

Resource-agnostic programming for many cores through a hardware/software co-design

T.A.M. Bernard, C. Grelck, M. Hicks, C.R. Jesshope, R. Poss

Institute for Informatics, University of Amsterdam, The Netherlands

Abstract

Many-core architectures are now a reality and programming them is still a challenge. Our approach is to promote hardware and software co-design in order to efficiently program a many-core architecture. In this paper, we present an abstract concurrent execution model, its possible hardware implementations, its programming model and the tool chain that we have implemented. The contribution of this paper, beyond exposing the novel architecture and programming model, is to present an important property of our data-driven, fine-grained concurrent execution model: the performance characteristics for a single program implementation match closely the expectations derived from architectural properties, across a range of different hardware configurations. We show through experiments how this environment allows to express diverse scientific problems using resource-agnostic programs and still approach the maximum performance allowable by the hardware on different configurations without a need to tailor the program code to the architectural settings.

Keywords: Concurrent execution model, high performance computing, configurable many-core architectures, resource-agnostic parallel programming language, multi-core menace.

1 Introduction

Many-core architectures are a potential solution to the limitations confronting advances of mainstream computing [8]. However, programming applications on such platforms is difficult [6, 1]: thinking in parallel is subject to pitfalls [7]; also, most existing paradigms [13] require programmers to express not only computations, but also to manage concurrency explicitly. This must then be tailored to the specific granularity of the target architecture, to balance concurrency overheads with computational costs. This forces a full development cycle each time the concurrency granularity evolves. We address this issue with a novel hardware/software co-design where where concurrency is default in the programming language and where the hardware and run-time system are in charge of mapping concurrency to the appropriate target granularity. In this direction, we provide both a concurrent execution model, new architecture concepts, a matching programming language and the corresponding tool chain.

Our work is based on a model called *SVP* [3, 5]. It combines fine-grained threads with dataflow synchronisation. This requires the maximum concurrency available to be *exposed*, without the need to explicitly *manage* it. Independent tasks are exposed as thread composition and data dependencies are captured using dataflow semantics. Hardware-supported dataflow synchronisation can then be used to implement efficient scheduling through arbitrary thread interleaving. Concurrent composition is available at all levels. Bulk synchronisation is implemented over concurrent regions hierarchically, which can be used for both fine-grained mappings (one SVP thread per hardware thread) or coarse-grain mapping at an arbitrary level of the composition hierarchy by sequentializing entire branches of the concurrency tree. With this approach, mapping and scheduling programs in hardware is thus decoupled from the program implementation.

In this paper, we illustrate that the performance of a resource-agnostic program representation in SVP adapts automatically to the hardware characteristics and yields adequate performance without changes across different architectural configurations. In other words, we promote our research goal: *“Implement once, compile once and run on all.”* To do so, we will show how SVP is expressed both in the architecture and the programming model, and combined through our tool chain. We then proceed to analyze the performance of a set of characteristic scientific benchmarks.

2 Related work

Work in parallel architectures, and here particularly dataflow-like architectures, has a long history of research which now receives focus given the impending limits of conventional performance scaling.

Historically, closely related work can be observed from around 20 years ago at MIT during Arvind’s research on combining dataflow principles with Von Neumann computing [14, 10]. This work explored the potential benefit of providing the scheduling of dataflow computing on conventional instruction sets. At the more modern and conceptual level, transactional memory [9] satisfies similar requirements as the work undertaken by our group. However, in transactional memory, dependencies are defined dynamically, in contrast to the static capturing of dependencies in the SVP model (see Section 3.1) which facilitates very fine grained concurrency.

More recent work similar to the principles we used is the WaveScalar architecture [16]. This achieves dataflow-like execution by using an implicit program counter for memory ordering. However, we found restrictions in the amount of concurrency that this can expose.

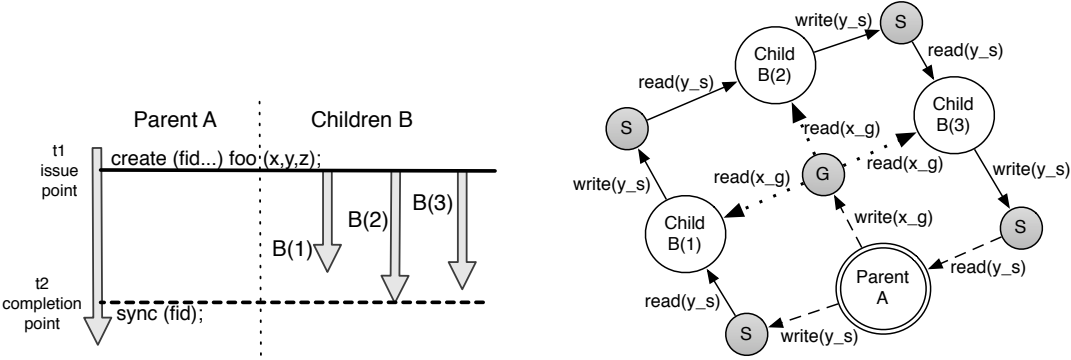
A characteristic example of typical hardware currently available in the industry is NVidia’s CUDA [15]. This makes use of GPU arrays by providing a programming interface that allows off-loading of computations from the CPU via a high speed bus, for example AGP. The OpenCL framework is commonly used to program these devices. The limitations of this model (both hardware and programming framework), when compared to the SVP approach are two-fold: the programming model requires highly explicit concurrency through the use of many code directives, and the hardware itself relies on an ‘accelerator’ model, where more general computations are carried out on a CPU, and hence computations carried out on the CUDA architecture are bound by bus speed limitations. Also, the scheduling of work/threads lacks the flexibility and concurrent composition allowed by SVP.

3 Overview of the SVP model and its implementation

SVP is implemented both as a parallel programming language called μ TC (Section 3.3) and an architecture called the Microgrid (Section 3.2). We also provide a compiler tool chain to map μ TC onto our hardware implementation.

3.1 The SVP execution model

SVP offers a uniform means of capturing concurrency. It uses a *create* action to generate threads grouped into families. Family creation uses a fork-join based pattern illustrated in Figure 1(a): *create* forks a named family of statically homogeneous, but dynamically heterogeneous threads. Family synchronisation on termination is performed with a *sync* action in the parent thread that completes after all child threads are terminated. This model captures concurrency hierarchically, from software component composition down to inner loops of computations.



(a) SVP family creation. Thread A creates child family B. Execution resumes after *sync* in A when all threads in B have terminated.

(b) SVP synchronisation. A creates a family B of three threads with two parameters x_g (global channel) and y_s (shared channel). Communication occurs when the synchronising objects are accessed in each thread (reads or writes).

Figure 1: SVP Family creation and inter-thread communication.

A family of threads is characterised by its index sequence, the code entry point for each thread and a definition of unidirectional dataflow synchronisation channels from, to and within the family. These are shared between the parent thread and between the threads in a family. They offer the semantics of I-structures [2], i.e. blocking reads and non-blocking writes. Figure 1(b) illustrates the two types of communication channels. *Globals* are written once by the parent and are available for reading by each child thread. *Shareds* are defined between every consecutive pair of threads. A thread writes to each outgoing shared channel once, allowing the next thread’s read from the incoming channel to succeed. The first and last threads are also connected to the parent thread.

Forward-only communication allows SVP to be free of communication deadlock under composition of families [17]. Also for any program, there exists a sequential schedule of the entire

concurrency tree which satisfies dependencies in order. This can be used at any level to ‘flatten’ a branch of the concurrency tree for sequential execution when the hardware’s concurrency granularity is reached. Another characteristic is the separation of concern between the program and its mapping to computing nodes (e.g. cores). The latter is achieved by binding a collection of computing nodes, called a *place*, to a thread family upon its creation. This can happen at any level in the family hierarchy and can be decided at run-time.

3.2 The SVP hardware implementation: Microgrids of SVP cores

The proposed hardware implementation of SVP is an architecture called a Microgrid. This is a composition of multiple clusters of SVP cores connected in rings. Each SVP core extends an existing instruction set architecture (ISA) with extra instructions that capture SVP’s semantics. These extensions are described in Figure 2.

Family/thread management		Family configuration	
<i>allocate</i>	allocate an entry in the on-chip family table and bind to a SVP place.	<i>setstart</i> , <i>setlimit</i> , <i>setstep</i>	set the index range.
<i>create</i>	start creating a family of threads.	<i>setblock</i>	set the maximum number of threads per core.
<i>sync</i>	wait for termination of a child family.		

Figure 2: List of SVP instructions which can be added to an existing ISA.

Next to the ISA extensions, each SVP core allows to configure on-chip physical registers as I-structures [2] and use them provide low-latency synchronisation between threads: a read on a pending register causes a thread switch at the next pipeline cycle. This mechanism together with support for a large number of threads per core (up to 256) is a key concept of our approach. The implementation details are presented in [5].

To create families, the *allocate* and *set* instructions acquire and configure a control structure in an on-chip memory that holds information for thread management at the family level. Then, the *create* reads the family table entry and starts asynchronous creation of threads. The *allocate* instruction also identifies the cluster of cores that a *create* instruction will be delegated to using a low-latency bus. The creation process uses the control structure and allocates threads across

cores on the cluster. On each core, thread resources are allocated both from control structures and blocks of registers in the register file. One thread can be created every cycle, making family creation affordable even for very few instructions per thread. The *sync* instruction causes a pending read on a special register which is asynchronously set and becomes readable upon family termination.

As said above communication occurs using registers. Pending registers contain bits that indicate the expected origin of the dependency (either current core or previous core on the ring). When a read occurs to a pending register, the thread is suspended within 1 cycle. A request is sent to the neighbouring core on the ring if the dependency is not local and the thread is scheduled again in the pipeline when the dependency is satisfied. Register configuration is performed by the create process based on information defined by the parent thread and the binary program code. This information defines a *virtual register window* for each created thread which separates the thread's register context (as visible from the ISA) in four regions: *dependents* that map to the dependencies from the previous thread (incoming values for SVP's *shared* channels), *globals* that map to the dependencies from the parent thread (SVP's *global* channels), *locals* for local use by the thread, and *shares* that provide the dependencies to the next thread (outgoing values for SVP's shared channels).

As an important optimization, the local window can use fewer physical registers than are available in the ISA, and the *dependent* and *shared* virtual registers can be physically shared between threads. This mapping is dynamic [5] and allows to efficiently utilize the register file.

3.3 The SVP programming language: μ TC

Introduced in [12], μ TC extends the C programming language with SVP extensions. *Thread functions* are elementary programs run by threads, defined by syntax similar to C functions. Thread families are distinguished by *thread indices*, integer variables automatically predefined to a different value when each thread starts. Family creation is triggered by a **create** construct. Synchronising channels are exposed as variables in the C language with special read and write semantics. Furthermore, this is further separated into *shared parameters*, shared between adjacent threads in a family, *global parameters* shared by all threads in a family and *termination synchronisers*, which cause a reading thread to wait for termination of an asynchronous operation.

```

1 thread ddot(shared double res,
2           double* x, double* y)
3 {
4   index i;
5   res = x[i] * y[i] + res;
6 }
7
8 thread main(void)
9 {
10  create(fid ;;0;1000;1;) ddot(x = 0,u,v);
11  sync(fid);
12 }

```

Figure 3: μ TC example: simplified reduction from the BLAS library. The reduction is computed by a family of 1000 threads indexed with 'i'. The read to 'res' synchronises with the previous thread; the two memory loads can be performed concurrently before synchronisation; the read to 'res' in the next thread completes after the current thread completes its write to 'res'. After the **sync** construct) in the parent, 'x' contains the result from the reduction.

The extra constructs of μ TC are language primitives. This can be compared to other approaches [13] where concurrency is available through external libraries. We use primitives to allow a finer concurrency granularity than what external libraries allow. [4] describes limits and dangers of library-based concurrency that we tried to avoid.

Listing 3 illustrates an example μ TC program. The main construct is the reflection of SVP's *create* action. Its syntax is: **create**(*family_id*; *place_id*; *start*; *limit*; *step*; *block*)*fun*(*args...*). A family is identified by *family_id* and its range is bounded by the values of *start*, *limit* and *step*, as in common uses of C's **for** statement. The other parameters are the *place identifier* for a SVP place where the family should be created; and the *blocking factor* which provides a hint to the run-time system about the number of threads to create on each core.

As required by SVP, μ TC allows for resource-agnostic concurrent programs. The concurrency-aware programmer is only responsible of exposing concurrency and revealing dependencies. Mapping and scheduling is resolved by the hardware and the program needs to be compiled only once to be executable on any Microgrid configuration.

3.4 SVP core compiler

In the SVP tool chain, the core compiler is the central *enabling technology* that translates the SVP language to code executable on the target SVP architecture implementation.

Next to this enabling role, it is also the point where architecture-specific optimizations can take place. The rest of this paper considers only the Microgrid architecture: no specific compiler optimizations are required since the hardware is responsible for managing concurrency and provides

tolerance to inefficient code. However, our compiler has been designed as an extension to an existing framework (GCC) where we expect any future optimizations to be readily implementable.

The challenge in the design and implementation of our current compiler lies in the mapping of μ TC special variables to register configurations for the create process (cf. Section 3.2). While each thread is a sequential process and the regular C translation of a function body can be applied in principle to any μ TC thread function, we had to consider that registers in the target SVP ISA also have a state and may be shared physically between threads. Unbeknown to a regular C compiler, certain registers must be written to only once and not read after they are written. Moreover, the μ TC shared thread parameters are mapped simultaneously to two registers, one used for reads and another for writes. These peculiarities not only required the creation of a new back-end to support our hardware, but also required changes to several existing optimization engines. We plan to release more details on these aspects in the future.

4 Evaluation framework

The evaluation was performed using a platform emulation of a Microgrid. The emulation captures state transitions down to the lowest level in the core pipelines. It also provides a fully parameterisable, cycle-accurate simulation of all hardware aspects of the Microgrid: core functional units, interconnects, network-on-chip, memory architecture, and external memory interface. The memory architecture is a COMA derivative described in [18]. We used parameters suitable for hardware that can be built using current technology [5, 11]: 128 SVP cores implementing the DEC Alpha ISA, sharing 64 FPUs with separate pipelines for *adds* (2 pipelines), *divs*, *mul*s and square root operations (1 pipeline each); and 32 L2 caches of 32KB each shared by group of 4 cores. L2 caches are in turn grouped in COMA rings of 8 caches connected to a COMA directory. The top level directory ring is connected via two DDR3-1600 channel to external storage of arbitrary size. The performance figures presented in the next section should be considered in the light of the following hardware characteristics: the two DDR3-1600 channels provide 1600 million 64-bit transfers/s of 64 bits each, i.e. a peak bandwidth of 25.6GB/s overall for the external memory interface; each COMA ring provides a total bandwidth of 64GB/s, shared among its participants; the bus between cores

and L2 caches provides 64GB/s of bandwidth, shared among 4 cores; the SVP cores are clocked at 1GHz. The cores are grouped in 8 clusters (SVP places) of 1, 1, 2, 4, 8, 16, 32 and 64 cores, to allow running benchmarks on places of varying sizes. The selection of cluster sizes has no impact on the performance of each cluster other than the number of cores. Figure 4 provides a schema of this configuration.

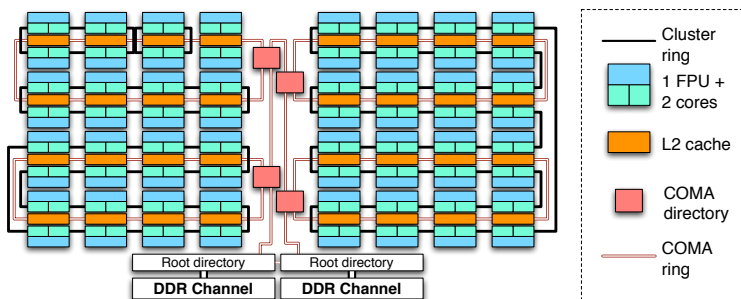


Figure 4: Functional schema of a Microgrid of 128 cores.

5 Experiments and results

In the following sections, we present a range of results spanning a selection of computation kernels, and two cipher algorithms. The selection was made among known scientific problems (Livermore kernels, linear algebra, cryptography) as they are commonly used in benchmarks. Our selection process was guided by the intent to expose the relationship between implementations, architecture parameters and actual observed performance.

5.1 Performance of essentially sequential code

We consider the function DNRM2 of the BLAS library. This function computes the Euclidean norm of a vector: sum up the squares of the vector elements, then computes the square root of the sum. In the sequential algorithm, each iteration requires one memory load, followed by a FP *multiply-add*. The *add* in each iteration has a data dependency on the previous iteration. We do not yet consider here the benefits of implementing this function using another algorithm based on e.g. a parallel reduction. This will be evaluated in Section 5.2.

Listing 1: BLAS DNRM2 in μ TC.

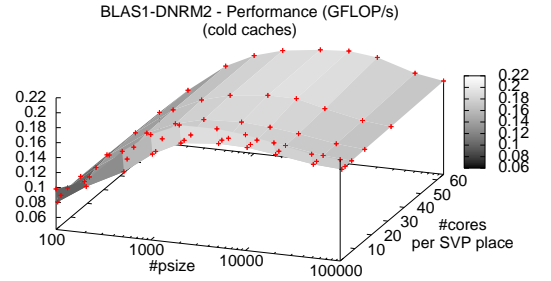
```

1 thread accum(shared double sum,
2             double* v) {
3     index i;
4     sum += v[i] * v[i];
5 }
6
7 thread dnrn2(shared double result,
8             int n, double *v) {
9     create(fid ;;0; n;1);
10    accum(sum = 0, v);
11    sync(fid);
12    result = sqrt(sum);
13 }

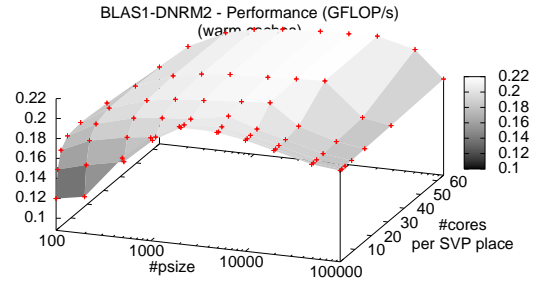
```

Here, one must consider that current architectures must deal with typical memory latencies of hundreds of cycles or more. Branch prediction, prefetching and out-of-order issue provide latency tolerance only for tens of cycles. This is sufficient to optimize performance when working from cache but causes the core to become I/O bound for larger data sets. In our approach, the hardware provides latency hiding through multithreading in the pipeline: the load and FP *mul* form an independent prefix to the dependent *add*, so latency hiding is possible by overlapping them across threads and across cores. In principle, we can expect that the latency of loads can be hidden even with cold caches and the resulting performance is the sequential performance of the dependent FP operations assuming no delays.

We have a naive implementation of DNRM2 in μ TC (Listing 1). This compiles to code with 4 instructions per thread, including 2 FP operations. In each thread, the 3 initial instructions are entirely independent and can run concurrently. So, latency hiding applies and we can expect a visible 1-cycle cost per instruction. This is added to 1 cycle required to start each thread. Then, the dependent adds are executed in sequence with a 4-cycle or 6-cycle latency (4 cycle FP latency + 3 cycle reschedule cost in the core pipeline, two of which may overlap). We note that no latency hiding is possible in this case, because there is a data dependency. There is thus a minimal theoretical cost of 8-10 cycles per thread on one core, i.e. a theoretical peak



(a) Cold caches



(b) Warm caches

Figure 5: Performance of DNRM2 on one SVP place.

performance of 0,20-0,25GFLOP/s. As Figure 5 shows, we actually observe up to 0,20GFLOP/s on one core, i.e. within the expected maximum range. We have confirmed experimentally that this sort of gain can be reproduced for a variety of memory architecture configurations with diverse delays.

As a side note, we observe that we can furthermore increase the performance by increasing the number of cores, because some of the thread creations and independent instructions can be scheduled concurrently across cores. However, not much gain is to be expected since the data dependency is then distributed between cores. The other behaviors are explained as follows. When the problem size is too small at constant number of cores (more than 2-4 cores), there are not enough threads per core to fill the pipeline and to hide latency, decreasing visible performance (e.g. problem size 100 with cold caches in Figure 5(a)). Also, as expected, the extra latency at small problem sizes is less noticeable with warm caches due to the lower access latency to caches. However, also as expected, this caching benefit disappears when the problem size is large enough (e.g. size 50000 at 32 cores in Figure 5(b)). Finally, overlapping of loads across cores does not increase performance past two cores, because once memory latencies are hidden, the algorithm is still sequential and subject to the distributed data dependency.

We conclude by highlighting that we have achieved the automatic, fine-grained parallelisation of the concurrent part of a purely sequential algorithm. This result is significant considering that our software implementation is naive, resource-agnostic and that the hardware scheduler is not specifically optimized for this algorithm.

5.2 Performance of parallel reductions

The case study in this section is the inner vector product (IP). For this algorithm, we have used the associativity of the reduction as an opportunity to parallelize across cores. Our program (cf. Figure 6) is a straightforward extension of the naive implementation in μ TC as the number of cores in the ‘current place’ is exposed to programs as a language primitive. It splits the reduction into two stages, where a first family is created with one thread per core, and each thread in that family creates a local family to perform a core-local reduction. In effect, this amounts to parallelizing the reduction among the cores. So, when the number of threads per core is significantly larger than

```

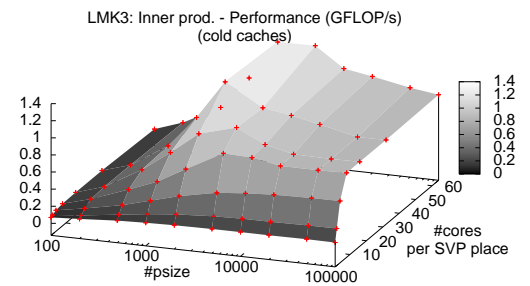
1 thread accum1(shared double sum,
2   int sp, double* x, double *y) {
3   index i;
4   create(fid; LOCAL; i; i+sp; 1; )
5     accum1(sum1 = 0, x, y);
6   sync(fid);
7   sum += sum1;
8 }
9 thread accum2(shared double sum,
10              double* x, double *y) {
11   index i; sum += x[i] * y[i];
12 }
13 thread lmk3(shared double result,
14   int n, double *x, double *y) {
15   int p = local_cores ();
16   create(fid; ; 0; p; 1; 1)
17     accum1(sum = 0, n/p, x, y);
18   sync(fid);
19   result = sum;
20 }

```

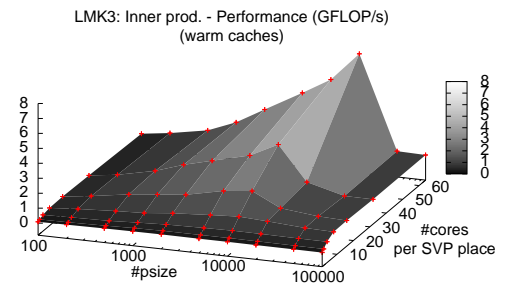
Figure 6: N/P parallel reduction for the inner product

the number of cores, the cost of the final reduction is negligible and the performance should scale linearly with the number of cores. Figure 7 shows the experimental results for this code.

To analyze our results, we consider the case of one core first. The compiled code for the inner thread contains 7 instructions, the first 6 of which are independent but contain two loads. The last instruction is a dependent *add*. Using the same methodology as before, if we assume that latency hiding occurs, we have an expected cost of 11-13 cycles per thread (1 create cycle + 6 independent instructions + 4-6 cycles for the dependent *add*), hence a theoretical maximum performance on one core of 0,15-0,18GFLOP/s. The actual performance observed falls within this range (0,151GFLOP/s). Assuming linear speedup on the number of cores, the expected performance for P cores is thus 0,15P. Again, this matches the experimental results up to 8 cores when the problem size provides enough threads per core (i.e. N=1250 for P=2, N=2500 for P=4 and N=5000 for P=8). Then, we consider the effect of adding more cores. For any pair of FP operations, the algorithm requires two loads (16 bytes) to complete. Therefore, to sustain an overall FP perfor-



(a) Cold caches



(b) Warm caches

Figure 7: IP performance, using N/P reduction.

mance of X , the memory system should sustain a memory bandwidth of $8 \times X$ bytes/s. However, as detailed in Section 4, the maximum external memory throughput is 25.6GB/s. This implies a first theoretical maximum of 3.2GFLOP/s for this program overall, assuming that all loads are served only once by the external interface. Then, when the working set does not fit in the L2 caches, loads from adjacent threads to shared cache lines on each core are interleaved with evictions due to loads from other threads to other cache lines. In the worst case, each pair of loads may require an entire cache line to be transferred from external memory. The ‘visible external bandwidth’ can then drop to 6.4GB/s, and the peak performance to 0,80GFLOP/s. As our experiment shows, with cold caches, the program becomes I/O constrained at $P=32$ for $N=2k$ and $P=8$ for $N \geq 10k$. The observed, reduced performance is then 0,88GFLOP/s above the expected minimum. With warm caches, this restriction is lifted and linear speedup is possible again, as long as the working set fits in the L1 caches, there are enough threads per core and until communication latencies dampen the benefits of latency tolerance.

We consider this ability of our resource-agnostic software implementation to be bound only by simple architectural limitations as yet another significant result.

5.3 Performance of data-parallel algorithms

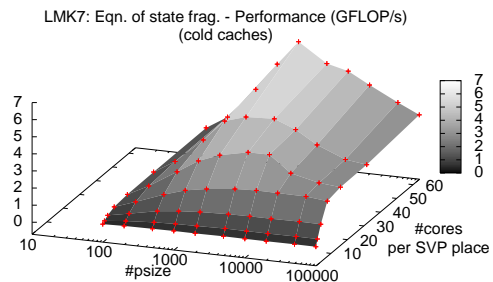
We show here the behavior of three data-parallel algorithms which exhibit three different behavior patterns that we have found to occur frequently across our research: the equation of state fragment (ESF) from the Livermore benchmark suite (loop 7), where the computation-intensive part actively helps with latency hiding; the matrix-matrix product from the Livermore benchmark suite (loop 21), which shows the benefits of locality of memory accesses; and the 1-D FFT, which is computation-intensive and has relatively poor locality of access. As for our other benchmarks, our μ TC implementation is straightforward and does not make attempts at explicitly mapping to hardware resources.

We consider the ESF first. The compiled code for the inner thread contains 33 instructions, including 9 loads, 1 *store*, 8 FP *adds* and 8 FP *muls*. As before, assuming latency hiding, each instruction should execute with a ‘visible cost’ of 1 cycle, plus 1 creation cycle. The expected performance on one core should thus be $1\text{GHz} \times 16 \text{ FPOPs/thread} / 34 \text{ cycles/thread}$, i.e. 0,47GFLOP/s.

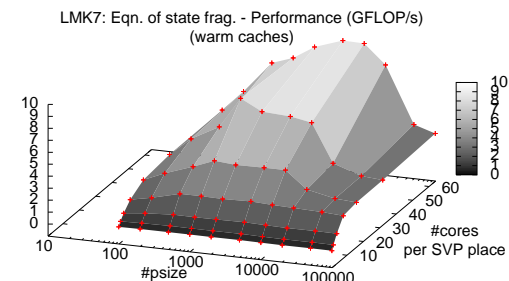
The observed performance is 0,43GFLOP/s, i.e. 90% of the expected maximum (see Figure 8(a)).

As for the IP above, when there are enough threads per core to hide latency, we observe linear speedup up to 8 cores. After this point, the program becomes I/O bound. For this program, a performance rate X requires $3,14 \times X$ bytes per second (the same item in array U is accessed by 7 different threads). With the external bandwidth limited to 25.6GB/s, the maximum performance reachable is thus 8,14GFLOP/s overall. Our experiments show that we reach 6,55GFLOP/s for $P=64$, i.e. 80% of this theoretical maximum. Again, as for the IP above when using larger problem sizes, L2 cache evictions cause the effective usable bandwidth to decrease and the performance drops accordingly. Conversely, when using warm caches and smaller problem sizes, greater speedups can be achieved (see Figure 8(b)). The program is then computation-bound: with $P=64$ the work is shared by 32 FPUs (see Section 4), each equipped with 2 *add/sub* pipelines and 1 *mul* pipeline. The theoretical peak performance for this program is then $32 \times 1 \text{ GHz} / 4 \text{ adds}$ (8 *adds* on 2 pipelines) + $33 \text{ GHz} / 8 \text{ muls} = 12\text{GFLOP/s}$. In our experiments, we reach up to 9,87GFLOP/s, i.e. 82% of this maximum. The behavior of this computation fragment thus matches adequately all expectations derived from the architectural parameters, without any optimization.

The next benchmark is a matrix-matrix product which implements naively the product of 25×25 and $25 \times N$ matrices. The IP operates in each core on rows of 25 consecutive elements in memory, thus achieving near-ideal locality of reference. Figure 9 shows the observed performance of this program. As expected, performance scales linearly across all SVP place sizes and all problem sizes. As previously, the IP requires 7 instructions to issue two floating-point operations. In this case, the peak performance is computationally limited: with one FPU per 2 cores, the theoretical peak

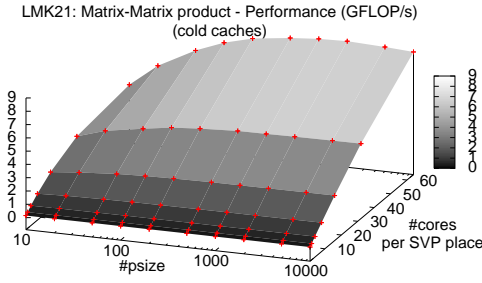


(a) Cold caches

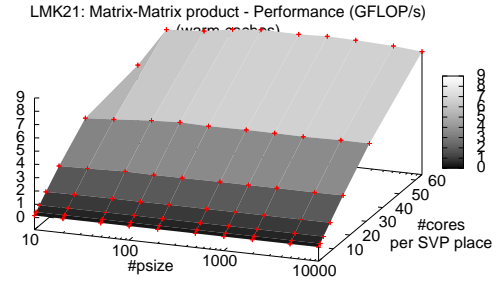


(b) Warm caches

Figure 8: Performance of the ESF.



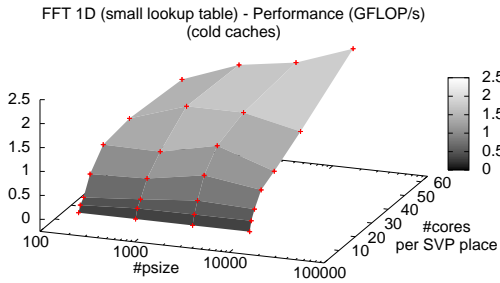
(a) Performance on cold caches



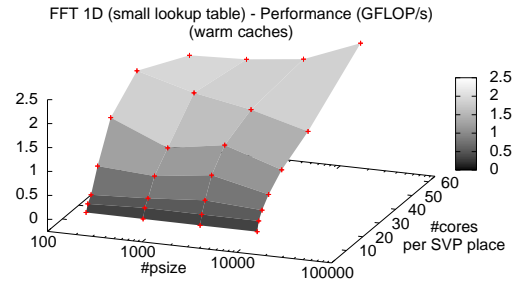
(b) Performance on warm caches

Figure 9: Performance of the matrix-matrix product.

rate is $P/7$, or 9,14GFLOP/s for $P=64$. Our experiments show an actual peak of 8,57GFLOP/s, i.e. 93% of the maximum. Again, the performance drops for small problem sizes and large place sizes when there are not enough threads per core to hide all latencies.



(a) Cold caches



(b) Warm caches

Figure 10: Performance of the 1-D FFT.

Again, we highlight this near-optimal use of resources across all sets of parameters in the context of a naive software implementation.

We conclude this section by looking at the 1-D FFT. Our μ TC implementation performs the forward FFT kernel without the bit reversal phase, as it would be used in a scientific application. The implementation, again, is straightforward. For a problem size N , the inner computation defines $N/2$ threads each containing 6 loads, 4 FP *muls*, 3 FP *adds*, 3 FP *subs* and 4 stores. This thread contains 33 instructions, so on one core, the theoretical peak performance assuming latency hiding is 34 cycles per thread (+ 1 create cycle), each containing 10 FPOPs, hence $10/34 = 0,29$ GFLOP/s.

The actual observed performance (cf. Figure 10) on one core is 0,23GFLOP/s, 78% of the expected maximum. When the number of cores and the problem size increase, the program becomes I/O constrained. To sustain performance X , this program requires $8 \times X$ bytes/sec of data. With the external bandwidth limited to 25.6GB/s, the peak theoretical performance is thus 3.2GFLOP/s. The actual observed peak performance for $P=64$ and $N=16k$ is 2.24GFLOP/s, 70% of this maximum. Again, our straightforward implementation yields nearly the maximum performance possible with the architecture, without any optimization.

5.4 Performance of stream ciphers

We consider stream cipher algorithms in the context of network communications where encryption/decryption occurs on multiple streams simultaneously, each of limited bandwidth. Typical applications usually require 1MB/s per stream, but with support for many streams. Here, we have implemented the popular AES and RC5 algorithms in μ TC. Our implementation is again resource-agnostic. Stream data is input and output through the external memory interface. The benchmarking driver inputs a number of streams and schedules the algorithms naively (round-robin) across available SVP places. Enough data is provided in each stream to obtain stable measurements.

As a first step, we have confirmed that we obtain more than 1MB/s per stream. We also confirmed that the program is not I/O bound, by checking that no memory interconnect becomes saturated. We observed that when running on multiple SVP places with the same number of cores, the overall performance scales linearly with the number of places as factor, up to the limit imposed by communication bandwidth. Thus, the performance is only dependent on the place size and the number of streams processed at each place. This in turn implies that once the best performance on a single SVP place is determined. The overall cipher performance using our implementation can be scaled arbitrarily without software changes, by changing the Microgrid layout to provide multiple places of optimal sizes and enough communication bandwidth. We then determine the optimal place size experimentally. When running on a single place, we can run AES on up to 5 simultaneous streams per place and RC5 on up to 7 streams. Figure 11 illustrates our results.

The main observation is that when there are more streams than cores, work on independent streams can be spread across cores and performance scales linearly, as expected. Additionally, we

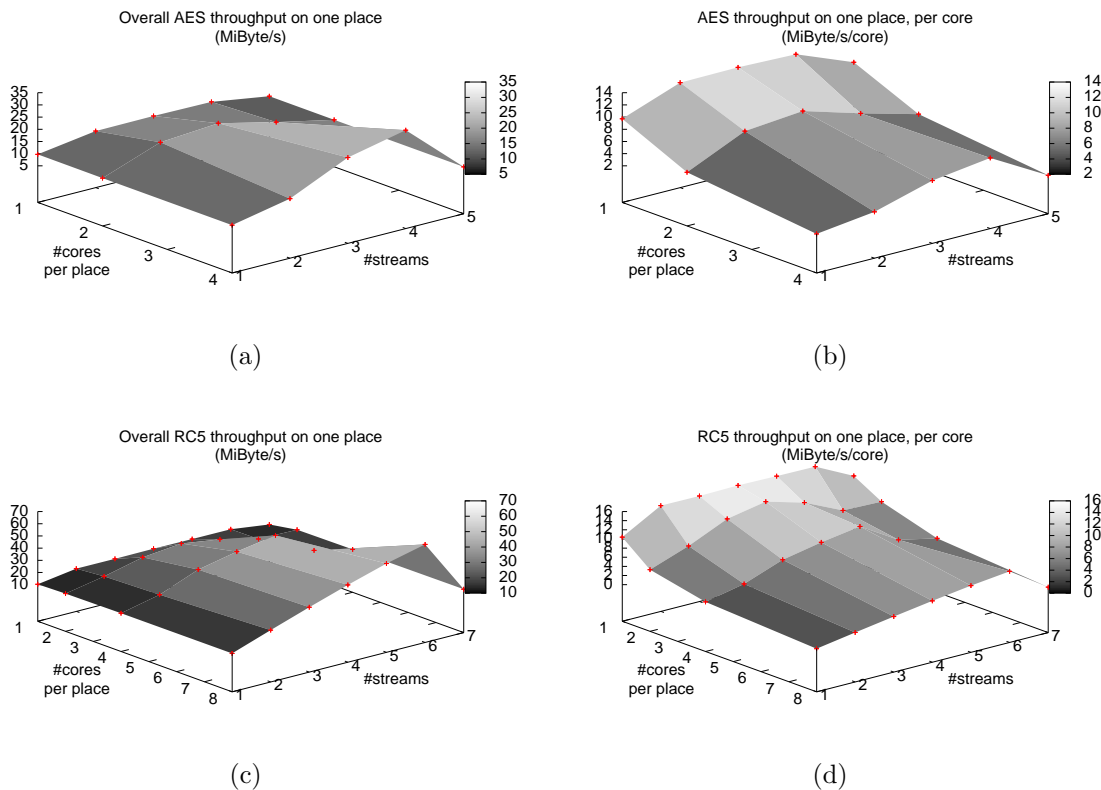


Figure 11: AES and RC5 performance on one SVP place.

note that on a single core, we achieve better throughput per core with multiple streams than with a single stream. This is again due to the latency hiding properties of the multithreaded pipeline: the algorithms have a high ratio of integer computations to memory operations. Latency hiding occurs until the average number of memory operations per integer instruction reaches the cache delay time, e.g. with AES and 5 streams, or RC5 with 7 streams.

We conclude from these observations that the performance of the ciphers is constrained by a simple function of architectural limitations and the performance on a place of one core. Within these constraints, performance scales freely without modifying the software. We have reproduced these results along a wider set of ciphers (RC4, RC6, SEAL, IDEA). In general, a set of performance requirements for stream ciphers can be reached either by optimising the algorithm for one stream on a single core, and/or *independently* by scaling up the number of SVP places. We see this independence of factors, together with the fact that *the software implementation stays entirely resource-agnostic*, as a significant contribution to the field.

6 Current and future directions

The results presented above show efficient use of the hardware resources of single SVP places by naive implementations of computation kernels. The next step in this research is therefore to compute mappings of entire applications (compositions of small software components), if possible dynamically, to SVP places of appropriate sizes. This will require feature extraction, preferably automatically, from complex concurrency trees (such as the expected throughput and latencies of computation nodes). The upcoming EU *ADVANCE* project will cover this area. From a language/compiler perspective, we intend to extend μ TC with software patterns and optimizations that further abstract mapping issues away from the programmer. For instance, primitives for parallel prefix sums and reductions could be added that would be automatically expanded by the μ TC core compiler to the type of construct introduced in Section 5.2. Finally, the *Apple-CORE* project aims at providing parallelizing compilers from existing languages to μ TC, as well as extra operating system support, as a path towards exploiting our novel architecture with existing application code.

7 Conclusion

Our SVP concurrency model allows to write concurrent programs once, and run efficiently on a variety of hardware configurations. We are able to express programs using a *resource-agnostic* representation in our system language μ TC, independent from the concurrency granularity offered by the target architecture.

Even with naive algorithm implementations, the hardware/software co-design allows for an efficient mapping and schedule at run-time. On one core, the performance achieved is close to the maximum performance allowed by the hardware assuming no latencies, which is a property of the multithreaded pipeline of the Microgrid architecture. With many cores, performance scales adequately and stays a simple function of the static program code, the architecture configurations and the one-core performance, and is thus reliably predictable.

We have thus achieved an environment *which decouples concurrency management from the implementation of programs*. Programmers or code generators can focus on exposing concurrency while our run-time and hardware system manages the concurrency automatically and efficiently.

Acknowledgements

The development of SVP, the Microgrid architecture and the μ TC compiler was initially supported by the NWO *Microgrids* project, then by the EU *Apple-CORE* project. SVP and its implementation is a group effort of the CSA group at the University of Amsterdam. The authors would like to thank especially Mike Lankamp for his work on emulation and the cipher algorithms in μ TC.

References

- [1] AMARASINGHE, S. (How) can Programmers Conquer the Multicore Menace? In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2008), ACM, pp. 133–133.
- [2] ARVIND, NIKHIL, S., R., AND PINGALI, K. K. I-Structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (1989), 598–632.
- [3] BERNARD, T., BOUSIAS, K., GUANG, L., JESSHOPE, C. R., LANKAMP, M., VAN TOL, M. W., AND ZHANG, L. A General Model of Concurrency and its Implementation as Many-core Dynamic RISC Processors. In *Proc. Intl. Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation, SAMOS-2008* (2008), pp. 1–9.
- [4] BOEHM, H.-J. Threads Cannot be Implemented as a Library. Tech. Rep. HPL-2004-209, HP Internet Systems and Storage Laboratory, Nov 2004.
- [5] BOUSIAS, K., GUANG, L., JESSHOPE, C., AND LANKAMP, M. Implementation and Evaluation of a Microthread Architecture. *Journal of Systems Architecture* 55, 3 (2009), 149–161.
- [6] CHAPMAN, B. M. The Multicore Programming Challenge. In *Advanced Parallel Processing Technologies* (2007), p. 3.
- [7] GABB, H., MATTSON, T., AND BRESHEARS, C. Thinking in Parallel - Three engineers' Viewpoints. *Intel Software Insight Magazine* 16 (Feb 2009), 24–26.

- [8] GEER, D. Industry Trends: Chip Makers Turn to Multicore Processors. *Computer* 38, 5 (2005), 11–13.
- [9] HERLIHY, M., AND MOSS, J. E. B. Transactional Memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News* 21, 2 (1993), 289–300.
- [10] IANNUCCI, R. A. Toward a dataflow/von Neumann hybrid architecture. *SIGARCH Comput. Archit. News* 16, 2 (1988), 131–140.
- [11] JESSHOPE, C., LANKAMP, M., AND ZHANG, L. The Implementation of an SVP Many-core Processor and the Evaluation of its Memory Architecture. *ACM SIGARCH Computer Architecture News* 37, 2 (2009), 38–45.
- [12] JESSHOPE, C. R. muTC - An Intermediate Language for Programming Chip Multiprocessors. In *Asia-Pacific Computer Systems Architecture Conference* (2006), pp. 147–160.
- [13] KASIM, H., MARCH, V., ZHANG, R., AND SEE, S. Survey on Parallel Programming Model. In *Network and Parallel Computing* (2008), vol. 5245/2008 of *LNCS*, Springer, pp. 266–275.
- [14] NIKHIL, R. S. Can dataflow subsume von Neumann computing? *SIGARCH Comput. Archit. News* 17, 3 (1989), 262–272.
- [15] NVIDIA. CUDA Compute Unified Device Architecture Programming Guide. Tech. rep., Santa Clara, CA, USA, 2008.
- [16] SWANSON, S., SCHWERIN, A., MERCALDI, M., PETERSEN, A., PUTNAM, A., MICHELSON, K., OSKIN, M., AND EGGERS, S. J. The WaveScalar Architecture. *ACM Trans. Comput. Syst.* 25, 2 (2007), 4.
- [17] VU, T. D., AND JESSHOPE, C. R. Formalizing SANE Virtual Processor in Thread Algebra. In *ICFEM* (2007), pp. 345–365.
- [18] ZHANG, L., AND JESSHOPE, C. R. On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores. In *Euro-Par Workshops* (2007), Bouge and et al., Eds., vol. 4854 of *LNCS*, Springer, pp. 38–48.