Apple-CORE: Microgrids of SVP cores

Flexible, general-purpose, fine-grained hardware concurrency management

(Invited Paper)

Raphael Poss, Mike Lankamp, Qiang Yang, Jian Fu, Michiel W. van Tol, and Chris Jesshope
Institute for Informatics, University of Amsterdam
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. The corresponding hardware implementation 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 intended to be used as components in *distributed systems on chip* that consider both clusters of small cores and optional larger cores optimized towards sequential performance as *system services* shared between applications. The key aspects of the design are asynchrony, *i.e.* the ability to tolerate operations with irregular long latencies, a scale-invariant programming model, a distributed vision of the chip's structure, and the transparent performance scaling of a single program binary code across multiple cluster sizes. This paper describes the execution model, the core micro-architecture, its realization in a many-core, general-purpose processor chip and its software environment. The reference chip parameters include 128 cores, a 4 MB on-chip distributed cache network and four DDR3-1600 memory channels. This paper 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.**

I. 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 would like to 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 the age of the Internet and ever-increasing support of computers for human activities. For example, while a user's focus may be geared towards the decoding of a film, another activity in the system may be dedicated to downloading the next stream, while yet another may be monitoring the user's blood nutrient levels to predict when to order food online, while yet another may be responsible for backing up the day's collection of photographs on an online social platform, etc.

Even programs that are fundamentally sequential now expose high-level parallelism at scales that were unexpected. For example, the compilation of program source code to machine code is inherently sequential as each pass must scan the program linearly and is dependent on the previous pass' output. However, meanwhile, entire applications have become increasingly larger in terms of their number of program source files, so even though one individual compilation cannot be accelerated via parallelism it becomes possible to massively parallelize an entire application build.

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 Apple-CORE project attempted to answer.

II. Context and design strategy

The first aspect considered is how much logic to invest *per core*, *vs. larger number of cores*. Once a large amount of concurrency is available in software, one can scale back on the number of transistors per core and frequency and multiply the number of cores to increase the throughput/watt ratio. Yet we acknowledge that some inherently sequential workloads will still matter in the foreseeable future, both from legacy software and few applications where no parallel or distributed algorithms are yet known. To support these while still taking advantage of available software concurrency, two options exist. The conservative approach is to favor homogeneity and optimize all cores towards increased sequential performance. This is the approach taken *e.g.* with the Niagara T4 [4]. This simplifies the machine model exposed to programmers, but comes

at the cost of less efficiency for more concurrent workloads. The other approach is to introduce *static heterogeneity* and allocate some areas of the chip towards throughput and others towards single-thread performance. This is the approach taken *e.g.* with the AMD Fusion architecture [5], where "accelerator" cores are placed next to general-purpose cores on the same die. However this latter approach has a possible pitfall: the appearance of *model assymmetry* as an historical artefact.

Indeed, the shift towards more on-chip parallelism has emerged from a background culture where a chip was a single processor. The availability of on-chip parallelism may thus appear as an *extension* of a well-known single processor. However, if a CMP is considered as a mostly-sequential processor with optional "parallel accelerators," this will encourage software ecosystems to keep their focus on the overall sequential scheduling of workloads. An *opportunity loss* ensues: the design of truly *distributed applications on chip*, which would consider both throughput-oriented and sequential-oriented cores as shared *services* in the system, is thereby discouraged. The Apple-CORE project avoids this pitfall by placing the focus on the *protocols* that coordinate workloads between cores on chip. It captures the performance assymmetry as mere component properties in the conceptual model offered through its SVP programming interface.

A. Feature specialization vs. fungible cores

Function specialization by dedicated logic is well known to increase overall application throughput at constant cost or reduce cost at constant throughput, *for given application scenarios*. In general-purpose processors without specific scenarios, only prevalent features benefit from specialization, for example floating-point arithmetic and cryptographic kernels. Yet the question remains of *how much* logic to invest into these specialized units as opposed to *e.g.* more general-purpose cores, larger on-chip memories or a faster interconnect.

Here we are able to recognize a limit on specialization. In general-purpose applications, workloads enter and leave the system at unpredictable times. The sharing of the chip's components between workloads thus requires *on-line, dynamic chip resource management*. To satisfy the need for allocation times within the scale of operation latencies, resource management must be supported in hardware [6]. Since the amount of state that can be maintained locally on chip is limited, the component models used by resource managers must be kept simple, which in turns implies that the *diversity of component properties* is kept low. An example of this can be found in the the replacement of multiple bus hierarchies by a common packet-switched network-on-chip (NoC, [7]). Moreover, as application requirements increase in complexity [8], the pressure to reduce component diversity to keep on-line resource managers fast increases further.

There are two ways forward from there. One is the full integration of *reconfigurable logic*, *e.g.* FPGA, in general-purpose chips, so that functions can be specialized *on demand*. Unfortunately, state of the art research has not yet come up with *update protocols* that can transparently substitute one specialized feature by another via reconfiguration *at a fast rate*. The other direction, followed by Apple-CORE, suggests *adaptive general-purpose cores* on a NoC, to simplify on-line resource management by making pools of resources *fungible*[1]. For example, a group of many in-order RISC cores on a mesh interconnect with configurable frequency and voltage may be an advantageous replacement for a fixed-frequency specialized SIMD unit with dedicated data paths, because it is reusable for other purposes without overhead.

B. On-chip latencies and hardware multithreading

The increasing disparity between the chip size and the gate size causes the latency between on-chip components (cores, caches and scratchpads) to increase relative to the pipeline cycle time: the wire delay increases relative to the transit time across a gate; the latter in turn constrains frequency and puts a lower bound on the pipeline cycle time. This divergence is the on-chip equivalent of the "memory wall" [9]. Moreover, these latencies will become increasingly unpredictable, both due to overall usage unpredictability in general-purpose workloads and due to soft errors in circuits.

These latencies cannot be easily tolerated using superscalar issue or VLIW, for the reasons outlined in [10]: superscalar execution mandates non-scalable complexity in coordination structures like register files, and VLIW requires energy-inefficient speculation to maximize throughput under unpredictable latencies. The known solution is hardware multithreading (HMT), *i.e.* the interleaving of fine-grained threads in the cores' pipelines via a hardware scheduler.

Yet there are two issues with previous approaches to HMT. With barrel processors ([11], [12]), throughput does not adapt to the number of ready threads. With dynamically scheduled threads over long pipelines (*e.g.* in the MTA [13], [14] and SMT processors [15], [16]), mispredicted branches negate the benefit of interleaving. This suggests that the benefits of HMT will be most apparent with *dynamic scheduling over shorter pipelines*.

C. Pressure for hardware-assisted concurrency management

Assuming CMPs with increasing number of cores and per-core HMT, *space scheduling* must be implemented to spread concurrent software tasks to the chip's parallel execution resources. Space scheduling can be done either in software or in hardware. With a software scheduler, each hardware thread is controlled by a space scheduler that assigns tasks using state taken from main memory. This can occur even when thread interleaving is performed at a fine grain in hardware. It is also relevant even when there is no need for time sharing of multiple tasks onto a single hardware thread, for example when the number of tasks is smaller or equal to the number of hardware threads, or when the software environment simply does not require preemptive task multiplexing. However, the choice of a software scheduler assumes that the workload per

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

task is always sufficient to compensate the non-local latencies incurred by memory accesses to task state during schedule decision making and task assignment.

This assumption traditionally holds for coarse-grained concurrency, for example external I/O. It can also hold for regular, wide-breadth concurrency patterns extracted from homogeneous sequential tight loops, via blocking aggregation (*e.g.* OpenMP). However the situation is not so clear with fine-grain heterogeneous task concurrency sourced from 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 introduce *hardware support for concurrency management* is not new [12], [17], [18], [19], [20]; however previous research met with resistance against the introduction of explicit concurrency in applications. Now that on-chip software concurrency is the norm, hardware support deserves renewed attention for two reasons.

One is the potential gain in resource fungibility obtained by the replacement of specialized SPMD/SIMD units by arrays of general-purpose cores, outlined above. For this to be tractable, the overhead to dispatch an SPMD task as a group of tasks over all participating pipelines must be comparable or smaller than the latency of the operation, *e.g.* a couple dozen pipeline cycles for most SPMD workloads. To make general-purpose cores an attractive substitute to SPMD/SIMD units, extra hardware support must exist with low-latency bulk work distribution and synchronization.

The second argument is cost predictability: when a software scheduler is involved, it competes with algorithm code for access to the memory components. The overhead of communication and synchronisation between software schedulers increases with the number of hardware threads, and interferes with communication for computations, introducing jitter [21]. This can be avoided by a dedicated task control network physically separated from the memory network.

III. Architecture components

The Apple-CORE architecture proposes to combine RISC cores optimized for latency tolerance with dynamically scheduled HMT, hardware units next to cores to organize software concurrency within and across clusters of neighboring cores, called Microgrids, and a common NoC protocol to assign workloads to different regions of a Microgrid, different Microgrids on chip, or to other core types in an heterogeneous design. The design combines with various memory systems, although Apple-CORE also proposes a custom distributed cache network for scalable throughput.

A. Core micro-architecture

The core design, illustrated in fig. 1, is derived from a 6-stage, single-issue, in-order RISC pipeline:

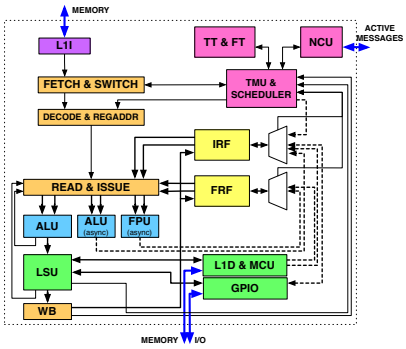


Fig. 1. Core micro-architecture.

- the register file is extended as *synchronizing storage*, where each register has a *dataflow state* which indicate whether it contains data or not (*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, so that 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;
- 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.

B. Core clusters and hardware concurrency management

Each core is equipped with a Thread Management Unit (TMU, table I). 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 II:

- *context allocation*, which reserves execution resources (PC, registers, bulk synchronizers) across one or multiple cores with a single request;

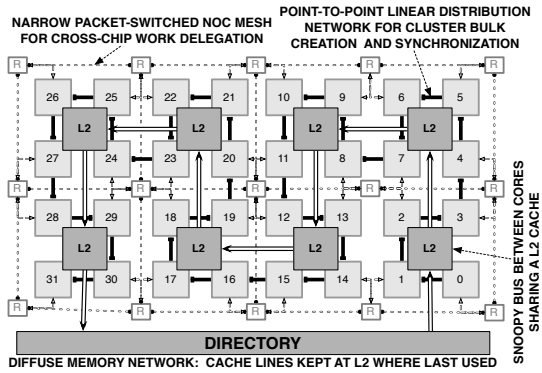


Fig. 2. Microgrid of 32 cores.

TABLE I
LOGICAL SUB-UNITS IN THE TMU.

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 II
CONTROL EVENTS HANDLED BY 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 III
PRIVATE STATE MAINTAINED 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

This state is maintained in dedicated hardware structures close to the TMU.

- *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 (fig. 2). 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.

C. 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 fig. 2): 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 [22], [23]: from the perspective of programs, it appears as a single shared memory with consistency resolved at concurrency management events.

D. 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. Concurrency semantics can then in turn be captured in various programming models, *e.g.* the bulk-synchronous parallelism (BSP [24]) and task parallelism constructs of OpenMP [25] and OpenCL [26]. To facilitate the implementation of multiple programming interfaces, 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. 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 between threads to implement the producer-consumer pattern (reads to *empty* block, writes to *waiting* by another thread wakes up);
- performing bulk creation and synchronization of *families* of tasks running thread programs, 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 [27] 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 [28] 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 access. 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 [29, 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, 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.

IV. Realization and evaluation

For evaluation, Apple-CORE has defined an overall chip design based on a single 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. 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. The cluster 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 [30]. 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 [31], to maximize bandwidth in streaming applications.

This chip was implemented in a discrete event, cycle-accurate, full-system simulator 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.

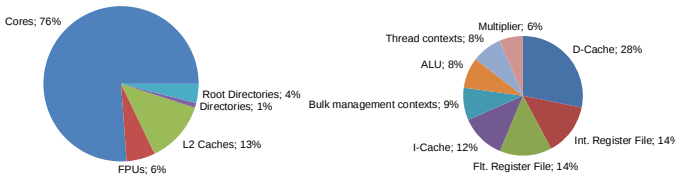


Fig. 3. Chip area breakdown: entire grid (left), per core (right).

A. Area and timing estimates

The area and access time requirements of the reference configuration have been evaluated using CACTI [32]. 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 (fig. 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.

B. Software environment

Apple-CORE has produced a C language implementation based on the GNU C compiler (GCC), and a run-time environment derived from FreeBSD [33]. 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 [34], similarly to [35]. This *platform* has fostered the separate development of a parallelizing C compiler targeting SVP [36], [37] and a SVP back-end to the array-oriented, functional productivity language SAC [38], [39].

C. Performance and scalability

Figure 4 illustrates the computational density and latency tolerance for FPU operations. This heterogeneous compute-bound workload of 40k imbalanced threads (left 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 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 IV-A above, this implies the baseline is exceeded at 20$\times$ smaller area budget and less than half the frequency. The round-robin distribution

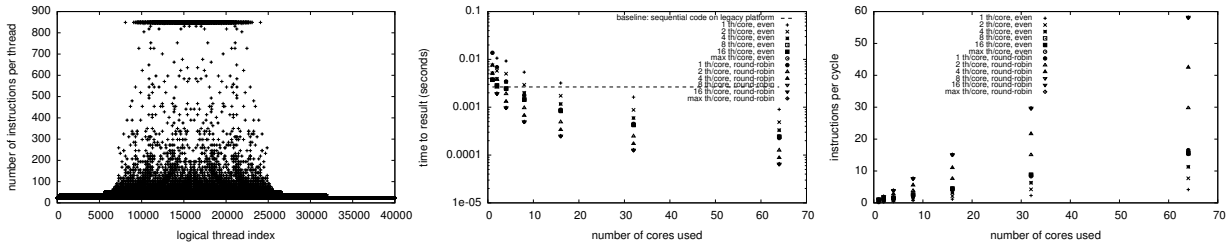


Fig. 4. Example heterogeneous workload: Mandelbrot set approximation.

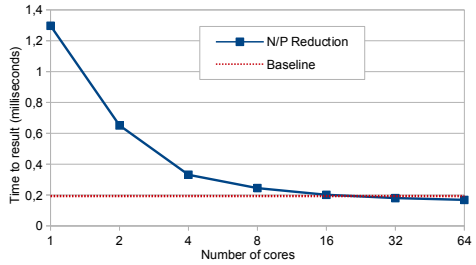


Fig. 5. Scalability of a N/P reduction of 64k floating-point values.

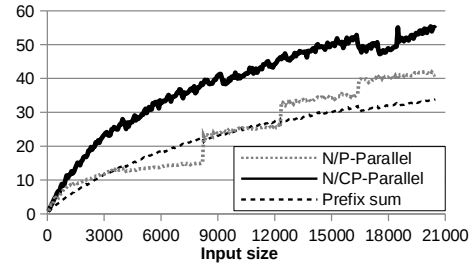
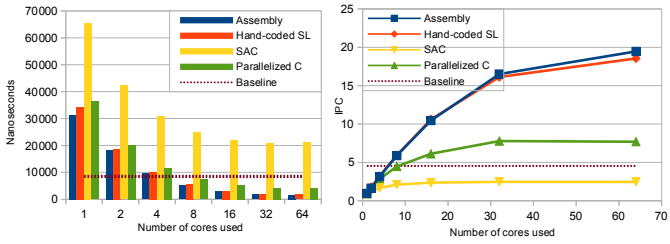


Fig. 6. Speedup of various parallel reduction strategies on 64 cores.



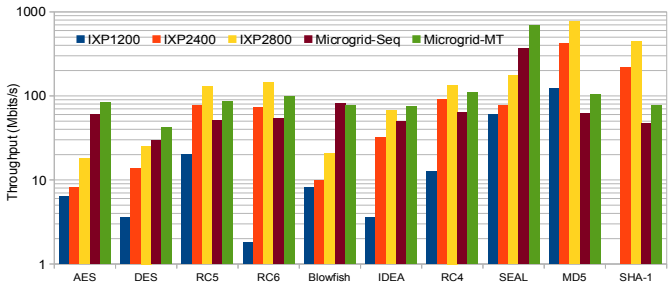


Fig. 8. Throughput for one stream on one core.

with 16 threads per core further scales nearly linearly with full pipeline utilization up to 64 cores (right in figure).

Figures 5 and 6 illustrate the scalability for parallel reductions. In fig. 5, 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 fig. 6, 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 [28] 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 onwards. 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 the memory throughput approaches the external bandwidth of the chip and the workload becomes memory-bound, preventing extra speedup past 32 cores.

D. Example throughput application: cryptography

In [40], [41], 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 ([40, fig. 4], [41, fig. 3]). Two Microgrid codes are used, one purely sequential and one where the inner loop is parallelized. As the results in fig. 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 and minimizes latency

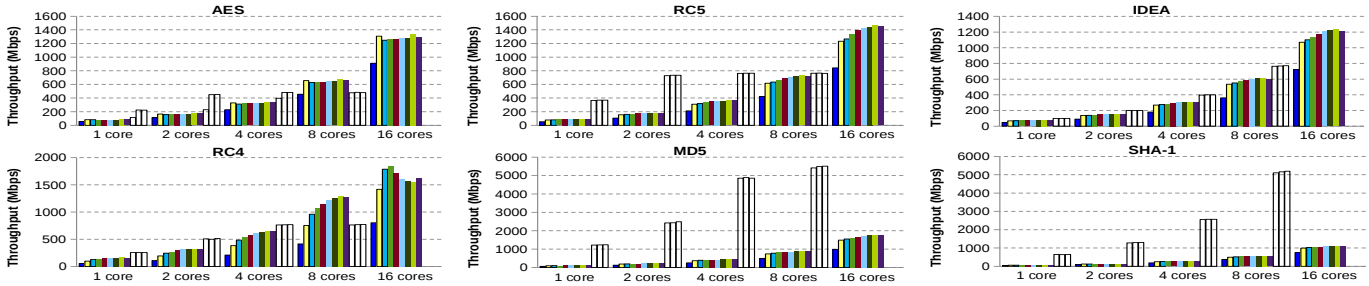


Fig. 9. Combined throughput for 1-8,16 streams per core on 1-16 cores (1-256 streams total). IXP2800 performance in leftmost 3 bars at each core group.

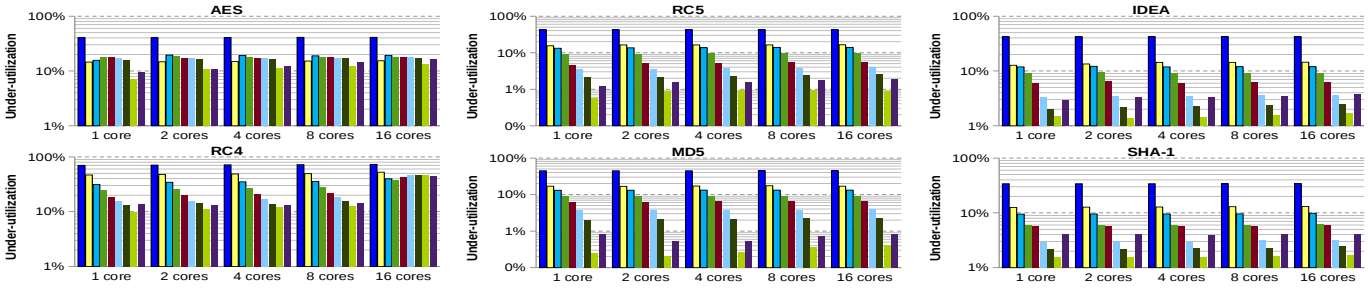


Fig. 10. Pipeline under-utilization for fig. 9.

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 9 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 ([40, fig. 6], [41, fig. 8]). For each subcluster size, increasing the number of flows per core increases utilization (fig. 10) 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.

V. Discussion and future work

The results in this paper show the potential of general-purpose, fine-grained concurrency in various simple scenarios. Unfortunately the Apple-CORE project has 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 [42]) 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.

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.

VI. Conclusion

This paper 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. Then its hardware management protocol in turn offers reliable multi-core throughput scalability up to the bandwidth capacity of the memory system.

Acknowledgements

This research was supported by the European Union under grant numbers FP7-215216 (Apple-CORE) and FP7-248828 (ADVANCE).

REFERENCES

[1] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. Shen, "Coming challenges in microarchitecture and architecture," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 325–340, mar 2001.

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

[3] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, 2005.

[4] M. Shah, R. Golla, P. Jordan, G. Grohoski, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja, "SPARC T4: A dynamically threaded server-on-a-chip," *IEEE Micro*, vol. PP, no. 99, p. 1, 2012.

[5] Advanced Micro Devices, Inc., "AMD Fusion APU era begins." [Online]. Available: http://www.amd.com/us/press-releases/Pages/amd-fusion-apu-era-2011jan04.aspx

[6] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *Proc. IEEE Computer Society Annual Symposium on VLSI*, 2002, pp. 105–112.

[7] J. Henkel, W. Wolf, and S. Chakradhar, "On-chip networks: a scalable, communication-centric embedded system design paradigm," in *Proc. 17th International Conference on VLSI Design*, 2004, pp. 845–851.

[8] O. Moreira, J. J.-D. Mol, and M. Bekooij, "Online resource management in a multiprocessor with a network-on-chip," in *Proc. 2007 ACM symposium on Applied computing*, ser. SAC '07. ACM, 2007, pp. 1557–1564.

[9] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, March 1995.

[10] I. Bell, N. Hasasneh, and C. Jesshope, "Supporting microthread scheduling and synchronisation in CMPs," *International Journal of Parallel Programming*, vol. 34, pp. 343–381, 2006.

[11] J. E. Thornton, "Parallel operation in the Control Data 6600," in *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, ser. AFIPS '64 (Fall, part II). New York, NY, USA: ACM, 1965, pp. 33–40.

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

[13] J. Boisseau, L. Carter, A. Snavely, D. Callahan, J. Feo, S. Kahan, and Z. Wu, "CRAY T90 vs. Tera MTA: The old champ faces a new challenger," in *Proc. Cray User's Group Conference*. 411 First Avenue South, Seattle, WA 9810, USA: Cray Inc., June 1998.

[14] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz, "Multi-processor performance on the Tera MTA," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–8.

[15] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," *SIGARCH Comput. Archit. News*, vol. 23, pp. 392–403, May 1995.

[16] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading technology architecture and microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 1–12, 2002. [Online]. Available: http://www.mendeley.com/research/hyperthreading-technology-architecture-and-microarchitecture/

[17] R. H. Halstead, Jr. and T. Fujita, "MASA: a multithreaded processor architecture for parallel symbolic computing," *SIGARCH Comput. Archit. News*, vol. 16, pp. 443–451, May 1988. [Online]. Available: 10.1145/633625.52449

[18] R. S. Nikhil and Arvind, "Can dataflow subsume von Neumann computing?" *SIGARCH Comput. Archit. News*, vol. 17, no. 3, pp. 262–272, 1989.

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

[20] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and 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*. New York, NY, USA: ACM, 1991, pp. 164–175.

[21] T. Anderson, E. Lazowska, and H. Levy, "The performance implications of thread management alternatives for shared-memory multiprocessors," *IEEE Trans. Comput.*, vol. 38, no. 12, pp. 1631–1644, dec 1989.

[22] L. Zhang and C. R. Jesshope, "On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores," in *Euro-Par Workshops*, ser. LNCS, Bouge and et al., Eds., vol. 4854. Springer, 2007, pp. 38–48.

[23] T. D.Vu, L. Zhang, and C. R. Jesshope, "The verification of the on-chip COMA cache coherence protocol," in *International Conference on Algebraic Methodology and Software Technology*, 2008, pp. 413–429.

[24] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, aug. 1990.

[25] OpenMP Architecture Review Board. (2008) OpenMP application program interface, version 3.0. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[26] Khronos OpenCL Working Group. (2009) The OpenCL specification, version 1.0.43.

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

[28] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, October 1980.

[29] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, June 1971.

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

[31] M. A. Hicks, M. W. van Tol, and 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, July 2010, pp. 341–348.

[32] S. Wilton and N. Jouppi, "Cacti: an enhanced cache access and cycle time model," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 5, pp. 677–688, may 1996.

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

[34] M. W. van Tol and C. R. Jesshope, "An operating system strategy for general-purpose parallel computing on many-core architectures," *Advances in Parallel Computing*, vol. High Performance Computing: From Grids and Clouds to Exascale, no. 20, pp. 157–181, 2011.

[35] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Trans. Comput. Syst.*, vol. 12, pp. 271–307, November 1994.

[36] D. Saougkos, D. Evgenidou, and 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, January 2009.

[37] D. Saougkos and 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)*, ser. Lecture Notes in Computer Science. Springer, 2011, pp. 151–157.

[38] C. Grelck and S.-B. Scholz, "SAC: a functional array language for efficient multi-threaded execution," *International Journal of Parallel Programming*, vol. 34, no. 4, pp. 383–427, Aug 2006.

[39] C. Grelck, S. Herhut, C. Jesshope, C. Joslin, M. Lankamp, S.-B. Scholz, and 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*, 2009.

[40] Z. Tan, C. Lin, H. Yin, and B. Li, "Optimization and benchmark of cryptographic algorithms on network processors," *IEEE Micro*, vol. 24, no. 5, pp. 55–69, September/October 2004.

[41] Y. Yue, C. Lin, and Z. Tan, "NPCryptBench: a cryptographic benchmark suite for network processors," *SIGARCH Comput. Archit. News*, vol. 34, no. 1, pp. 49–56, September 2005.

[42] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and 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*, ser. ASPLOS '12. ACM, 2012, pp. 37–48.