

Implementation and Evaluation of a Microthread Architecture

K. Bousias, L. Guang, C.R. Jesshope, M. Lankamp

*Informatics Institute, University of Amsterdam, Kruislaan 403, 1098 SJ
Amsterdam, The Netherlands*

Abstract

Future many-core processor systems require scalable solutions that conventional architectures currently do not provide. This paper presents a novel architecture that demonstrates the required scalability. It is based on a model of computation developed in the AETHER project to provide a safe and composable approach to concurrent programming. The model supports a dynamic approach to concurrency that enables self-adaptivity in any environment so the model is quite general. It is implemented here in the instruction set of a dynamically scheduled RISC processor and many such processors form a microgrid. Binary compatibility over arbitrary clusters of such processors and an inherent scalability in both area and performance with concurrency exploited make this a very promising development for the era of many-core chips. This paper introduces the model, the processor and chip architecture and its emulation on a range of computational kernels. It also estimates the area of the structures required to support this model in silicon.

Key words: architecture, evaluation, implementation, microgrid, microthread, SVP

1 Introduction

Although frequency increases from Moore's law have hit the power wall, functional density is expected to grow exponentially for at least the foreseeable future [4, 6]. Existing processor architectures can not really exploit this functional density as the methods currently used to achieve speedup do not scale well and must deal with increased power dissipation and circuit complexity

Email address: (bousias,liang,jesshope,mlankamp)@science.uva.nl (K. Bousias, L. Guang, C.R. Jesshope, M. Lankamp).

as a result. Many-Core Chip Multiprocessors (MCCMPs) appear to be the only way to utilise this massive amount of on-chip resources in the future, yet issues concerning the programming model of these systems, the architecture and interconnection of the cores are often neglected or hinder the takeup of this approach. This is particularly true in commodity processor systems, where users have come to expect binary compatibility across a range of generations of processor implementations and moreover to program them using safe and composable programming models. This is a non-trivial problem when using an explicitly concurrent architecture.

The first issue in solving this situation is to adopt a model of concurrency that has similar characteristics to the sequential model, which is comparatively easy to understand and program in, is deadlock-free and provides deterministic results. There is a case for continuing to program in a sequential model and this has been the holy grail of the parallel computing community for decades. However, to do so without first adopting a concurrency model that reflects its attributes has been shown, by the lack of success to date, as a doomed exercise. Several coherent models of concurrent programming (together with many ad-hoc approaches) have been proposed, e.g. the CSP-based occam, now finding a reincarnation as XC (www.xmos.com), Pthreads, and Transactional Memory (TM) [11]. However, the requirements above do not seem to be met by any of these models. The first issue is that the model should be abstract and able to capture all parallelism available in a program, not just the characteristics of an implementation. Pthread programming falls at this first hurdle, as programming in this model must match the granularity constraints of the implementation, which may take tens of thousands of cycles to create a thread or synchronise between threads. Secondly, the programming model should be composable, namely if two correct programs are composed in any way, the result should be correct. Occam falls at this hurdle as it is not free of deadlock under composition (although constraints on the model can provide this). Lastly the model should be conservative, i.e. it should execute only those operations that contribute to a result, speculative threading and models such as TM fall at this hurdle. TM aims to provide a lock-free programming model and does this by dividing programs into parallel transactions but ignoring dependencies between them. The implementation then detects dependencies dynamically when they commit. The result is that any conflicts will cause one or more transactions to roll back and be re-executed. Although TM may provide a simple model for programmers to obtain correct programs, the same can not be said for the efficiency of these parallel programs.

The work described here adopts a model developed within the AETHER project [1] that has all of the required characteristics defined above: it is abstract, composable and conservative in its execution. The SVP model, captured in the language μ TC [12], is an abstract model based on the concurrent composition of *families* of blocking threads. It is abstract, as even in the pro-

cessor's ISA this model captures program concurrency and is implemented subject to resource constraints; families can be of any size (even infinite) and threads are created only when resources are available. The model has to be restricted in terms of how threads communicate in order to bound the resources required to span a family. In fact, the limit in all cases is a requirement for the resources to execute a single thread per family, i.e. the model falls back, when constrained by resources to the sequential execution model. All created threads run concurrently on one or more processors and the family provides a unit of work, which can be used to synchronise with memory, as memory is shared and bulk synchronised in this model. Synchronisation between threads uses a thread's context of register variables, which implement dataflow i-stores and blocking reads. These enable the scheduling of only active threads' instructions into a pipeline.

Implementing a thread-based model in hardware is not new. For example, the Niagara II [14] and X-core (www.xmos.com) both support 8 threads per core in multi-core chips. Both implement fine-grained scheduling of threads, the former based on access to shared memory and the latter on input from channels. In both cases however, threads do not provide a programming model but a resource that acts as a place holder for some other model. Our motivation is to show that the abstract SVP model can be implemented in a processor's ISA, so that the translation from sequential code into the SVP model can be automated and provide efficient code for a large range of applications. In implementing this the proposals in [6] have been met; the cores support a coherent and abstract programming model that is implemented in a simple and efficient way so as to provide scalable performance and power efficiency. The core and chip architecture are discussed in more detail in section 4, and emulation results and core-area estimations are presented. We show that when running various kernels, SVP MCCMPs always provide linear speedup to the limit of concurrency in the kernel and moreover do so with a conservative use of power. Area estimation demonstrates the scalability of the architecture in both number of processors (for throughput) and number of threads per core (for tolerance to latency). A comparison of the mainstream Itanium 2 processor against a cluster of SVP processors of similar area reveals the performance edge of the latter over the former.

2 Related Work

Multithreading has been widely used to provide virtual parallelism; it is often used combined with multiple pipelines and multi-cores to offer physical parallelism. *Niagara II* [14] is an example of fine-grained temporal multi-threading combined with a multi-core implementation. Each processor has eight cores, each supporting eight logical threads; every four threads execute on a sin-

gle pipeline, so there are two pipelines on a single core. The threads in each core are issued in an interleaved manner with thread switching possible on each cycle. When one thread stalls for some reason, it is not issued until it is reactivated. Other multi-threading architectures adopt different mechanisms; for instance, the *Power5* [15] processor is an example of simultaneous multithreading (SMT). The Power5 supports two threads per core and contains two cores on a single die. Instead of interleaving the threads as in the Niagara II processor, both threads in each Power 5 core are issued and executed simultaneously. *Cell* [13] provides high performance by incorporating one PPE (Power processor element, a power architecture based, two-way multithreaded core) on a single chip, acting as the controller for the eight SPEs (Synergistic processing element) which handle most of the computational workload. The latest release of *Tilera* (www.tilera.com) has 64 cores on one die, where each core has a 3-way VLIW pipeline and is able to work independently of the others. On all these processors thread support is very limited.

The SVP model is based on more than 10 years of research, starting with the dynamic RISC processor [7] which outlined a processor supporting a large number of threads, called microthreads, to achieve high memory latency tolerance. It did this using programmed interleaving of those microthreads and demonstrated that by supporting a small number of threads the processor already provides high latency tolerance. Based on this initial work, the Microthread execution model was developed, which was refined and generalised into the SVP model.

There has been an effort to make thread scheduling dynamic and efficient in conventional processors which has resulted in the *DDM-CMP* architecture [17]. This incorporates data-flow semantics into conventional multi-threading by issuing ready-to-execute threads only. The scheduling of threads is managed by a separate thread synchronisation unit which keeps track of the availability of threads and maintains a ready thread queue. This technique can be applied to any core but is more coarse grain than the implementation of SVP described here.

3 The SVP model

The Self-adaptive Virtual Processor (SVP) is a programming model based on the concurrent and hierarchical composition of homogeneous families of blocking threads. The novel aspect that supports self-adaptation is that the model abstracts the concept of resource (*place*) and the remote execution of units of work (a sub-tree in the program hierarchy) on a specified resource (*delegation*). The SVP model defines all concurrency and communication in abstract terms and all mapping and scheduling of the threads created is transparent

```

thread void simple (float * a, float * b, float * c) {
    index i;
    c[i]=a[i]*a[i]+b[i]*b[i];
}
thread void main() {
    float a[100],b[100],c[100];
    family fid; place pl;
    ...
    create(fid;pl;0;99;1;;) simple (a,b,c);
    sync(fid);
    ...
}

```

Figure 1. Concurrent composition in the SVP model illustrated in μ TC

to the programmer. SVP also defines mechanisms for the dynamic control of concurrently executing programs and its implementation may be considered as an operating system kernel. In this paper we consider the implementation of that model in the ISA of a regular RISC processor. Thus, SVP implemented at this level is an operating system in silicon.

3.1 Families of blocking threads

SVP programs are based on blocking threads and use only concurrent composition. Any thread may read and write an asynchronous shared memory. A thread may create a subordinate *family* of threads but changes to shared memory from that family may only be read consistently by the parent thread once all threads in the family have terminated (i.e. on *sync*). The threads in a family are created from a single thread definition that contains an *index*, which is automatically initialised on thread creation to a value in the range that defines the extent of the family.

The example in figure 1 illustrates this; it creates 100 threads to compute in *c*, the sum of the squares of the elements of arrays *a* and *b*, where each thread computes one index value and all of this is executed at a remote place (*pl*). The parameters of the create action specify a family identifier (*fid*), which is set to uniquely identify the family and a triple defining the start, limit and step of the index set. In this example, create replaces an independent loop in the equivalent C code. In translating C to SVP, all function calls would also be replaced with a create/sync pair, although no concurrency would be exposed unless the parent thread performs work between create and sync. Programs in the model are therefore a dynamically evolving *concurrency tree*, in which sub-trees may be delegated (again dynamically) to processing resources. This is illustrated in figure 2.

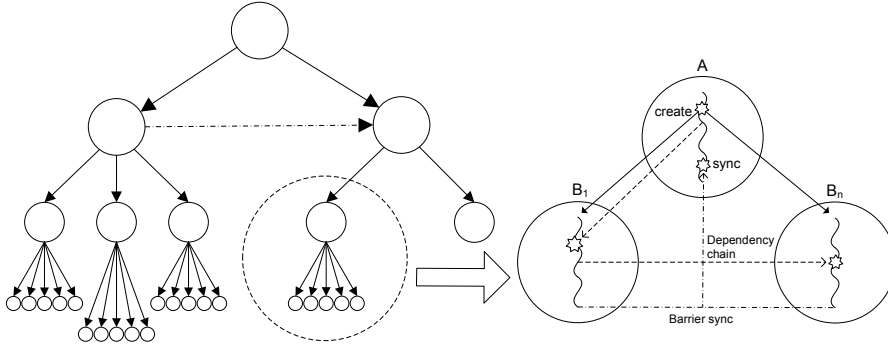


Figure 2. SVP programs as a dynamically evolving concurrency tree

Blocking threads are used in the SVP model to decouple the action of thread creation from its computation. The fine-grain blocking of threads also provides a scheduling mechanism at the instruction level. Each thread is therefore given a context of synchronising scalar variables that are registers in an implementation at the level of a processor’s ISA. These variables are initialised empty, block on read (suspending the reading thread) and reschedule any suspended thread when written. This provides a fine-grain dataflow synchronisation between threads but, in order to enforce composability and locality, the model limits this quite severely. A thread may *share* a variable with the first thread in a family it creates and each thread in that family may share a value with its successor in the family. This is illustrated in figure 2 with a code example in figure 3. This example uses a *shared* variable to maintain the sum of the first 10 elements in the array `a`, it defines a dependency chain through each thread, initialised in the parent thread. This code implements what would be a dependent loop in the equivalent C that updates a scalar variable `sum` in each iteration of the loop. In SVP the variable `sum` is created in each thread’s context and its synchronising characteristics, ensure the sequential semantics captured by the original C code. This code does not capture any significant amount of concurrency (just the accesses to array `a` may be performed concurrently) but as will be shown in section 5, this code can be transformed by reordering the reduction’s operations. In this example the shared variable `sum` is initialised before the create action, however this is not generally necessary and asynchronous functional concurrency can be exploited by initialising shared variables following a thread’s creation. In this case the thread’s creation and any computation required before a shared variable is defined can be executed concurrently with the creating thread.

In summary, the create action can be used to implement both independent and dependent loops as concurrent families of threads, to replace a function call and to capture it concurrently due to the blocking nature of the threads, when passing arguments using shared variables.

```

thread void sum_int (int * a, shared int sum) {
    index i;
    sum = sum + a[i];
}
thread void main() {
    int * a;
    family fid;
    ...
    int s_in = 0;
    create(fid;;0;9;1;;) sum_int (a, s_in);
    sync(fid);
    ...
}

```

Figure 3. Concurrent composition in SVP using shared variables to implement a dependent loop or function call

3.2 Places, delegation and mutual exclusion

As illustrated in figure 1, the threads in a family (and all of their subordinate threads) can be delegated to a remote place for execution. The place abstraction is implementation dependent and defines a set of processing resources. In the context of this paper and the implementation described, a place is a cluster of processors on the MCCMP. In order to manage dynamic placement of computation through the concept of place, the system's environment must provide a mechanism for setting place variables. Depending on how resources are partitioned, this is achieved by defining one or more System Environment Places (SEPs) that execute place server threads for multiple clients threads. This is one of the few examples where a place will be shared and the SEP must provide mutual exclusion between concurrent requests for places, as the allocation thread will update a usage map of that SEP's portion of the chip's resources. The allocation thread is similar to malloc in C, it simply takes a request for a number of processors and returns a place where the requesting thread can delegate work.

As can be seen, the concept of place provides a number of abstractions. It induces communication through the remote execution of a family of threads, it also serves as a mechanism to introduce mutual exclusion between threads that can not safely communicate via shared memory, because of its bulk-synchronous semantics. A final overloading of place is to provide security. Each place embeds a unique security key that must be matched in order for the creating thread's action to succeed. The reason for this is that places are normally used exclusively by a single thread in order for the system (either compiler or run-time) to be able to manage resource deadlock deterministically. Like all data-driven models, if the concurrency exposed exceeds the available resources then deadlock is the result. In SVP, because the minimum resources required

for a family is always that required by a single thread, then any constraint comes about through the depth of the concurrency tree. Solutions in avoiding this are virtualisation of resources (i.e. synchronising registers, and thread and family state) or delegation.

3.3 Control of concurrently executing programs

Although not important to the results presented in this paper, for completeness we conclude this section with a brief description of the management of families in a system context. As should be clear from the above discussion, `create` manages all issues of mapping and scheduling in the model's implementation and is therefore the kernel on which any operating system must be built. To complete its functionality, SVP has a number of other actions to control the termination of families. An asynchronously executing thread may *kill* or *squeeze* (a form of preemption) a named family of threads. Alternatively, a thread from a given family may *break* that family's execution. To provide reflection on these actions, a return code on the sync of a family will provide the cause of its termination, along with any system defined message.

4 Microgrid and processor architecture

The SVP model described in section 3 has been implemented in the instruction set of an in-order issue Alpha core. This core forms the basis of an MCCMP or *microgrid*, a sixteen-core example is illustrated in figure 4. It comprises a number of SVP processors sharing an on-chip COMA memory. A place in this architecture comprises a number of processors configured into a ring network to implement the SVP actions (create, sync, etc.) and to support register sharing between adjacent processor's register files. Note that in this small example a single processor is designated as the SEP and all requests for resources are resolved here. All delegation in the microgrid is performed by a network on chip, linking all processors.

The operations defined by the SVP model are implemented as additional instructions to the core's ISA, while the base ISA remains unchanged. Hence a legacy binary program may be executed on a single core using an SVP instruction to create a family of one thread (the binary code). Each SVP core is typically able to execute hundreds of threads from tens of families simultaneously. Thread management is efficient, requiring only a few processor cycles to create a parameterised family. Thread creation requires no pipeline cycles at all and occurs in the background at a rate of one thread per cycle, while resources are available. It takes zero cycles to context switch between two

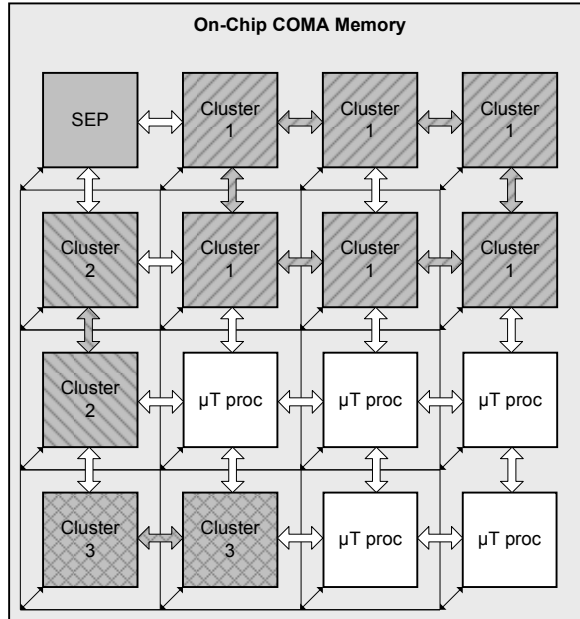


Figure 4. Overview of a configurable Microgrid. Three configured clusters are shown with one System Environment Place (SEP) which deals with resource management, and some non-configured processors.

threads and threads may context switch on every cycle, i.e. execute a single instruction and yield. It is imperative therefore that the core provide dedicated storage for thread and family state as well as a large register file for storing each thread's context of synchronising variables. In this paper we evaluate a core with an unbounded number of families, creating up to 256 threads with 1024 Integer and 512 floating-point registers for their contexts.

Every created family occupies one entry in the core's *family table* and similarly, each thread created occupies one entry in a dedicated *thread table*. Together, thread and family table entries comprise all the information needed to create, execute and clean up a complete family of threads. The implementation of all major structures has an area which is linear in the number of families, threads or registers as each is implemented with a fixed number of ports. In the current implementation, the width of each entry in the family table is around 500 bits, and around 180 bits for each thread table entry. Section 5.2 will demonstrate the area scalability for various-sized thread and family tables.

4.1 Register file and asynchronous operations

The register file needs further explanation as it implements both synchronisation and scheduling of threads in the SVP core. The synchronising action is required when communicating by shared variables between threads but the same functionality can be used to decouple certain operations within a single

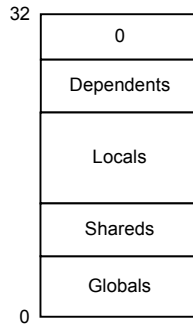


Figure 5. Layout of the virtual register space of one thread. R31, and possibly others below it, are read-as-zero.

thread from the execution of other instructions. Examples are memory accesses, thread table allocation, family synchronisation and floating point operations, all of which may execute asynchronously and complete out-of-order. Synchronisation ensures that intra-thread dependencies are preserved on any operations completing out of order and the thread is suspended when waiting for these operations.

Each thread is created with its own context of register variables which are all initialised to empty. The exception is the index value which is written by the thread creation process. The layout of this context, as it appears to the thread, is illustrated in figure 5. The core simulated can address at most 32 registers of one type and the top register(s) in the context are read-as-zero. The remainder of the register space is subdivided into four classes of registers: *globals*, *dependents*, *locals* and *shareds*.

The *globals* are read-only and are broadcast to all threads in a family from values stored in the creating thread's local class. The *dependent* and *shared* registers implement dependencies between threads and values written to a thread's shared registers are available in the corresponding dependent registers of the next thread in index sequence. A dependency chain is initialised and terminated also using a register in the creating thread's local class. Shared registers may only be written once. There are no constraints on *local* registers.

Figure 6 illustrates the relationship between the registers of different threads in different families. It shows a thread in family A creating a family with three threads (family B). It illustrates the relationship between the virtual contexts of the creating and created threads and demonstrates hierarchy as the parent thread also has globals and is also part of a dependency chain on shared variables.

The virtual register context for a thread is mapped onto a single processor's register file and is heap allocated by block for all threads in a family. Communication between processors is only induced when two adjacent threads are mapped to adjacent processors, otherwise the shared and dependent registers

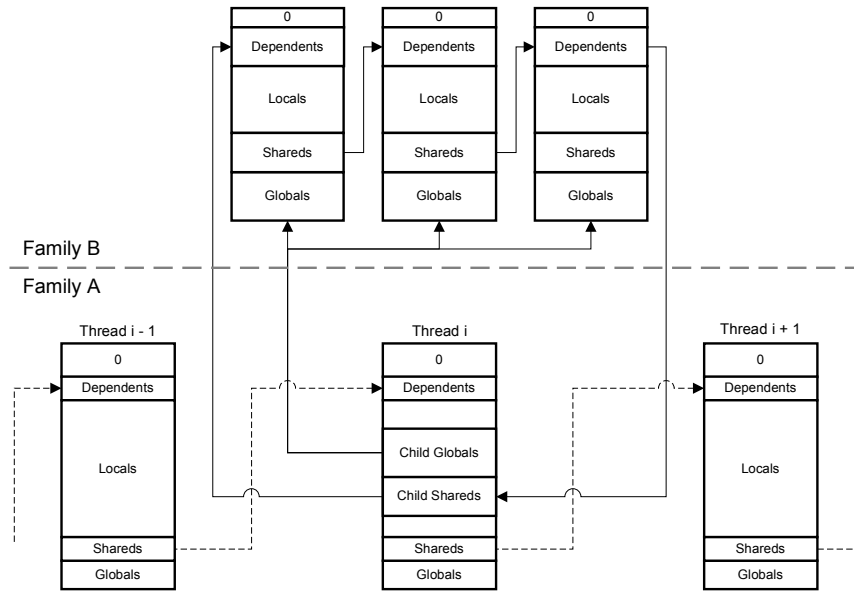


Figure 6. Register mapping for a family and its child family. Family A is the parent family and one of its child threads, thread i , creates another family B. All types of register connections are shown in the figure.

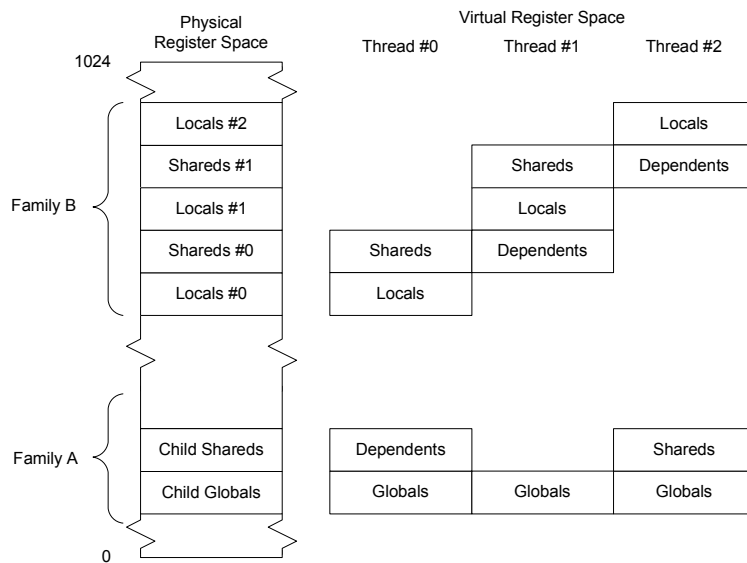


Figure 7. An example mapping of three virtual register spaces onto a large register file. Shareds and dependents of communicating threads overlap and all globals are mapped to the same registers.

of the two threads are mapped to the same physical register. An example of such a mapping is shown in figure 7; it shows the case of figure 6, where a thread in family A creates a family B of three threads on the same processor.

For families that have threads on multiple processors, the mapping on processors that do not host the creating thread is illustrated in figure 8. It has a local cache of the globals from the creating thread, which have been sent

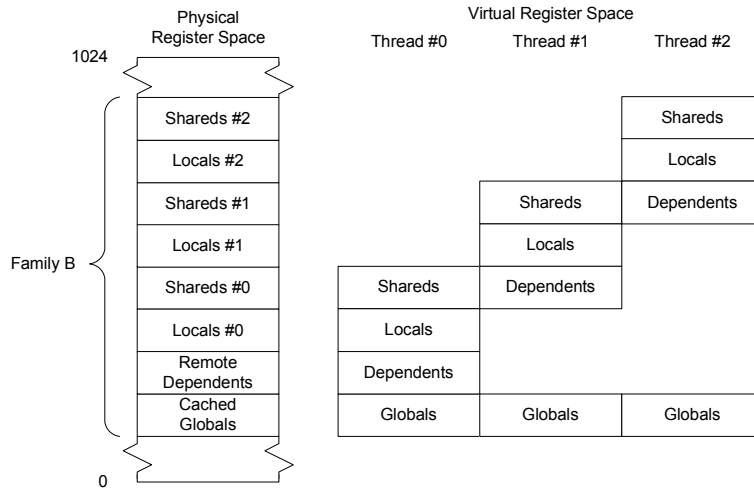


Figure 8. An example mapping of three virtual register spaces onto a large register file on processors that do not hold the creating thread.

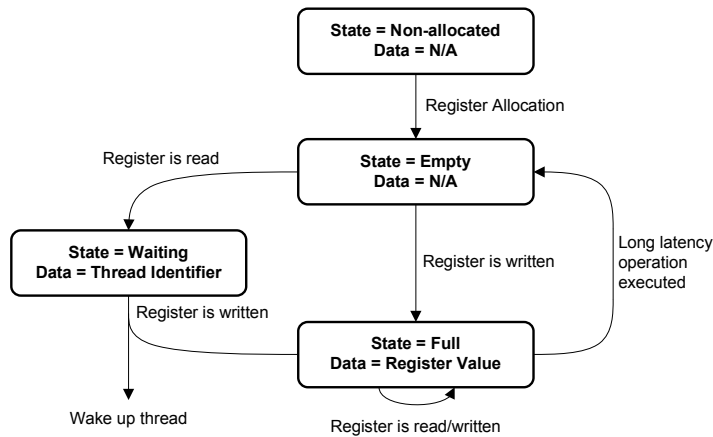


Figure 9. State Transitions of Thread Registers

over the cluster network as part of the family creation process, and a cache of the shareds from the previous thread, which is running on the neighbouring processor. Shareds are transmitted between processors when the producer thread writes them, and stored in the "remote dependents" cache on the next processor, where they remain available for the consumer thread.

Register synchronisation uses four states (*not-allocated*, *empty*, *waiting* or *full*) stored as *state bits* with the associated data. Figure 9 illustrates the state transitions of the registers for read and write operations. Registers are allocated in the empty state and may be written resulting in the full state or read resulting in the waiting state. In this state a reference to the reading thread is written to the register. When a waiting register is written to, then as well as becoming full, the suspended thread will be rescheduled. All asynchronous operations also set the state of their target register to empty at the writeback stage of the pipeline.

4.2 L1 caches

Both the L1 I-cache and D-cache are similar in principal to their counterparts in conventional processors, but with some significant changes to deal with the blocking nature of microthreads and their scheduling. Lines in the I-cache are extended with a reference count and fields to maintain a list of threads waiting on the cache line (also see 4.3). This is required as threads are only scheduled when their instructions are present in the cache line. Thus on a branch or when the PC increments over a cache line boundary, a context switch will be forced. Only on a hit to the corresponding cache line will such threads be rescheduled. Threads suspended on a register will also touch the cache before being scheduled.

The process invoked by a cache miss allocates an unreferenced line, requests the data from the next level of memory and links the thread in question to the waiting list. The list is maintained in the thread table using a pointer for that purpose. Once the data returns, this list of waiting threads is appended to the active-thread list by a single append operation. All active threads maintain a reference in the thread table to the cache-line containing their instructions, which is also reflected in the reference count in the cache line. When this counter drops to zero, no more threads are referencing the cache line and it can be replaced. Note that the instruction cache typically has fewer lines than thread table entries as usually not all threads will be active and many will share the same cache line.

The D-cache is also modified to keep track of outstanding memory reads on that line. A line is allocated on a cache miss, the request is sent to the next level of memory and the memory read request (which gets stored in the target register) is put on a linked list of requests for that line. When the data returns, these requests are serviced one by one after which the line can be reused. Finally, note that the D-cache can be much smaller than in a conventional processor since the architecture supports as many outstanding reads as there are registers in the core and will not stall the pipeline as long as there are active threads.

4.3 Thread management

A family of threads is created either by an instruction in the local core or as a request from another processor. The mechanism for managing thread state has already been illustrated in 4.2, where the *waiting* state was identified as a linked list of thread-table entries. The other logical states a thread can be in are: *empty*, *active*, *running*, *suspended* and *unused*, as illustrated in figure 10.

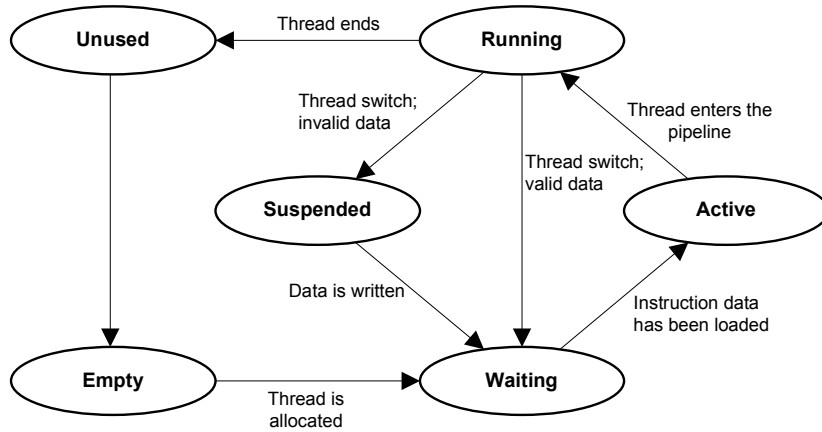


Figure 10. Thread State Transitions

Empty threads are thread-table entries not yet allocated to a family. Active threads are maintained in a queue in a similar manner to the waiting state. When a thread is selected to be run in the pipeline, it is considered to be in the running state and on a context switch goes into the suspended state and may be stored in a register. The compiler can tag certain instructions to context switch after those instructions have been fetched. This technique is used to tag dependencies on asynchronous operations, i.e., any instructions that could cause a thread switch at the register read stage of the pipeline. Proper use of the switch flag in the instruction stream prevents the pipeline from being flushed if the required data is not available. When the thread completes, it enters the unused state where it remains until the whole family has been terminated. At that point, all unused thread entries are released and returned to the empty state.

Each thread can be linked into two types of queue: a *state queue* and a *membership queue*. These two queues are not mutually exclusive and require two fields in each thread table entry. The membership queue links all the threads that belong to a single family in order to simplify family clean up, especially when forcibly terminated with a kill or break. There are three kinds of state queues: the *empty queue*, the *waiting queue* and the *active queue*. On a thread switch, the thread at the head of the active queue will be removed from the queue and used by the pipeline to fetch instructions. Threads in the suspended, running and unused states, are not linked into any state queue but are, however, still linked on their family's membership queue.

4.4 Memory Architecture

The memory system in a microgrid must provide the abstraction of a shared memory but achieve this across potentially thousands of processing cores, while providing scalable throughput both on- and off-chip. We propose to use

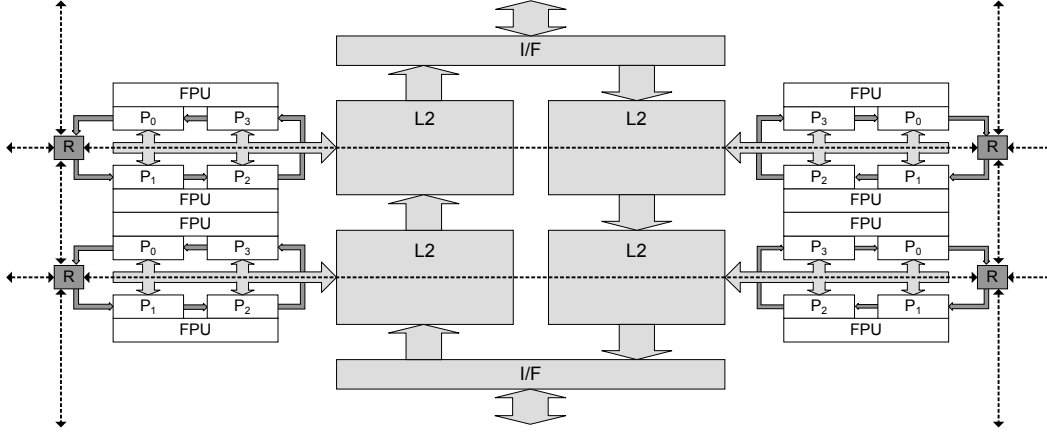


Figure 11. Microgrid tile of 4 clusters of 4 cores (P_0 to P_3), sharing an FPU between every 2 cores and an L2 cache between every 4 cores

the COMA memory model [2, 10] in a microgrid due to its latency-tolerant characteristics. The proposed memory system has multiple banks of on-chip caches and one or more off-chip communication interfaces to a multi-bank memory system. On-chip there are just two levels of cache. Each core has an L1 data and instruction cache and access to an L2 cache bank using a snooping bus. To support thousands of processors on chip, a large number of L2 caches would be required and implemented using a hierarchical ring network that can exploit locality at multiple levels. The memory consistency model plays an important role in balancing programming complexity and system performance. Since the SVP model only requires memory synchronisation on a family creation and termination, the memory model implements location consistency (LC) [8], one of the weakest consistency models. This combination of on-chip COMA and LC will provide an effective utilisation of the necessarily limited off-chip memory bandwidth. The implementation and evaluation of the memory architecture is discussed in detail in [18].

4.5 Network and Chip Architecture

A microgrid is designed to support thousands of processing cores on a single chip, connected by several logical networks. The pool of SVP cores on a microgrid chip will be configured (at design time or possibly at run time) into clusters which implement the model's abstraction of a place. Figure 11 illustrates the internal and external connections of several clusters. Clusters may be of any size and, if pre-configured, can be heterogeneous.

A chip-wide packet-switched network between clusters is used to handle resource management and delegation from one cluster to another. The delegated create involves a number of messages between clusters, the first to the SEP is to request and allocate a remote cluster for the delegation. Then the creating

thread will request a remote family table entry at the cluster allocated and initialise it. Finally the create instruction will pass any register values required to initialise shared and global variables remotely. All other data is retrieved via the shared memory network.

The COMA network provides most of the on-chip bandwidth. This is a cache-line wide hierarchy of rings connecting the L2 cache blocks local to each cluster. This network implements the cache-coherency protocol described in section 4.4 and provides the interface to the off-chip memory.

The final on-chip network is the SVP protocol network, which is local to a cluster of cores and implements the SVP operations between processors within a place, i.e. family creation, shared-register communication and family termination, however achieved. This is implemented as a ring network. Family creation and termination protocols use token propagation and a dedicated part of this ring network is used to implement register sharing between processors.

5 Results and Evaluation

This section presents the results of emulating various code fragments run on a microgrid, followed by an evaluation of the SVP MCCMP architecture in light of those results. For these results, we first describe our methodology and then present the results, showing performance scalability as well as efficiency of execution. Finally we present estimations of the implementation requirements in terms of on-chip area.

5.1 Emulation Results

The results presented here were measured using a cycle-accurate implementation of a cluster of SVP cores. This software emulator is based on the specification of the model as described in section 4. It uses the Alpha ISA as its base ISA, thus each core in the cluster executes Alpha machine code with SVP extensions. Since a μ TC compiler is still in development we are currently only able to hand-compile and test small code fragments which, although small, represent a range of different types of computation found in large scale applications. Our code fragments include several Livermore kernels, a microthreaded version of the sine function using Taylor expansion and a Fast Fourier Transform. Currently, our efforts have been focused on validating the reference implementation but we have also investigated processor performance. For this reason, while our core emulator is precise, the memory system is ideal; it is able to handle multiple requests in parallel, without conflict.

```

for (i = 0; i < N; i++) {
    x[i] =
        u[i ] + R*(z[i ] + R*y[i ]) +
        T*(u[i+3] + R*(u[i+2] + R*u[i+1]) +
        T*(u[i+6] + Q*(u[i+5] + Q*u[i+4])));
}

```

↓ ↓

```

main:
    ldl $R0, Q
    ldl $R1, R
    ldl $R2, T

    lda $R3, u
    lda $R4, x
    lda $R5, y
    lda $R6, z

    allocate $R7
    setlimit $R7, N
    cre $R7, loop
    bis $R31, $R7, $R31
end

loop:
    .registers 8 0 6 0 0 0

    s4addq $LR0, $GR3, $LR1
    ldl $LR5, 16($LR1)
    ldl $LR6, 20($LR1)
    ldl $LR7, 24($LR1)
    ldl $LR2, 4($LR1)
    ldl $LR3, 8($LR1)
    ldl $LR4, 12($LR1)
    ldl $LR1, 0($LR1)
    mull $GR0, $LR5, $LR5
    swch

    addl $LR6, $LR5, $LR5
    swch

    mull $GR2, $LR2, $LR2
    s4addq $LR0, $GR6, $LR4
    ldl $LR4, ($LR4)
    mull $GR1, $LR3, $LR3
    swch

    mull $GR2, $LR5, $LR5
    mull $GR1, $LR2, $LR2
    addl $LR4, $LR3, $LR3
    swch

    addl $LR3, $LR2, $LR2
    swch

    mull $GR1, $LR2, $LR2
    addl $LR2, $LR5, $LR2
    addl $LR4, $LR2, $LR2
    swch

    mull $GR2, $LR2, $LR2
    s4addq $LR0, $GR5, $LR3
    ldl $LR3, ($LR3)
    end

    mull $GR0, $LR5, $LR5
    addl $LR7, $LR5, $LR5
    swch

    s4addq $LR0, $GR6, $LR4
    ldl $LR4, ($LR4)
    mull $GR1, $LR3, $LR3
    swch

    addl $LR4, $LR3, $LR3
    swch

    mull $GR1, $LR3, $LR3
    addq $LR3, $LR2, $LR2
    addq $LR1, $LR2, $LR1
    swch

    s4addq $LR0, $GR4, $LR0
    stl $LR1, ($LR0)
end

```

Figure 12. C and Assembly Code for the Livermore 7 Kernel

Our first tests involved kernels with no intra-iteration dependencies. Compiling such kernels for a microthreaded processor is a straightforward process as illustrated in figure 12. There are two important things to note in the code shown in this figure. The first is that the code contains no information for mapping iterations to processors since scheduling is handled dynamically by the architecture, thus making the binary code independent of the number of cores it executes on. The second thing to note is the use of annotations (*swch*, *end*) on instructions that use operands which are targets of asynchronous operations (e.g. memory load instructions) as described in section 4.3.

We have executed the code shown in figure 12, as well as the Livermore 1 kernel, on clusters with different numbers of processors and recorded the time (in cycles) required to execute the kernel with the problem size set to 64K iterations. Each processor in the cluster has a thread table size of 256 entries and a register file with 1024 registers. A very important parameter is the size of the L1 data cache; we performed the tests with a 1 kB and a 32 kB L1 data cache per processor. Figure 13 shows the speedup achieved for these kernels, relative to 1 processor, when increasing the number of processors in the cluster. The kernels were executed on clusters with 1, 2, 4, 8, 16, 32, 64 and 128 processors. As can be seen the speedup scales almost linearly (with a 20-40% deviation from the ideal) .

A very interesting result shown in figure 13 is the stall rate. We define the stall rate as the percentage of time when the pipeline was stalled because of a hazard. For this, we do not consider the time when the pipeline is completely empty because in that case the clock can be gated. This measurement gives an indication of the pipeline’s power efficiency, where a small stall rate indicates

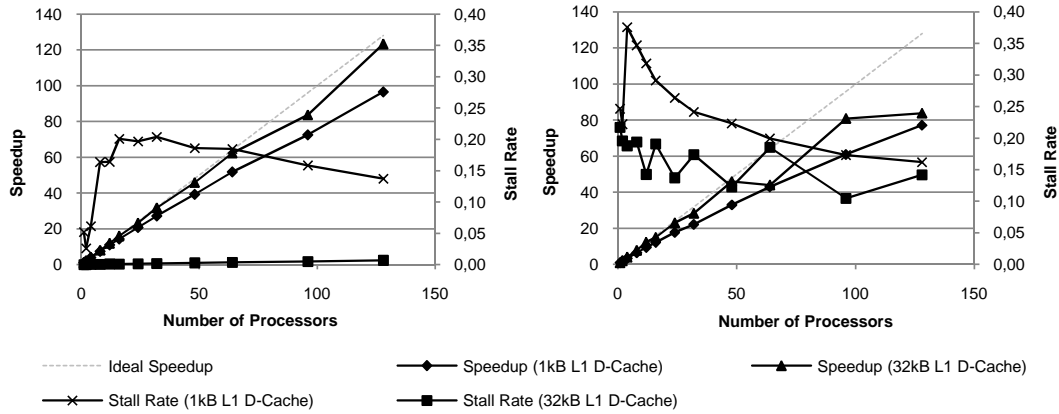


Figure 13. Speedup over number of processors in a cluster for the Livermore kernels 1 (left) and 7 (right)

high efficiency. Since our model uses simple in-order pipelines without features such as multiple instruction issue and execution or branch prediction, this metric is quite important as it shows that even without those features, which are mainly employed in conventional architectures for the purpose of avoiding pipeline hazards, SVP cores can still maximise the pipeline efficiency by thread interleaving.

The second test involved kernels which contain loop-carried dependencies, including the Livermore 3 kernel, which calculates an inner product, and an SVP version of the *sine* function. As with the previous tests the same assembly code was executed on clusters with different numbers of cores and using the same parameters. Figure 14 shows the speedup achieved. Again, the speedup is relative to a cluster with a single core but for the *sine* function speedup is also compared against the sequential version of the function and shows super-linear speedup. The SVP version of the sine function is 40% faster than the sequential version due to the absence of any loop control overheads and a high pipeline utilisation from the few local threads.

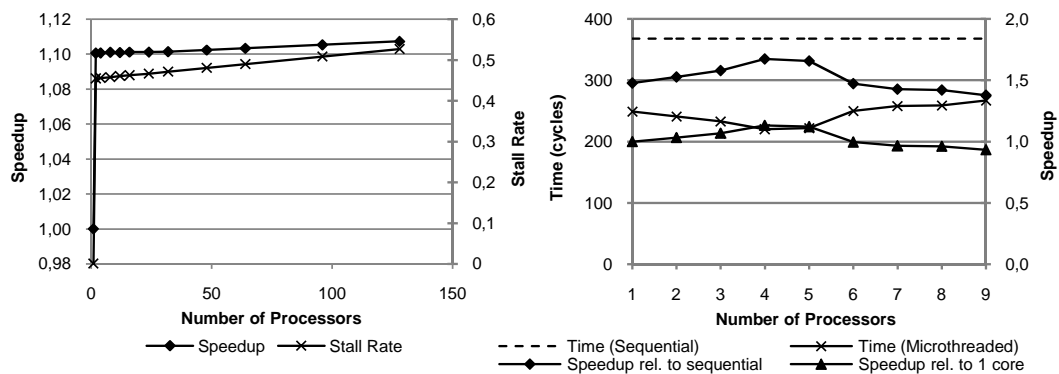


Figure 14. Speedup of Livermore 3 kernel (left) and of the microthreaded version of the *sine* function (right) over the number of processors

The SVP model makes parallelising such kernels easy, as the sequential se-

mantics of the equivalent loop are captured as a family of threads. While this concurrency can be exploited locally to allow the asynchronous execution of instructions within a single core, the speedup from multiple processors (about 10% in both examples) is limited due to the sequential constraints on the execution of the dependency. In families with dependencies, the concurrency exposed (i.e. the limit on speedup) is proportional to the number of independent instructions in a thread's schedule and inversely proportional to the number of cycles required to carry a dependency between processors (just a few cycles). In the case of the sine function, the small number of threads (9 in this test) is also another factor for the limited speedup.

In general, loop-based code captures three classes of computation: data-parallel, where the iterations are independent, recurrence relations and reductions. Because of the commutative and associative properties of the latter, reductions can be implemented with significant concurrency. Ideally, a reduction tree shows $O(N)$ concurrency and a latency of $O(\log N)$ operations. `Sine` implements a recurrence but the `Livermore 3` kernel is a reduction and its implementation can be transformed to enable several partial reductions to be performed concurrently across all processors used with a final reduction across the processors. Two features in SVP allow for such automatic optimisations, recursion on create and the ability to control where a family is executed (i.e., a place). Using those two features the code for the `Livermore 3` kernel can be rewritten to create P threads, where P is the number of processors on a cluster; these will automatically be distributed one per processor. Each of those threads then creates a family of threads to locally compute a partial sum before reducing those sums across the P cores. Figure 15 shows this transformation in μ TC code. The outer create uses the default place (second parameter omitted), which distributes one thread to each processor available. Each of these threads then creates a subordinate family at the *local* place, which creates all threads on the local processor to perform a local partial reduction.

This transformed code was compiled and executed and speedup against number of cores is shown in figure 16. As can be seen, the transformed version is significantly faster. Speedup scales almost linearly with the number of processors. The figure also shows an approximate schedule for the computation. An important fact to note is that this code is also independent of the number of cores used, even though it must be parameterised by it. We assume a system thread will provide the number of processors, which in practice just reads a register on each processor set on configuration.

One of the most widely used algorithms, particularly in signal processing, is the fast Fourier transform. Because of its importance the algorithm is also widely used for architecture benchmarking. Given the complexity of current architectures, the FFT presents a challenge, especially to compiler writers and library developers. For these reasons, the FFT has been subject to extensive

```

#define N 65536

int x[N];
int y[N];

int i;
int sum = 0;

for (i = 0; i < N; i++) {
    sum += x[i] * y[i];
}

```

↓

```

#define N 65536

int x[N];
int y[N];

family fid;
int sum = 0;

create(fid;; 0; N - 1; 1)
    sum_loop(sum, x, y)

thread void sum_loop(shared int s,
    const int* x, const int* y)
{
    index i;
    s += x[i] * y[i];
}

```

→

```

#define N 65536
#define N2 64
#define N1 (N / N2)

int x[N];
int y[N];

family fid;
int sum = 0;

create(fid;; 0; N2 - 1; 1)
    outer_loop(sum, x, y)

thread void outer_loop(shared int s,
    const int* x, const int* y)
{
    index j;
    family fid;
    int local_sum = 0

    create(fid; local; j; N - 1; N1)
        inner_loop(local_sum, x, y);
    sync(fid);
    s += local_sum;
}

thread void inner_loop(shared int s,
    const int* x, const int* y)
{
    index i;
    s += x[i] * y[i];
}

```

Figure 15. Transformation of the Livermore 3 kernel to parallel partial reductions with sequential version of the kernel (left), the simple statement by statement translation to microthreaded code (middle) and the optimised version which performs partial reductions (right)

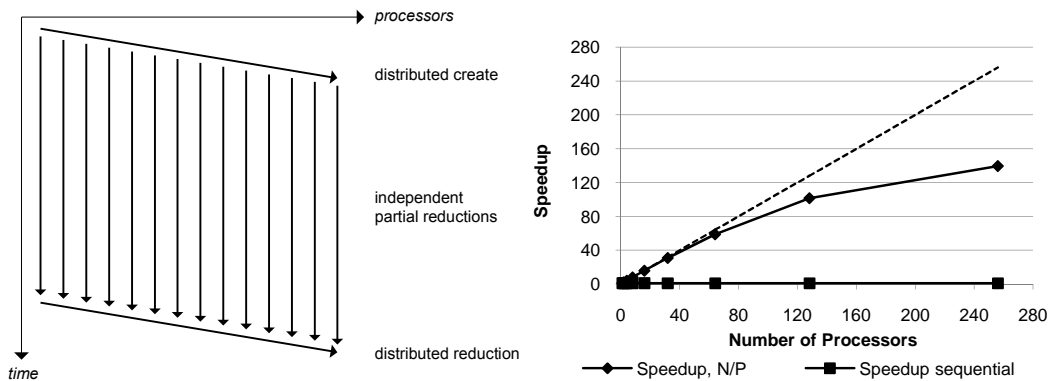


Figure 16. Implementation of reduction on P processors executing a partial reduction on each processor; an approximate schedule (left) and the results of execution (right) with a comparison to the sequential reduction.

research [3]. Due to its importance we ran an SVP version of the FFT to examine how an SVP-based architecture performs with this algorithm. This algorithm also gives us a first chance for performance comparison with other architectures as performance results for this algorithm have been widely published .

It is shown in [5] how a 4-processor Itanium-2 SMP running at 1.5 GHz, with a theoretical peak performance of 24 GFLOPS (4 FLOP/cycle/processor × 1.5 GHz × 4 processors), has a peak performance of roughly 5.2 GFLOPS

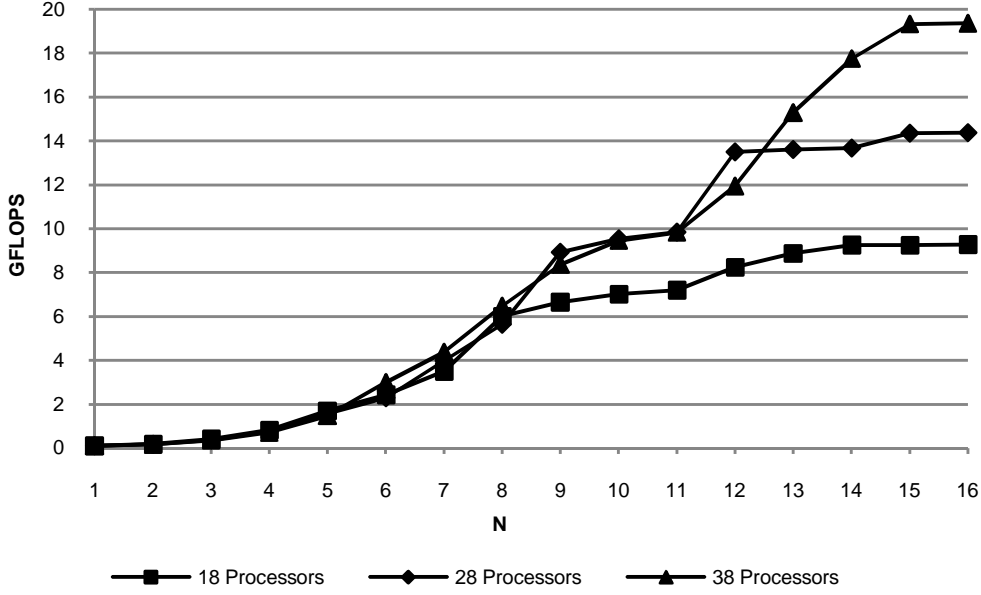


Figure 17. Performance (expressed in GFLOPS) of FFT executed on clusters with 18, 28 and 38 cores at 1.5GHz against log of problem size (N).

(22% of the theoretical maximum) running a double precision complex FFT application with N (the \log_2 of the number of points of the problem size) as 15. Using comparisons in 5.3 it can be seen that 28 SVP cores, with a theoretical peak performance of 21 GFLOPS (1 FLOP/cycle/ 2 processors \times 1.5 GHz \times 28 processors), will fit in the same area and achieve a peak performance of 14.4 GFLOPS (67% of the theoretical maximum) running a double precision complex FFT kernel for the same input size.

5.2 Component Level Area Estimation

This section presents a component level core area estimation and a comparison with several conventional processor cores. The components in a SVP core can be categorised into three types:

- functional units, including an ALU, multiplier and floating point unit (n.b. the latter two may be shared by a number of cores based on relative frequency of use)
- on-chip memory including registers, family table, thread table and L1 data and instruction cache
- control logic

The size of the functional units was estimated based on [9]. The size of the on-chip memory, implemented as SRAM arrays, was estimated using CACTI [16] and the control logic was omitted considering its relatively small area contribution. As discussed, one SVP core can support any number of threads

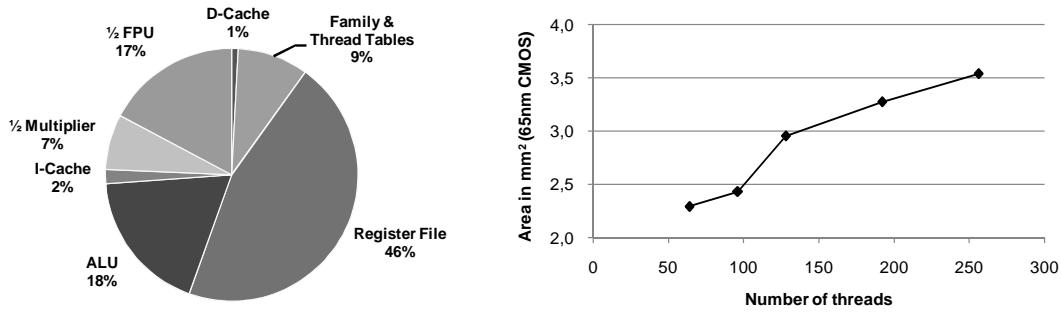


Figure 18. Area contribution of each SVP core component (left) and core area estimation for processors supporting approximately this number of threads (right)

by varying the sizes of the thread supporting structures (i.e., registers, family and thread table), which should be designed with certain relative size ratios. For instance, the family table is typically smaller than the thread table, since a single family is a group of threads. Currently, given the number of integer registers as N , we assume $\frac{N}{2}$ floating point registers, $\frac{N}{4}$ thread table entries, $\frac{N}{32}$ family table entries and $\frac{N}{16}$ instruction cache lines.

Figure 18 shows the area contribution of the components of a microthread core where two cores share an FPU and a multiplier. The components are: a register file (1024 integer and 512 floating point registers), a 32-entry family table, a 256-entry thread table, a 1 kB L1 data cache and a 4 kB L1 instruction cache. The figure shows that the major contributions to core area in conventional processors (the L1 caches) are insignificant in an SVP core and instead the register file dominates the area. It should be noted that this area scales with the required tolerance to latency; the area of a core of this configuration with 65nm CMOS technology is approximately $3.53mm^2$ but can be reduced to about $2.30mm^2$ when reducing the number of supported threads per core to 64. Figure 18 illustrates the area of a core for various sizes of the support structures indexed by the number of threads it supports.

5.3 Core Area Comparison against Conventional Processors

In the same technology, an SVP-based processor is able to provide more cores than conventional processors. Even assuming half the chip size is given to on-chip COMA L2 memory, we get the following comparisons (all sizes are scaled to the same technology):

- A McKinley (Intel Itanium 2 [3]) chip with one core supporting one thread can accommodate 7 SVP cores supporting around 1,800 microthreads.
- An UltraSPARC T2 (Niagara 2 [14]) with 8 cores supporting 64 threads can accommodate over 48 SVP cores supporting nearly 12,400 microthreads.
- A Power5 [15] with 2 cores supporting 4 threads can accommodate nearly

14 SVP cores supporting over 3,500 microthreads.

Section 5.1 compared the performance of the Itanium 2 SMP processor versus an SVP-based CMP of the same area and shows that the SVP CMP had significant performance gains.

6 Conclusion and Future Work

With the end of frequency increases in sight, many parallel architectures have sprung up to answer the need for a continued increase in performance. And while the step away from sequential architectures to parallel architectures is necessary, no architecture so far has been able to both scalably support a varying number of threads and solve the problems inherent to programming such a parallel architecture. In this paper we presented the answer to these problems in the form of SVP, an abstract parallel programming model, and microgrids of SVP-based RISC processors, a parallel processor architecture. SVP uses families of threads to capture concurrency at any level in a program, without complex and deadlock-inducing constructs such as semaphores. It defines communication restrictions in one place and relaxations in other places, both of which allow efficient implementations to be constructed without loss of either generality or determinism. SVP programs can be run on a microgrid, which employs many simple cores to provide the resources to scalably support any number of threads. Each core is able to efficiently handle many threads at once by employing synchronising registers to deal with inter-thread dependencies and operations with a long or unknown latency. By being able to suspend threads each core ensures the pipeline always runs at maximum efficiency. Without complex logic to extract parallelism from sequential code streams, each core can scale to support any number of threads in order to cope with increasing instruction latencies. Results from running various compiled SVP code fragments on an emulated microgrid show that SVP and microgrids are able to offer scalable speedup, even performing significantly better than conventional processors of identical size.

References

- [1] Aether project. <http://www.aether-ist.org/>.
- [2] KSR1 technical summary. Technical report, K.S.R. Corporation, 1992.
- [3] Inside Intel Itanium 2 processor: an Itanium processor family member for balanced performance over a wide range of applications. Technical white paper, Hewlett Packard, July 2002.

- [4] International technology roadmap for semiconductors update. Technical report, Semiconductor Industry Association, 2006.
- [5] A. Ali, L. Johnsson, and J. Subhlok. Scheduling FFT computation on SMP and multicore systems. In *Proc. of the 21st annual international conference on Supercomputing*, pages 293–301, 2007.
- [6] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.
- [7] A. Bolychevsky, C.R. Jesshope, and V.B. Muchnick. Dynamic scheduling in RISC architectures. *IEE Trans. E, Computers and Digital Techniques*, 143:309–317, 1996.
- [8] G.R. Gao and V. Sarkar. Location consistency—a new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 49(8):798–813, 2000.
- [9] S. Gupta, S.W. Keckler, and D. Burger. Technology independent area and delay estimates for microprocessor building blocks. Technical report, Computer Architecture and Technology Laboratory, 2000.
- [10] E. Hagersten, A. Landin, and S. Haridi. DDM—a cache-only memory architecture. *IEEE Computer*, 25(9):44–54, 1992.
- [11] L. Hammond, B.D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6):92–103, Nov–Dec 2004.
- [12] C.R. Jesshope. μ TC—an intermediate language for programming chip multiprocessors. In *ACSAC06, LNCS 4186*, pages 147–160, 2006.
- [13] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, July 2005.
- [14] H. McGhan. Niagara 2 opens the floodgates. *Microprocessor Report*, 20(11):1–12, 2006.
- [15] B. Sinhary, R.N. Kalla, J.M. Tender, R.J. Eickenmeyer, and J.B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4):505–521, 2005.
- [16] D. Tarjan, S. Thoziyoor, and N.P. Jouppi. Cacti 4.0. Technical report, Western Research Laboratory, Compaq, 2006.
- [17] P. Trancoso, P. Evripidou, K. Stavrou, and C. Kyriacou. A case for chip multiprocessors based on the data-driven multithreading model. *Int. J. Parallel Program*, 34(3):213–235, June 2006.
- [18] L. Zhang and C.R. Jesshope. On-chip COMA cache-coherence protocol for microgrids of microthreaded cores. In *Proc. Workshop on Highly Parallel Processing on a Chip (HPPC)*, 2007.