

Evaluation of a hardware implementation of the SVP concurrency model

M. Lankamp, T.A.M. Bernard, M. Hicks, C. Jesshope and L. Zhang

Abstract

SVP is a general concurrency model that has been implemented in the ISA of a multi-threaded core, both of which support dataflow synchronisation with imperative programming. This core is used as a building block to design systems-on-chip comprising many cores, either for general-purpose use or for specific applications. The major advantages of this implementation include asynchrony, i.e. the ability to tolerate long latency operations without impacting performance and the binary compatibility of programs when executed on an arbitrary number of cores. This paper describes the execution model, the processor design and its emulation in software as a many-core, general-purpose processor chip. This chip comprises a feasible 128 cores, a 1 MB on-chip COMA and two external DDR3-2400 memory channels. We present cycle-accurate simulation performance results for various key algorithmic and cryptographic kernels. The results show very good efficiency in terms of the utilisation of hardware despite the high-latency memory accesses and good scalability across relatively large clusters of cores.

1 Introduction

The computer industry currently faces a dilemma. On the one hand, the requirement to manage power dissipation on chip suggests the use of many simple cores rather than fewer, more complex and less efficient ones. On the other hand, the many-core approach requires a tool chain that supports massive explicit concurrency, which is both difficult to implement and error prone to use. There is precedence for the use of many cores in the embedded systems domain, where chips already exist that can issue in excess of one thousand instructions in every clock cycle [1]. However, these chips are focussed on a limited set of applications and hence effort can be targeted in finding and mapping the applications' concurrency.

For the commodity market such an approach is not feasible as the effort required is large and compounded by the many different applications required. Here, a more automated approach from the tool chain is necessary. In addition, there is a requirement for binary-code compatibility between different generations of processors. The current solution to this problem uses sequential binary code where concurrency is extracted dynamically through the out-of-order issue of instructions. To provide binary compatibility when exploiting explicit concurrency requires a standard kernel interface for the tool chain supporting concurrency creation and synchronisation, either in the form of an efficiently implemented API or, more radically, in the ISA of the processor itself. As different implementations of such an API are likely to have different overheads, the point at which it is viable to exploit concurrency (i.e. the thread grain size) will vary from one system to another. This makes it imperative that the tool chain extract binary code describing all available concurrency—not just the concurrency required for a specific target. Binary compatibility thus requires an abstract API, in which the available concurrency is defined and where any system will transform this code to a grain size suitable for the given target. This is achieved by providing a suitable sequential schedule for the fine-grain concurrency up to a suitable grain size.

The Sun UltraSPARC T1 (Niagara) processor [2] has 8 cores and 4 threads per core and is probably the most concurrent commercial chip available in this sector of the market and yet this does not provide such an API. The problem arises in trying to define a ubiquitous concurrent execution model to replace the sequential one. This is particularly difficult as there is no agreement on such a model. We suggest that such a model should have the following properties:

- it should capture maximal concurrency;
- it should be able to capture locality without specifying explicit communication;

- it should support asynchrony to allow for high latency in both on- and off-chip communication;
- the concurrency and synchronisation specified should always admit a well-defined sequential schedule;
- it should promote safe program composition, i.e. guaranteed deadlock freedom when two binary programs are combined.

This paper describes work based on a concurrent execution model called the *Self-adaptive Virtual Processor* (SVP).

The results presented in this paper evaluate an implementation of SVP in the instruction set of a conventional RISC ISA. The focus here is on the architecture, although a significant effort is also being expended on the tool chain for this architecture, including compilers based on GCC for a language called μ TC [3] that captures the SVP model and higher-level compilers from C [4] and SAC [5] that target this core language and provide the automation of concurrency capturing referred to above. The results presented are generated using a semantically verified software emulation of a *Microgrid* of cores that execute Alpha binary code with SVP extensions. We show that this provides full binary compatibility across an arbitrary number of cores. We also show that it can provide backward compatibility with sequential binaries for the base ISA without the SVP instructions. Moreover, a significant effort has been made to ensure that the simulated execution time is cycle accurate by considering all transactions at every stage in the pipeline and relating them to their silicon implementation. Using this tool and the tool chain from the language μ TC, we present results that show the performance of this architecture on a variety of benchmarks.

2 Related Work

Perhaps the most relevant work related to that presented in this paper was performed some 20 years ago at MIT, where Arvind’s group investigated the fusion of dataflow and von Neuman computing [6, 7]. Both these papers explore the benefit of providing dataflow scheduling over sequences of RISC-like instructions. However, with the development of an efficient matching mechanism using the Explicit Token Store (ETS) proposed by Papadopoulos, development at MIT veered back to mainstream dataflow in the Monsoon architecture [8].

Currently, perhaps the most explored abstract model that satisfies many of the properties described above is transactional memory [9], which has a clear advantage in certain applications. Those are where dependencies in a program are dynamically defined and where there is non-determinism in the updates made by transactions. Such applications however, are in the minority and the majority can be expressed with either statically or dynamically defined dependencies. Hence, the dataflow model provides another potential solution to this problem. Here, dependencies are captured in the binary code rather than by sequence, which provides very fine grain concurrency that can be implemented in the ISA of the processor. However, dataflow does not support the imperative form of programming well as it deals with values and not stored variables. WaveScalar [10] overcomes this using an implicit program counter on memory ordering to support compilation from imperative languages. Each memory operation identifies a predecessor and a successor as well as a memory no-op for balancing conditional operations. However, this scheme rather restricts the amount of concurrency exposed and requires the explicit notion of threads to overcome this restriction. In WaveScalar, a thread is defined as an independent ordering on memory references. Another problem with dataflow (and all data-driven models) is the potential for resource deadlock, when the concurrency exposed exceeds the amount of hardware resources available to describe that concurrency, e.g. matching store in dataflow.

3 The SVP Model

The SVP execution model [11] was developed to capture the requirements described for a generic concurrency model that could become as ubiquitous as the sequential one we have used for decades. It captures threads hierarchically down to the lowest level of loops and function bodies. This section gives a definition of the various components of the SVP model in its most generic form.

3.1 Threads and Families

An SVP *family* is a parameterized group of statically homogeneous, but dynamically heterogeneous threads that are created en masse. Every thread can create families of its own, making the model hierarchical. A family is characterized by the index sequence of the threads to be created, a reference to the thread body and a definition of uni-

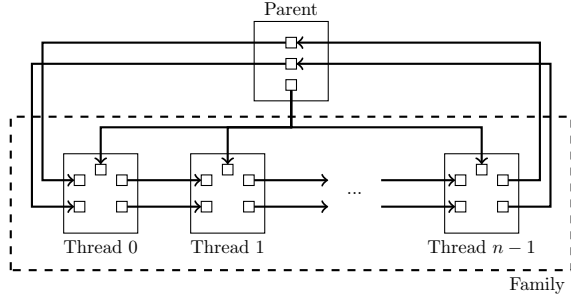


Figure 1: Unidirectional synchronization channels between threads. Shown here is one global and two shared channels.

directional synchronization channels from, to and within the family. Using parameterized families avoids the state explosion and creation overhead for large numbers of individual threads. Note that in low-resource conditions a family can be executed entirely in the parent thread’s context; the parent thread is executed up to the point of synchronization after which the child threads can then be executed in index order (to satisfy communication dependencies), followed by the remainder of the parent thread.

3.2 Places

SVP provides a separation of concerns between the definition of a program’s function and its resource mapping. The latter is achieved by binding a named resource, called a *place*, to a family on its creation, which can happen at any level in the family hierarchy. Depending on the implementation, a place can be a virtual resource or a physical one and can have various properties, allowing threads with specialized code to be run on heterogenous systems via a common mechanism. It has been shown [12] that the default model is free of deadlock under composition. Regardless of the implementation, there is a special class of place that must be supported, the *exclusive place*. This class has the property that of all families created at that place, only one will run at a time. Exclusive places provide synchronisation between unrelated threads and add non-determinism to the execution model. Typically they will be used by system code, e.g. in sharing or allocating resources.

3.3 Synchronization

Communication and synchronization between threads does not happen via memory, but through special hardware-supported channels called *globals* and *shares*. These are unidirectional communication channels between the parent thread and the threads in a family using blocking reads and non-blocking writes. Global channels are written once by the parent thread and are available for reading to each thread in the family. Shared channels are defined between every consecutive pair of threads in the family. A thread can write a value to its outgoing shared channel once, which can only be read by the next thread’s corresponding incoming shared channel. The first thread’s incoming shared channel and the last thread’s outgoing channel are connected to a variable in the parent. If a compiler targeting SVP wishes to capture dependent loops concurrently, then it must transform any loops with dependencies to meet these constraints [4]. Figure 1 shows these channels and how they connect between the threads in a family and their parent.

3.4 Memory

The memory model is conceptually a single, flat address space with a restricted consistency model, allowing greater freedom for the implementation. Informally, the model defines memory consistency as follows: at any time, each thread has a consistent view on subsection of memory, such that reads and writes to this view are well defined as long as that thread is the only one writing to that location between synchronization events. The consistency view is shared between threads on the synchronization events: create, sync, reading/writing global/shared channels and creating on an exclusive place. This means, specifically, that:

- for references passed through a global channel, threads in a family can only see and use what the parent thread could see at the point of the create, as long as the parent does not change the memory before the family terminates.
- the parent thread cannot see the changes made by the child threads until the it has synchronized on the family’s termination;
- a thread cannot see the writes made by a previous thread in the family except for writes to objects which are sent via shared channels.
- in order to share data between unrelated threads, all threads must create a family on the

```

void thread work(shared int f1,shared int f2)
{
    int n = f1 + f2;
    f2 = f1;
    f1 = n;
}

void thread fibo(int n, shared int result)
{
    family f;
    create(f;2;n;1;;) work(f1 = 1, f2 = 1);
    sync(f);
    result = f1;
}

void thread main()
{
    int result;
    family f;
    create(f;;;;) fibo(10, result);
    sync(f);
    /* result now contains the
       10th fibonacci number */
}

```

Listing 1: μ TC code for fibonacci

same exclusive place to synchronize consistent access to a shared location.

Specifically, this consistency model makes no guarantees for a thread seeing writes made by an unrelated thread at some point in time. This absolves the memory implementation from ensuring global consistency, allowing more experimentation and optimization with implementations.

3.5 Microthreaded C

The programming model for the SVP model is Microthread C, or μ TC [3]. It is ISO C99 with extensions added and certain constructs stripped out due to inherent incompatibility with the threaded model. Listing 1 shows an example of μ TC code calculating the 10th fibonacci number. New keywords are the **thread** function specifier to identify functions as thread bodies, the **shared** type specifier to identify thread arguments as shared (all non-shared arguments are globals), the **create** construct to create a family of microthreads and **sync** to synchronize (“join”) with a created family. **create** takes (among others) a start, step and limit value, a reference to the thread body and a list of arguments.

This language has been implemented in a compiler [13] which targets the hardware implementation described in this paper.

4 Microgrid

A *microgrid* is a chip-multiprocessor using *microthreaded* processors that implement the SVP model directly in hardware [14]. This section explains both the chip architecture and the processor architecture, which is based on a DRISC processor [15] and executes an extended standard instruction set.

4.1 Processor Overview

A microthreaded processor uses an in-order issue pipeline with both in-order and out-of-order completion of instructions. The core principle of the processor is to avoid stalls and speculative execution, in order to maximize utilization of the pipeline and energy efficiency, respectively. Cache misses, family and thread creation, synchronization and FPU operations are all implemented by issuing the operation without keeping track of outstanding dependencies in the pipeline, and continuing to execute instructions from the same thread. Any instruction that attempts to use the result of an operation which has not yet completed has its thread suspended on the target register and is woken up when it is written. The register file is modified by adding state bits to each register and logic for handling these states on reads and writes.

This ability to execute many threads in a single pipeline without stalling gives a microthreaded processor the ability to hide large amounts of latency. To support this number of threads, the processor has data structures to store and manage the families and threads and a relatively large register file to hold all of the threads’ contexts. Dynamic allocation of register contexts to families allows the thread context size to be tailored to its code, optimizing register file usage.

4.2 Family and thread management

When a thread creates a family, an entry in the *Family Table* is allocated to store the family’s state. This state includes the number of threads created so far for the family, parameters for creating new threads and a link to the parent thread. An independent hardware process is responsible for creating the threads of created families, up to a program-specified upper bound. All non-register state for the threads created on a processor is stored in the *Thread Table*. Threads start as soon as they are created and terminate after executing any instruction annotated by the special END annotation (see

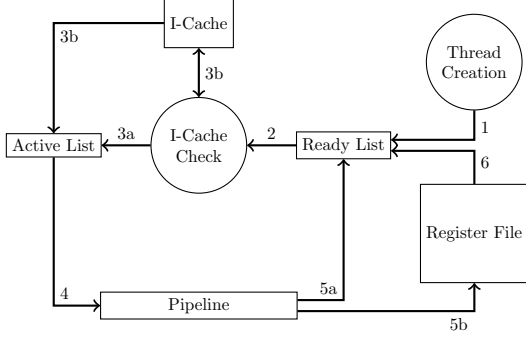


Figure 2: State of a thread during its lifetime. After being created, it goes onto the Ready List (1), where it eventually gets selected for its I-Cache check (2). It proceeds onto the Active List after a hit (3a) or indirectly after the cache-line is read (3b), after which it is eventually selected by the pipeline (4). After a thread switch, it either goes back onto the Ready List immediately (5a), or suspends on a register (5b) and goes onto the list when the register is written (6).

section 4.4). When a thread terminates its context is reused for the next thread in the family unless all threads in the family have been created. All resources related to a family are released when all threads in that family have terminated and no more references to the family exist.

Threads can be in different states besides *running* and threads in a one state may need to be moved to another state en masse. To do this efficiently, these states are managed with a linked list with a field in the Thread Table, such that moving threads from one state just means appending one linked list to another. This mechanism is used in the following cases:

- When a register is written to that has several threads waiting on it, these threads are woken up by taking the head and tail pointer from the register and appending that list to the processor’s list of threads that should be executed, the *Ready List*.
- Threads that are waiting on an I-Cache line for their instructions are linked in a list with the head pointer in the cache-line entry itself. This allows the I-Cache, when the line has been read, to immediately make all threads that were waiting on it available to the pipeline via the *Active List*.
- All allocated threads are also maintained in a

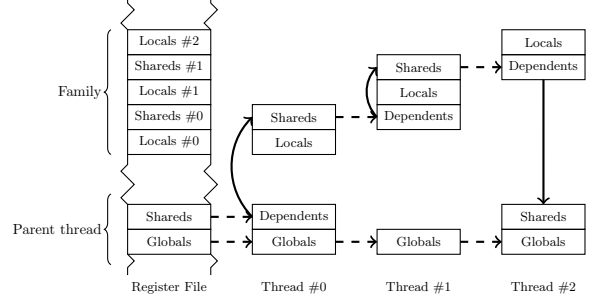


Figure 3: Physical register mapping with synchronization flow. On the left is the register file with the registers for the parent thread and the family it created. On the right is the mapping to the contexts for the three threads in the family. The dependents, shareds and globals of certain threads map physically onto the same registers.

per-family membership list (requiring a second link field per thread). This list allows the processor to append all threads of a family, regardless of their state, to the empty list when the family is forcibly terminated.

Figure 2 shows the high-level flow of threads during their lifetime as they move between pipeline, register file and I-Cache.

4.3 Registers

Each processor has a relatively large register file to allow many concurrent threads to have their own context of registers, which can (and if possible, should) use fewer registers than the maximum of the ISA. Each register has state bits that allow threads to suspend on empty registers. This mechanism is used to tolerate instruction latencies by suspending threads on the results of e.g., memory loads and family creation, and to implement the *shared* and *global* synchronisation channels. Certain registers from one thread are physically mapped, by simple register address translation in the pipeline, onto certain registers of another thread. The consumer thread can then suspend on a channel simply by reading its register, and will be woken up once the producer writes to it. The contents of a register is interpreted differently depending on its state. Non-Full registers contain, among others, a list of threads suspended on that register and information about the memory load for that register.

Figure 3 shows how registers are mapped between threads. Each thread has a virtual window visible

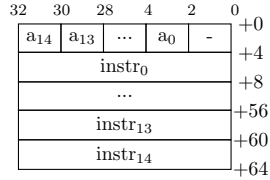


Figure 4: Packing of instruction annotations. The first word in a cache-line contains the 2-bit annotations for all instructions in that cache-line.

to the ISA with *Global*, *Shared*, *Local* and *Dependent* registers which are mapped in various ways to the physical registers in the register file.

4.4 Pipeline

The pipeline in an SVP core is a simple in-order issue RISC pipeline with modifications. When the Read stage determines that one or more of the operands of an instruction are not full, it will change the instruction to a no-op that writes the thread's ID back to that register and marks it *Waiting*. The Fetch stage of the pipeline uses the processor's Active List to get the next thread to execute from. It will read the Thread Table, Family Table and I-Cache and buffer the information while it executes that thread. A switch to another thread is required when:

- the current thread reaches the end of the cache-line,
- the thread executes a jump or branch,
- an instruction in the thread uses a register that is not *Full*, or
- an instruction has been annotated with a *SWCH* or *END*.

In the first two cases, the thread proceeds to a cache-line that may not be present in the I-Cache and to avoid stalling the pipeline, execution switches to the next thread on the Active List. At the end of the pipeline, the thread is put onto the Ready List to fetch its new cache-line. In the third case, the thread needs data that is not present and cannot continue. The thread will suspend in the register and be put on the Ready List when the register is written.

Instruction annotations

The last case requires a more detailed explanation. The second and third cases cannot be identified un-

til the instruction has passed the Decode, Read or Execute stage, depending on the situation, after which the pipeline is flushed and another thread is selected to be run. To avoid bubbles in the pipeline because of the flush, the compiler annotates instructions that could cause a flush with a *SWCH*, which instructs the Fetch stage to switch to another thread. Should the tagged instruction cause one of the thread-switch scenarios, the thread switch has already occurred so the pipeline does not have to be flushed.

To preserve instruction alignment and compatibility with the base ISA, the *SWCH* bits for all instructions in a cache-line are packed into the first instruction word in that cache-line, which has to be skipped as an instruction during execution. Figure 4 shows the layout of a cache-line for an architecture with 32-bit instructions and 64-bit I-Cache lines, with the annotations packed into the the first instruction word. There are in total three annotations: *CONTINUE* (implicit), *SWCH* and *END*. Any instruction annotated with the latter terminates the thread after being successfully executed. As a general rule, *SWCH* tags should be placed on any instruction that uses the result of a long-latency operation (e.g. memory loads) or that change the PC of the thread (e.g. branches). Note that an explicit thread switch is not necessary for the end-of-cache-line scenario, since the Fetch stage can trivially detect it.

4.5 Caches

The instruction and data cache in a microthreaded processor are smaller than in conventional processors, due to the architecture's ability to tolerate latency, and are extended with additional fields in each cache-line to allow for decoupled memory accesses.

In the Instruction Cache, each line has a pointer into the Thread Table to support a linked list of threads waiting on that line. Whenever a thread needs to be run, the I-cache is checked to see if the thread's instruction-line is present. On a miss, a line is selected for replacement, reinitialized and the thread is put into the linked list for that line. Should the line already be present, but is still being fetched from memory, the thread is appended to the linked list for that line. When the cache-line has been read from memory, the processor can append the entire linked list for that line to the Active List, from which the pipeline will select threads to run. A per-cache line reference counter prevents cache-lines from being evicted while being needed by threads in the Active List.

The Data Cache supports decoupled memory operations by using a linked list in the Register File for each cache-line. When a read from memory into a register misses the cache, the read information (e.g., number of bytes and offset in the cache-line) is stored in the target register, and the register is appended to the linked list for that cache-line. When the cache-line has been fetched from memory, the processor will (over multiple cycles) iterate over the list of registers of the cache-line and finalize those reads.

4.6 Memory Interface

Since all memory operations are decoupled, memory requests are tagged with a value that identifies the destination of the response, which must contain the same tag. The tag is two bits identifying the request type (I-Cache read, D-Cache read and Write) and an index into the appropriate cache. This tag system allows the processor to properly handle the response when it is received and relaxes the interface to memory; responses may arrive in any order, as long as the correct tag is attached to the response.

To provide memory consistency on family creation and termination, each processor maintains a counter of outstanding reads and writes for threads and families, which requires writes to be acknowledged by the memory system.

5 Evaluation

To evaluate the architecture described in this paper, we ran several key algorithmic and cryptographic kernels written according to the SVP model on a simulated microgrid system. The software simulator keeps track of the full state of the system, simulates the pipeline, caches, network and memory and takes contention on those buffers and ports into account to provide an extremely cycle-accurate simulation. The simulated system consists of 128 1.6 GHz cores spread among several differently-sized places. Each core has a 4-way set associative 1 kB I-Cache and D-Cache, storage for 32 families, 256 threads and 1024 integer and 512 FP registers. A 4-way set associative 32 kB L2 cache serves 4 cores via a bus and a COMA directory is connected with 8 L2 caches in a ring. Two DDR3-2400 channels are connected to the top-level directory ring.

The simulated algorithms are the inherently sequential *inner product* and *prefix sum* on floating point arrays and nine key cryptographic kernels, consisting of hashes and block and stream ciphers.

In all cases, the input data is located in the off-chip memory and program setup time (such as allocations and creating the key schedule) was not measured.

5.1 Inner Product

The inner product on arrays a and b returns $\sum_{i=1}^N a_i b_i$ and is representative for the general *reduction* operation used in many algorithms. This algorithm has been implemented in two ways. The *linear* version is a naive implementation consisting of a create of N threads with a single dependency that carries the sum. The *parallel* version does several nested creates. The first create creates a thread per core in the place. Each of those threads performs another two nested create to do c reductions locally on that core (c is a constant). This results in cp independent reductions, after which the result is reduces over the p cores by the top-level family. Figure 5 shows the results.

The results show that the linear version, although multithreaded, is constrained by the single dependency. Although the memory loads for all threads are dispatched before the dependency is required, the pipeline and communication latency mean that executing it on multiple cores yields negligible benefit. The parallel version, however, can utilize all cores due to every core doing several independent reductions. Both performance and utilization of the hardware is significantly higher. However, since the main threads are small with about two loads, a multiply and an addition, the memory latency causes the utilization to be relatively low overall.

The difference between warm and cold caches is insignificant, which is not surprising considering that with 32 cores, there is only 32 kB of L1 cache, and the linear access pattern of the warm-up results in the L1 caches containing the second half of the array, which is evicted for the first half in the timed run. The only caching benefits occur at the L2 caches.

5.2 Prefix Sum

The prefix sum on array a returns array b such that $b_i = \sum_{j=1}^i a_j$, i.e. the sum of all preceding elements. We chose the prefix sum because it can be used to express many algorithms [16]. Again, two implementations were tested. The *linear* version is again a single create with a dependency that carries the sum. The *parallel* version has two levels of creates to perform $\log N$ steps of $N/2$ independent operations. While the total number of performed

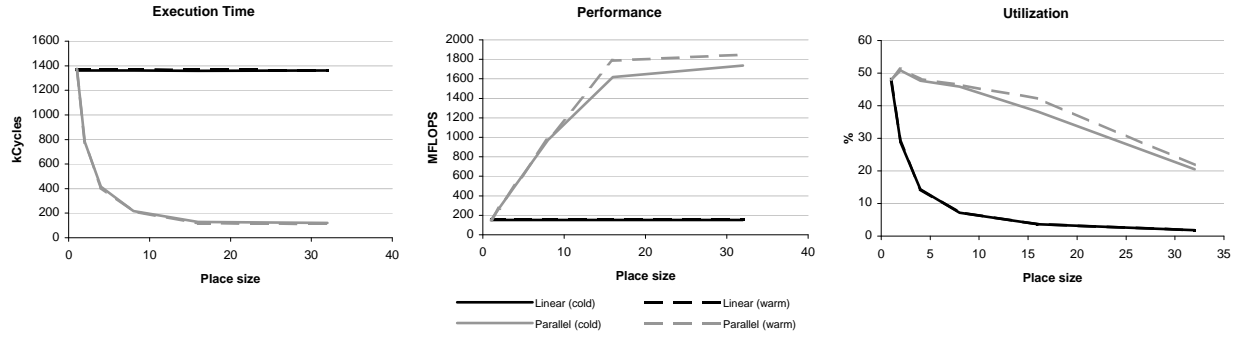


Figure 5: Results of an inner product on a 64k element floating-point array on places of 1–32 cores. Utilization is defined as the theoretical best time-to-result ($\frac{\#executed\ instructions}{\#cores}$) divided by actual time-to-result.

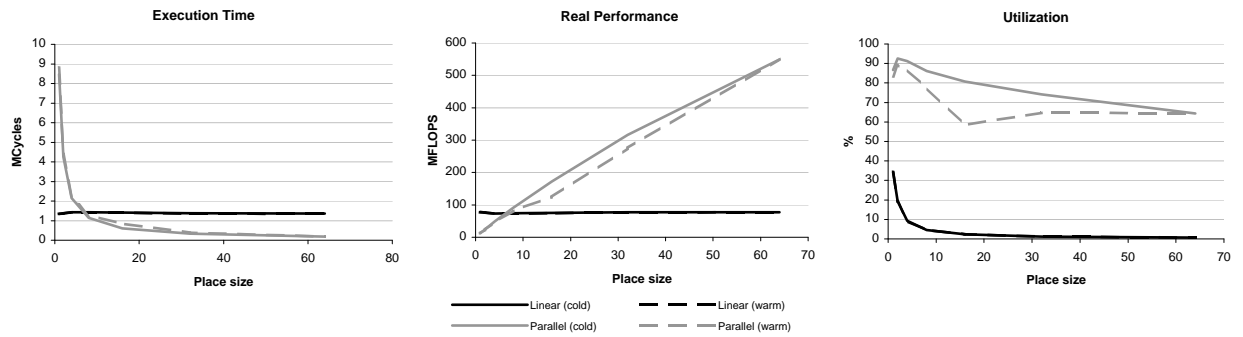


Figure 6: Results of a floating-point prefix sum on a 64k element array on places of 1–64 cores. Real performance is based on the FLOPs for the linear version. Utilization is defined as the theoretical best time-to-result ($\frac{\#executed\ instructions}{\#cores}$) divided by actual time-to-result.

Name	Block Size (bits)	Rounds	Key Size (bytes)	Table Size (bytes)
AES	128	10	16	8192
DES	64	16	8	4096
RC5	64	16	16	0
RC6	128	20	16	0
IDEA	64	9	16	0
RC4	-	-	16	256
SEAL	-	-	20	< 4096
MD5	512	64	-	0
SHA-1	512	80	-	0

Table 1: The cryptography kernels and their parameters

operations is greater, more work can be done in parallel. Both versions work in-place on the input array. Figure 6 shows the results.

As with the inner product, the linear version of the prefix sum has no significant performance benefit from running on multiple cores due to being constrained by the dependency. The parallel version scales much better, but care has to be taken to realize that it performs more work in the base case. As such, its performance, which is calculated based on the number of operations from the linear version, is worse on 8 cores and less, which is not surprising considering that it performs 8 times more operations for a 64k element array. Besides scaling well up to at least 64 cores, the utilization of the hardware is significantly better, as it is at worst only around 66% slower (60% efficiency) than the best possible time-to-result. A high utilization implies that memory latencies are effectively hidden.

More interestingly, running the prefix sum with a warm cache makes it perform slightly (although not significantly) worse. This is most likely due to the non-linear access pattern of memory in the different steps of the algorithm during warm-up, causing memory to be moved to caches far from where it is actually needed in the first timed step.

5.3 Cryptography

Two hashes, two stream ciphers and five block ciphers in CBC mode were ran on several parallel streams to evaluate the throughput of the system. Table 1 lists the kernels that were timed, and their parameters. Sixteen places of p cores were used to process 1–32 independent streams. The runs were repeated for $p = 1, 2, 4$ and 8 , to test how the ability to distribute a stream among more cores affected the performance. Note that the same binary code was run across all configurations.

The performance of the cryptographic kernels on

the microgrid has been compared against the kernels running on the Intel IXP 2800 Network Processor (NP) [17]. The IXP 2800 has 16 32-bit MicroEngines running at 1.4 GHz and dedicated hardware hash units. For this comparison, the microgrid cores have also been scaled to 1.4 GHz. Figure 7 shows the resulting throughput on the microgrid in comparison with the IXP 2800’s. In general, the microgrid performs either similarly or significantly better. However, the code for the IXP 2800 has many specific optimizations such as inline assembly, manually threading specific regions of rounds and tweaking the burst performance of the memory system. This code, which runs on the MicroEngines, has to be compiled separately from the control code around it. In contrast, all parts of the SVP version run on the same kind of processors in the microgrid. The code is written generically without inline assembly or splitting statements into threads. Threads are defined for streams, blocks in streams and rounds in blocks where the shared variables carry the hash, previous block value (for CBC ciphers) or state (for stream ciphers).

There are four cases where the microgrid performs worse. With RC4 the modified state must be made consistent in memory before the next thread can proceed, which highly sequentializes execution. For SEAL, the next block depends on several memory loads that are made late in the current block, reducing the latency hiding benefits of the architecture. Regardless, they still perform comparably and SEAL might be improved with specific optimizations such as reordering memory loads. MD5 and SHA-1 perform worse although the difference is reduced with higher number of cores/place. But, since the IXP 2800 has a dedicated hardware hash unit, it is no surprise that it performs better.

6 Conclusion

In this paper we presented the SVP model, a hierarchical, iterative programming model with constrained dataflow synchronization and its implementation in hardware as a microgrid of multi-threaded processors. The latter efficiently supports the features of the model in hardware with dedicated instructions for mass thread creation without sacrificing binary compatibility. Dataflow communication occurs through the register file with dynamically allocated register contexts. Sequentialization on resource exhaustion is trivial due to the uni-directional communication channels between threads.

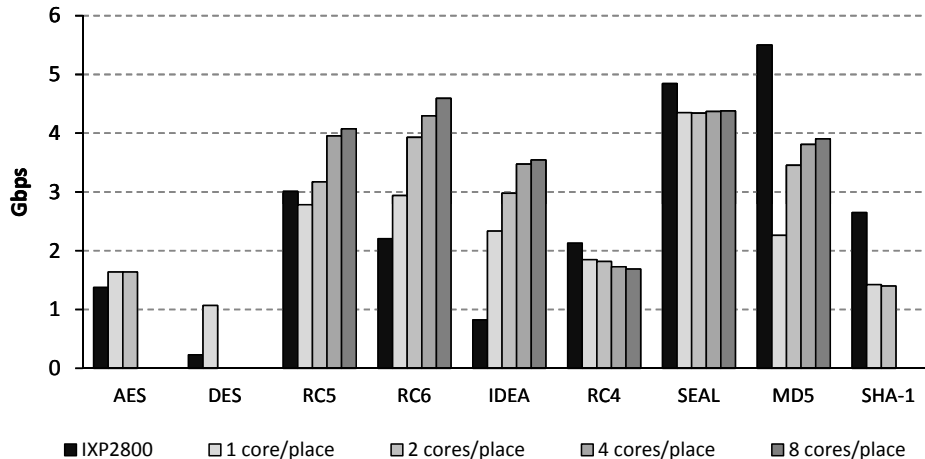


Figure 7: Maximum throughput of the various cryptographic algorithms on 1–32 streams of the IXP 2800 and the microgrid with 16 places, with 1–8 cores/places.

We ran key algorithmic and cryptographic kernels to illustrate the potential of the model and implementation. The two inherently sequential algorithms are still able to be parallalized and achieve respectable performance and utilization of the hardware. The cryptographic kernels running on the general-purpose grid are at least comparable with highly optimized and specialized code running on dedicated commercial hardware, and at best perform significantly better. Taking into account that the SVP code requires the programmer to only express the algorithmic concurrency to achieve this similar performance, we believe the SVP programming model and its microgrid implementation offer a viable path into future manycore designs.

References

- [1] G. Panesar, D. Towner, A. Duller, A. Gray, and W. Robbins. Deterministic parallel processing. *International Journal of Parallel Programming*, 34(4):323–341, 2006.
- [2] H. McGhan. Niagara 2 opens the floodgates. *Microprocessor Report*, pages 1–9, 2006.
- [3] Thomas A. M. Bernard, C. R. Jesshope, and P. M. W. Knijnenburg. Strategies for Compiling μ TC to Novel Chip Multiprocessors. In S. Vassiliadis et al., editors, *SAMOS*, pages 127–138, 2007.
- [4] D. Saougkos, D. Evgenidou, and G. Manis. Specifying Loop Transformations for C2 μ TC Source to Source Compiler. In *Proc. Compilers for Parallel Computers*, 2009.
- [5] C. Grelck, S. Herhut, C. R. Jesshope, C. Joslin, M. Lankamp, S.-B. Scholz, and A. Shafarenko. Compiling the Functional Data-Parallel Language SAC for Self-Adaptive Virtual Processors. In *14th Workshop on Compilers for Parallel Computing CPC’09*, IBM Research Center, Zurich, Switzerland, 2009.
- [6] R. S. Nikhil. Can dataflow subsume von neumann computing? *SIGARCH Comput. Archit. News*, 17(3):262–272, 1989.
- [7] R. A. Iannucci. Toward a dataflow/von neumann hybrid architecture. *SIGARCH Comput. Archit. News*, 16(2):131–140, 1988.
- [8] G. M. Papadopoulos. Monsoon: an explicit token-store architecture. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, pages 82–91, 1990.
- [9] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.
- [10] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The WaveScalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4, 2007.
- [11] C. R. Jesshope. A model for the design and programming of multi-cores. *Advances in Par-*

- allel Computing*, High Performance Computing and Grids in Action(16):37–55, 2008.
- [12] T. D. Vu and C. R. Jesshope. Formalizing SANE Virtual Processor in thread algebra. In *ICFEM*, pages 345–365, 2007.
- [13] T. Bernard, K. Bousias, B. de Geus, M. Lankamp, L. Zhang, A. D. Pimentel, P.M.W. Knijnenburg, and C.R. Jesshope. A Microthreaded Architecture and its Compiler. In *Proc. of the Int. Workshop on Compilers for Parallel Computers*, 2006.
- [14] K. Bousias, L. Guang, C. R. Jesshope, and M. Lankamp. Implementation and evaluation of a microthread architecture. *J. Syst. Archit.*, 55(3):149–161, 2009.
- [15] A. Bolychevsky, C. R. Jesshope, and V. B. Muchnick. Dynamic scheduling in RISC architectures. In *IEE Proceedings Computers and Digital Techniques*, volume 143, pages 309–317, 1996.
- [16] G. E. Blelloch. Prefix Sums and Their Applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.
- [17] Y. Yue, C. Lin, and Z. Tan. NPCryptBench: a cryptographic benchmark suite for network processors. *SIGARCH Comput. Archit. News*, 34(1):49–56, 2006.