Apple-CORE: harnessing general-purpose many-cores with hardware concurrency management

R. Poss*, M. Lankamp, Q. Yang, J. Fu, M.W. van Tol, I. Uddin, C. Jesshope

University of Amsterdam, The Netherlands

Abstract

To harness the potential of CMPs for scalable, energy-efficient performance in general-purpose computers, the Apple-CORE project has co-designed a general machine model and concurrency control interface with dedicated hardware support for concurrency management across multiple cores. Its SVP interface combines dataflow synchronisation with imperative programming, towards the efficient use of parallelism in general-purpose workloads. Its implementation in hardware provides logic able to coordinate single-issue, in-order multi-threaded RISC cores into *computation clusters on chip*, called Microgrids. In contrast with the traditional "accelerator" approach, Microgrids are components in *distributed systems on chip* that consider both clusters of small cores and optional, larger sequential cores as *system services* shared between applications. The key aspects of the design are asynchrony, i.e. the ability to tolerate irregular long latencies on chip, a scale-invariant programming model, a distributed chip resource model, and the transparent performance scaling of a single program binary code across multiple cluster sizes. This article describes the execution model, the core micro-architecture, its realization in a many-core, general-purpose processor chip and its software environment. This article also presents cycle-accurate simulation results for various key algorithmic and cryptographic kernels. The results show good efficiency in terms of the utilisation of hardware despite the high-latency memory accesses and good scalability across relatively large clusters of cores.

Keywords: chip multiprocessors, many-cores, hardware multithreading, dataflow scheduling, operating systems in hardware, microgrids

*Corresponding author

1. Introduction

Ever since the turn of the century, fundamental energy and scalability issues have precluded further performance improvements for *single threads* [1]. To "cut the gordian knot," the industry has since shifted towards multiplying the number of processors on chip, creating increasing larger Chip Multi-Processors (CMPs) by processor counts, to take advantage of efficiency gains made possible by frequency scaling [1, 2].

This shift to multi-core chips has caused a commotion in those software communities that had gotten used to transparent frequency increases and implicit instruction-level parallelism (ILP), without ever questioning the basic machine model targeted by programming languages and complexity theory. "The free lunch is over" [3], and software ecosystems now have to acknowledge and understand explicit on-chip parallelism and energy constraints to fully utilize current and future hardware.

We propose that while general-purpose programmers have been struggling to identify, extract and/or expose concurrency in programs during the last ten years, a large amount of untapped higher-level parallelism has appeared in applications, ready to be exploited. This is a consequence of the increasing number of features, or *services* integrated into user-facing applications. In other words, while Amdahl's law stays valid for individual programs, we should recognize that Amdahl did not predict that *single* users would nowadays be routinely running so *many* loosely coupled programs *simultaneously*. This thus begs the question: *assuming that multi-scale concurrency in software has become the norm*, what properties should we expect to find in general-purpose processor chips? This is the question that the project *Architecture Paradigms and Programming Languages for Efficient programming of multiple CORES* (Apple-CORE) attempted to answer.

The Apple-CORE strategy to answer this question was holistic, groundbreaking and perhaps suprisingly also conservative. The holistic side was to co-design a processor chip, together with its programming model, tool chain and specialized benchmark applications. We outline the corresponding innovations in section 3. The conservative side was a careful attention for compatibility with existing application code, in particular standard C and the traditional C/Unix execution environment. We touch on these aspects in section 5. The outcome of Apple-CORE is intriguing: it is possible to give up many hardware features found in traditional general-purpose core designs, such as branch predictors and interrupt-based control flow preemption, and replace them with new features that are slightly less advantageous for purely sequential workloads, but greatly advantageous for heterogeneous, mixed sequential/parallel workloads. To demonstrate this result, Apple-CORE has produced both FPGA-based prototypes and production-grade simulation and compiler software, and applied this technology to common software workloads; an extract of the corresponding empirical experiments is reported on in section 5. The Apple-CORE technology has also been made publicly available for further research in this area, even for third parties without access to a physical implementation of the proposed design.

2. Context and design motivations

We have detailed the arguments that have justified Apple-CORE's design directions in a previous publication [4]; we provide a summary here.

The first concern was *avoiding model assymmetry*: while hardware heterogeneity is a proven approach to increase efficiency and decrease costs, the desired adoption of multi-cores is only possible if the machine model exposes a single programming model for all resources, instead of different programming interfaces for each component. Towards this, Apple-CORE has centered its approach on its SVP protocol, which coordinates the various computing resources on chip with a single set of semantics.

A related interest was to *acknowledge the limitations of specialization*: while desirable for specific application scenarios, hardware specialization increases the amount of component properties that must be described to resource managers in operating systems and compilers. Apple-CORE instead promotes simple, adaptable cores that can be pooled in clusters of configurable sizes depending on application requirements, thereby introducing resource *fungibility*[1].

As the chip size grows relative to the gate size, so does the latency between on-chip components (cores, caches and scratchpads) relative to the pipeline cycle time; this divergence is the on-chip equivalent of the "memory wall" [5]. Moreover, inter-component latencies will become increasingly

[1] *Fungibility* is the property of a good or a commodity whose individual units are capable of mutual substitution. Example fungible commodities include crude oil, wheat, precious metals, and currencies.

unpredictable, both due to overall usage unpredictability in heterogeneous application scenarios and due to soft errors in circuits. These latencies cannot be easily tolerated using superscalar issue or VLIW, for the reasons outlined in [6]; Apple-CORE instead promotes the known solution, that is, *tolerate unpredictable on-chip latencies using hardware multi-threading (HMT) with dynamic scheduling* over shorter pipelines.

Finally, Apple-CORE has acknowledged that concurrency in software comes with varying granularity. The state of the art in concurrency managers deals well with coarse-grained concurrency, for example external I/O or regular, wide-breadth concurrency patterns extracted from homogeneous sequential tight loops: these can be managed in software schedulers with coarse threads or via blocking aggregation (e.g. OpenMP [7]). Meanwhile, software schedulers struggle with fine-grain heterogeneous task concurrency, such as found in graph reduction, irregular tight loops or data flow algorithms. In these latter cases, a strain is put on compilers and run-time systems: they must determine the suitable aggregate units of concurrency from programs that both optimize load balancing and compensate concurrency management costs. This motivates the acceleration of space scheduling, considered as a system function, using dedicated hardware logic. This idea to re-introduce *hardware support for concurrency management* after a hiatus of twenty years [8, 9, 10, 11, 12] is further motivated by three new arguments: the first is that software ecosystems have become more accepting of expressing concurrency explicitly (e.g. in the output of automated parallelization, which has recently tremendously matured); the second is that accelerated hardware support makes it possible to reuse clusters of simple cores as an efficient substitute for specialized SPMD/SIMD units; third, placing system functions in dedicated hardware units makes the cost of concurrency management more predictable by avoiding the interleaving of computations with system activities on the memory subsystem.

3. Outline of the project's strategy

The management of concurrency in hardware is central to Apple-CORE's strategy. The main principle is to expose all known concurrency in software at the hardware/software interface using new instructions in the ISA. To achieve this, Apple-CORE combines two previous developments.

3.1. Contributions to processor architecture: Microgrids

At the hardware level, the project exploits the D-RISC microprocessor core design [13], more specifically its combination of hardware multithreading, dataflow scheduling and specialized hardware manager for inter-core work coordination (creation, distribution, synchronization, communication). This capitalizes on hardware microthreading, an architectural approach previously demonstrated [14, 15, 16] to be successful at tolerating varying latencies on chip, and at providing a simple machine model and powerful primitives in the ISA to control the management of software concurrency over multiple cores. Apple-CORE has extended D-RISC with new multi-core coordination abilities. The result consists of chip components called *Microgrids*, whose cores can be targeted by software either individually, as a group, or partitioned at run-time into multiple computing resources. We detail the Microgrid architecture in section 4.

What distinguishes Microgrid-based CMPs from the traditional Symmetric Multi-Processor (SMP) architecture vision is that each hardware thread does not appear as a fully-fledged processor from the perspective of operating systems (OSes) in software. In other SMP approaches, control flow and work distribution is under the control of software OSes by means of a timer interrupt, which can preempt any individual hardware thread; notions of software threads and processes merely emerge as artefacts of the software OS's structure. In Microgrids, individual hardware threads are not preemptible; to gain control of a hardware thread and dispatch work to it, software instead uses a *create* instruction in the ISA which triggers a control signal to the on-chip hardware concurrency manager. The notion of software thread thus maps one-to-one to hardware threads. Similarly, synchronization and communication are handled by hardware events instead of the traditional interrupt-based and memory-based management mechanisms found in other OSes, resulting in orders of magnitude shorter management latencies (cf. table 4). We review this protocol in section 4.2.

In short, Microgrids capture in hardware the scheduling and control roles traditionally assigned to operating software. As a consequence, the OS code is not any more needed on most cores, since application software can now directly direct concurrency using the proposed ISA extensions. This enables a simpler core design, which in turn translates into higher utilization and processing density, i.e. more operations per watt and per transistor. We outline some of the corresponding empirical results in section 5.

3.2. Software strategy and programming interface

The success of the Apple-CORE approach was highly dependent on the direct exploitation of the hardware concurrency management by application code. In particular, existing application programming interfaces (APIs) to control concurrency, e.g. POSIX threads, would be too coarse-grained to dispatch work composed of only few instructions to hardware threads on Microgrids. This in turn mandated the direct control of hardware via low-level primitives in the *de facto* standard system-level language, that is C, and their subsequent utilization by application benchmarks.

The Apple-CORE consortium did examine established C language extensions to control Microgrids, for example OpenMP, Cilk [17] or OpenCL [18]. However, prior work was found somewhat inadequate: at the point this analysis was performed, most existing approaches required either strong memory coherency, mandatory run-time support in a software operating system (e.g. for work distribution or memory management) or considered that concurrent sections can only form "leaves" in a computation call graph otherwise centered on a sequential processor. Instead, Apple-CORE used μTC [19, 20, 21] then SL [22] as its system-level interface to the Microgrid hardware. These language extensions transparently expose the proposed hardware protocol for use in application code without additional semantics or software logic.

The addition of new features in C does not necessarily imply that application code must be modified to use them. Indeed, Apple-CORE considered that it is the task of other compilation tools to either offer higher-level language primitives that simplify the use of parallelism, or parallelizing compilers able to extract concurrency from existing sequential code. To demonstrate this, Apple-CORE also supported two related sub-projects: Microgrid support in the functional, productivity-oriented language Single-Assignment C [23] (SAC), and a parallelizing C compiler able to target Microgrids. The status and achievements of these components has been already published [24, 25, 26, 27].

The software strategy largely benefited from the advent of multi-core architectures in the last decade, since this recent development has created a culture around software ecosystems that was favorable to explicit concurrency constructs towards new forms of parallelism on chip. One of the main remaining obstacles was to determine which of the existing services of operating software could be substituted by hardware processes, and how to adapt

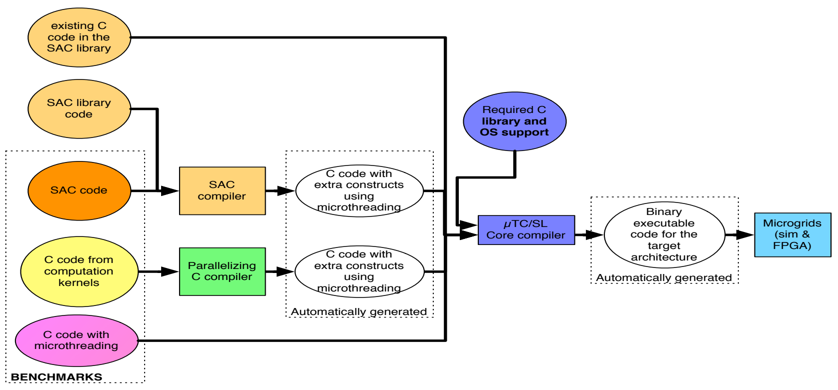

existing C code in the SAC library
SAC library code
SAC code
C code from computation kernels
C code with microthreading
BENCHMARKS
SAC compiler
Parallelizing C compiler
C code with extra constructs using microthreading
C code with extra constructs using microthreading
Automatically generated
Required C library and OS support
μTC/SL Core compiler
Binary executable code for the target architecture
Automatically generated
Micrograds (sim & FPGA)

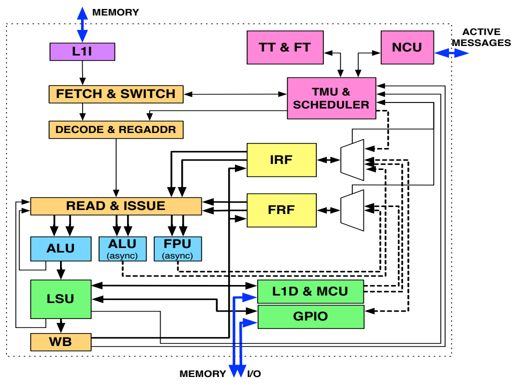
Figure 1: The Apple-CORE software deliverables.

the remaining services so that applications using them could still run on Microgrids. To analyse this aspect, Apple-CORE did not choose to port an entire existing OS to its proposed architecture, and instead opted for a hybrid approach: application-level OS services were ported to run on Microgrid cores, whereas services based on unportable legacy system code (e.g. precompiled device drivers) could be accessed *remotely* through the on-chip network (NoC) via lightweight remote procedure calls. The corresponding system architecture has been published previously [28]; it is inspired from the heterogeneous OS integration in the Cray XMT.

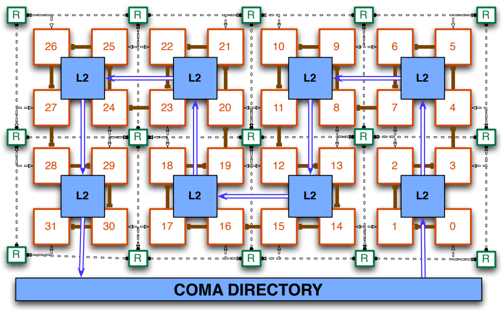
The software produced in the Apple-CORE project is summarized in Figure 1. Next to the system-level compilation and operating software dedicated to Microgrids, higher-level compilation tools that hide the platform details to application code were also developed. Finally, to demonstrate the soundness and effectiveness of the approach, existing application benchmarks were selected and new benchmarks were created across a variety of application domains. These were then subsequently evaluated on both cycle-accurate simulators and FPGA prototypes.

4. Architecture components

The Apple-CORE architecture proposes to combine a) RISC cores optimized for latency tolerance with dynamically scheduled HMT, b) hardware units next to cores to organize software concurrency within and across clusters of neighboring cores, called Microgrids, and c) a common NoC protocol to assign workloads to different regions of a Microgrid, different Microgrids



(a) Core micro-architecture.



(b) Microgrid of 32 cores.

Figure 2: Micro-architectural components.

on chip, or to other core types in an heterogeneous design. The design can be combined with various memory systems, although Apple-CORE also proposes a custom distributed cache network with a diffusion protocol that ensures evicted lines are kept close to their point of last use on chip.

4.1. Core micro-architecture

The core design, illustrated in Figure 2a, is derived from a 6-stage, single-issue, in-order RISC pipeline:

- the register file is extended as *synchronizing storage*, where each register has a *dataflow state* which indicate whether it contains data (*full*/*empty*) or is *waiting* for an asynchronous operation to complete;
- upon issuing an instruction that requires more than one cycle to complete, or whose input operands are not *full*, the *waiting* state is written back to the output operand and the value is overwritten with the identity of the issuing thread. This way, the thread can be put back on the schedule queue when its dependency becomes available. Meanwhile, further instructions in the pipeline can continue;
- the L1-D cache is modified so that loads are issued to memory asynchronously, constructing in the line's storage a list of registers to notify when the load completes;
- the fetch stage is connected to an *active thread queue*. It reads instructions and *switch hints* from the program counter at the head of the queue. Switch hints force a switch at every instruction that *may* suspend the current thread, and are ignored if only one thread is active;

Unit	Description
Scheduler	Wakes up threads upon writes to waiting registers or L1-I load completions
Thread Control Unit (TCU)	Performs bulk thread creation and logical index distribution
Register Allocation Unit (RAU)	Allocates and deallocates register ranges dynamically
Network Control Unit (NCU)	Receives and sends active messages and responses on the NoC

Table 1: Logical sub-units in the TMU.

Event category	Parameters
context allocation	minimum/maximum number of cores
bulk creation	allocated context identifier, common PC, common register window layout, overlap factor, logical thread index range
request for bulk synchronization	context identifier, network address of remote register to write to upon termination
remote register access	context identifier, relative address of register

Table 2: Control events handled by the TMU.

State	Update events
Program counters	Bulk creation, branches
Mappings from logical register windows to the register file	Bulk creation
Logical index ranges	Bulk creation
Bulk synchronizers	Bulk creation, bulk synchronization

Table 3: Private state maintained by the TMU.
This state is maintained in dedicated hardware structures close to the TMU.

- each thread is associated with a *configurable logical window in the register file*, including a *configurable number of registers per thread*; the decode stage computes the absolute register address for the read stage.

As the register files only require five ports, more registers can be provisioned, and thus more *hardware thread contexts*, for the same area budget as a smaller number of registers in a wide-issue core. The reference configuration uses 256 thread contexts and 1024 registers, for a minimum of 32 threads with a full logical register window and a maximum of 256 threads using 4 registers each.

Event	Cost on software critical path	Extra latency in asynchronous hardware process
context allocation	1 pipeline cycle	NoC round-trip to Microgrid cluster + 2 network cycles per core effectively targeted
context configuration	1-4 pipeline cycle	NoC single trip + 1-4 network cycle at first core in target cluster
bulk creation	1 pipeline cycle	NoC single trip + 2 network cycles per core in cluster (created threads are immediately schedulable in the pipeline of each core)
request for bulk synchronization	1 pipeline cycle	NoC round-trip + time to termination of target thread(s)
remote register access	1 pipeline cycle	NoC single or round-trip + 1 network cycle

Table 4: Concurrency management operation latencies.

4.2. Core clusters and hardware concurrency management

Each core is equipped with a Thread Management Unit (TMU, table 1). The TMU is responsible for the local scheduling of threads, and the TMUs of adjacent cores coordinate to offer automated multi-core concurrency management. TMUs accept *control events* either locally from ISA extensions, or from the NoC. The main events are listed in table 2:

- *context allocation*, which reserves execution resources (PC, registers, bulk synchronizers) across one or multiple cores with a single request;
- *bulk creation*, which starts the *autonomous, asynchronous creation of multiple logical threads* over a previously allocated context;
- *bulk synchronization*, which instructs the TMU to notify the thread issuing the bulk synchronization upon completion of all threads bound to a previously allocated context;
- *remote register access*, for non-blocking point-to-point communication and broadcasts. Remote writes may wake up thread(s) waiting on the written register(s).

Core clusters for context allocation are identified by a simple, generic addressing scheme: each cluster address is a value $2P + S$, where $P = cS$ is the address of the first core in the cluster and $S = 2^M$ is the cluster size. Requests for context allocation, bulk creation and synchronization and register broadcasts are sent to the first core in the cluster using the NoC, then negotiated asynchronously across TMUs from the first core based on the size field. Inter-TMU coordination occurs using a linear, point-to-point *distribution network* (DN). The DN follows a space filling curve to maximize locality at any cluster start position and size (Figure 2b). Although Apple-CORE uses a dedicated separate physical network, the DN can be implemented as a virtual network over a single common NoC using QoS to guarantee latency independence of concurrency management between regions of the chip.

4.3. Memory architecture

The proposed design is distinct from most other CMP architectures in that work distribution and synchronization is coordinated by mechanisms distinct from memory. This allows the chip integrator to use Microgrids with various memory systems. However, the design is optimized to tolerate memory latencies using multiple, asynchronous in-flight operations, and is thus best used with memory systems that support split-phase transactions and/or request pipelining. To demonstrate this, Apple-CORE has also developed an on-chip memory network implementing a custom *distributed cache protocol* (also illustrated in Figure 2b): memory stores are effected at the local L2 cache without invalidating other copies, and are only propagated and *merged* with other copies upon explicit barriers or bulk creation or synchronization of threads. This protocol is derived from [29, 30]: from the perspective of programs, it appears as a single shared memory with consistency resolved at concurrency management events.

4.4. Programming methodology

The TMU control events are exposed via ISA extensions and can be used from any thread, ensuring that concurrency control can be truly distributed across application components co-located on the chip. The intent is to enable capturing these concurrency semantics in various programming models, e.g. the bulk-synchronous parallelism (BSP [31]) and task parallelism constructs of OpenMP [7] and OpenCL [18], although the Apple-CORE project did not explicitly explore these opportunities yet.

As we discussed in section 3, Apple-CORE provides a single set of extension primitives to the C language, called *SVP*, intended for use by higher-level code generators or language libraries. It has been concretely implemented in the μTC and SL extensions to C. SVP features:

- defining *thread programs*, analogous to OpenCL's "kernels" but allowed to invoke any valid, separately compiled C function for truly general-purpose computations;
- declaring and using *dataflow channels*, which are translated to physical register sharing to implement the producer-consumer pattern (reads from *empty* block, writes to *waiting* by another thread wakes up);

- performing bulk creation and synchronization of *families* of logical threads, each identified by a *logical index* in a configurable range. This can be used to implement both the BSP pattern and nested fork-join parallelism found in Cilk [32] and functional languages.

For reductions, within one core multiple threads can share a single register and all reducing instructions using that register as both input and output operand will serialize automatically using the dataflow scheduler. Across cores, parallel prefix sums [33] or standard distributed reductions can be used for scalable throughput.

Furthermore, by encouraging an overall program structure with forward-only chains of dataflow channels, SVP favors program styles that are serializable and can be run deterministically using any cluster size, down to only one thread on one core. For system and library code, non-deterministic constructs are also possible. In particular, priviledged code can construct any point-to-point communication pattern using remote register accesses. For mutual exclusion and atomic state updates, programs can either send state updates as a remote thread creation to a single, previously-allocated execution context where all bulk creations are automatically serialized by the TMU (Dijkstra's "secretary" pattern [34, p. 135]), or they can send state updates to a previously agreed core cluster sharing common coherent caches (e.g. a single L2 cache in the proposed memory architecture) and then negotiate atomicity locally using standard memory transactions.

To summarize, SVP was designed as a set of acceleration primitives for operating systems and general-purpose concurrency management frameworks. Its "killer feature" is perhaps that the time overhead of thread creation and point-to-point communication is driven down to a few pipeline cycles (table 4), cheaper than most C procedure calls. Moreover, this overhead can be made nearly invisible to computations as it can overlap in the TMU with instructions from other threads in the pipeline.

5. Realization and evaluation

For evaluation, Apple-CORE has defined two platforms.

To confirm the realizability of D-RISC in hardware and estimate its actual implementation costs, a single-core but multithreaded FPGA prototype was developed, called UTLEON3 [35, 36, 37]. This provided SVP extensions over the 32-bit SPARC ISA, 1-4K instruction (I) and data (D) caches, 512 registers, 16 bulk synchronizers and 128 thread contexts.

Furthermore, to exercise the software tool chain and provide insight about the scalability of the design, a simulation-based platform was also defined providing a 128-core Microgrid cluster. Each core runs a 64-bit Alpha-derived ISA, has a 2KiB L1-I cache, 4KiB L1-D cache, 1024 registers, 32 bulk synchronizers and 256 thread contexts.

In the multi-core configuration, asynchronous FPUs are shared between adjacent cores. The corresponding distributed cache network has 32 L2 caches of 128KiB each, connected in 4 rings themselves connected to a single top-level ring with 4 DDR3-1600 memory channels.

In both platforms, the Microgrid hardware shares the NoC with a small *companion core* able to run OS services that cannot be implemented directly on the Microgrid cores, as discussed in [28]. Virtual memory is implemented using a MMU shared by all cores, so that all threads appear to run in the same logical address space. Some Microgrid cores have a direct asynchronous interface to off-chip I/O which supports multiple in-flight split-phase transactions [38], to maximize bandwidth in streaming applications.

The multi-core chip was implemented as a model in the MGSim [39, 40] framework. MGSim provides a discrete event, cycle-accurate, full-system simulation kernel where all component behaviors down to individual pipeline stages, register file ports, arbitrators, functional units, FIFOs, DDR controller, etc. have their own detailed model. This enables a slightly higher level, and thus faster simulation than a circuit-level simulation, while preserving the timing accuracy of thread scheduling and TMU operations relative to the pipeline cycle time. This simulation-based platform receives the focus of the remainder of this section.

5.1. Area and timing estimates

The area and access time requirements of the reference configuration have been evaluated using CACTI [41]. Using conservative technology parameters at 45nm CMOS, the Microgrid occupies an estimated 120mm^2, not counting the physical links and routers on the NoC and memory system (Figure 3). This can be compared e.g. to the Intel P8600 chip (Core 2 Duo) which provisions 3MB L2 cache and 2 cores on 107mm^2 using the same gate size. The register files have the longest access time at .4ns, and with two subsequent accesses at the read and writeback stages this constrains the maximum core frequency at 1.25GHz. Experiments subsequently used 1GHz clocks for pipelines.

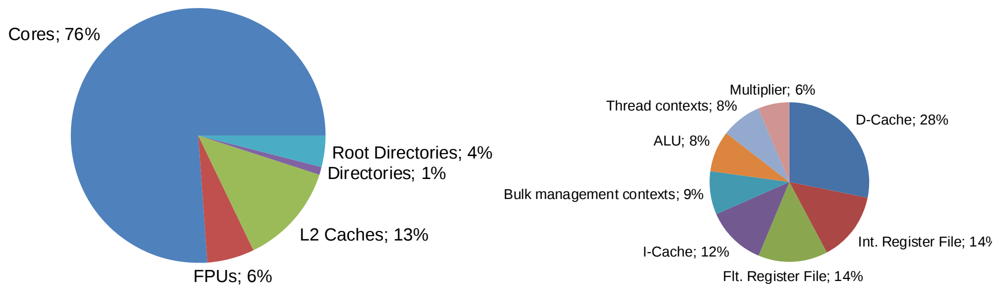


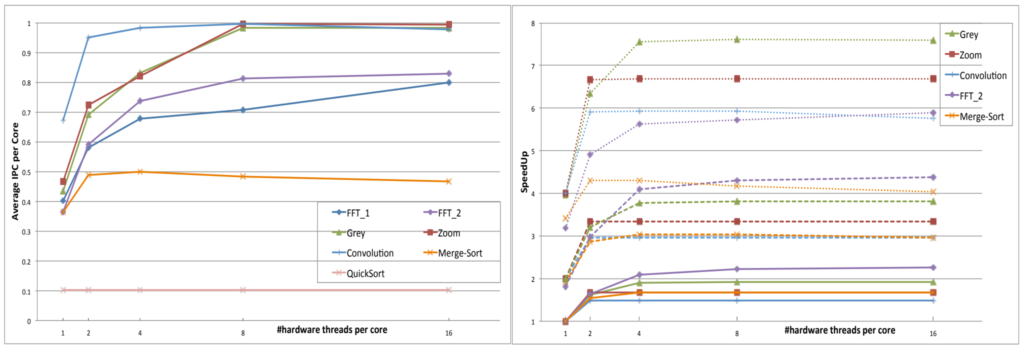
Figure 3: Chip area breakdown: entire grid (left), per core (right).

5.2. Software environment

Apple-CORE has produced a C language implementation based on the GNU C compiler (GCC), and a run-time environment based on FreeBSD [42]. GCC's Alpha back-end was extended to support the SVP primitives. The run-time environment provides access to the entire C library from any core on a Microgrid cluster, performing memory management locally on each core and *delegating* services (e.g. I/O) that operate on system structures to a reserved sub-cluster or the integrated companion core. Instead of using memory-based protection, isolation between processes is achieved using capabilities [43], similarly to [44]. As we outlined in section 3, this infrastructure has fostered the separate development of a parallelizing C compiler targeting SVP [26, 27] and a SVP back-end to the array-oriented, functional productivity language SAC [23, 25].

5.3. Performance and scalability

The performance benefits are two dimensional: the increase efficiency per core due to the interleaving of multiple threads, and the increased throughput enabled by multi-core execution. We illustrate this in Figure 4. As the left side diagram shows, the pipeline occupation increases as the number of threads per core increases for the selected numeric benchmarks, up to nearly full utilization in numeric code from 8 threads per core. This result shows also that a naive quicksort, in particular, performs poorly due to its mostly sequential pivot selection and the absence of branch prediction in hardware; merge sort instead exhibits more concurrency and benefits from multiple threads. The right side diagram shows the scalability of the same group of benchmarks as the number of cores used increases (plotted with different line styles).



(a) Pipeline utilization per core (b) Scalability

Figure 4: Orthogonal benefits of multithreaded and multi-core execution.

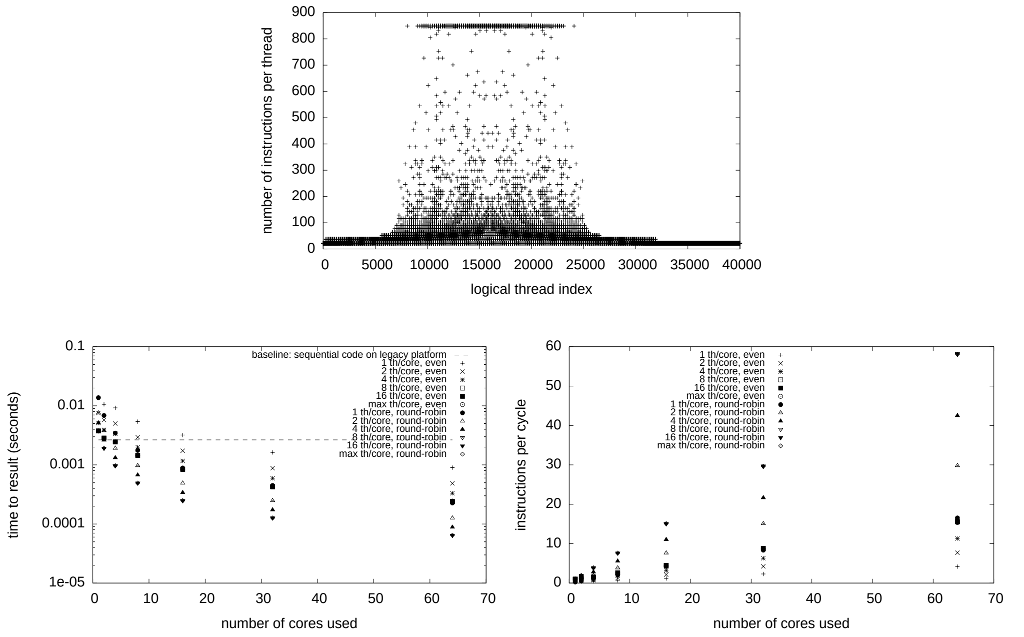
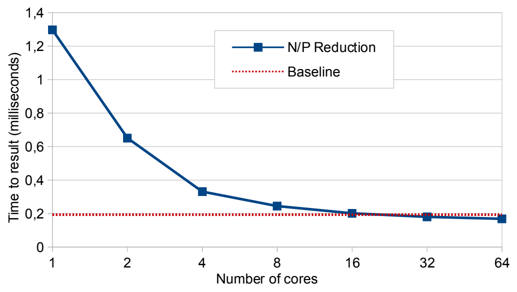
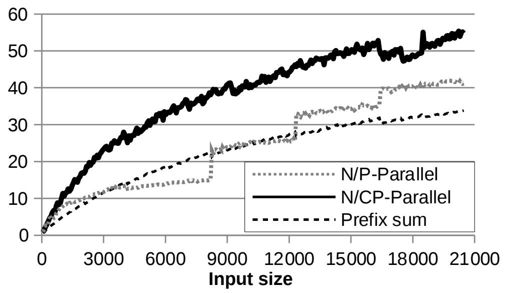


Figure 5: Example heterogeneous workload: Mandelbrot set approximation.

Figure 5 illustrates the computational density and latency tolerance for FPU operations. This heterogeneous compute-bound workload of 40k imbalanced threads (top in figure) has been run over Microgrid sub-clusters of different sizes, using two distribution strategies. Sequential code on the Intel P8600 at 2.4GHz was used as baseline. Using an even distribution and



(a) Scalability of a N/P reduction of 64k floating-point values.



(b) Speedup of various parallel reduction strategies on 64 cores.

Figure 6: Performance of parallel reductions.

one thread/core, the performance exceeds the baseline at 32 cores. Using a round-robin distribution to spread the load over the cluster, at 1 thread/core the baseline is exceeded at 8 cores. At 16 threads per core and a round-robin distribution, the baseline is exceeded at 2 cores. As per section 5.1 above, this implies the baseline is exceeded at 20× smaller area budget and less than half the frequency. The round-robin distribution with 16 threads per core further scales nearly linearly with full pipeline utilization up to 64 cores (right in figure).

Figures 6a and 6b illustrate the scalability for parallel reductions. In Figure 6a, sub-vectors are first summed locally on each core, then the partial sums are summed on one core. The baseline performance (same chip as above) is matched from 32 Microgrid cores onwards. In Figure 6b, multiple reduction strategies are used on a single sub-cluster of 64 cores, and compared (speedup) against the performance of the sequential version on 1 core. The parallel prefix sum [33] scales regularly with the input size but is disadvantaged as it executes more instructions. The best strategy (N/CP) is to run multiple local reductions on each core in different threads and then combine the partial sums on one core.

Figure 7 illustrates the scalability for a scientific kernel using different programming interfaces. Using hand-coded C or assembly code, the baseline is matched from 4 cores. With code automatically parallelized from C and a software-based dynamic loop scheduler, the baseline is matched from 8 cores. The higher-level SAC code has a large run-time overhead, but still benefits from multi-core scalability. As can be seen on the right side, from 32 cores

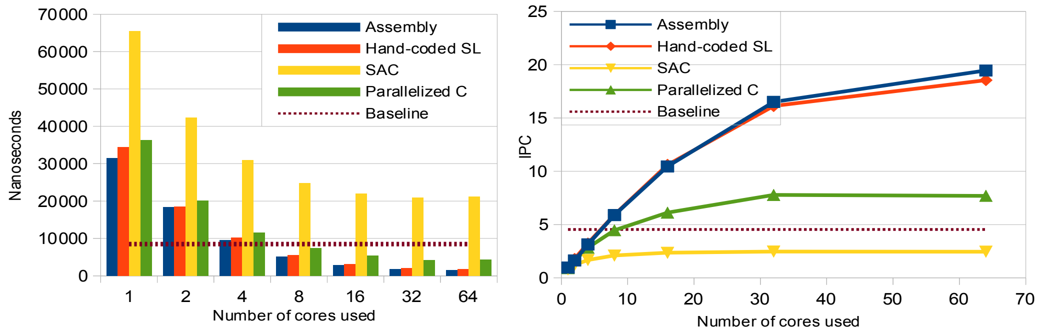


Figure 7: Performance of the equation of state fragment.

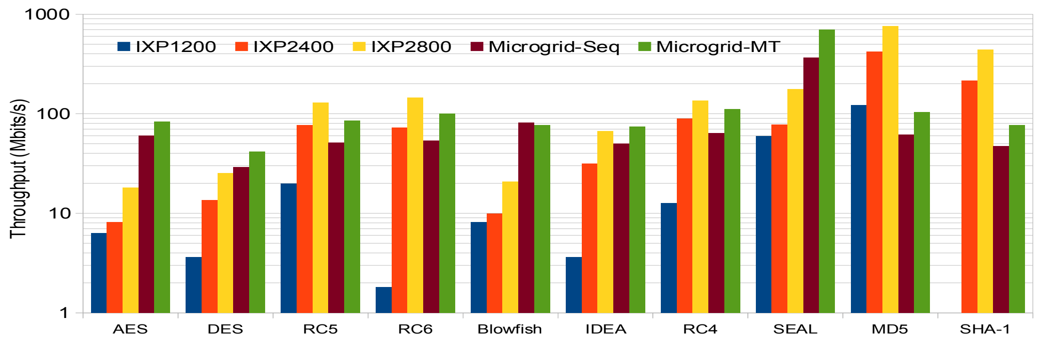
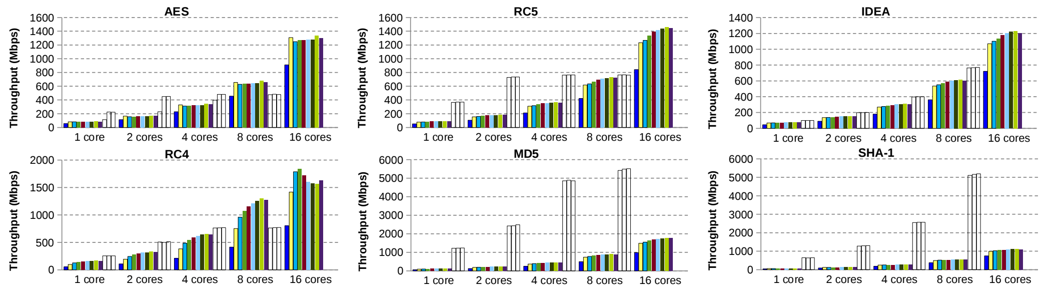


Figure 8: Throughput for one stream on one core.

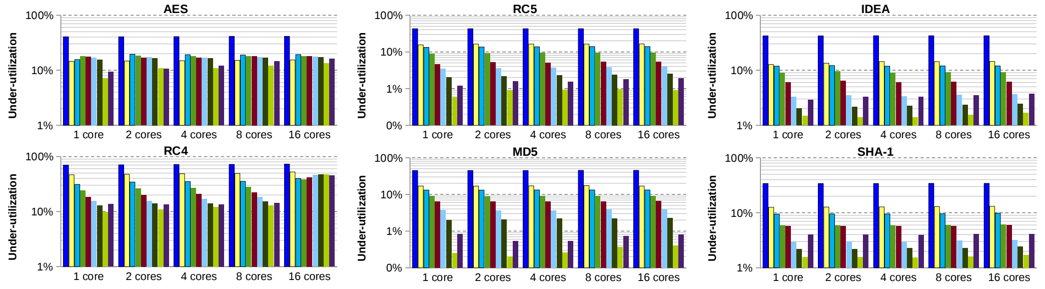
the memory throughput approaches the external bandwidth of the chip and the workload becomes memory-bound, preventing extra speedup.

5.4. Example throughput application: cryptography

In [45, 46], the authors introduce NPCryptBench, a benchmark suite to evaluate network processors. We have run unoptimized code for these ciphers and hash algorithms on the Apple-CORE chip. First the throughput of the unoptimized code for one flow on one core is compared against the unoptimized throughput on the Intel IXP chips ([45, fig. 4], [46, fig. 3]). Two Microgrid codes are used, one purely sequential and one where the inner loop is parallelized. As the results in Figure 8 show, the Microgrid hardware provides a throughput advantage for the more complex AES, SEAL and Blowfish ciphers, whereas the dedicated hardware hash units of the IXP accelerate MD5 and SHA-1. For the other kernels, the Microgrid hardware is slower: with RC5, RC6 and IDEA, a carried dependency serializes execution



(a) Combined throughput for 1-8,16 streams per core on 1-16 cores (1-256 streams total). IXP2800 performance in leftmost 3 bars at each core group.



(b) Pipeline under-utilization.

Figure 9: Performance of cryptographic kernels.

and minimizes latency tolerance. With RC4, the modified state at each cipher block must be made consistent in memory before the next thread can proceed, which also partly sequentializes execution. Further throughput deviation from the IXP should be considered in the light of the frequency difference (1.4GHz for the IXP vs. 1GHz for the Microgrid) and the fact the Microgrid hardware was not designed specifically towards cryptography.

Figure 9a shows the scalability of the most popular cryptographic kernels, using the purely sequential, unoptimized code for each stream on the Microgrid and the Level-2 optimized code for the IXP2800 ([45, fig. 6], [46, fig. 8]). For each sub-cluster size, increasing the number of flows per core increases utilization (Figure 9b) and thus overall throughput. Throughput is furthermore reliably scalable up to 16 cores. With RC4 and 64 flows on 16 cores the workload reaches the memory bandwidth of the chip; with additional flows, contention on the internal memory network appears, the utilization is reduced slightly and so is the throughput. The througput then stabilizes at 96 flows around 1.6Gbps.

6. Discussion and future work

6.1. Related work

As we outlined in sections 3 and 4, the proposed design diverges from traditional general-purpose SMPs in that preemption and branch predicition are replaced by hardware thread management and groups of threads created and synchronized in bulk. Meanwhile, it does not fit the current landscape of "compute accelerators" either: each D-RISC core in Microgrids is fully general-purpose and application code needs not be separated into its principal skeleton and its "leaf" computation kernels.

The two platforms that were contemporary to Apple-CORE and received most attention at the time were perhaps Intel's Single-Chip Cloud Computer [47] (SCC), a research platform, and Tilera's TILE64 [48], a product offering for network applications.

Both integrate more general-purpose cores on one chip than contemporary multi-core product offerings: 48 for the SCC, 64 for TILE64. All cores are connected to a common network-on-chip. On both chips, each core consists of a traditional processor architecture—the MIPS pipeline for TILE64, the P54C pipeline for the SCC—and a configurable mapping from cores to external memory able to provide the illusion of arbitrarily large private memory to each core.

For I/O, connectivity is provided on the TILE64 by direct access to external I/O devices on each core via dedicated I/O links on the NoC; on the SCC, the approach is more similar to Apple-CORE: the NoC is connected to an external service processor implemented on FPGA which forwards I/O requests from the SCC cores to a host system.

For parallel execution, TILE64 and SCC only support one hardware thread per core. Both offer preemption to time and space schedulers in software as the means to multiplex multiple software activities, and are thus less equipped to deal with fine-grained heterogeneous concurrency than the Apple-CORE proposal.

For communication, TILE64 has the most comprehensive offering: it supports 4 hardware-supported asynchronous channels to any other cores, a single channel to a configurable, static set of peers and two dedicated channels to external I/O devices per core. Alternatively, cores can also implement virtual channels over a coherent, virtually shared memory implemented over another set of NoC links, although the communication latency is then higher.

This diversity of communication means offer more optimization opportunities for concurrent software, but at the expense of a largely more complex machine model for the implementers of operating software. In contrast the SCC provides a single, much simpler memory-mapped interface for arbitrary point-to-point communication between cores, but the lack of dataflow synchronization in each core means that multi-core concurrency management rely on expensive network round-trips for every coordination step. The Apple-CORE proposal can be described as a middle point: it offers two virtual networks on chip, one for global distribution and one for local distribution, and a single protocol for simplicity, but it also equips cores with extra hardware for low-latency synchronization and multi-core work distribution.

6.2. Achievements and follow-up research

The infrastructure developed by Apple-CORE, namely platforms and associated software tool chains, have been released publicly towards third-party scrutiny and reuse[2]. The consortium partnerships have been extended past the end of the project, with the intent to continue collaboration in the promotion and development of the platform. Current follow-up research is focused on developing abstract resource models to support the dynamic mapping of varying, heterogeneous application workloads onto the available resources on chip. Once further results confirm confidence in the field that the proposed platform is a sound approach to multi-core system management, the consortium intends to mobilize towards more hardware realizations.

6.3. Evaluation challenges and continued work

The results of Apple-CORE have demonstrated the potential of general-purpose, fine-grained concurrency in various simple scenarios. Unfortunately, the project has also struggled to carry out more extensive evaluations. Indeed, most current benchmark suites towards architecture design focus on the performance of single programs (e.g. SPEC) and thus fall out of the scope of the proposed design. Larger benchmark suites that test multi-application scenarios (e.g. CloudSuite [49]) in turn assume the existence of physical hardware able to run datacenter-grade workloads, and are thus inadequate for architecture research where experiments are performed in detailed simulations running at a few hundred MIPS.

[2]`http://www.apple-core.info/` and `http://svp-home.org/`

To sustain further activity in this field, in particular the analysis of chip behavior under multiple scales of concurrency, both lighter multi-application benchmarks and faster simulations must be developed. The Apple-CORE consortium is committed to continue further research in this direction. Two strategies are currently planned. Towards increasing the evaluation space, the platform will be extended to provide more compatibility with existing application code, so as to increase the number of benchmark applications that can be used for evaluation. Simultaneously, to increase the evaluation scale in application complexity, the SVP interface will be emulated in software on existing multi-core platforms, such as Intel's SCC or Tilera's TILE64. This will enable the evaluation of the Apple-CORE software infrastructure for existing production-grade applications.

7. Conclusion

This article has sought to motivate a renewed effort in microprocessor architecture design towards many-core chips with smaller, more efficient cores using hardware multi-threading and hardware acceleration for concurrency management. It further described the architecture developed in the Apple-CORE project as a step in this direction, and illustrated its performance using several benchmarks.

As the results show, traditionally sequential workloads can be parallelized in presence of fine-grained multithreading and hardware-supported concurrency management on chip. The proposed hardware achieves higher throughput per unit of area and per clock cycle than contemporary state-of-the-art components. Its hardware management protocol in turn offers reliable multi-core throughput scalability up to the memory bandwidth.

Follow-up work will focus on broadening the scope of the evaluation of the platform, towards larger-scale and more heterogeneous workloads. This work will in turn favor increasing compatibility between the proposed platform and existing software environments. The international consortium that composed Apple-CORE, key to the success of this project, intends to extend this partnership beyond its current achievements and towards the hardware realization of the proposed innovation.

Acknowledgements

This research was supported by the European Union under grant number FP7-215216 (Apple-CORE).

References

[1] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, J. Shen, Coming challenges in microarchitecture and architecture, Proceedings of the IEEE 89 (2001) 325–340.

[2] L. Spracklen, S. G. Abraham, Chip multithreading: opportunities and challenges, in: Proc 11th International Symposium on High-Performance Computer Architecture, HPCA'05, IEEE, 2005, pp. 248–252.

[3] H. Sutter, The free lunch is over: A fundamental turn toward concurrency in software, Dr. Dobb's Journal 30 (2005).

[4] R. Poss, M. Lankamp, Q. Yang, J. Fu, M. W. van Tol, C. Jesshope, Apple-CORE: Microgrids of SVP cores (invited paper), in: S. Niar (Ed.), Proc. 15th Euromicro Conference on Digital System Design (DSD 2012), IEEE Computer Society, 2012.

[5] W. A. Wulf, S. A. McKee, Hitting the memory wall: implications of the obvious, SIGARCH Comput. Archit. News 23 (1995) 20–24.

[6] I. Bell, N. Hasasneh, C. Jesshope, Supporting microthread scheduling and synchronisation in CMPs, International Journal of Parallel Programming 34 (2006) 343–381.

[7] OpenMP Architecture Review Board, OpenMP application program interface, version 3.0, 2008.

[8] B. Smith, Architecture and applications of the HEP multiprocessor computer system, Proc. SPIE Int. Soc. Opt. Eng.; (United States) 298 (1981) 241–248.

[9] R. H. Halstead, Jr., T. Fujita, MASA: a multithreaded processor architecture for parallel symbolic computing, SIGARCH Comput. Archit. News 16 (1988) 443–451.

[10] R. S. Nikhil, Arvind, Can dataflow subsume von Neumann computing?, SIGARCH Comput. Archit. News 17 (1989) 262–272.

[11] D. May, R. Shepherd, Occam and the transputer, in: Advances in Petri Nets 1989, volume 424 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 1990, pp. 329–353.

[12] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, J. Wawrzynek, Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine, in: ASPLOS-IV: Proc. 4th international conference on Architectural support for programming languages and operating systems, ACM, New York, NY, USA, 1991, pp. 164–175.

[13] A. Bolychevsky, C. Jesshope, V. Muchnick, Dynamic scheduling in RISC architectures, IEE Proceedings - Computers and Digital Techniques 143 (1996) 309–317.

[14] K. Bousias, N. Hasasneh, C. Jesshope, Instruction level parallelism through microthreading – a scalable approach to chip multiprocessors, The Computer Journal 49 (2006) 211–233.

[15] C. Jesshope, M. Lankamp, L. Zhang, The Implementation of an SVP Many-core Processor and the Evaluation of its Memory Architecture, ACM SIGARCH Computer Architecture News 37 (2009) 38–45.

[16] K. Bousias, L. Guang, C. Jesshope, M. Lankamp, Implementation and evaluation of a microthread architecture, Journal of Systems Architecture 55 (2008) 149–161.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, Cilk: an efficient multithreaded runtime system, SIGPLAN Not. 30 (1995) 207–216.

[18] Khronos OpenCL Working Group, The OpenCL specification, version 1.0.43, 2009.

[19] C. R. Jesshope, μTC - an intermediate language for programming chip multiprocessors, in: C. Jesshope, C. Egan (Eds.), Advances in Computer Systems Architecture, volume 4186 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2006, pp. 147–160.

[20] T. Bernard, C. Jesshope, P. Knijnenburg, Strategies for compiling muTC to novel chip multiprocessors, in: S. Vassiliadis, M. Berekovic, T. Hämäläinen (Eds.), Embedded Computer Systems: Architectures, Modeling, and Simulation, volume 4599 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2007, pp. 127–138.

[21] T. Bernard, C. Grelck, C. Jesshope, On the Compilation of a Language for General Concurrent Target Architectures, Parallel Processing Letters 20 (2010).

[22] R. Poss, SL—a “quick and dirty” but working intermediate language for SVP systems, Technical Report arXiv:1208.4572v1 [cs.PL], University of Amsterdam, 2012.

[23] C. Grelck, S.-B. Scholz, SAC: a functional array language for efficient multi-threaded execution, International Journal of Parallel Programming 34 (2006) 383–427.

[24] S. Herhut, C. Joslin, S.-B. Scholz, C. Grelck, Truly nested data parallelism: Compiling SaC for the Microgrid architecture, in: M. Morazán (Ed.), Proc. 21st International Symposium on Implementation and Application of Functional Languages (IFL’09), Technical Report SHU-TR-CS-2009-09-1, Seton Hall University, Department of Mathematics and Computer Science, South Orange, USA, 2009, pp. 141–153.

[25] C. Grelck, S. Herhut, C. Jesshope, C. Joslin, M. Lankamp, S.-B. Scholz, A. Shafarenko, Compiling the Functional Data-Parallel Language SaC for Microgrids of Self-Adaptive Virtual Processors, in: 14th Workshop on Compilers for Parallel Computing (CPC’09), IBM Research Center, Zurich, Switzerland.

[26] D. Saougkos, D. Evgenidou, G. Manis, Specifying loop transformations for C2μTC source-ro-source compiler, in: Proc. of 14th Workshop on Compilers for Parallel Computing (CPC’09), Zürich, Switzerland, IBM Research Center, 2009.

[27] D. Saougkos, G. Manis, Run-time scheduling with the C2uTC parallelizing compiler, in: 2nd Workshop on Parallel Programming and Run-Time Management Techniques for Many–Core Architectures, in Workshop Proceedings of the 24th Conference on Computing Systems (ARCS 2011), Lecture Notes in Computer Science, Springer, 2011, pp. 151–157.

[28] R. Poss, M. Lankamp, M. I. Uddin, J. Sýkora, L. Kafka, Heterogeneous integration to simplify many-core architecture simulations, in: Proc. 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO ’12, ACM, 2012, pp. 17–24.

[29] L. Zhang, C. R. Jesshope, On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores, in: Bouge, et al. (Eds.), Euro-Par Workshops, volume 4854 of *LNCS*, Springer, 2007, pp. 38–48.

[30] T. D.Vu, L. Zhang, C. R. Jesshope, The verification of the on-chip COMA cache coherence protocol, in: International Conference on Algebraic Methodology and Software Technology, pp. 413–429.

[31] L. G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (1990) 103–111.

[32] C. E. Leiserson, The Cilk++ concurrency platform, in: DAC '09: Proceedings of the 46th Annual Design Automation Conference, ACM, New York, NY, USA, 2009, pp. 522–527.

[33] R. E. Ladner, M. J. Fischer, Parallel prefix computation, J. ACM 27 (1980) 831–838.

[34] E. W. Dijkstra, Hierarchical ordering of sequential processes, Acta Informatica 1 (1971) 115–138.

[35] M. Danek, L. Kafka, L. Kohout, J. Sykora, Instruction set extensions for multi-threading in LEON3, in: Z. K. et al. (Ed.), Proc. 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS'2010), IEEE, 2010, pp. 237–242.

[36] J. Sykora, L. Kafka, M. Danek, L. Kohout, Analysis of execution efficiency in the microthreaded processor UTLEON3, volume 6566 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 110–121.

[37] M. Daněk, L. Kafka, L. Kohout, J. Sýkora, R. Bartosinski, UTLEON3: Exploring Fine-Grain Multi-Threading in FPGAs, Circuits and Systems, Springer, 2012.

[38] M. A. Hicks, M. W. van Tol, C. R. Jesshope, Towards Scalable I/O on a Many-core Architecture, in: International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), IEEE, 2010, pp. 341–348.

[39] M. Lankamp, R. Poss, Q. Yang, J. Fu, I. Uddin, C. R. Jesshope, MGSim—Simulation tools for multi-core processor architectures, Technical Report arXiv:1302.1390v1 [cs.AR], University of Amsterdam, 2013.

[40] R. Poss, M. Lankamp, Q. Yang, J. Fu, I. Uddin, C. Jesshope, MGSim—a simulation environment for multi-core research and education, in: Proc. Intl. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), IEEE, Samos, Greece, 2013. (to appear).

[41] S. Wilton, N. Jouppi, Cacti: an enhanced cache access and cycle time model, Solid-State Circuits, IEEE Journal of 31 (1996) 677–688.

[42] M. K. McKusick, G. V. Neville-Neil, Design And Implementation Of The FreeBSD Operating System, Addison Wesley, 2004.

[43] M. W. van Tol, C. R. Jesshope, An operating system strategy for general-purpose parallel computing on many-core architectures, Advances in Parallel Computing High Performance Computing: From Grids and Clouds to Exascale (2011) 157–181.

[44] J. S. Chase, H. M. Levy, M. J. Feeley, E. D. Lazowska, Sharing and protection in a single-address-space operating system, ACM Trans. Comput. Syst. 12 (1994) 271–307.

[45] Z. Tan, C. Lin, H. Yin, B. Li, Optimization and benchmark of cryptographic algorithms on network processors, IEEE Micro 24 (2004) 55–69.

[46] Y. Yue, C. Lin, Z. Tan, NPCryptBench: a cryptographic benchmark suite for network processors, SIGARCH Comput. Archit. News 34 (2005) 49–56.

[47] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, The 48-core scc processor: the programmer's view, in: Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–11.

[48] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, J. Zook, TILE64 processor: A 64-core SoC with mesh interconnect, in: IEEE International Solid-State Circuits Conference, 2008 (ISSCC 2008). Digest of Technical Papers., IEEE, 2008, pp. 88–598.

[49] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, B. Falsafi, Clearing the clouds: a study of emerging scale-out workloads on modern hardware, in: Proc. 17th international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '12, ACM, 2012, pp. 37–48.