. 4). The FEs are connected to a typical *Symmetric Multiprocessor* (SMP)*/Non-Uniform Memory Access* (NUMA) style memory system and the BEs are connected to distributed shared memory system modules via a high-bandwidth multimesh network. TPA is a part of our multiprocessor framework aiming for high performance and easy programmability.

FEs aim at low-latency processing of TCF-level control and data common to all fibers. They borrow many features, including a number of parallel functional units (FU) and hierarchical cache-assisted SMP/NUMA memory system, from current CPUs. In this paper we assume that the FE uses a static superscalar execution scheme known as *Very Long Instruction Word* (VLIW) architecture, where instructions are composed of a number of subinstructions commanding explicitly the FUs, and where scheduling and register allocation happens at compile time [19,20]. However, utilizing a traditional dynamic superscalar architecture with out-of-order execution, register renaming and speculation would be possible as well if the interface with the BEs is implemented properly. A special requirement for the FEs is that they should support fast switching of TCFs. The inner memory system typically consists of level 1 instruction and data caches as well as level 2 unified caches. The details of the outer memory system are, however, out of the scope of this paper.

BEs have long pipelines for high-throughput processing of individual fibers. They utilize the ESM scheme augmented with the TCF support mechanisms to provide high performance and simple programmability. This is implemented via the latency compensation mechanism that employs interleaved multifibering to hide the shared memory system latency by executing other fibers while one is referring the memory if the number of fibers is large enough [16]. Low-cost synchronization is achieved by using wave synchronization where a wave-shaped group of special messages separates the references belonging to consecutive steps of execution and prevents violating the execution order [8,11]. To maximize exploitation of low-level parallelism, the FUs are organized as a chain so that subinstructions can use the results from preceding functional units as operands [9]. The shared Memory Unit (MU) is

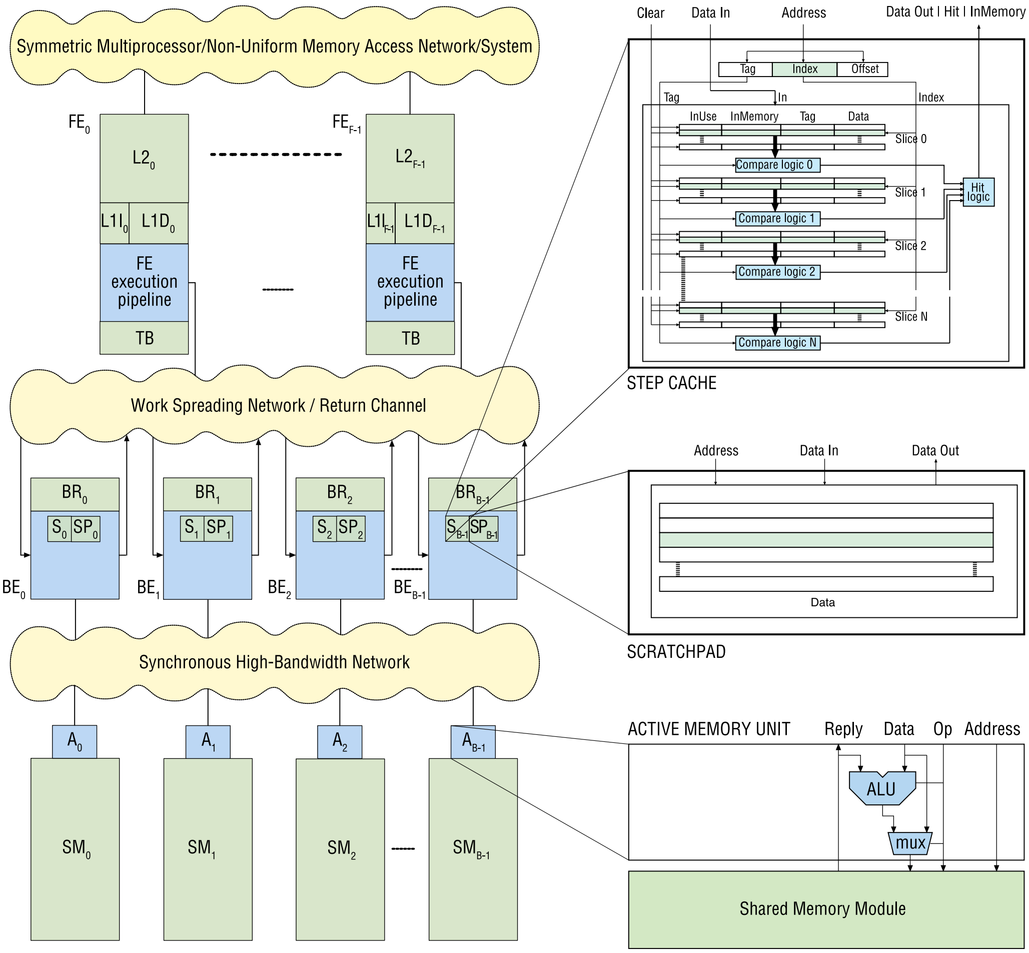


**Fig. 5.** (Left) The overall structure of the Thick Control Flow Processor Architecture with the proposed multioperation and multiprefix support (L1D = SMP/NUMA level 1 data cache, L1I = SMP/NUMA level 1 instruction cache, L2 = SMP/NUMA level 2 cache, FE = processor frontend unit, TB = TCF Buffer, BE = processor BackEnd unit, BR = Backend Register block, S = step cache and SM = Shared Memory module, SP = scratchpad, A = active memory unit). The outer memory hierarchy is not shown. (Right top) N-way set associative step cache. (Right middle) Scratchpad. (Right bottom) Active memory unit.

placed in the middle of the chain to enable both address computation prior to memory access and processing of memory data after it. This implies that code with inter-instruction dependencies can be executed within a step as long as there are suitable FUs available. TCFs are supported in BEs via flexible register block and pipeline mechanisms allowing arbitrarily thick TCFs and providing access to FE data via the work spreading network and return channel [16].

Execution of instructions belonging to a TCF happens in a single FE and optionally for each fiber synchronously in the BEs assigned for that particular TCF. As a result, execution of a step lasts potentially for a number of clock cycles. More specifically, we can identify the following three FE phases and three BE phases:

**For each** active FE **do**

**F1.** Select the next TCF from the TCF buffer if requested by the previous instruction or TCF management logic.

**F2.** Fetch a VLIW instruction pointed by the PC of the current TCF from the FE memory system.

**F3.** Execute the subinstructions specified by the instruction in the FUs including control transfer ones. Memory subinstructions are typically targeted to the FE memory system. If the instruction contains a BE part, select the operands and send them along with the part to the BEs assigned to the FE 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 or TCF management.

**For each** BE **do**

**B1.** When the BE has finished executing the previous instruction, fetch the next instruction from the spreading network and determine the fibers to be executed in the BE.

**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 FE data and BE register block.

**B3.2** Execute the BE subinstructions in a chained manner. Memory subinstructions targeted to the shared memory system are executed in the MU residing in the middle of the chain.

**B3.3** Write back the BE register block and send the optional reply data back to the FE via the return channel built into the spreading network.

After all active TCFs from the FE have been in execution for a single instruction, TPA issues a special synchronization TCF of thickness zero per BE that only sends and receives a synchronization message to/from the BE shared memory system.

TPA is a shared memory processors that is aimed to be used as a parallel central processing unit (CPU) of a computer system. It resembles processors that are optimized for processing multiple homogeneous components in parallel but unlike vector processors [21–23], SIMD units of chip multiprocessors [24] and hybrid architecture GPUs [25, 26], the vector size in our TPA is unbounded, execution of parallel components is synchronous over the entire machine and therefore BE-initiated parallel access fulfills the requirements for strict memory consistency. TPA therefore frees a programmer from the connection of software threads to hardware execution units (hardware threads) that limits the usage of parallelism in programs with inter-thread dependencies in current multicore processors.

The mapping of individual subinstructions from an FE to BEs happens according to a programmable mapping function. Fig. 3 shows an example of the stacked mapping but alternatives, such as interleaved mapping, can also be applied when necessary.

We also assume that the processors use concurrent memory access techniques implemented with step caches as described in [17]. These techniques allow parallel programs, where all fibers read from/write to a single memory location, to execute without any congestion overhead. *Step caches* are associative storage blocks holding copies of the latest references to the shared memory system [27] (see Fig. 4 top left). Unlike ordinary caches, there are no coherency issues in step caches since the lifetime of cache entries range only until the end of the current step and references within a step are independent by definition. The main idea of step caches is to reduce the traffic to target locations in the case of concurrent memory access so that a BE sends only one reference per step per accessed address. In a TPA, step caches are used by most memory subinstructions and placed as a part of the memory access machinery of BEs (see Fig. 4) so that a memory subinstruction utilizing the step cache can access it prior to making a decision whether an actual memory reference needs to be done or not. In the case of a write operation, the shared memory unit accesses the step cache. If an entry corresponding to the target location is found, the write operation can be ignored since a write is already done, otherwise the write will proceed normally to the shared memory system. Similarly, in the case of a read operation, the MU accesses the step cache. In the case of hit, data is already in the BE unit and no memory system reference is needed, while in the case of a miss, memory reference is generated and sent to the shared memory system. Thus, a BE unit-wise step cache is attached to the MU of each unit.

In the following two sections, we describe how multioperations and multiprefixes can be implemented in TCF processors, such as TPA.

## 3. Multioperations for TCF processors

In order to support multioperations in TCF processors, one needs to implement reductions for arbitrarily large number of fibers in hardware using the blocking algorithm. This prevents applying existing multioperation techniques relying on a step cache and scratchpad entry per thread [13]. Our idea is to employ step caches, scratchpads, active memory units, FEs and load operations along with processors's BE register blocks organized differently than in [13] to support TCFs with unbounded number of fibers.

We propose a solution consisting of $L$-line $N$-way set associative step caches, $L$-entry scratchpads, a pre-memory ALU (pre-ALU) and a post-memory ALU (post-ALU), where the default value for $L$ is the maximum latency of the shared memory that can be hidden. Fig. 5 shows the modifications needed to the TCF processors architecture opening up step cache, scratchpad and active memory blocks. The step caches in this solution are similar to those used for the concurrent memory access support for TCF processors [17]. They are used to associatively reduce the traffic to the shared memory and group the references targeted to the same address, i.e., belonging to the same multioperation. Step caches consist of $N$ cache arrays, comparators and the hit logic. Since caches hold their content only during the end of ongoing step, scratchpads are used to hold the data and intermediate results. The scratchpads resemble those used in some ESM architectures [13] but do not have capacity to hold data for each fiber. Scratchpads consist of an array of data addressed by an operand determined as a function of step cache address. Besides step caches and scratchpads, a multioperation-aware active memory unit on each shared memory module is needed. They consist of an ALU and multiplexer attached into the BE shared memory module. Active memory operations, such as multioperations and multiprefixes, fetch the old value of the target memory location, perform the operation in the ALUs and write the result back to the memory location while non-active memory operations just pass the ALUs.

We identify five different techniques (summarized also in Fig. 10 of Section 5 for reference purposes) to realize multioperations in TCF processors:

1. **Fast multioperations (FS).** These are implemented with M$op$ subinstructions realizing the reduction $op$ by relying on active memory units in the memory module (see Fig. 6 left): Each BE executes its instructions for each fiber one by one in an overlapped way in phase B3.2. The fibers participating in multioperations send out their data (marked with blue color) to a memory module where the target location is placed. The active memory unit of the target module fetches the old data of the location (marked with green color), performs $op$ with the incoming fiber data and stores the result back to the target location. Fast multioperations execute in one step. If there are more than $T/B$ participating fibers per multioperation ($T$ is the thickness of the TCF taking care of the functionality, $B$ the number of BEs) the execution will slow down accordingly. The minimum execution time of a step is $T/B$. If all fibers of a TCF participate to a single fast multioperation, the speedup is lost entirely due to congestion caused by $T$ requests within $T/B$ clock cycles in the module access port.

2. **Symmetric two phase multioperations (S2).** This consists of a sequence of two subinstructions — BM$op$ and EM$op$ that perform the main work in the phase B3.2 of the BEs (see Fig. 6 right). The former begins execution by calculating backend unit-wise results for the reduction $op$ with a help of the step cache, the pre-ALU and the scratchpad without referring to the shared memory system. For this, each fiber sends its references along with the data (marked with blue color) to the step cache for grouping together the values belonging to the same multioperation based on target addresses and assigning data values to corresponding scratchpad address (marked with gray color). The fibers then proceed to the pre-ALU logic located in the BE pipeline just before the logic sending out memory references. There, previous value from the determined scratchpad address is fetched, the $op$ with the fiber data is performed in the pre-ALU and the result stored back to the scratchpad. After all the fibers have been executed, the scratchpad entries contain BE-wise multioperation results. They are used to pass these BE-wise results to the second instruction that ends execution by taking care of the actual computing of the processor-wise results in the same way as the FS technique does and writing the final result to the target location (marked with green color). A symmetric two-phase multioperation takes two full steps to execute and requires an overflow mechanism if more than $N$ multioperations per a cache set needs to be supported as the step cache sets and scratchpad can get full. In the best case it takes $2T/B$ clock cycles.

3. **Backend–frontend multioperations (BF).** This can be used only in the case of single multioperation and consists of two phases

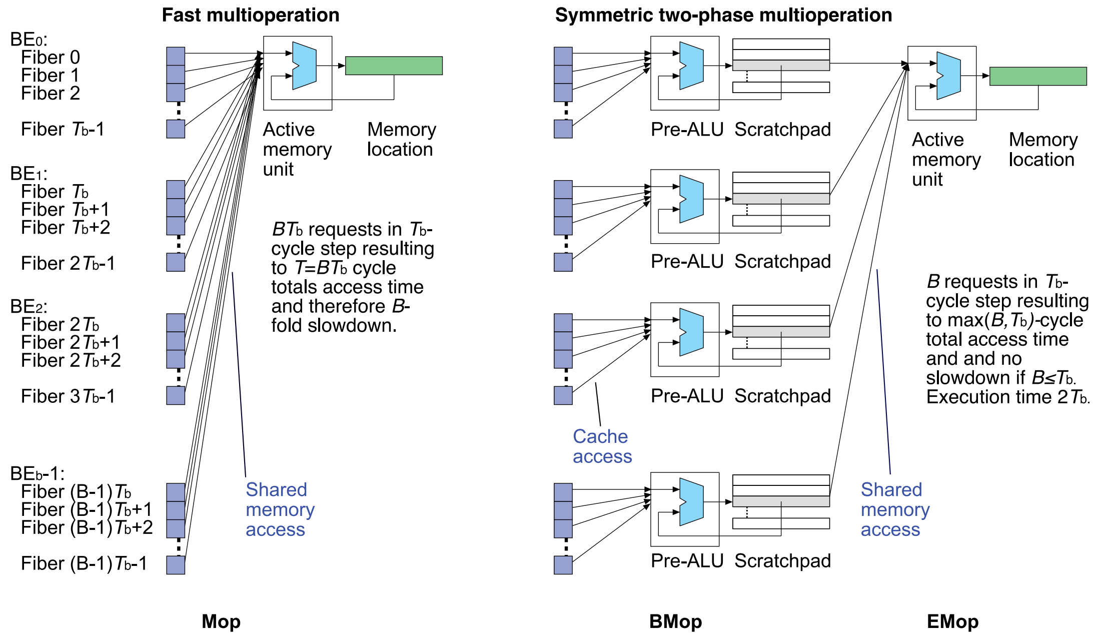


**Fig. 6.** Fast multioperation (FS, left) and symmetric two-phase multioperation (S2, right) techniques. Step caches are not shown. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

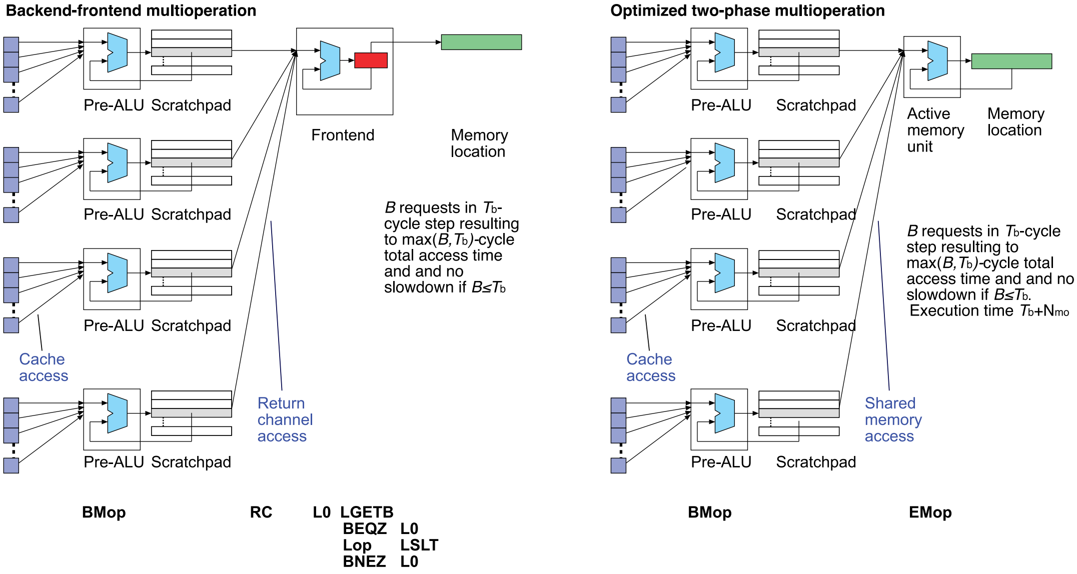

Backend-frontend multioperation
Optimized two-phase multioperation
Pre-ALU
Scratchpad
Frontend
Memory location
B requests in Tb-cycle step resulting to max(B,Tb)-cycle total access time and no slowdown if B≤Tb
Cache access
Return channel access
BMop
RC
L0 LGETB
BEQZ L0
Lop LSLT
BNEZ L0
Active memory unit
B requests in Tb-cycle step resulting to max(B,Tb)-cycle total access time and and no slowdown if B≤Tb. Execution time Tb+Nmo
Shared memory access
EMop

**Fig. 7.** Backend–frontend multioperation (BF, left) and optimized two-phase multioperation (O2, right) techniques. Step caches are not shown. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

like S2 (see Fig. 7 left). The first phase is identical to S2 but the processor-wise results are also sent to the frontend for completing the operation there. The second phase consists of reducing the received $B$ BE-wise results into the final result in FE phase F3 using a FE memory location (marked with red color) and writing the obtained value to the target BE memory location (marked with green color). This takes $T/B+B+L_f$ clock cycles, where $L_f$ is the latency of FE write to the BE shared memory system. A variant of this writes the final result to the FE memory only and potentially saves up to $L_f-1$ cycles.

4. **Optimized two phase multioperations (O2).** The first phase of this technique works like in S2 but allocation of scratchpad is done from the beginning towards the end. Due to this sequential addressing scheme, the scratchpads need to keep also the multioperation target addresses. The second phase sends out the contents of the occupied entries of the scratchpad only. If more than $L$ multioperations need to be supported, O2 requires an overflow mechanism of S2 but here for the first phase. The execution time is normally $T/B+M$, where $M$ is the number of simultaneous multioperations (see Fig. 7 right).

5. **Multioperation loads (ML).** An obvious weakness of the previous techniques is that they work efficiently only for register to memory reductions, not for memory to memory reductions. Multioperation loads work similarly as O2 but the first phase

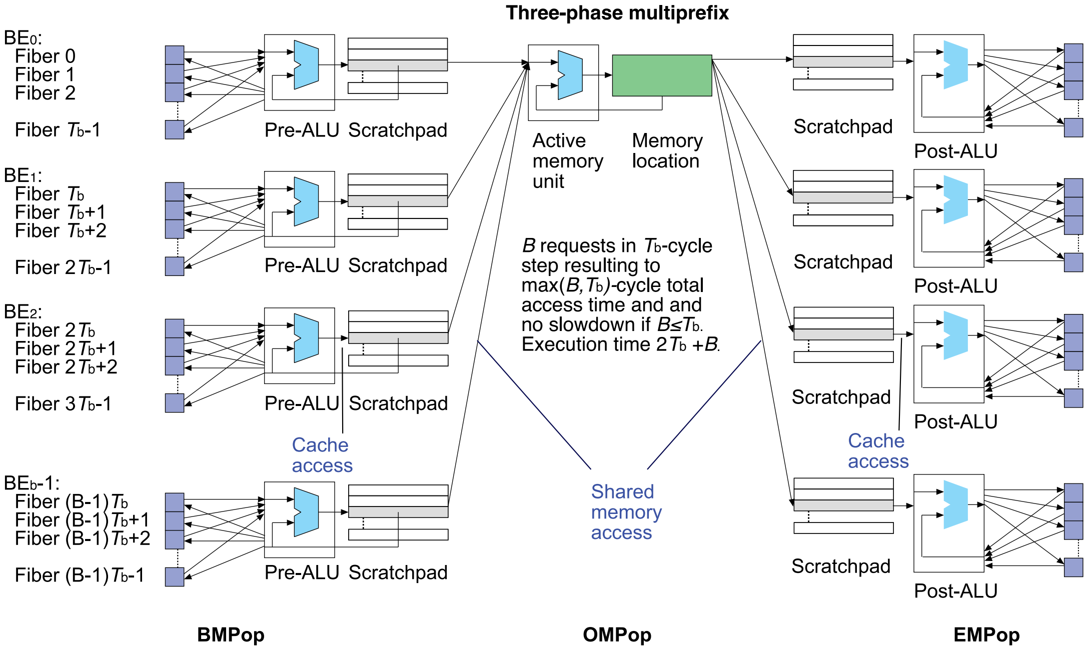


**Fig. 8.** Three phase multiprefix (3MP) technique. Step caches are not shown.

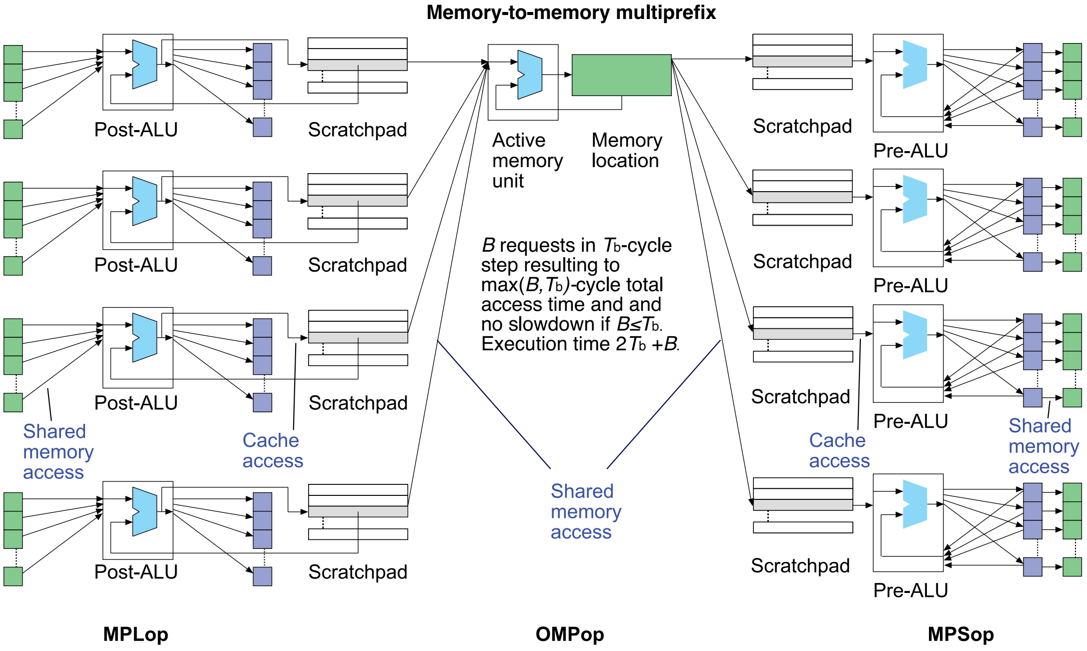
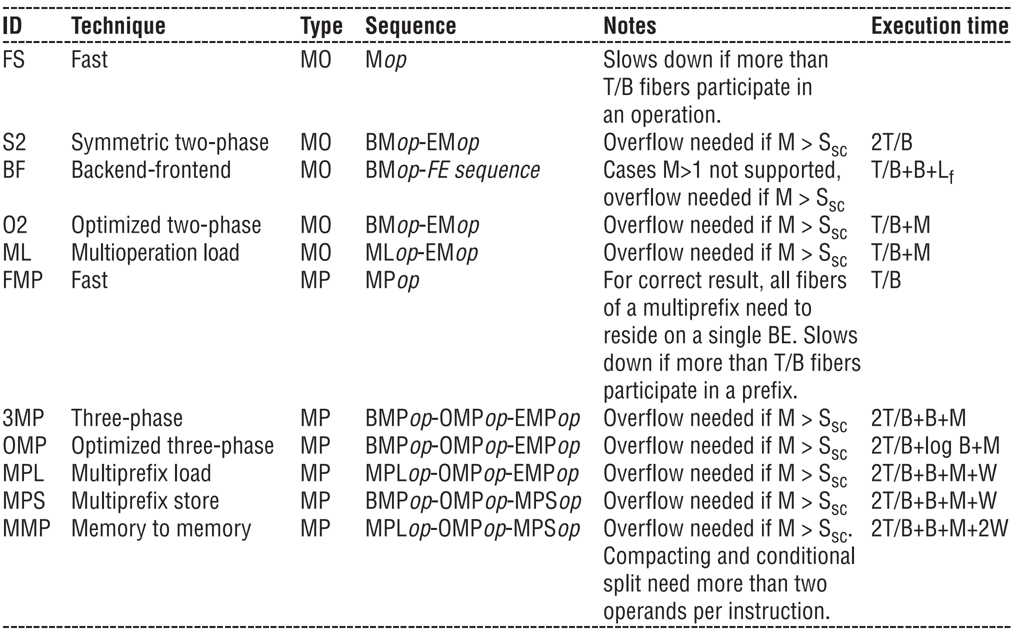


**Fig. 9.** Memory-to-memory multiprefixes (MMP) technique. Step caches are not shown.

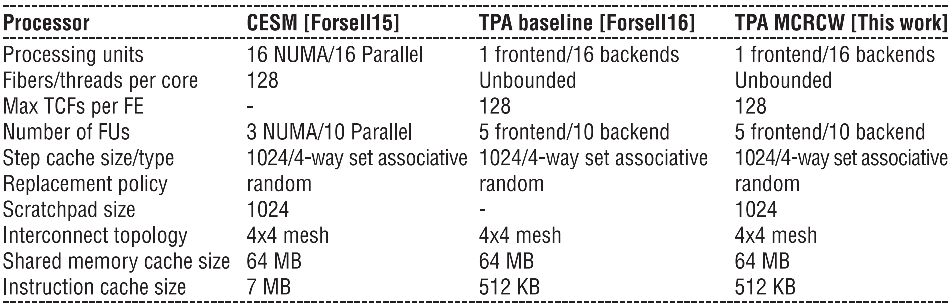
loads data from the memory and performs the backend-wise reduction of incoming data with an additional post-ALU following the memory unit in the BE pipeline. The minimum execution time for memory-to-memory multioperation loads with the O2 s phase operation is $T/B+M$. The overflow mechanism for this technique is harder to design than for the other techniques due to the fact that loaded data already occupies the memory interface.

Note that for this kind of techniques, step caches, scratchpads and data memory modules employing active memory units should be able to perform a read and write to the same address within a clock cycle. For step caches and scratchpads this is not a major problem since they reside inside BE units and have typically small size. For data memory modules that are much larger and typically implemented with a fast SRAM technology, a read–write access can be implemented with a double speed memory technology that is very expensive or may limit



| ID | Technique | Type | Sequence | Notes | Execution time |
|---|---|---|---|---|---|
| FS | Fast | MO | M*op* | Slows down if more than T/B fibers participate in an operation. | |
| S2 | Symmetric two-phase | MO | BM*op*-EM*op* | Overflow needed if M > $S_{SC}$ | 2T/B |
| BF | Backend-frontend | MO | BM*op-FE sequence* | Cases M>1 not supported, overflow needed if M > $S_{SC}$ | T/B+B+$L_f$ |
| O2 | Optimized two-phase | MO | BM*op*-EM*op* | Overflow needed if M > $S_{SC}$ | T/B+M |
| ML | Multioperation load | MO | ML*op*-EM*op* | Overflow needed if M > $S_{SC}$ | T/B+M |
| FMP | Fast | MP | MP*op* | For correct result, all fibers of a multiprefix need to reside on a single BE. Slows down if more than T/B fibers participate in a prefix. | T/B |
| 3MP | Three-phase | MP | BMP*op*-OMP*op*-EMP*op* | Overflow needed if M > $S_{SC}$ | 2T/B+B+M |
| OMP | Optimized three-phase | MP | BMP*op*-OMP*op*-EMP*op* | Overflow needed if M > $S_{SC}$ | 2T/B+log B+M |
| MPL | Multiprefix load | MP | MPL*op*-OMP*op*-EMP*op* | Overflow needed if M > $S_{SC}$ | 2T/B+B+M+W |
| MPS | Multiprefix store | MP | BMP*op*-OMP*op*-MPS*op* | Overflow needed if M > $S_{SC}$ | 2T/B+B+M+W |
| MMP | Memory to memory | MP | MPL*op*-OMP*op*-MPS*op* | Overflow needed if M > $S_{SC}$. Compacting and conditional split need more than two operands per instruction. | 2T/B+B+M+2W |

**Fig. 10.** Evaluated techniques (MO = multioperation, MP = multiprefix, *op* = multioperation or multiprefix operation, $B$ = number of backends, $L_f$ = latency of FE write to the BE shared memory, $M$ = number of simultaneous multioperations, $S_{sc}$ = size of step cache, $T$ = thickness of the current TCF, $W$ = memory system segment length in the BE pipeline).



| Processor | CESM [Forsell15] | TPA baseline [Forsell16] | TPA MCRCW [This work] |
|---|---|---|---|
| Processing units | 16 NUMA/16 Parallel | 1 frontend/16 backends | 1 frontend/16 backends |
| Fibers/threads per core | 128 | Unbounded | Unbounded |
| Max TCFs per FE | - | 128 | 128 |
| Number of FUs | 3 NUMA/10 Parallel | 5 frontend/10 backend | 5 frontend/10 backend |
| Step cache size/type | 1024/4-way set associative | 1024/4-way set associative | 1024/4-way set associative |
| Replacement policy | random | random | random |
| Scratchpad size | 1024 | - | 1024 |
| Interconnect topology | 4x4 mesh | 4x4 mesh | 4x4 mesh |
| Shared memory cache size | 64 MB | 64 MB | 64 MB |
| Instruction cache size | 7 MB | 512 KB | 512 KB |

**Fig. 11.** Processors in the evaluation.

the clock rate, use a (patented) module cache technology adding a small cache read–write port cache to speed up access, or accept the performance hit of half-speed access. In this paper, we assume that the module caches are used and no performance penalty is caused.

Since the TCF architecture typically does not access memory locations in processor order, non-commutative multioperations cannot be implemented this way.

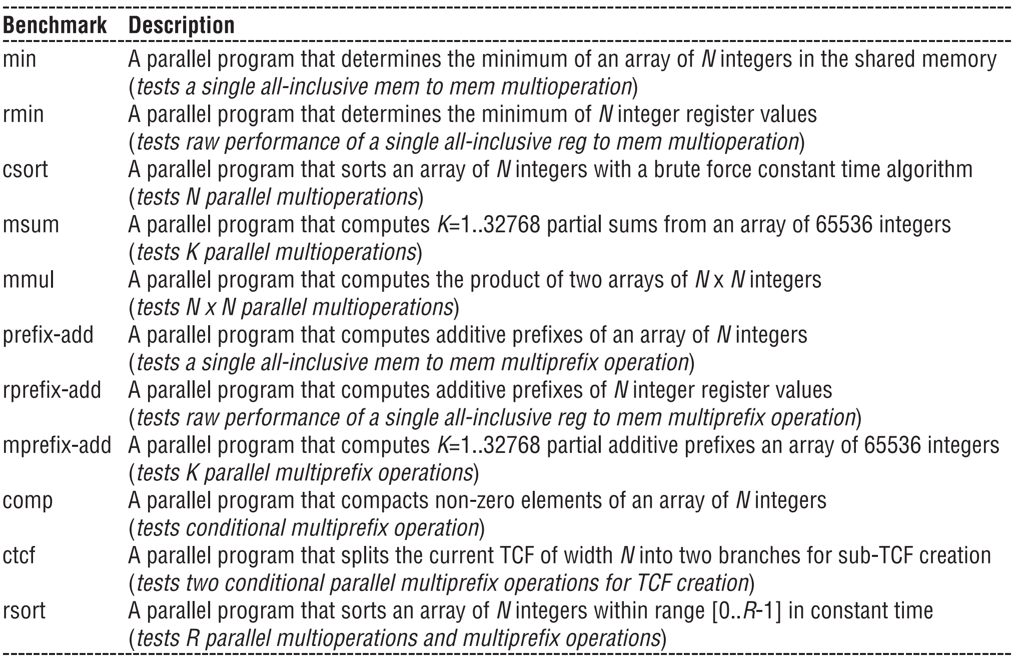
## 4. Multiprefixes for TCF processors

Implementing multiprefixes in TCF processors resemble that of multioperations but have two main differences — there is a return value for each fiber, and the operations must be executed in order to guarantee correct results. Our idea is to use the blocking multiprefix algorithm with $L$-line $N$-way set associative step caches, $L$-entry scratchpads, pre-memory ALU, post-memory ALU and multiprefix-aware active memory units. Unlike the case for multioperations, employing frontends in multiprefix computation is not practical since each fiber/block needs unique return values and passing such values via the spreading network can be slow.

We identify six different techniques (summarized also in Fig. 10 of Section 5 for reference purposes) to realize multiprefixes in TCF processors:

1. **Fast multiprefixes (FMP).** These are implemented with MP*op* subinstructions realizing reduction *op* and returning the ordered reduction of the preceding fibers. The implementation employs active memory units in memory modules and executes in one step. If there are more than $T/B$ participating fibers per multiprefix ($T$ is the thickness of the TCF taking care of the functionality, $B$ the number of backend units) then it will slow down the execution accordingly. The minimum execution time of a step is $T/B$. Furthermore, this primitive gives the correct ordered result only if the participating fibers belong to the same backend. However, one can use it to perform $\left\lfloor \sqrt{T} \right\rfloor$ simultaneous multiprefixes useful in many parallel algorithms. Even if the result is not an actual multiprefix in the case of multiple participating backends, the result can be used for purposes where the ordering is not required, such as compacting.

2. **Three phase multiprefixes (3MP).** This consists of a sequence of three subinstructions — BMP*op*, OMP*op* and EMP*op*. The first subinstruction begins execution by calculating backend unit-wise multiprefix *op* results with a help of the step cache, scratchpad and pre-ALU without referring to the shared memory system but storing intermediate BE-wise results back to the registers of fibers (see Fig. 8). The second instruction performs multiprefix



| Benchmark | Description |
|---|---|
| min | A parallel program that determines the minimum of an array of $N$ integers in the shared memory<br>(*tests a single all-inclusive mem to mem multioperation*) |
| rmin | A parallel program that determines the minimum of $N$ integer register values<br>(*tests raw performance of a single all-inclusive reg to mem multioperation*) |
| csort | A parallel program that sorts an array of $N$ integers with a brute force constant time algorithm<br>(*tests N parallel multioperations*) |
| msum | A parallel program that computes $K$=1..32768 partial sums from an array of 65536 integers<br>(*tests K parallel multioperations*) |
| mmul | A parallel program that computes the product of two arrays of $N$ x $N$ integers<br>(*tests N x N parallel multioperations*) |
| prefix-add | A parallel program that computes additive prefixes of an array of $N$ integers<br>(*tests a single all-inclusive mem to mem multiprefix operation*) |
| rprefix-add | A parallel program that computes additive prefixes of $N$ integer register values<br>(*tests raw performance of a single all-inclusive reg to mem multiprefix operation*) |
| mprefix-add | A parallel program that computes $K$=1..32768 partial additive prefixes an array of 65536 integers<br>(*tests K parallel multiprefix operations*) |
| comp | A parallel program that compacts non-zero elements of an array of $N$ integers<br>(*tests conditional multiprefix operation*) |
| ctcf | A parallel program that splits the current TCF of width $N$ into two branches for sub-TCF creation<br>(*tests two conditional parallel multiprefix operations for TCF creation*) |
| rsort | A parallel program that sorts an array of $N$ integers within range [0..$R$-1] in constant time<br>(*tests R parallel multioperations and multiprefix operations*) |

**Fig. 12.** Benchmark programs for the proposed multioperation TPA processor.

computation of BE results like the FMP technique does. In order to obtain correct results, processing of BE results is sequentialized by issuing $i$ synchronizations for BE $i$ prior to actual memory operation and $B-i$ synchronizations after it. Finally, the third instruction finalizes the fiber-wise results by computing the obtained BE prefixes into fiber-wise partial results with a help of post-ALU. This primitive needs an overflow mechanism resembling that used in the O2 multioperation technique. The execution time is normally $2T/B$+$B$+$M$, where $M$ is the number of simultaneous multiprefixes over the BEs.

3. **Optimized three phase multiprefixes (O3M).** The first and last phase operate like in the 3MP technique but the middle phase is optimized to work using the logarithmic algorithm instead of sequential one used in 3MP. The execution time is normally $2T/B + \log B + M$, where $M$ is the number of simultaneous multiprefixes over the backends.

4. **Multiprefix loads (MPL).** An obvious weakness of the previous techniques is that they work efficiently only if the data is in the registers before execution, not in memory. Multiprefix loads work similarly as 3MP but the first phase loads data from memory and performs the backend-wise prefix computation of incoming data at the post-ALU. The (memory-to-register) minimum execution time for multiprefix loads with second and third phase operation from the 3MP technique is $2T/B + B + M + W$, where $W$ is the memory system segment length in the backend pipeline. The overflow mechanism for this technique is harder to design than for the other techniques due to the fact that loading data already occupies the memory interface.

5. **Multiprefix stores (MPS)** Another drawback of the multiprefix primitives FMP and 3MP is that they write the prefixes to the registers rather than in memory. MPS solves this problem by performing the last phase directly to the memory with a help of the pre-ALU. The minimum execution time for register-to-memory multiprefix stores employing first two instructions from the 3MP technique is $2T/B + B + M + W$.

6. **Memory-to-memory multiprefix (MMP).** This combines reading from memory used in MPL, backend-wise computation phase of 3MP and storing to memory used in MPS to implement a true memory-to-memory multiprefix operation (see Fig. 9). The minimum execution time for memory-to-memory multiprefix operation employing the middle instruction from the 3MP technique is $2T/B + B + M + 2W$.

Besides these implementation techniques, we identify a couple of special needs to accelerate key primitives in TCF processors with multiprefix variations, namely compacting a vector and splitting a TCF conditionally to two sub-TCFs. Both variations need to handle fiber-wise data independently and conditionally which is not possible with standard multiprefixes and TCF operation. To implement these primitives efficiently, we propose a variant of the MMP technique in which there are two targets in additive multiprefixes, one for data meeting the condition and another for data not meeting the condition. An additional requirement is handling extra input and output operands for the conditions and selection of the data to be written into the memory. The execution time characteristics for these multiprefixes are similar to 3MP.

## 5. Evaluation

We conducted a performance evaluation of the techniques described in the previous sections. The baseline TPA used in the evaluations consists of a single 7-functional unit MPA VLIW frontend [20] and sixteen 10-functional unit backends. We have implemented the FS, S2, BF, O2 and ML multioperation techniques, as well as the FMP, 3MP, O3M, MPL, MPS, MMP including conditional multiprefix techniques for ADD, SUB, AND, OR, MAX, MAXU, MIN and MINU multi(prefix)operations (see Fig. 10) on the TPA to demonstrate that the proposed operations can be implemented and meet the expectations. For reference purposes, we also evaluated single core execution in the frontend using the sequential reduction algorithm (SEQ) and the best existing ESM solution using the configurable emulated shared memory (CESM) architecture [14]. For a summary of the measured architectures and their configuration, see Fig. 11.

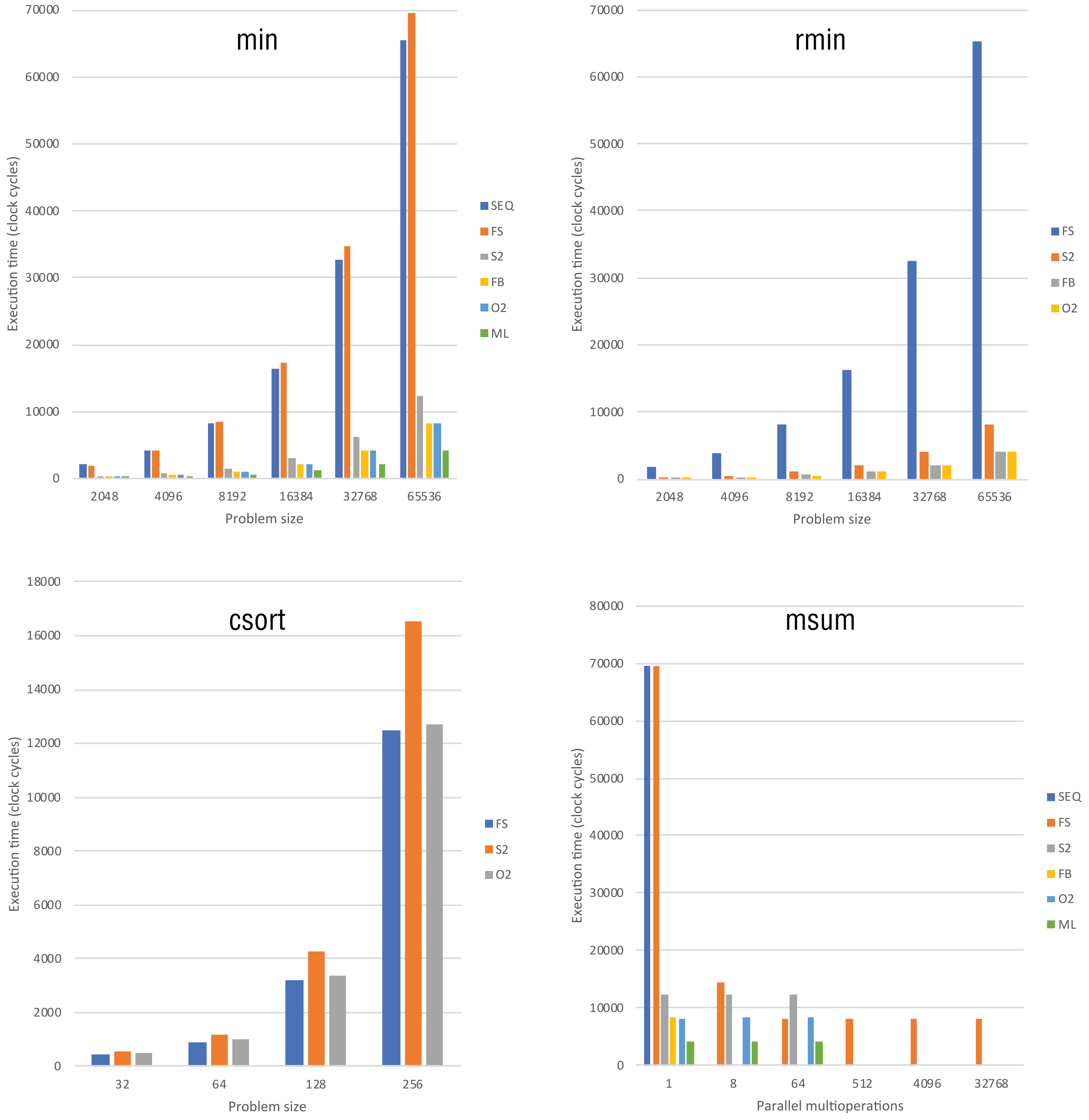


**Fig. 13.** The execution time in the tested processors as the function of problem size $N$ for the multioperation benchmark programs.

### 5.1. Performance

We implemented 11 parametric kernel benchmarks representing different use cases of multioperations and multiprefixes (see Fig. 12), executed them in our clock and RTL accurate TPASim configurable to simulate the baseline TPA and TPA MCRCW, and measured the execution time in clock cycles.

The results of the measurements comparing the multioperation techniques are shown in Fig. 13 while Fig. 14 demonstrates the speedup potential of the best technique with respect to the baseline TPA and CESM. Our observations are:

- The FS technique performs poorly in the case of a single multioperation. The execution time of the `min` benchmark is higher than that of the sequential algorithm with most values of $N$ due to congestion in the shared memory system. However, as the number of simultaneous multioperations grows (see the `msum` benchmark), it becomes the second best performing multioperation technique.
- The S2 technique performs up to 5.33 and 7.98 times faster than FS in the `min` and `rmin` benchmarks in the case of a single multioperation. This is because S2 reduces the contention in the target memory module. As the number of parallel multioperations grow, the performance of S2 stays the same until it runs out of step cache.
- The BF technique works only for single multioperation cases, where it gives 48.6% and 96.3% higher performance than S2 in the `min` and `rmin` benchmarks. This improvement comes from reduction of the second phase of the multioperation.
- The O2 technique performs slightly better than BF but it works also for parallel multioperations that fit in the step cache. It increases the performance by up to factor of 15.95 in single register to memory multioperations with respect to FS. The `msum` benchmark reveals that the advantage diminishes fast as the number of parallel multioperations increases.
- The ML technique gives the best performance for memory to memory multioperations used in the `min` benchmark. It speeds up single multioperations by up to factor of 15.57 with respect to the sequential execution with our 16-backend system. As the

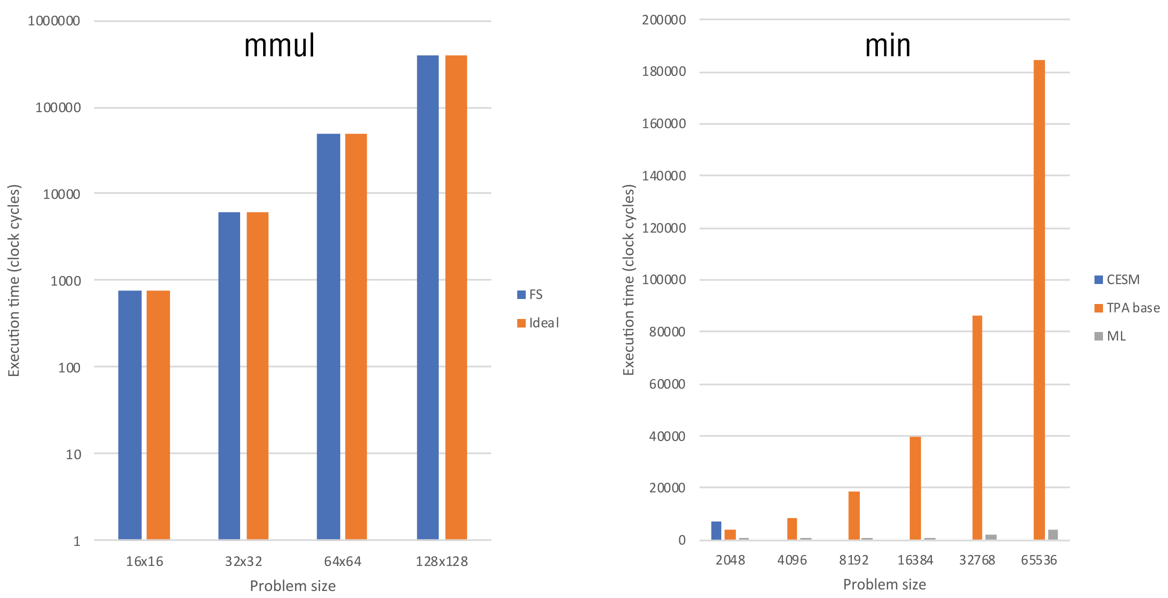


**Fig. 14.** The execution time of the `mmul` benchmark versus that in a machine with ideal memory system. The execution time of the `min` benchmark for the best single multioperation technique versus that in the CESM without multioperations and TPA baseline processors.

number of parallel multioperations grows, it remains the fastest alternative until it runs out of step cache.

- For the `csort` benchmark, the FS technique is the fastest as expected, while the S2 technique is the weakest. O2 performs a bit weaker than FS since it runs a second phase. Due to the nature of the algorithm reducing the parallel comparison results with a multioperation, the ML technique is not applicable in it.
- The best proposed single multioperation technique executes the `min` benchmark up to 43.82 times faster than the baseline TPA using the logarithmic algorithm shown in Fig. 1. CESM (without multioperations) performs 1.75 times slower than the TPA baseline in the case where size matches the number of threads. Otherwise the overhead would be much higher due to thread switching costs.

To determine the optimality of the proposed solution, we compared the proposed TPA to a similar machine with an ideal memory system. The results for the `mmul` benchmark implemented with the FS technique are shown in Fig. 14. We observe that measured TPA execution times are very close to those in the ideal machine. The execution time overhead 0.39% in 16 × 16 case drops down to 0.00076% in 128 × 128 case and corresponds with the synchronization wave overhead.

The CESM architecture (including multioperation support) gives exactly the same results if the number of hardware threads (2048 for this configuration) equals to $N$ for fast multioperations and two-phase multioperations resembling the FS and S2 techniques of TCF-aware processors but it does not have anything similar to BF, O2 or ML. If $N$ is less than the number of threads then the utilization of processor pipeline will drop proportionally. Likewise, if $N$ is greater than the number of threads, there will be a high thread switching penalty. Alternatively one can use iterative methods, e.g., looping to match the number of HW treads with the problems size. In the case of a simple multioperation, the loop overhead can be eliminated but the loop header takes at least one instruction with minimum execution time of 129 clock cycles in the CESM configuration. This corresponds to a slowdown of up to 6.4% and 33.3% in the min benchmark for FS and S2 techniques, respectively.

The results of multiprefix tests are shown in Fig. 15 while the conditional multiprefix tests are shown in Fig. 16. Our notes of interest include:

- The FMP technique performs modestly in the case of a single multiprefix operation. It outperforms the sequential algorithm in the `prefix-add` benchmark only by a factor of 69%–85% due to memory system congestion but gives an incorrectly ordered result. In the `mprefix-add` test, it outperforms all other techniques if the number of parallel prefixes is at least 512 and is the only possible solution for over 1024 parallel prefixes due to inappropriate step cache size.
- The 3MP technique performs up to 4.49 and 7.97 times faster than FMP in the `prefix-add` and `rprefix-add` benchmarks in the case of single multiprefix. As the number of multiprefixes increases, the performance drops gradually to half of the FMP. This technique always gives the correct results, but it cannot be used if the number of simultaneous multiprefixes exceeds the size of the step cache.
- The O3M performs slightly worse than 3MP with our test data. The slowdown compared to it drops from 2.07 down to 1.03 as the problem size increases in `prefix-add` benchmark. In the `rprefix-add` case, the slowdown ranges from 3.07 down to 1.07. Like 3MP, this technique gives always the correct results, but it cannot be used if the number of simultaneous multiprefixes exceeds the size of the step cache.
- The MPL technique gives somewhat better results than 3MP in memory-to-memory multiprefix tests. The speedup ranges from 1.03 to 1.32 in the single multiprefix case. In the case of multiple parallel multiprefixes the speedup over FMP drops from 5.93 down to 0.60.
- Like MPL, the MPS technique performs better than 3MP in the case of a single multiprefix operation. The speedup ranges from 1.02 to 1.32. In the case of parallel multiprefixes the speedup over FMP collapses from 5.93 down to 0.60.
- The MMP gives the best performance for memory to memory multiprefixes. It speeds up single multiprefixes by up to a factor of 15.52 with respect to the sequential version with our 16-backend system. As the number of parallel multioperations grows, it remains the fastest alternative except for the largest setup. Like other step cache assisted multiprefix techniques, it cannot handle the cases where number of multiprefixes exceeds the step cache size.

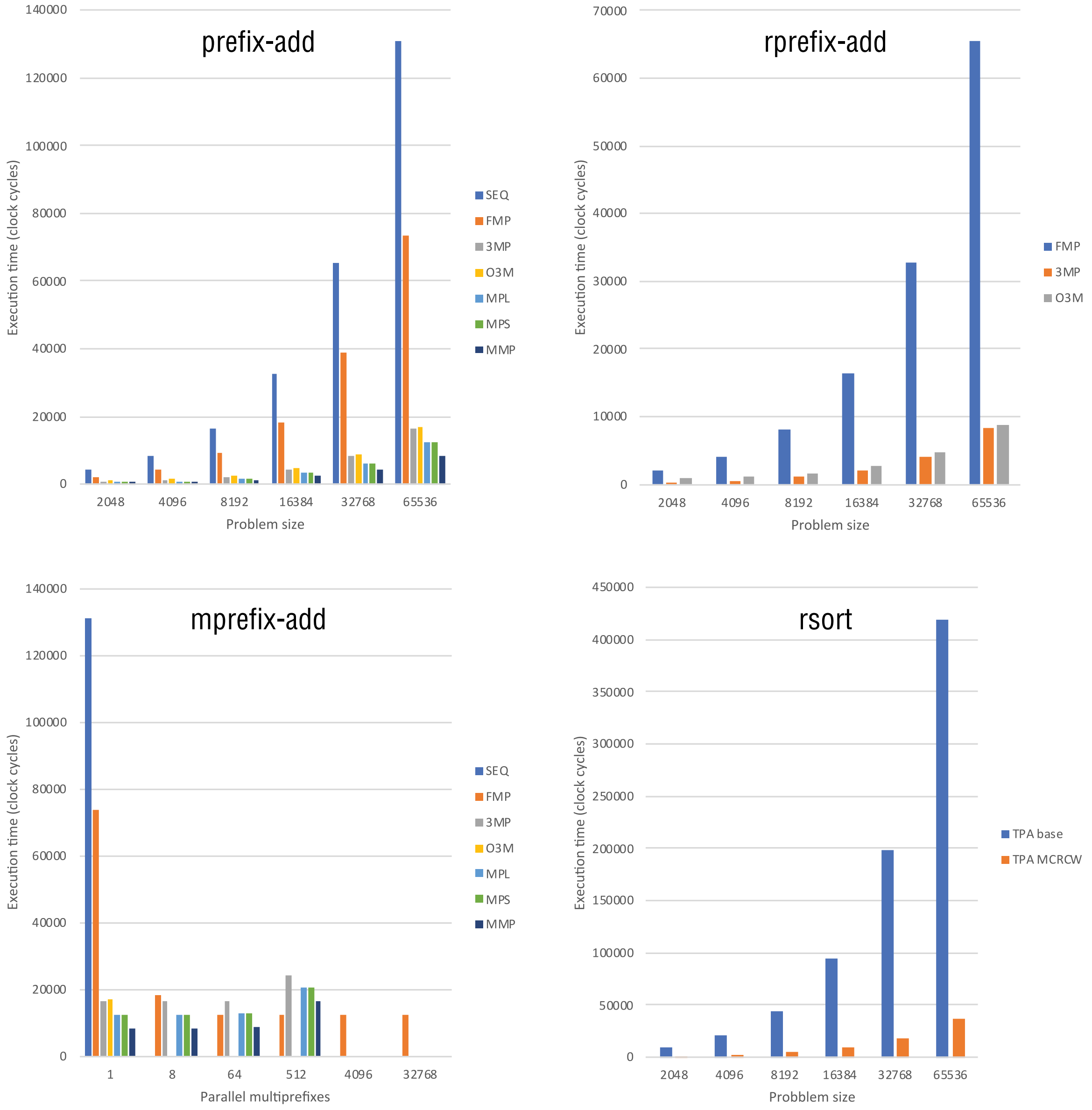


**Fig. 15.** The execution time in the tested processors as the function of problem size $N$ for the multiprefix benchmark programs.

We compared the proposed conditional multiprefixes against those implemented with the baseline TPA without multiprefixes using the comp and ctcf benchmarks. The former compacts an input data array using a separate mask array distinguishing the actual values from the empty space. The latter divides the fibers of a TCF into two sub-TCFs according to a fiber-wise condition array. Since TCFs always number the fibers from 0 to $T-1$, this algorithm provides an output array containing the fibers of the parent TCF for both sub-TCFs. The results are shown in Fig. 16. According to the measurements, the proposed conditional multiprefixes indeed accelerate the `comp` and `ctcf` benchmarks over the baseline TPA by the factors from 8.96 to 23.45 and from 9.22 to 23.93, respectively. The main portion of the speedup comes from trading an iterative logarithmic prefix algorithm to the conditional three-phase memory-to-memory MMP technique.

In order to compare the implementation options for OMP*op* instructions for three-phase multiprefixes, we executed the `prefix-add` benchmark with the standard sequential, balanced sequential and logarithmic realizations. The standard sequential version sends out $B+M$ references while the balanced sequential version schedules the sending of references so that it takes $B(M+1)$ clock cycles in total to avoid tweaking of synchronization waves. The results are shown in Fig. 17. According to the measurements, the standard sequential version is up to 5.6% faster in dual memory access techniques (MPL and MPS) and up to 8.1% faster in the three memory access technique (MMP). In the single memory access case, the logarithmic techniques (O3M) turned out to be 3.5% slower than the standard sequential version (3MP) in this benchmark due to extensive pipeline delays not present in the sequential version. We believe that for very large TPAs the logarithmic version would be faster than the standard sequential one due to its asymptotically lower time complexity. However, we did not experiment with such large TPAs due to extensive simulation times and extra memory required by the O3M technique.

Finally, to get a rough idea how the performance of TPA MCRCW would compare to that of existing multicore processors in multiprefix computation, we implemented a 262 144-element memory-to-memory additive blocking multiprefix algorithm with a stacked allocation shown in Fig. 2 using C/Pthreads for a MacBook Pro 15'' laptop featuring a 2.7 GHz 4-core Intel Core i7 processor with 3 memory units per core and equally wide lines to the shared L3 cache as TPA MCRCW has to the on-chip BE shared memory. We compiled the algorithm

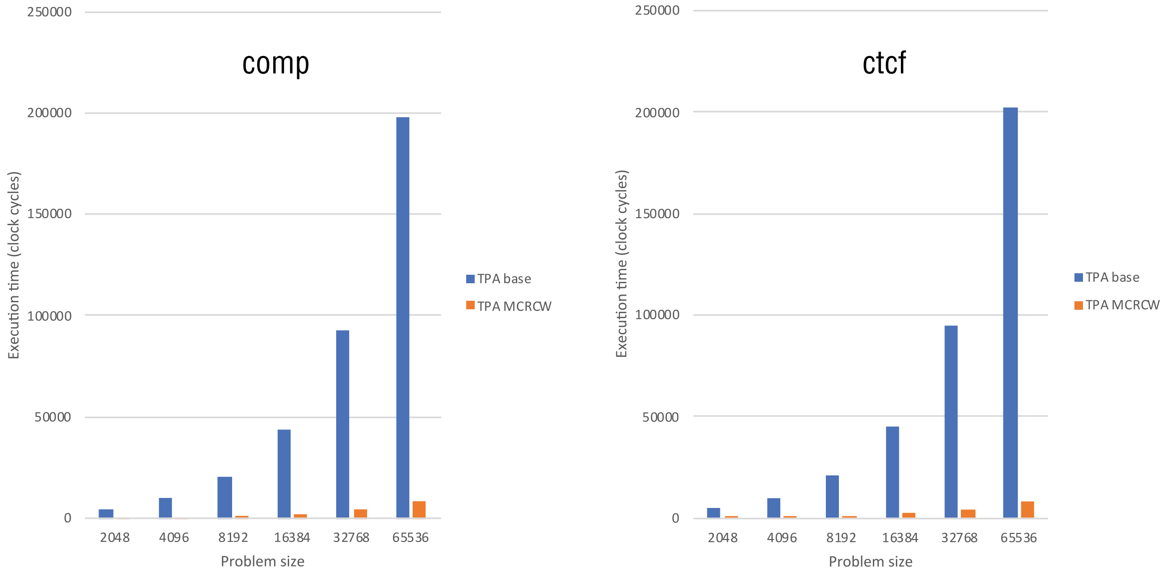


**Fig. 16.** The execution time in the baseline and proposed architecture as the function of problem size $N$ for the `comp` and `ctcf` benchmark programs.

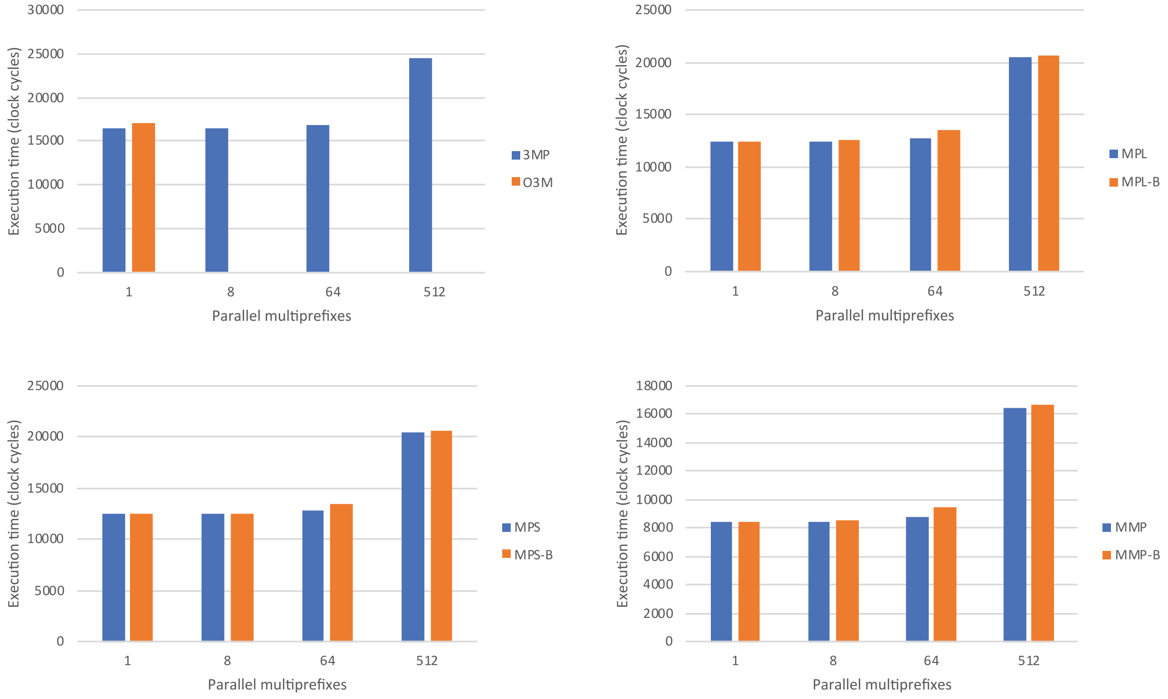
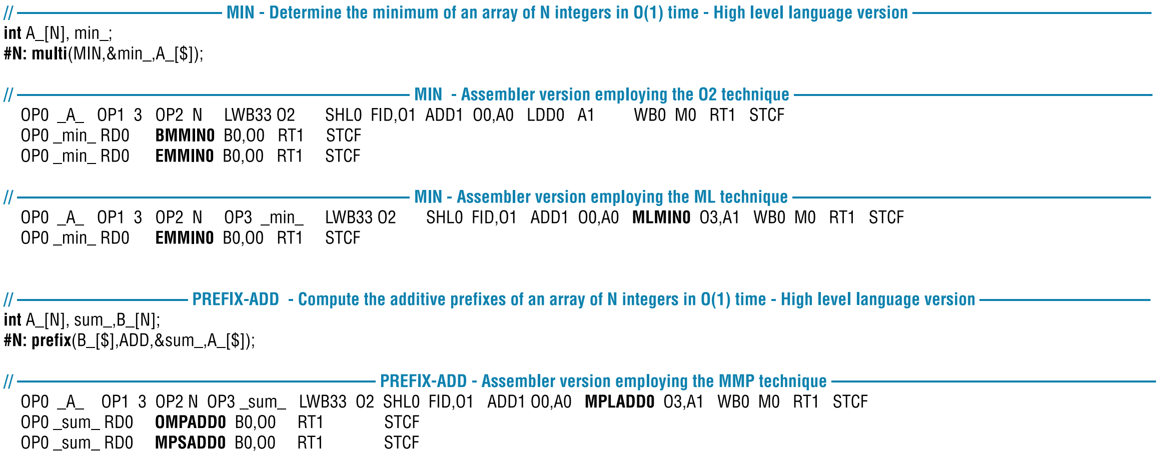


**Fig. 17.** The execution time in the proposed processors employing sequential, balanced sequential and logarithmic variants of the OMP*op* instruction as the function of the number of parallel multiprefixes for the `mprefix-add` benchmark program.

with Apple LLVM compiler with -O3 optimizations and measured the average execution time of five 100 000-iteration runs to guarantee that the execution in the processor is ramp-up properly from possible power saving modes and that the data is in the on-chip caches. We implemented the same functionality for TPA MCRCW with the MMP-technique and measured the execution time. If we assume that the processors would run at the same clock rate, the comparison shows a significant 9.28-fold performance advantage for TPA.

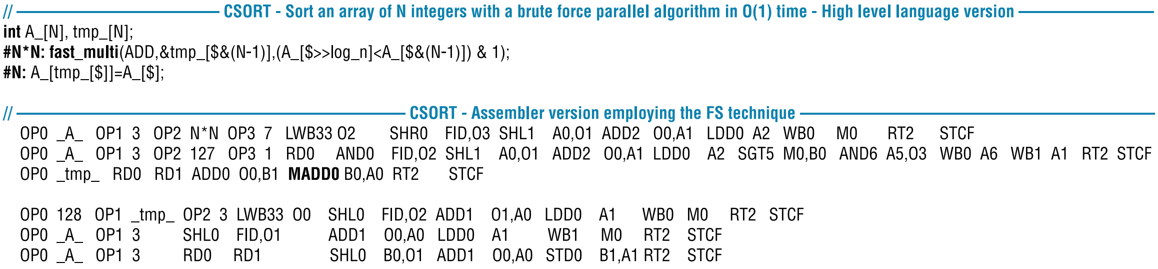
### 5.2. Hardware implementation considerations

Extending the TPA baseline processor into TPA MCRCW version does not imply major modifications. Essential parts for supporting multioperation techniques FS, S2, FB, O2 and ML for operations ADD, SUB, AND, OR, MAX, MAXU, MIN, MINU as well as multiprefix techniques FMP, 3MP, O3M, MPL, MPS and MMP including the vector compaction and conditional TCF splitting include step caches as projected in [13], active memory units and scratchpads. There is no need to add step caches to TPA MCRCW, since they are assumed to be already included in the baseline TPA to support concurrent memory access [17].



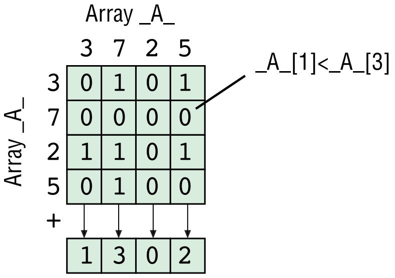
```
//———————— MIN - Determine the minimum of an array of N integers in O(1) time - High level language version ————————
int A_[N], min_;
#N: multi(MIN,&min_,A_[$]);

//———————————————— MIN  - Assembler version employing the O2 technique ————————————————
   OP0  _A_   OP1  3   OP2  N    LWB33 O2     SHL0  FID,O1  ADD1  O0,A0   LDD0   A1       WB0  M0   RT1   STCF
   OP0 _min_ RD0       BMMIN0  B0,O0   RT1    STCF
   OP0 _min_ RD0       EMMIN0  B0,O0   RT1    STCF

//———————————————— MIN - Assembler version employing the ML technique ————————————————
   OP0  _A_   OP1  3   OP2  N    OP3  _min_    LWB33 O2     SHL0  FID,O1   ADD1  O0,A0  MLMIN0  O3,A1   WB0  M0   RT1   STCF
   OP0 _min_ RD0       EMMIN0  B0,O0   RT1    STCF


//———————— PREFIX-ADD  - Compute the additive prefixes of an array of N integers in O(1) time - High level language version ————————
int A_[N], sum_,B_[N];
#N: prefix(B_[$],ADD,&sum_,A_[$]);

//———————————————— PREFIX-ADD - Assembler version employing the MMP technique ————————————————
   OP0  _A_   OP1  3  OP2 N  OP3 _sum_   LWB33  O2  SHL0  FID,O1   ADD1 O0,A0   MPLADD0  O3,A1   WB0  M0   RT1   STCF
   OP0 _sum_ RD0      OMPADD0  B0,O0    RT1          STCF
   OP0 _sum_ RD0      MPSADD0  B0,O0    RT1          STCF
```

**Fig. 18.** Implementation of the `min` and `prefix-add` benchmarks with a high-level programming language and assembler using the O2, ML and MMP techniques (# = thickness of the TCF, $ = fiber identifier, _ = shared variable). Each assembler text line represents a single VLIW instruction consisting of a multiple subinstructions.



```
//———————— CSORT - Sort an array of N integers with a brute force parallel algorithm in O(1) time - High level language version ————————
int A_[N], tmp_[N];
#N*N: fast_multi(ADD,&tmp_[$&(N-1)],(A_[$>>log_n]<A_[$&(N-1)]) & 1);
#N: A_[tmp_[$]]=A_[$];

//———————————————— CSORT - Assembler version employing the FS technique ————————————————
   OP0  _A_   OP1  3   OP2  N*N  OP3  7   LWB33 O2      SHR0   FID,O3  SHL1   A0,O1  ADD2   O0,A1   LDD0  A2   WB0    M0     RT2    STCF
   OP0  _A_   OP1  3   OP2  127   OP3  1   RD0    AND0   FID,O2  SHL1   A0,O1   ADD2   O0,A1  LDD0    A2   SGT5  M0,B0   AND6  A5,O3   WB0  A6   WB1  A1   RT2  STCF
   OP0  _tmp_   RD0   RD1  ADD0  O0,B1  MADD0 B0,A0  RT2      STCF

   OP0  128   OP1  _tmp_  OP2  3  LWB33  O0    SHL0    FID,O2  ADD1    O1,A0   LDD0    A1      WB0   M0    RT2   STCF
   OP0  _A_   OP1  3       SHL0    FID,O1         ADD1    O0,A0   LDD0    A1         WB1     M0     RT2    STCF
   OP0  _A_   OP1  3       RD0     RD1             SHL0    B0,O1   ADD1    O0,A0   STD0    B1,A1  RT2    STCF
```

**Fig. 19.** Implementation of the `csort` benchmark with a high-level programming language and assembler using the FS technique (# = thickness of the TCF, $ = fiber identifier, log_n = precomputed $\log N$, _ = shared variable).



| | 3 | 7 | 2 | 5 |
|---|---|---|---|---|
| 3 | 0 | 1 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 1 |
| 5 | 0 | 1 | 0 | 0 |
| + | 1 | 3 | 0 | 2 |

**Fig. 20.** The `csort` algorithm compares all elements of the source array _A_ = (3,7,2,5) to each other and sums the results of comparisons together column-wisely. The column sums give the rank of each element in the sorted array.

Adding the 1024-entry scratchpads corresponds to 0.51% increase of the amount of on-chip memory. Extra logic is required in the form of ALUs needed before the memory access, within the active memory units and after the memory access. Based on the gate count estimates of typical ALU components, the area overhead of these three ALUs involving adders/comparators and support for basic boolean logic with respect to eight full-size ALUs of the baseline TPA BE containing also multipliers and barrel shifters is considered even smaller than 0.51%. Since the scratchpads and ALUs are utilized only for multioperations and multiprefixes, we estimate that the energy consumption increases with less than 0.51%. Thus, all in all, the estimated additional resources needed are minor compared to gained performance advantages.

Regarding the implementability of active memories, a notable fact is that they are associated with on-chip memory/cache resources implemented with SRAM technology rather than external DRAM. This makes silicon implementation of them many ways simpler and more efficient.

To demonstrate the feasibility of the techniques proposed in this paper, we have built a VHDL realization of a full TPA BE system implementing the most relevant FS, BF, O2, ML multioperation and FMP, 3MP, MPL, MPS, MMP multiprefix techniques and synthesized it against FPGA libraries. The resource usage and performance figures in HDL simulator were identical to those in our RTL software simulator.

### 5.3. *Programming mechanism*

Multioperation instruction-aware programming can be implemented with the aid of simple constructs (intrinsics) compiling directly to multioperation instructions. Since there is no standard way to express multioperations in popular parallel programming languages, we will use here the notation of the e-language [28]. It defines high-level constructs, `multi(OP,tgt,src)` and `fast_multi(OP,tgt,src)`, that implement multioperation `OP` for data obtained from `src` and save the result in `tgt` using the S2 and FS techniques of the target machines, like CESM, respectively. The constructs are not subprograms but compile to inline sequences of multioperation instructions. For example, `multi(ADD,t_,s_)` compile to pseudocode sequence of `BMADD s_,t_ | EMADD s_,t_` instructions that executes additive reduction in constant time. The number of parallel multioperations is defined by the target address values `t_` and the data for each multioperation is defined by the combination of `s_` and `t_`.

Our idea is to use the same constructs for expressing multioperations for TCF-aware processors. The semantic of the constructs remains the

```
// ―――――― RSORT - Sort an array of N integers within range [0..R-1], R=CN+K with a constant time parallel algorithm T=O(1) - High level language version ――――
int A_[N], B_[N], D_[R];
int SUM_;
#N: fast_multi(ADD,B_[A[$]],1);
#R: prefix(ADD,SUM_,B[$]);
#N: D_[B_[A_[$]]]=A_[$];

// ――――――――――――――――――――――― RSORT - Assembler version employing the FMP and MMP techniques ――――――――――――――――――――
   OP0  _A_    OP1  3 OP2  N      SHL0   FID,O1  ADD1  O0,A0   LDD0    A1      WB0    M0      RT1    STCF
   OP0  _B_    OP1  3 OP2  1 RD0 SHL0   B0,O1   ADD1  O0,A0   MADD0  O2,A1   RT1     STCF

   OP0  _B_  OP1  3 OP2 R  OP3 _SUM_  OP4  0  LWB33 O2  AND0 FID,O4  SHL1 A0,O1 ADD2 O3,A1  SHL3 FID,O1  ADD4  O0,A3  MLADD0  A2,A4 WB0 M0 RT1 STCF
   OP0  _SUM_ OP1 R LWB33 O1  RD0          OMPADD0 B0,O0                RT1          STCF
   OP0  _SUM_ OP1 0 OP2  3      OP3  4096  LWB33   O3  RD0  AND0   FID,O1   SHL1   A0,O2   ADD2  O0,A1  MPSADD0 B0,A2  RT1          STCF

   OP0    _A_  OP1  3 OP2  N      LWB33  O2       SHL0  FID,O1  ADD1    O0,A0  LDD0   A1      WB0   M0      RT2     STCF
   OP0    _B_  OP1  3 RD0          SHL0    B0,O1   ADD1  O0,A0  LDD0   A1       WB1    M0      RT2    STCF
   OP0    _D_  OP1  3 RD0  RD1   SHL0    B1,O1   ADD1  O0,A0  STD0   B0,A1   RT2     STCF
```

**Fig. 21.** Implementation of the rsort benchmark with a high-level programming language and assembler using the FMP AND MMP techniques (# = thickness of the TCF, $ = fiber identifier, _ = shared variable).

same as for the fixed threading scheme machines but the number of participating fibers is now defined by the thickness of the executed TCF. Due to the bounded size of the assisting step caches and scratchpads, S2, FB, O2 and ML techniques do not work properly for excessive number of parallel multioperations unless some kind of an overflow mechanism is used. If the thickness of the TCF is constant, the selection of correct technique can be done automatically by the compiler. The multioperation implementations shown in Section 3 require that the mapping of individual operations from the FE to the BEs needs to follow the default stacking function shown in Fig. 3. Selecting the correct mapping can easily be handled by the compiler.

In order to demonstrate multioperation programming, let us consider the min, prefix-add, csort and rsort benchmarks that compute a minimizing reduction, additive prefix sum and sort an array of integers in constant number of steps, respectively. The high level implementation of min is shown in Fig. 18. It consists of only a single code line performing the multioperation. The assembler version compiled with the O2 techniques consists of a sequence of a load instruction and BMMIN-EMMIN sequence but not a loop to adapt the software parallelism to hardware one while the ML version consists the sequence MLMIN-EMMIN only. The high level implementation of prefix-add is shown in Fig. 18. Also it consists of a single line implementing the multiprefix operation. The assembler version is compiled using the MMP technique resulting into a program with a sequence of MPLADD-OMPADD-MPSADD.

The high level implementation of csort is shown in Fig. 19. It consists of just two statements executed with thicknesses $N^2$ and $N$, respectively. The first statement performs $N$ parallel multioperations using the construct fast_multi() over the results of comparing all $N$ data elements to each other. The operations of the algorithm is illustrated in Fig. 20 for $N = 4$. For the FS techniques, the algorithm compiles to just two sequences of three assembler instructions that all involve execution in both the frontend and backends. The TPA assembler listing of the csort is shown in Fig. 19. The execution time of this program is six steps in TPA. The simulations showing the execution time in clock cycles are shown in Fig. 13 under the label FS.

For multiprefixes, our idea is to likewise reuse the CESM notation [28]. In it the standard three-phase multiprefix is denoted with a high-level construct prefix(p,OP,tgt,src) and fast_prefix(p,OP,tgt,src), that implement multiprefix OP for the data obtained from src, save the corresponding multioperation result in tgt, and return the prefixes in p.

The high level language implementation of rsort benchmark is shown in Fig. 21. It consists of just three statements executed with thicknesses $N$, $R$ and $N$, respectively. The first statement performs a fast multioperation for counting the occurrences of each integer in range $[0, \ldots, R-1]$ into array B. The next statement computes the additive multiprefix of B revealing the number of predecessors for each integer. Finally, the third statement uses this information to move each input integer onto its own place in array D so that the input array is sorted. The TPA assembler listing of the rsort is shown in Fig. 21 grouped to three groups of instructions corresponding to the three statements of the high-level language version. The execution time of this program is eight steps in TPA (see Fig. 15 under the label TPA MCRCW).

Programming and analyzing parallel algorithms for ESM computers is discussed beyond multioperations and multiprefixes, e.g., in [3,8,18,28,29].

## 6. Conclusion

In this paper we have described architectural solutions for realizing reductions with multi(prefix)operations in thick control flow (TCF) architectures, such as TPA. On the processor side, our solution relies on step caches and equally sized multioperation scratchpads, while on the memory side, we make use of active on-chip memory modules. We also discussed solutions utilizing the frontend unit instead of active memory units, or conditional splitting of fibers of thick control flows. Asymptotically, the proposed techniques can speed up execution of certain $N$ data element algorithms by a factor of $\log N$ when comparing to similar processors without multioperations. The actual speedups depend on the selected techniques, number of concurrent multioperations, execution units and problem size. The highest practical speedup factor – 43.82 for 64K data elements on a 16-backend TCF-aware processor – was achieved in the min benchmark using the multioperation load technique. The best acceleration factor in multiprefix operations, 23.93, was achieved with memory-to-memory multiprefix technique in the case of conditional multiprefix computation. We also provided insights into the selection and limitations of the proposed multi(prefix)operation techniques. Based on the needed additional memory capacity for scratchpads and small ALUs, we estimate that the silicon area and power consumption overheads in this case are modest compared to the baseline processors even when all the proposed techniques are implemented. Finally, we show how these operations can be used to accelerate real programs both in assembler and high-level languages.

We intend to develop TCF computing hardware and methodology further. This includes, e.g., implementing a TPA with the proposed additions on an FPGA and hopefully also in silicon, and comparing it more widely to existing CPU architecture and accelerators, such as GPUs. Our aim is to eventually commercialize the TCF processor technology to speed up future multiprocessors and better support parallel application development.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

The data that has been used is confidential.

## Acknowledgments

This work was funded by VTT, Finland and the grant 1982/31/2021 of Business Finland.

## References

[1] M. Forsell, J. Roivainen, V. Leppänen, J.L. Träff, Implementation of multioperations in thick control flow processors, in: IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshops, 2018, pp. 744–752.

[2] M. Snir, Depth-size trade-offs for parallel prefix computation, J. Algorithms 7 (2) (1986) 185–201.

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

[4] J. Schwartz, Ultracomputers, ACM Trans. Prog. Lang. Syst. 2 (4) (1980) 484–521.

[5] D. Gajski, D. Kuck, D. Lawrie, A. Sameh, CEDAR-A large scale multiprocessor, in: Proceedings of International Conference on Parallel Processing, 1983, pp. 524–529.

[6] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E. Melton, V.A. Norton, J. Weiss, The IBM research parallel processor prototype (RP3): Introduction and architecture, in: Proceedings of International Conference on Parallel Processing (ICPP), 1985, pp. 764–771.

[7] A. Ranade, S. Bhatt, S. Johnsson, The Fluent Abstract Machine, Tech. rep., Technical Report BA87-3, Department of Computer Science, Yale University, 1987.

[8] J. Keller, C.W. Keßler, J.L. Träff, Practical PRAM Programming, John Wiley & Sons, 2001.

[9] M. Forsell, A scalable high-performance computing solution for networks on chips, IEEE Micro 22 (5) (2002) 46–55.

[10] U. Vishkin, Using simple abstraction to reinvent computing for parallelism, Comm. ACM 54 (1) (2011) 75–85.

[11] A.G. Ranade, How to emulate shared memory, J. Comput. System Sci. 42 (3) (1991) 307–326.

[12] M. Forsell, Realising constant time parallel algorithms with active memory modules, Int. J. Electron. Bus. 3 (3/4) (2005) 255–263.

[13] M. Forsell, Realizing multioperations for step cached MP-SOCs, in: Proceedings of the International Symposium on System-on-Chip (SOC), 2006, pp. 77–82.

[14] M. Forsell, J. Roivainen, REPLICA T7-16-128 - a 2048-threaded 16-core 7-FU chained VLIW chip multiprocessor, in: 48th Asilomar Conference on Signals, Systems, and Computers, Special Session on Multicore, Manycore and Distributed Systems, 2014, pp. 1709–1713.

[15] M. Forsell, V. Leppänen, An extended PRAM-NUMA model of computation for TCF programming, Int. J. Netw. Comput. 3 (1) (2013) 98–115.

[16] M. Forsell, J. Roivainen, V. Leppänen, Outline of a thick control flow architecture, in: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) Workshops, 2016, pp. 1–6.

[17] M. Forsell, J. Roivainen, V. Leppänen, J.L. Träff, Supporting concurrent memory access in TCF processor architectures, Microprocess. Microsyst. Embedded Hardw. Des. 63 (2018) 226–236.

[18] M. Forsell, S. Nikula, J. Roivainen, Preliminary performance and programmability comparison of the thick control flow architecture and current multicore CPUs, in: Advances in Parallel & Distributed Processing, and Applications (PDPTA), Springer International Publishing, 2021.

[19] J. Fisher, Very long instruction word architectures and ELI-512, in: Proceedings of the 10th Annual International Symposium on Computer Architecture, 1983, pp. 140–150.

[20] M. Forsell, Minimal pipeline architecture - an alternative to superscalar architecture, Microprocess. Microsyst. Embedded Hardw. Des. 20 (5) (1996) 277–284.

[21] R. Hintz, D. Tate, Control data STAR-100 processor design, in: COMPCON, 1972, pp. 1–4.

[22] W.J. Watson, The TI ASC: a highly modular and flexible super computer architecture, in: Proceedings of the AFIPS Fall Joint Computer Conference, 1972, pp. 221–228.

[23] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, first ed., Morgan Kaufmann Publishers Inc., 1990.

[24] G. Lento, Optimizing Performance with Intel® Advanced Vector Extensions, Tech. Rep., Intel, 2014, https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf.

[25] Introducing DNA Architecture: The All New Radeon™ Gaming Architecture Powering NAVI, Tech. Rep., Advanced Micro Devices, 2019, https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[26] Nvidia Ampere GA102 GPU Architecture: Second-Generation RTX, Tech. Rep., NVIDIA, 2021, https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[27] M. Forsell, Step caches–a novel approach to concurrent memory access on shared memory MP-SOCs, in: Proceedings of the 23rd IEEE NORCHIP Conference, 2005, pp. 74–77.

[28] M. Forsell, Faster implementation of e for multioperation concurrent read concurrent write MP-SOCs, WSEAS Trans. Comput. 6 (1) (2007) 103–110.

[29] F. Ghanim, U. Vishkin, R. Barua, Easy PRAM-based high-performance parallel programming with ICE, IEEE Trans. Parallel Distrib. Syst. 29 (2) (2018) 377–390.

**Martti Forsell** is Principal Scientist of processor architecture and parallel computing at VTT, Oulu, Finland as well as Adjunct Professor of computer architecture in the Faculty of Information Technology and Electrical Engineering at the University of Oulu. He is known for his work for the Flow-computing multiprocessor framework aiming to solve the performance and programmability bottlenecks of current multicore CPUs. Prior to joining VTT, he was affiliated with the University of Joensuu, where he also received his M.Sc. in 1991, Ph.Lic. in 1994 and Ph.D. in 1997.

**Jussi Roivainen** is Senior Scientist at VTT, Efficient Computation and Communications, Oulu, Finland. His research interests include digital logic implementations of processors and baseband.

**Ville Leppänen** is 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 a Vice dean (education) of Faculty of Technology and leader of 7 research and development projects.

**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.