Outline of a Thick Control Flow Architecture

Martti Forsell, Jussi Roivainen
Computing Platforms
VTT
Martti.Forsell@VTT.Fi, Jussi.Roivainen@VTT.Fi

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

Abstract*—The recently invented thick control flow (TCF) model packs together an unbounded number of fibers, thread-like computational entities, flowing through the same control path. This promises to simplify parallel programming by partially eliminating looping and artificial thread arithmetics. In this paper we outline an architecture for efficiently executing programs written for the TCF model. It features scalable latency hiding via replication of instructions, radical synchronization cost reduction via a wave-based synchronization mechanism, and improved low-level parallelism exploitation via chaining of functional units. Replication of instructions is supported by a dynamic multithreading-like mechanism, which saves the fiber-wise data into special replicated register blocks. The architecture facilitates programmers with compact, unbounded notation of fibers and groups of them together with strong synchronous shared memory algorithmics. According to evaluations, the architecture is able to efficiently handle workloads featuring computational elements with the same control flow, independently of the number of elements. In its turn, this pays out as improved performance and lower power consumption due to elimination of redundant parts of computation and machinery.

***Keywords*—parallel computing; processor architecture; programming model; TCF; multithreading; chaining**

I. Introduction

The history of parallel processor architectures has been a series of trade-offs between bold ideas towards stronger computational models (and higher performance) versus compromises to be able to implement technically and commercially viable products [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. So far, the implementation related aspects have shown dominance over the theoretical strength [11, 12, 13, 14]. As a result, current parallel processors support asynchronous programming models rather than synchronous ones, prefer cache-based memory hierarchies over parallel slackness-based solutions, and make use of dynamic superscalar instruction-level parallel techniques [2, 15] instead of more general combination of low-level inter-thread and instruction-level parallelism [16]. Despite of many practical advantages, this dominant approach has introduced a number of problems, including tedious programmability and weak portability among machines with different parameters as well as limited performance in applications with frequent intercommunication and interthread dependencies [16, 17, 18, 19].

The *emulated shared memory* (ESM) architectures take the opposite approach building on idealized shared memory rather than straightforward replication of hardware and therefore provide simpler programmability [20]. They provide a mechanism to compensate the intercommunication and synchronization latency as well as speed difference between processors and memories. Although the first attempts to this direction were not able to catch up the performance of commercial offerings, the latest designs have shown potential to surpass current chip multiprocessors and GPGPUs [21].

At high level, most CPU architectures make use of the *multiple instruction stream multiple data stream* (MIMD) model, which picks up P instructions from separate instruction flows and executes them in P processing elements. While MIMD is a flexible model, it potentially wastes resources for code containing self-similarities. Consider e.g. the case of T data parallel threads, where all the threads are executing the same code but with different data. This includes potentially T-way replicated base addresses, registers containing them, part of the intermediate results, and even processor elements, like sequencers and fetchers (see Figure 1). A more optimal solution for this kind of code would be to use the *single instruction stream multiple data stream* (SIMD) in which the same instruction for P data elements is executed in P processing units. While SIMD is more cost-efficient, easier to program and implementations of it feature lower power consumption, it does not execute efficiently code with control parallelism and heterogeneity between the threads (see Figure 1). Another dimension of problems is raised by the fact that in implementations of MIMD and SIMD architectures the number of hardware threads is fixed and causes overhead when the number of software threads exceeds the supported one. To seamlessly adapt to code portions with different characteristics with respect to homogeneity, heterogeneity, number of threads and be able to execute them efficiently, we study a possibility to take advantage of the SIMD-style optimizations in ESM architectures and retain possibility to have multiple streams of the MIMD model where it is necessary. The key mechanism in this would be applying the *thick control flow* (TCF) model [22] in which homogeneous threads going via the same control path are combined into entities called TCFs. We give an overview of the model in Section II.

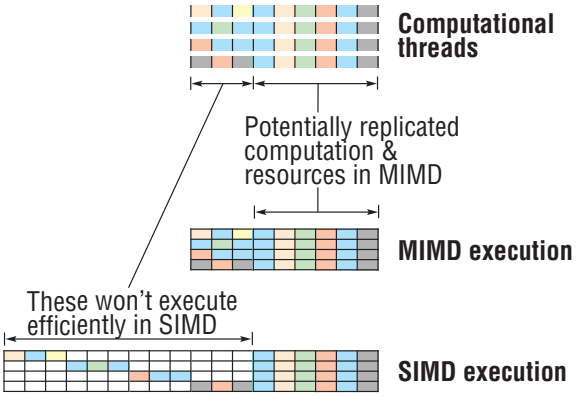

Figure 1. Computation organized as threads containing homogeneous and heterogeneous computation and corresponding MIMD and SIMD execution. Horizontal lines represent execution in a processing elements, vertical lines parallel operations and colors represent different kinds of operations.

A. Previous results

There are no known architectural implementation of the TCF model but some previous results overlap with the work presented in this paper.

The XMT architecture [23] implements the idea of forking to unbounded number of threads for independent parallel portions of code. This happens by picking up available threads to thread units and executing them from the beginning to the end of the portion. As soon as a thread unit has completed a thread it picks up one of the remaining threads until there are no threads left. Compared to our proposal, XMT does not provide synchronous execution of instructions but executes entire threads from the fork they are created to corresponding join and lacks the possibility for exploiting low-level interthread parallelism.

GPGPUs use multithreading to tolerate latency and multiple SIMD executions to reduce the silicon area and power consumption like the proposed architecture. On the other hand, the pipeline, connection to CPUs, threading, synchronization scheme, dynamic nature of TCFs and memory system differentiate the proposal from GPGPUs.

The Cilk programming language supports simple management of threads and data parallelism. The latest version by Intel adds support for array notation and SIMD enabled functions. The proposed TCF-architecture provides native implementation of many primitives and constructs for it and thus, potentially increase performance.

As a part of our work with the REPLICA chip multiprocessor framework, we have defined a number of *configurable emulated shared memory* (CESM) constellations [24]. These share the synchronization, latency hiding and low-level parallelism mechanisms with the proposal but do not support dynamic unbounded number of fibers neither simultaneous use of machineries for modestly and highly parallel parts of the code.

Finally, in our previous works we have introduced the TCF model [22], linked it the shared memory emulation at the model level [10] but have not attempted to present an implementation architecture for it.

B. Contribution

In this paper we outline an architecture for efficiently executing programs written for the TCF model. It features scalable latency hiding via replication of instructions, radical synchronization cost reduction via a wave-based synchronization mechanism, and improved low-level parallelism exploitation via chaining of functional units (FU). Replication of instructions is supported by a dynamic multithreading-like mechanism, which saves the fiber-wise data into special replicated register blocks. The architecture facilitates a programmer with compact, unbounded notation of fibers and groups of them together with strong synchronous shared memory algorithmics. According to evaluations, the architecture is able to efficiently handle workloads featuring computational elements with the same control flow, independently of the number of elements. In its turn, this pays out as improved performance and lower power consumption due to elimination of redundant parts of computation and machinery.

The rest of the paper is organized so that Section 2 describes the TCF programming model, Section 3 outlines the TCF processor architecture, Section 4 evaluates the architecture with simulations, and Section 5 draws conclusions as well as outlines some future work.

II. TCF Programming Model

The key for simpler programming is to raise abstraction related to composition and synchronization of parallel programs. The TCF model does this by packing together computational elements containing similarities for exposing natural synchronicity of parallel execution and providing a simple explicit mechanism for dynamically adjusting the number of elements executed in parallel.

The *Thick Control Flow* (TCF) model is a programming model combining homogeneous computations having the same control flow into data parallel entities controlled by a single control rather than keeping individual control for each computation [22]. The resulting entity is simply called *TCF* and components of it are called *fibers* to distinguish them from more independent threads. The number of fibers in a TCF is called *thickness* of it.

Execution of a TCF happens one instruction at the time. The time, during which all fibers of a TCF execute an instruction, is called a *step*. TCF execution resembles SIMD execution but there can be multiple TCFs executing simultaneously and their thicknesses can vary arbitrarily. The TCF model guarantees synchronicity and strict memo-

ry consistency between consecutive instructions so that all shared memory actions launched by the previously replicated instruction are guaranteed to complete before the operations of the current one take place. This reduces the cost of synchronization w.r.t. ESM since synchronizations happen once per step defined by software not hardware. If there are multiple TCFs, the mutual execution ordering of them is not strictly defined but the programmer can instruct them to follow certain ordering via explicit inter-TCF synchronizations. Since the thickness of a TCF can be defined by the programmer without bounds, it is easy to express intrinsic parallelism of a wide spectrum of computational problems without a worry about running out of threads or having to matching the software parallelism with the hardware one with loops or explicit threads. Besides reducing the number of loops, the model simplifies programming also by eliminating index calculations related to threads. The example of Figure 1 can now be expressed with 4 TCFs and a change of thickness from one to four between the heterogeneous and homogeneous parts (see Figure 2).

The TCF model can basically be programmed like other parallel programming models but the nature of model opens up possibilities for novel conventions that have substantial implications to notation, behavior and interpretation of computation [22].

III. Thick Control Flow Processor Architecture

Our proposal for a TCF-aware architecture leans on four observations on the TCF model and typical execution of instructions:

(1) **Single Control:** A TCF features a single control flow common to all fibers.
(2) **Mostly shared state:** Most of the state of TCF is shared between the fibers. This includes e.g. base addresses of arrays and register copies of shared variables.
(3) **Homogeneity:** Parallelism between the fibers of TCF is always homogeneous and refer to fiber-dependent data.
(4) **Dynamic thickness:** The degree of parallelism within a TCF can range from modest to unbounded.

Unfortunately (1)-(2) point to traditional single core architecture while (3)-(4) suggest that highly parallel ESM-style multicore architecture would be the best fit if it could be made to work smoothly with dynamic parallelism. Finally the architecture should ultimately support execution of multiple TCFs both in parallel and overlapped way.

In the following we outline a TCF-aware architecture based on the observations as well as explain the structure and operation of it but rule out lower-level details.

A. Structure and used architectural techniques

The *Thick Control Flow Processor Architecture* (TPA) is an architecture implementing natively the TCF programming model. A TPA chip multiprocessor consists of S scalar[1] processor frontends attached to instruction memory modules, local memories and TCF buffers, as well as P parallel processor backends, attached to replicated register blocks, and shared memory modules via the communication network (see Figure 3).

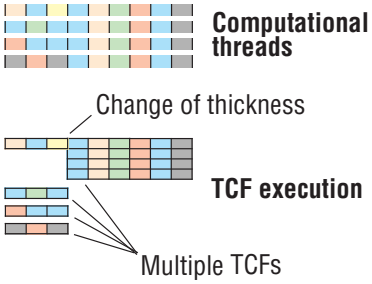

Figure 2. Computation organized as threads containing homogeneous and heterogeneous computation and corresponding TCF execution with a change of thickness.

The processor frontends support fast switching between a number of TCFs, management of them, execution of control and other common parts of the code. Architecturally they resemble multithreaded superscalar processors but instead of threads, TCFs with their own register sets kept in the TCF buffers. The memory systems of frontends consist of instruction memories and local data memories optionally interconnected via a network making use of the *non-uniform memory access* (NUMA) convention with cache coherence maintenance [12].

The processor backends support streamlined shared memory system with fiber-wise data and execute homogeneous parallel parts of the code. The structure of them resembles that of ESM pipelines with capability of dynamic fiber execution. Like ESM processors they use scalable latency hiding via multifibering, radical synchronization cost reduction via wave synchronization, and improved low-level parallelism exploitation via chaining of FUs to provide high performance in parallel execution.

The idea in *multifibering* is to execute other fibers while a reference of a fiber proceeds in the memory system. If the number of fibers is high enough and the intercommunication network is not congested, the reply arrives before it is needed by the fiber. The first synchronization method exploits the fact that fibers are independent within a step of execution and performs the synchronization action only once per step. The second one allows to overlap synchronizations with memory references. These together define the *low-cost synchronization wave mechanism* in which the amortized overhead caused by synchronization drops down

[1] The term *scalar* refers here to processing of single thread rather than single functional unit to emphasize the difference between the frontend and parallel/vector backend. Thus, we allow the scalar processor use static or dynamic superscalar execution.

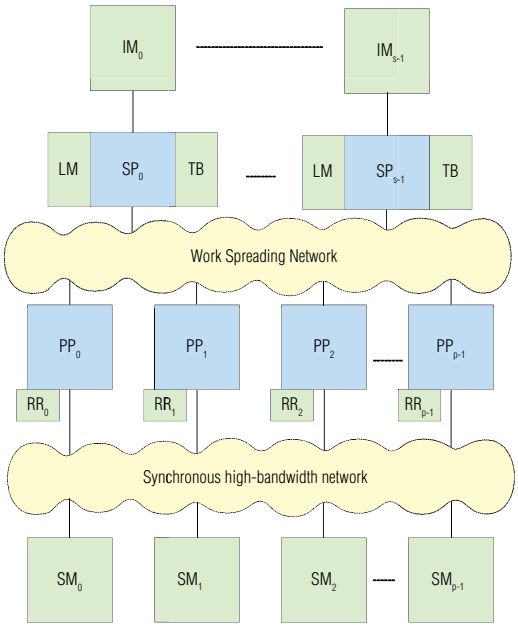

Figure 3. The overall structure of TPA (IM=instruction memory, SP=processor frontend, LM=local memory, TB=TCF buffer, WS=work spreading, PP=processor backend unit, RR=replicated register block, SM=shared memory module).

to 1/thickness [20]. The idea in *low-level parallelism exploitation* is to connect FUs as a chain so that a unit can use the results of its predecessors as operands rather than connect units in parallel requiring operations to be independent. This increases the utilization of FUs by allowing the pipeline to execute dependent instructions.

The interplay between frontends and backends happens by assigning a TCF to a single frontend and multiple frontends in an overlapped way. This kind of a single mode dual-purpose operation avoids the dual mode operation and switching overheads between them present in ESM architectures [24]. Support for *unbounded thickness* is implemented by generating fibers (or replicating instructions) dynamically on the pipeline and saving the fiber-wise data into special replicated register blocks that overflow their content to external memory system if necessary.

B. Operation

Execution of instructions in the TPA architecture differs somewhat from that of the NUMA or ESM architectures. While an F_n-FU NUMA core executes at most F_n independent (sub)instructions in parallel and an F_e-FU ESM executes an instruction with up to F_e subinstructions for a fixed number of threads per step in interleaved manner, the TPA executes sequences of frontend instructions and parallel backend instructions for a variable number of TCFs with non-constant thickness in interleaved manner. More specifically, execution of a single TCF assigned to a frontend and a number of backends units in TPA happens as follows:

- First the frontend (responsible of managing TCFs) fetches the next TCF from its TCF buffer and executes the sequence of scalar instructions defined by the program counter and instruction memory. As it meets an instruction containing a backend operation, it tries to send the operation along with its operands to the related backend units via the FIFO-like *work spreading* (WS) network. If the network is full then the frontend waits until there is room for the TCF and then continues until there is an explicit TCF switch request.
- Execution in the related backend units starts by checking whether all of them are free. In the positive case, the parallel operations, their operands and necessary TCF info on the head element of WS gets fetched into execution to all related backend units. In the negative case, the instruction in the FIFO waits for parallel units to become free.
- The backend then splits the TCF horizontally between the related backend units as evenly as possible and starts to generate and process fibers in parallel until they run out.
- During the fiber generation each fiber gets its operands, fiber identifier and instructions for FUs from the data sent by the frontend while the fiber-wise intermediate results are fetched from the replicated register block.

The length of the sequence for the frontend is at least one instruction and containing zero or more backend instructions. For a TPA with P backend units and a TCF of thickness T, the utilization of execution units achieves its maximum as the length of sequence is T/P instructions and there is exactly one backend instruction in the sequence.

IV. Evaluation

In order to show that TPA meets our design goals, we implemented it as a clock accurate low-level simulation with an equally accurate method as our FPGA proof-of-concept reference [24]. The evaluated configuration has 16 64-bit 8-FU backend units and one slightly modified 64-bit version of a 5-FU *Minimal Pipeline Architecture* (MPA) processor [25] as the frontend (see Table 1).

We wrote 4 kernel programs for initial evaluation (see Table 2) and simulated their execution in TPA. For each program, we determined the execution time, amount of data transferred from the frontend to backend, and usage of replicated register block. For comparison purposes, we measured also the execution time of $\#_{CESM}$-threaded ESM versions of them with REPLICA CESM [24] with a identical configuration. CESM uses the same latency hiding, and low-level parallelism techniques as TPA but features fixed number of threads. As a side efect, its synchronization overhead is fixed rather than tied to data set size.

The results of simulations are shown in Figure 4. We can make the following observations from the results:

Processor	CESM [Forsell15]	TCF [This proposal]
Processing units	1 NUMA/16 Parallel	1 frontend/16 backend
Threads per processor core	128	Unbounded
TCFs per frontend	-	128
Number of functional units	8	5/8
Interconnect	4x4 mesh	4x4 mesh

Table 1. Tested processors.

Benchmark	Description
addmat	A constant step time parallel program that adds two arrays of *N* integers (*tests simple matrix operations*)
fftdec	A parallel program that performs bitreversal shuffling of elements (fft decomposition) of an array of *N* integers (*tests elimination of intermediate variables in parallel dependent operations*)
logprefix	A parallel program that determines the prefix sums of an array of *N* integers in the shared memory with the logarithmic algorithm (*suboptimal, tests a highly dependent memory access pattern with changing thickness*)
parseq	A program that performs a parallel block copy of *N* integers in the shared memory followed a sequential block copy of *N/P* integers in local memory and finally repeats the parallel block copy (*tests shared and local memory system operation as well as switching between parallel and sequential execution*)

Table 2. Test programs for the proposed TCF-aware architecture.

- The performance of the proposal turns to be higher in all measurements. The main reason for this is higher utilization of fibers/threads due to the TCF model's better ability to assign workload seamlessly to execution units.
- The highest speedup is achieved in the **parseq** test due to TPA's ability to overlap sequential parts executed in the frontend and parallel parts executed in the backend as well as absence of switching time between them.
- The advantage of TPA decreases as the problem size grows. This is caused by relatively reducing fraction of slowly executing code in CESM, e.g. loop overhead, copyback of data in the case of dependent data fulfilling the condition $N > \#_{\mathrm{CESM}}$.
- In these benchmarks, TPA shows smooth execution time growth very close to algorithm theoretical one while CESM shows saw-shaped growth (see Figure 4/**addmat**). The reason for this quantization is rounding the effective size upwards to the next multiple of $\#_{\mathrm{CESM}}$.
- The amount of data transferred from the frontend to backend ranges from zero to 4 words. The average traffic per step is below 0.02 words per clock cycle for small *N* and below 0.002 for large *N*. The usage of replicated register block ranges from 0 to 2 words per fiber.

V. Conclusions

In this paper we have outlined a processor architecture allowing a programmer to combine "threads" following the same control flow and containing enough homogeneity into flexible computational entities called thick control flows and executing them efficiently. The architecture makes use of the key techniques of ESM—scalable latency hiding, low-cost synchronization and efficient low-level parallelism exploitation—to provide high performance and easy programmability. It addresses also typical limitations of MIMD and SIMD architectures by providing native support for unbounded number of "threads", eliminating unwanted replication of registers and instruction memory system resources leading to reduced energy consumption.

According to our evaluation, the processor indeed implements the TCF model, provides a higher performance than CESM with similar configuration especially for small problem sizes. Prior implementation of CESM [24] and the fact that in the proposal the amount of data transferred from the frontend to backend and stored in the replicated register block turned out to be modest indicating that the architecture would be implementable.

In the future we aim to develop a number of additional architectural and methodological optimizations on the top of this proposal, including multiple scalar units, multioperation support, detailed memory system, replicated register block implementation as well as prototype it with FPGA.

Acknowledgment

This work was funded by VTT, the grant 289773 of Academy of Finland and the Celtic-Plus project CONVINcE.

References

[1] E. Bloch, The engineering design of the Stretch computer, *Proc. of the Fall Joint Computer Conference*, 1959, 48-59.

[2] R. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM Journal of Research and Development* **11**, 1 (1967), 25-33.

[3] M. Flynn, Some Computer Organizations and their Effectiviness, *IEEE Trans. Comput.* **21**, 9 (1972), 948-960.

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

[5] J. Fisher, Very Long Instruction Word Architectures and ELI-512, *Proc. 10th Annual Int. Symp. on Computer Architecture*, Computer Society Press, Washington, 140-150.

[6] J. Kowalik (editor), *Parallel MIMD computation: the HEP supercomputer and its applications*, MIT Press, Cambridge, 1985.

[7] D. Skillicorn and D. Talia, Models and Languages for Parallel Computation, *ACM Computing Surveys* **30**, 2 (June 1998), 123-169.

[8] S. Rajasekaran and J. Reif, *Handbook of Parallel Computing—Models, Algorithms and Applications*, Chapman & Hall/CRC, Boca Raton, 2008.

[9] G. Bilardi, K. Ekanadham and P. Pattnaik, On approximating the ideal random access machine by physical machines. *Journal of the ACM* **56**, 5 (August 2009), Article 27:1-57.

[10] M.Forsell and V. Leppänen, An Extended PRAM-NUMA Model of Computation for TCF Programming, *Int. Journal of Networking and Computing* **3**, 1 (2013), 98-115.

[11] R. Swan, S. Fuller and D. Siewiorek, Cm*—A Modular Multiprocessor, *Proc. NCC*, 645-655, 1977.

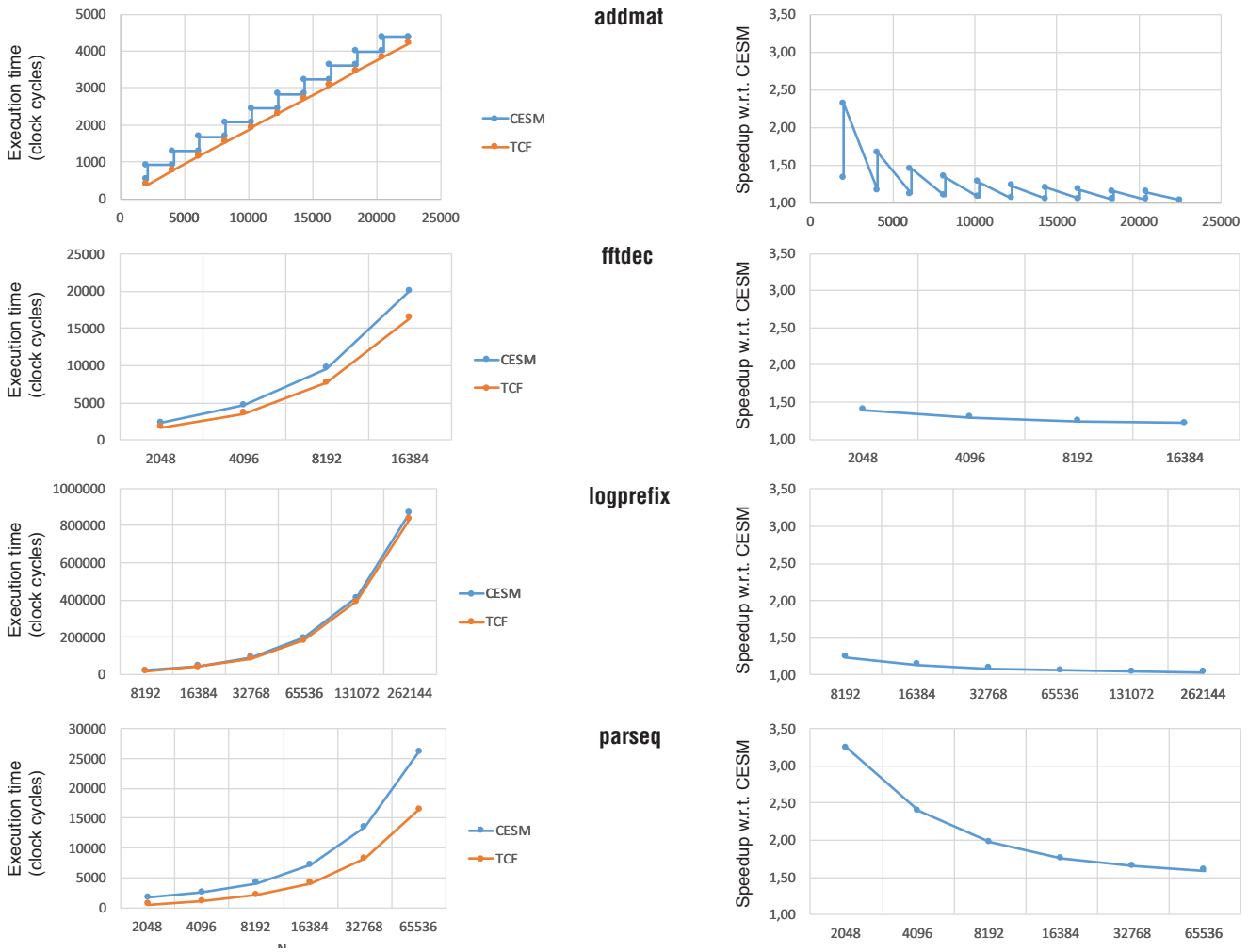

Figure 4. The execution time in TCF and CESM and speedup with respect to CESM as the function of problem size *N* for the benchmark programs.

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

[13] D. Culler and J. Singh, *Parallel Computer Architecture—A Hardware/ Software Approach*, Morgan Kaufmann Publishers Inc., San Fransisco, 1999.

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

[15] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., Palo Alto, 1990.

[16] M. Forsell, Architectural differences of efficient sequential and parallel computers, *Journal of Systems Architecture* **47**, 13 (July 2002), 1017-1041.

[17] R. Bocchino, V. Adve S. Adve and M. Snir, Bocchino, *First USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*, March 30–31, 2009, Berkeley, USA.

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

[19] The HiPEAC Vision for Advance Computing in Horizon 2020, HiPEAC Network of Excellence, 2013; http://www.hipeac.net/system/files/hp-roadmap-2013.pdf.

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

[22] V. Leppänen, M. Forsell and J-M. Mäkelä, Thick Control Flows: Introduction and Prospects, *Proc. 2011 Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, July 18-21, 2011, Las Vegas, USA, 540-546.

[21] E. Hansson, E. Alnervik, C. Kessler and M. Forsell, A Quantitative Comparison of PRAM basedcEmulated Shared Memory Architectures to Current Multicore CPUs and GPUs, *Proc. 11th Workshop on Parallel Systems and Algorithms (PASA'14)*, February 25-26, 2014, Luebeck, Germany, 1-7.

[23] U. Vishkin, Using Simple Abstraction to Reinvent Computing for Parallelism, *Communications of the ACM* **54**, 1 (January 2011), 75-85.

[24] M. Forsell and J. Roivainen, REPLICA T7-16-128 - A 2048-threaded 16-core 7-FU chained VLIW chip multiprocess*or, 48th Asilomar Conference on Signals, Systems, and Computers*, November 2-5, 2014, Pacific Grove, USA, 1709-1713.

[25] M. Forsell, Minimal Pipeline Architecture—an Alternative to Superscalar Architecture, *Microprocessors and Microsystems* **20**, 5 (1996), 277-284.