

OPERATING SYSTEMS IN SILICON AND THE DYNAMIC MANAGEMENT OF RESOURCES IN MANY-CORE CHIPS

CHRIS JESSHOPE

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

Received
Revised
Communicated by

ABSTRACT

This discussion paper explores the problems of operating systems support when implementing concurrency controls at the level of the instruction set in processors designed for multi- and many-core chips. It introduces the SVP model and its implementation in DRISC processors to a level of detail required to understand these problems. The major contribution of the paper is in analysing the issues faced in porting operating system functionality onto such processors. The paper covers the issues of job scheduling, dynamic resource management, memory protection and security. It provides examples in μ TC (a language based on the SVP model) of how resource management and security issues are managed. It concludes that the implementation of concurrency controls in a processor's instruction set is very disruptive. However, the author sees no alternatives if mainstream computing is ever to be served effectively by multi- and many core processors.

1 Introduction

1.1 *Technology and Architecture*

If it were not so serious it would be amusing as companies abruptly embrace concurrency as the new panacea following the abrupt halt in their pursuit of higher clock frequency for achieving better performance. The scaling issues with implicitly parallel, out-of-order, commodity processors were quite predictable and yet we remain unprepared as the computer architecture community scrabbles for new solutions. It is accepted now, that the only avenue for increasing performance in future computer systems will be to build circuits that contain many processor cores. Scalable performance requires distributed rather than centralised solutions to the problem of energy dissipation, slow on-chip communications and reliability. These are all discussed in a recent report from the Electrical Engineering and Computer Sciences department of the University of California at Berkeley [1]. This is a seismic shift and, as this paper will illustrate, is disruptive to more than just computer architecture.

These chips resemble our current distributed systems, where processors communicate asynchronously and work is distributed dynamically, both of which are difficult problems to solve. Embedded and commodity multi-cores exist today but programmers of these devices (or compilers with auto-parallelisation) must explicitly map and schedule the components of an application onto the available resources, which means the problems of application programming and concurrency engineering are intertwined. What we need is a systematic

approach that captures concurrency as an abstraction (programming) and where processor implementations perform mapping and scheduling (concurrency engineering).

One example of such an approach was the transputer concept [2], which was realised in the INMOS T800 processor chip in 1987. The T800 comprised a 64-bit floating-point processor, 4KByte of memory and 4 communications channels in a single chip. Moreover, it provided instructions to create/terminate processes and to communicate between them. Its main contribution was that communication was abstracted in the binary code using channels. These were resolved by mapping them to a memory word for two processes communicating on the same processor or one of the chip's physical channels, when communicating between processors. The compiler was not aware of this mapping thus separating programming and concurrency engineering issues.

What is also interesting, is that were the T800 to be implemented in today's technology, its 1/4M transistors could be tiled a thousand times and this gives a perspective on how inefficient today's processors are in exploiting the available silicon.

1.2 Programmability

In practice, there is no principle difference between grid clusters and multi-core chips, the difference is only in the implementation parameters, e.g. communication and synchronisation times relative to computational performance. These change significantly as data is moved on chip, close to the processing resources. The infrastructural implications, on the other hand, are immense, as a whole new generation of engineers and scientists will be forced to tackle the challenges of concurrency, whereas in the past it was only a few who were at the cutting edge of high-performance, parallel computing who would be exposed to this. This is the most significant problem in the exploitation of future of many-core devices. The programming of asynchronous concurrency is not only a difficult problem [3] it is also a recurring one. It requires the programmer to deal with both algorithmic aspects as well as concurrency-related issues. Concurrency needs to be identified, captured, distributed, synchronised and scheduled. This requires expertise, and is time consuming and error-prone. It is a recurring problem, as typically reprogramming is required as soon as the underlying parameters change (e.g. number of processors, type of memory organisation, latency of communication and synchronisation etc.). Clearly these are not characteristics that can be tolerated in mainstream computing and this explains why many vendors are back peddling on providing many simple cores on one chip.

The foundation for the work presented here is based on ten years of research into novel computer architecture. This work started in the mid 1990s [4] with the goal of developing an architecture that could provide an efficient, distributed programming platform based on the data-parallel abstraction [5][6]. The key concept was to decouple instruction execution from communication (including accessing data from memory), as this relieves the programmer or compiler from having to schedule every operation statically. In 1996, a novel processor architecture called DRISC was proposed that achieved this goal [4]. That first model was quite restricted but has been refined over the intervening decade from a processor model to an abstract programming model based on microthreads [7]. This model has been adopted as the SANE Virtual Processor (SVP) in the AETHER project, where SANE is derived from Self-Adaptive Network Entity (see <http://www.aether-ist.org/>). The SVP model addresses issues

such as the safe composition of concurrent programs, the avoidance of deadlock and exposes communication and locality in an abstract manner to be managed by compilers for the model. The simple fact of restricting communication patterns makes the problem static, even when concurrency is dynamically scheduled. SVP is a true replacement for the von Neumann model of computation used in all sequential machines and adapted in many concurrent ones. Moreover the Microgrids project (see: <http://www.science.uva.nl/research/csa/projects.html>) has shown that the SVP model can be implemented at the level of instructions in a processor's ISA, providing abstraction in the capture and implementation of concurrency similar to the transputer concept. The advantage of SVP is that it addresses both the programmability issues, i.e. deadlock-freedom by design, as well as the technological issues described in section 1.1.

1.3 Disruption vs. Incrementalism

Computational models implemented at the level of an instruction set are disruptive and presents many difficulties at the system level. They require significant research and tool development to provide an infrastructure, without which an evaluation of the technology becomes impossible. But the alternative, to continue with ad-hoc models of concurrency, implemented in software over conventional instruction sets, is not efficient. One of the biggest issues is how to program many-core systems. The goal should be to provide the programmer with a sequential language and to support the compiler in parallelising code automatically. This strategy was successful for early vector computers but was considered too difficult for asynchronous, distributed computers and low-level solutions such as MPI have been developed. Although MPI is considered portable in a programming sense, it is not so in terms of concurrency engineering and is non-portable under changes to implementation parameters (Section 1.2). This occurs in many-core chips, where number of cores etc. will change over time. Neither is MPI appropriate for mainstream computing and non-expert programmers. With SVP however, the compilation of sequential languages concurrent binary code is quite feasible and this is one of the biggest contributions that an abstract concurrency model can make, especially when implemented at the level of the ISA. The biggest difficulty with an incremental or ad-hoc approach is that the compiler must perform the mapping and scheduling of any concurrency exposed in its analysis. In SVP this task is delegated to the implementation of the instructions that capture the SVP model and this simplifies the compilation significantly. Concurrency extraction, is a well-researched area, e.g. [8]-[9], and the compilation of languages like C to SVP become a reality given a model that abstracts concurrency and provides a handle on locality for efficiency analysis.

2 Concurrency in the Processor's ISA

2.1 Background

It is surprising how little research there is on computers that capture concurrency controls explicitly in the ISA. Excluding dataflow models, there are just two significant developments in this area, one is the transputer, which has already been described in Section 1.1. The earliest by far however, was the pioneering work by Burton Smith on the Delencor HEP [10], the Horizon, and eventually the Tera architecture [11] (or Cray MTA).

HEP supported multiple blocking processes interleaved on a cycle-by-cycle basis in an eight-stage pipeline and required at least eight active processes to keep the pipeline full. HEP was able to tolerate latency in accessing its distributed memory, by suspending processes during memory accesses. HEP also supported synchronisation between processes by adding full-empty bits to data in memory, so that a read to an empty location or a write to a full location would block the process. However, the HEP did not force an abstract model on programming, as by providing both normal and blocking operations to memory, various programming models could be supported. In fact these mechanisms were exposed in HEP-FORTRAN, which added asynchronous variables for synchronisation between processes, and a CREATE construct for creating processes. The Tera (Cray MTA) incrementally improved on previous designs and has been shown to outperform the fastest supercomputers in graph processing algorithms while executing on a modest four-processor system [12]. This shows the power of a latency-tolerant processor and is a significant result as it demonstrates that adding synchronisation at the level of the ISA can extend the range of applicability of a concurrent computing system.

Most other concurrent architectures support scheduling and synchronisation in a software layer, typically using an inter-processor interrupt mechanism. The exception is in dataflow processors, e.g. the Monsoon architecture [13], which had many similarities with the work by Burton Smith. The difference is that in Monsoon, programs are implemented in the context of dataflow graphs rather than the explicit management of concurrent threads. This approach is inherently inefficient and there are difficulties in addressing applicative memory semantics in this model. Papadopoulos and Culler confirm this inefficiency in evaluating simple arithmetic using an implicit model in [14].

If we look at recent developments in this field, we see one of two directions taken. The first is exemplified by the CELL processor, where instead of providing abstractions in terms of a memory model and virtual concurrency, CELL exposes the local memory on each SPE and also uses the processor as a thread, exposing all scheduling to the compiler. The result requires threads/processors to interact with each other via a globally coherent DMA engine. As discussed in [15], this programming model imposes a significant burden on the programmer, since it requires explicit mapping of concurrency and communication onto the PPE and SPE portions of code. Moreover, it does not provide portability between different generations of device that will require different mappings as the number of SPEs increases.

The second direction is constrained by issues of compatibility with existing ISAs. Intel's Multiple-instruction-stream processor (MISP) [16] introduces the abstractions of threads via a signal instruction in the ISA that represents a thread continuation. This gives compatibility with existing operating systems and programming approaches. The signal takes a program counter, a stack frame and a thread identifier as parameters and provides easy migration from conventional threaded code and gives a measure of abstraction at the level of the instruction set. However, it is a coarse-grained model as the signal instruction is estimated to take some 5000 cycles to implement in microcode. In contrast the SVP model can be implemented in a processor's ISA with synchronisation between threads taking a few cycles using a large distributed-shared register file. Like the HEP it has data-driven instruction scheduling but it combines this with the abstraction mechanisms pioneered by the transputer. Thus in contrast

to MISIP, overheads for thread creation, signalling, context switching, etc. are insignificant i.e. $O(1)$ pipeline cycle.

2.2 The Microthread or SVP Model

The SVP model extends the microthreaded model by capturing reflection on dynamic concurrency creation. SVP captures concurrency across all scales in a computer system, i.e. operating system, programming model, execution model right down to the micro-architecture model. It is based on the sequential model and possesses the same properties of deadlock freedom under composition and determinism of results. Moreover, it does this while exposing unbounded concurrency in the programs it describes.

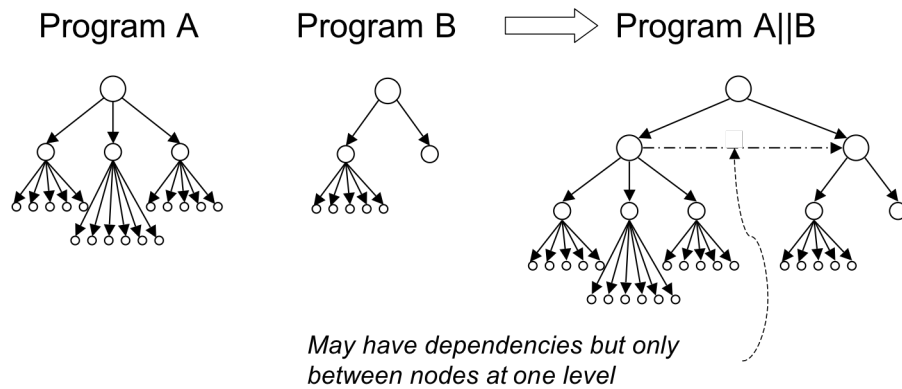


Fig. 1. Concurrent composition of programs in SVP, nodes represent threads, branching represents concurrent composition

2.2.1 Composition in SVP

The SVP model is based on blocking threads that are strict sequences of actions. Composition is by a *create* action. Whereas in the sequential model we have loops and function calls, in SVP both are implemented by creating one or more threads using a *this* action. These threads run concurrently with the creating thread and data dependencies between them are captured and respected. This composition in SVP is concurrent and dynamic. A program in the model can be viewed as a dynamically evolving concurrency tree, a snapshot of which is illustrated in Figure 1. The *create* action defines parameterised families of threads that are bound to resources. In a DRISC processor, *create* is an instruction and blocking is implemented in a distributed-shared register file with synchronisation bits. The nodes in Figure 1 are microthreads, typically comprising just a few machine instructions and working on a small context of registers, which fixes the level of granularity of the implementation at the instruction-level. It should be emphasised that the model is self-similar and all nodes in this concurrency tree are managed by the same machine instructions.

Sequence is introduced into the model in one of two ways:

- i. the microthreads are strict sequences of one or more machine instructions. However, a leaf node may comprise a complete binary program that has no understanding of the model and this provides backwards compatibility with the base processor's ISA.
- ii. explicit dependencies are captured between microthreads in a restricted manner. They are defined between the creating thread and its subordinate threads and between subordinate threads at one level as an acyclic dependency chain from the creating thread through each thread created in a well-defined order.

The create action is based on a single thread definition and its events are clarified in Figure 2. The creating thread A executes a create action, which asynchronously creates an ordered set of threads, B_0 through B_n , subject to resource constraints, i.e. not all threads are created simultaneously. In the DRISC processor, creation is autonomous and overlapped with pipeline execution, allowing deep pipelining of concurrency to be achieved; this is similar to chaining in a vector computer. The number of threads can also be unbounded, by creating the family block by block and terminating the family dynamically.

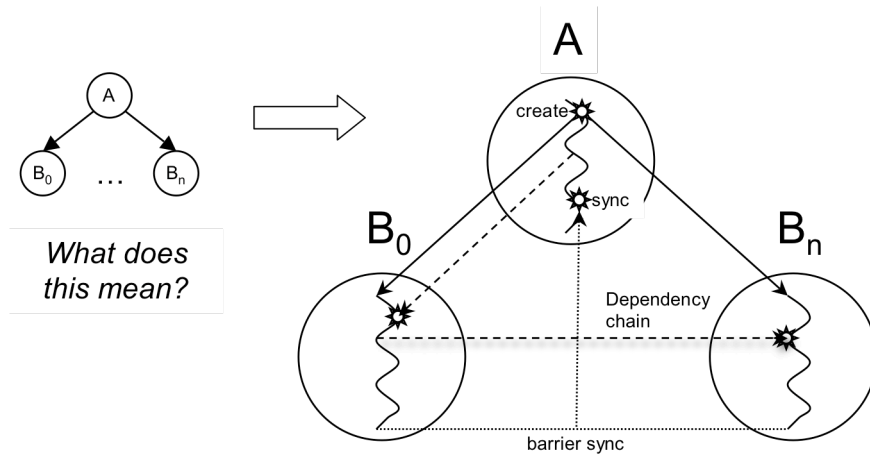


Fig. 2. Explanation of the events that occur in creating a family of threads, where the wavy lines represent a thread's sequence of instructions.

The second event in the creating thread, A, is when all threads in the family have terminated and the distributed memory they have updated is well defined. This is captured by a *sync* action. Note that thread A and any families it creates execute concurrently and asynchronously but A must wait on the sync actions associated with a family it creates before using any of the results the family generates in memory.

The create action is a concurrent analogue of both function invocation and loop structure in the sequential model. Parameterisation of the family captures both static and dynamic loop bounds and is defined by parameters: {start, limit, step}. Each thread in a family has available to it, through the implementation of create, a unique value from this implied sequence. Unbounded families represent dynamic loops with termination on the execution of a *break* action in any of the threads created. In fact if SVP is serialised by executing each thread to completion in index sequence, the analogue to the sequential model is precise.

2.2.2 Communication in SVP

Threads are blocking and participate in synchronising communication using a blocking read. This fine grain communication between threads is deliberately restricted so that communication is localised and exposed to the compiler, as even long-range, on-chip communication will be expensive in future silicon systems. Communication is constrained to be between adjacent threads in a family's index sequence. It is specified by reads and writes to specific register classes in the threads' contexts. If the two threads are mapped to the same processor, communication is implemented by a synchronised read after write to a single physical register shared between the threads' contexts. Otherwise it is implemented as a local communication between two adjacent processors' register files. Synchronisation is implemented by two bits on each register that implement a dataflow i-structure [17].

The events defined by an SVP communication cascade through the threads in a family. The creating thread, A in Figure 2, may write data only to the first thread created in the family, B_0 , which will block on the first action requiring that data. Thread B_0 can in turn write to the next thread in index sequence and so on, so that each thread writes to the next in index sequence until we get to B_n . To provide closure to this sequence, corresponding data written by B_n is available to the creating thread, A, but this is not a synchronising communication; it relies on the sync action to signal that this data is defined. To maintain symmetry and enable pre-emption of families of threads, this data overwrites the location in A's context that initiated the dependency chain. This restriction on communication guarantees freedom from communication deadlock in a family and by composition in complete programs [18]. It also exposes local communication to the compiler and finally, it allows the compiler to analyse resource deadlock in the case of bounded recursion of families of threads.

The restriction is not as severe as it first may seem due to the recursive nature of the model. To achieve it, compiler transformations are required that are localising in terms of communication. For example, any regular dependency over an iteration space with a skip distance of greater than one (the skip distance may be dynamically defined) can be transformed by nesting families of threads to isolate a local dependency and expose an independent family of threads, whose extent is equal to the skip distance. Other techniques can transform multiple non-local dependencies to local ones by generating code that routes data values between threads. Given that the routing step is a register-to-register move instruction in a DRISC processor, this technique is very efficient.

There are two other forms of communication in the SVP model. These occur at a higher level of abstraction (and of granularity) and are also implicit in the binary code. The first is that the model assumes a shared-memory abstraction and while this may or may not be a basis for implementation, there will certainly be communication involved in any implementation in order to obtain distributed access to data. As DRISC processors are highly tolerant to latency [6], we are investigating an on-chip COMA, cache-coherent protocol for the chip's L2 cache implementation. This is similar to a Data-diffusion memory [19], but is implemented with a memory consistency model based on but even more relaxed than the location consistency model proposed by Gao and Sakar [20].

The final form of communication in a multi-core implementation of the SVP model is encapsulated in the implementation of create, which is the point at which a binding occurs between a computation and the processors it uses. Create deterministically distributes threads

to a set of processors at a *place*. Two special places are defined in SVP, the *default* place, which in a DRISC many-core chip distributes the threads to the same set of processors as the creating family's threads and the *local* place, which places all threads created on the same processor as creating thread. Finally SVP supports *named* places, which are set by a place server and subsequently used in a create action, which delegates the execution of the family of threads to a new set of processors, i.e. it distributes work in similar manner to a remote procedure call, and induces communication between the creating thread and the remote place.

The named place is set dynamically by a resource-management system call and the focus of this paper is on exactly this aspect of the model, i.e. the management of processor (and memory) resources in a multi-core of DRISC processors using this delegation mechanism. Before this is investigated in detail however, we must complete the definition of the SVP model.

2.2.3 Reflection on concurrency

To provide a closure of concurrent composition within the SVP model, some form of concurrency management is required for reflection (observation and control). This is necessary to implement dynamic binding of computation to resources. In a retrospective on the Monsoon architecture [14], Papadopoulos and Culler observed: "Imagine in a conventional instruction set architecture that the register reservation bits are exposed in the instruction set, so it would be possible to branch on the result of a load being 'not yet present.'" Although the solution they propose is not at an appropriate level of abstraction for SVP, this action of reflection on the state of a concurrent computation has been introduced into the SVP model to provide support for self-adaptive computation.

The create, sync and break actions have already been described. To these are added two further actions on families of threads, namely *kill* and *squeeze*. Kill terminates a family of threads losing its state and squeeze terminates a family by identifying and capturing an intermediate state, so that the family can be re-executed from that point on different resources. These two actions are executed in an independent and asynchronous control thread that is monitoring some aspect of the computation or its environment (see Figure 3). These actions require families be identified in SVP and for this we define a *family identifier*.

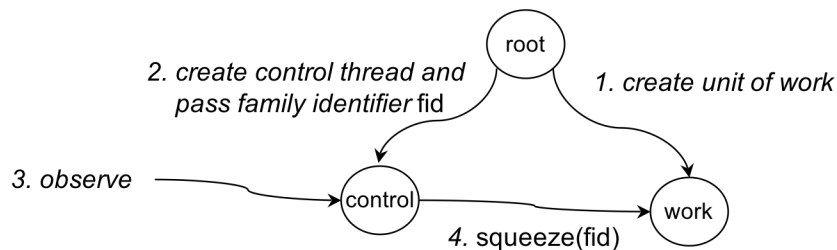


FIG. 3. Reflection using named families of threads and concurrency control

Managing concurrency asynchronously also requires that these actions be observed. For example, *root* in Figure 3 needs to know whether *work* was squeezed by *control* or whether it executed to completion, so that if necessary it can recreate work from its squeezed state in

order to complete it. In SVP this is achieved by defining a return code and sometimes a return value to the creating thread via the sync action. The return code identifies how the family terminated (normal, break, kill, squeeze) and a return value is set by break and squeeze. In the former, the thread that succeeds in executing the break sets the “return” value (threads contend for this) and in the squeeze it captures a unique point in the index sequence where the computation was gracefully terminated.

This reflection has far-reaching consequence in the model, i.e. only *root* in Figure 3 should be allowed to terminate *work*. In a conventional processor these actions would be controlled by user authentication in the operating system but in a DRISC processor these actions are implemented as instructions in its ISA, which means security issues must also be dealt with at the level of instructions in an ISA!

3 SVP-based Operating Systems for Microgrids

3.1 General Principles

In this section we explore some aspects of adding concurrency based on the SVP model to a processor’s ISA. This is a design-oriented paper and hence the issues are identified here with only some solutions proposed. It is emphasised that many of these issues are relevant to the dynamic allocation of resources in other concurrent systems, where SVP is not implemented at the level of the ISA. For example, in [21], an implementation of SVP based on POSIX threads is described, which supports all of the actions described above. Notwithstanding this, the major thrust of this paper is directed towards many-core devices (microgrids).

In general there are two approaches in asynchronous concurrent systems in supporting software through an operating system. The first, which is used on many grid clusters, is to support a conventional operating system on every node in the cluster. The second, which is normally adopted on multi-core processor chips, like the CELL processor, is to run an operating system on a single node and to run a thin kernel on all other nodes. That kernel would normally support job submission/termination, synchronisation and communication, and it may also support the scheduling of multiple tasks to provide tolerance to long latencies in communication and synchronisation. If we consider the SVP model implemented in a DRISC processor, this is exactly what it gives! These operations cannot therefore be retrospectively added as a layer of software and must be resolved in the design phase of the processor. The implementation of create provides job submission both locally and remotely. Kill and squeeze give job termination, with or without checkpoints. Communication and synchronisation in the model are implicit and arise from the mapping of threads to individual processors and families to clusters of processors. Synchronisation and communication are restricted by the model but this is to obtain security through determinism and freedom from deadlock and is an advantage.

SVP implementations also provide for the scheduling of tasks, in this model a task (or *unit of work* as we call it) is the result of a create action, i.e. a family of threads and possibly further subordinate families. These are not scheduled as tasks, instead all threads in a unit of work are interleaved with all other threads executing on the same processor. Currently we do not support priority between threads, although it is possible to implement this. To date

however, we have not yet found a need to do so. Latency in DRISC implementations of SVP is tolerated at the instruction-level as any instruction waiting on a communication or memory load will block and cause its thread to suspend until data is available. Fairness is built into the instruction set as no thread can monopolise the processor it is executing on. Context switches are forced under the following circumstances in our implementations of DRISC processors:

- i. where the compiler identifies a dynamic dependency and flags the consumer instruction to context switch (e.g. an instruction that depends on a prior load word or a communication);
- ii. a branch instruction (the hardware must check that the target instruction is in the cache before the next instruction can be scheduled);
- iii. the program counter is incremented over a cache line boundary (as above).

Fairness is guaranteed, as in straight-line code, a thread always yields on cache line boundaries (iii). Also, a thread can execute no more than one iteration of a loop before yielding at a branch point (ii). Consequently in our current implementation, no thread can execute more than 8 instructions before yielding to another thread, even if it is able to continue and then it is placed at the tail of an active queue of threads. Item (i) in practice means that threads will often execute less instructions before yielding.

The concept of job priority is one entrenched in the timesharing of a single processor when executing jobs concurrently. We take the position that this is inappropriate for large-scale concurrent systems. The idea behind prioritising jobs is to share the processors cycles in some ratio, which is proportional to the priority of a job, so that high priority jobs are executed more quickly. In a DRISC processor implementing SVP, binary code can be executed on any number of processors hence the same result can be achieved in a microgrid by mapping code to a number of processors that is proportional to the concept of priority in a legacy system. Indeed the AETHER project's self-adaptive approach to computation is to achieve this dynamically by monitoring the environment in some way.

There is a final reason why it is not advisable to share tasks on a processor in the SVP model. If a compiler knows that a unit of work will execute exclusively on a set of processors, it can statically determine resource usage in terms of depth of recursion in the unit of work (i.e. number of levels in the concurrency tree) and hence set an optimal number of threads per block at each level. This resource management is independent of the number of processors, as the block size for creating a family is defined as the maximum number of threads per family allocated to a processor at any given time. Finite recursion can be fully analysed; unbounded recursion, must use dynamic and potentially less efficient methods to avoid the resource deadlock that occurs when one of the following is fully utilised with recursive creates (register stack, thread table or family table).

3.2 Resource Management

An SVP create takes as its main parameter a thread definition. This defines the code that is executed by each thread. The family's extent is defined by a triple of {start, limit, step} which defines the number of threads and the index value of each, and finally, a block size that limits the number of threads allocated to a processor at any one time. In addition to this, create has a parameter that returns a family identifier to identify it and a place that defines where the family will execute. The example below in the language μ TC [22], creates a family

of 100 threads (index = 0..99 by 1) at the default place (second parameter, which is elided) with no more than 10 threads per processor.

```
family f1; ... create(f1; ; ; 0; 99; 1;10) thread_def(...);
```

In order to discuss resource management, the SVP's implementation of mutual exclusion must be defined. In conventional shared-memory programming, exclusion is implemented by acquiring a lock, entering a critical section and then releasing the lock. This is an extremely inefficient way of implementing mutual exclusion, especially if polling is required in the acquisition of the lock. In SVP, mutual exclusion is implemented through places. A an exclusion bit in a place variable, when set, ensures that no more than one unit of work can be created at that *exclusive* place at a time. Multiple concurrent create's to that place are serialised by queuing. This is an elegant and efficient mechanism, which can apply backpressure through the queue and network used for delegation so that under extreme load the creating thread will simply suspend on its create instruction. In the remainder of this paper we assume a System Environment Place (SEP), which is the exclusive place where a model of processor usage is maintained and updated by serialising requests from concurrent threads. The following example illustrates how a thread acquires a new place and delegates a unit of work to it.

```
family f1, f2; place new_place; int num_proc=8;
create (f1; SEP;; 0; 0;1;1) SEP_allocate(new_place, num_proc);
sync (f1);
if (num_proc >0 ){
    create (f2; new_place; ; 0; 99; 1; 10) thread_def(...);
    ...
    sync (f2);
    create (f1; SEP; ; 0; 0;1;1) SEP_release(new_place);
}
else ...;
```

In this example *create* translates to the same SVP instruction implemented in the DRISC processor, which returns a family id when it has a family table entry and a return code when complete. The *sync* simply waits on the return code's register and *family* and *place* are implementation-dependent, non-computational types for a family and a place respectively. This thread creates an instance of *SEP_allocate* at the SEP, requesting a place comprising 8 processors. This request is non-binding and if fewer than 8 processors are available the thread may return a place comprising fewer processors than requested. The interface definition of this system environment function is as follows:

```
thread p_alloc( shared place a_place; shared int req_proc) {};
```

Here the keyword *shared* identifies synchronising (register) variables between the creating thread and system function, in this case a place that is "returned" and the number of processors requested, *req_proc*. This thread uses the number of processors requested, its map of processor usage and an allocation strategy to decide how many processors to allocate,

it updates *req_proc*, configures those processors into a cluster and sets the address of the root processor as a component of the variable *a_place*. At the sync on family *f1*, both shared variables can be used by the creating thread. If processors were allocated to this thread, it then delegates a unit of work to them and on completion it frees the place using the system function *SEP_release*, so that the resource model can be updated.

Figure 1 illustrates the fact that an SVP program is hierarchical. Under certain conditions a different resource management strategy may be required at some node in this tree, which has an understanding of the nature of the algorithm being implemented. Also a single SEP may become saturated with allocation requests to the detriment of the system's performance. Below therefore, we illustrate how easy it is for any thread to set up a new SEP to allocate resources from a pool acquired from the original SEP.

```

family f1, f2; place new_place; pool new_pool; int num_proc=64;
/*allocate pool of processors*/
create (f1; SEP;; 0;0;1;1) SEP_partition(new_pool, num_proc);
sync (f1);
if (num_proc >0 ){
    /*initialise new SEP using that pool and redefine the SEP*/
    create (f2;local;0;0; 1;1) SEP_init(new_place, new_pool);
    SEP = new_place;
    ...
    num_proc=8;
    create (f1; SEP; ; 0; 0) SEP_allocate_local(new_place, num_proc);
    sync (f1);
    if (num_proc >0 ){
        create (f2; new_place; ; 0; 99; 1; 10) thread_def(...);
        ...
        sync (f2);
        create (f1; SEP; ; 0; 0;1;1) SEP_release(new_place);
    }
    else ...;
}
else ...;

```

In this example a new SEP is created to control the allocation of processors in some subtree of a program. To do this another non-computational type, *pool*, is used and a request is made for the allocation of a pool of processors using *SEP_partition*, which allocates an unconfigured set of processors. This returns the pool *new_pool*. A new SEP is then initialised using this pool, which involves setting up an SEP database and assigning a place, *new_place* where mutual exclusion in allocating resources will occur. Allocation to sub-families can then proceed in exactly the same manner after redefining the variable SEP.

This discussion outlines the basic protocols for hierarchical resource management in microgrids. An implementation may well be more complex and may implement contracts over time, markets to determine the cost of resources and even futures on resources. That discussion however, is outside the scope of this paper.

3.3 Memory protection

In a traditional operating system, the basic unit of execution is the process, which encapsulates mechanisms for addressing and memory protection, both of which are solved by allocating a private virtual address space to the process. This approach of using a private virtual address spaces carries with it two main advantages; it increases the amount of address space available to each process and it provides memory protection boundaries between concurrently executing processes. There are also disadvantages in this approach, as it becomes difficult for concurrently running programs to cooperate at a fine level of granularity. Pointers in one address space are meaningless in another and some means for translating pointers across address space boundaries is necessary. The alternative is that the processes execute in the same address space with no protection.

Where processes are independent jobs, as is the case in single processor systems, this is not an issue but in SVP, we have a model where concurrent units of work can be defined to an arbitrary level. Also the management of concurrency is in the hardware and concurrent threads in different virtual address spaces may be executing in the pipeline at the same time and hence thread interleaving (potentially every clock cycle) would require a new translation table to be used invalidating the contents of the TLB and possibly the cache.

With a 64-bit address space, addressing range is not a big issue and we are left with the requirement of protecting concurrently executing units of work from each other. In many high-performance concurrent systems even this requirement is dropped. However, to make concurrent systems mainstream, some form of protection is imperative, as we will surely have multiple users executing jobs concurrently on the same chip, even if they may be executing on different processors. The shared memory model in the SVP would still allow them to interfere with each other. An alternative approach to virtual memory and corresponding address translation is the concept of a single-address-space. A Single Address Space Operating System (SASOS) manages one system-wide address space. This approach is predicated on the fact that addressing and memory protection are two different issues, which can be solved by separate mechanisms. In a SASOS, memory protection is decoupled from addressing. In many SASOS systems, including Mungi [23], Opal [24] and Sombrero [25], processes run in a protection domain (PD). Thus, even though a process is capable of addressing any data item throughout the entire system, the OS is able to check whether a memory access should be allowed and raises a protection exception in the event of an invalid access. The advantage of this approach is that it facilitates the safe sharing of data between processes, which don't fully trust each other.

In DRISC implementations of SVP we are investigating the SASOS approach. We define memory *islands* as contiguous ranges of memory using base-range pairs to which access rights have been set (e.g., read, read/write, destroy, etc.). A memory protection *domain* is then defined as a list of islands that a unit of work is allowed to access. The novelty of this approach in SVP is that this now becomes an integral part of instruction processing. Every family is associated with a protection domain and this information is used on any instruction, which misses the L1 cache to access the protection table. We also use a small cache on this table similar to a conventional TLB, which caches protection table entries.

Domains are added to the SVP model in the same manner as places for processor resources. A create associates a protection domain with the unit of work that it is creating. All threads in that family will use the same domain. Similarly, threads in families they create will use the same domain unless a new domain is specified. All the examples above in μ TC use the *default* domain (parameter elided). To specify a new domain, an implementation-specific, non-computational type *domain* is used in the third parameter to create. Domains can be created destroyed, added to etc. using system threads created at the SEP.

```
family f1; place new_place; domain new_domain;
...
create(f1, new_place; new_domain; 0; 0;1;1) new_job(...);
...
sync(f1);
```

3.4 Security

In DRISC implementations of SVP, we have implemented an operating system kernel in silicon. We have implemented actions that are extremely powerful and need to be secured against unintended or even malicious use. In a conventional operating system, these actions are performed by users or user scripts and are secured by a user's login authentication. Within a user's environment, he or she can do as they please. Typically, only at the level of the file system does any sharing occur between users. In SVP a new solution is required, as we must support security from the hardware up. It should be noted that this approach will also make systems and software built on top this kernel more secure. There are a number of examples in the above discussion that we can identify as security issues:

- i kill and squeeze actions should only be initiated by the thread (or its delegate) that created a family being squeezed or perhaps as a last resort by some higher-level authority to enforce a contract on the use of resources the family is using;
- ii units of work should only be created at a place by the thread that was granted the resource by the SEP, again the use of that place may be delegated;
- iii access to protection domains should only be allowed by a thread that created the domain or a thread that was granted access to the domain by its creator.

In SVP, the approach we take is based on the concept of a capability [25]. Moreover this approach is built in at the level of instruction execution, providing the ability to build efficient and secure capability-based systems. A capability can be likened to a key that must be presented in order to gain access a certain object, service or facility. In our case, the key is a sufficiently large random or pseudo-random variable incorporated into the non-computational types of family, place and domain. Only a successful match on that key will succeed in the execution of create, kill or squeeze.

When a family is created, the thread creating it is returned a family identifier, *Fid*. This identifier must perform two functions. The first is to identify the family within its context, in a DRISC microgrid, this reduces to a processor identifier, *Pid*, and the index into that processor's family table, *FTi*. Thus the *Fid* comprises the tuple: {*Pid*, *FTi*}. In implementing kill and squeeze actions we noted the need to recycle family table entries as *FTi* is not unique

over time. As these actions are asynchronous, some additional identification is required to secure these operation, even in a benign environment. The second requirement on Fid therefore, is to provide a key to distinguish between different instances of the same family table entry on the same processor. It is a small step from here to defining this as a capability. On executing a create action therefore, the hardware generates a random key, *Cap*, of some length, which is stored along with the other information in the family table. Fid now becomes the tuple: {*Pid* , *F_{Ti}*, *Cap*} and to execute a kill or squeeze action, a thread must present the complete identifier, from which *Cap* is extracted and matched with the entry stored in the family table. As we have seen in the example given on page 8, the family identifier (capability) is delegated from the creating thread to a control thread in order to build a simple distributed system. Only in this way can these reflective actions be implemented in a manner that can be guaranteed to be secure. It should be noted that passing Fids via shared variables is also guaranteed to be secure as the channel cannot be eavesdropped upon. A shared variable is a register, whose access is hardwired as the sum of the thread's context location in one or more register files and an offset provided in the binary code. The context information is held in the thread table and is accessible only to the two threads sharing that variable!

Points (ii) and (iii) above are solved in a similar manner. In (ii) a place must identify a *Pid*, which is the root processor of the cluster, to which the delegated create message is sent. This processor also generates a random key of some length, which is returned to the SEP on configuration of the cluster and then passed to the requesting thread as a component of the place. In order for the delegated create to succeed at that place, a *Cap* component of the place must match that stored at the root of the cluster. Finally in (iii) a capability must be matched in order for the domain id (*Did*) to be added to the family table, from which it is used to authenticate requests to memory.

4 Conclusions

In this paper we have outlined the approach we are taking in developing DRISC microprocessors that support concurrency control instruction at the level of its ISA. These processors are specifically designed to provide a coherent hardware infrastructure for multi- and eventually many-core chips. As we proceed with implementations of these processors, we have been surprised at how disruptive this approach has been. Not only are we faced with developing new compilers for these processors to support code generation to the SVP actions, we have also had to consider basic operating system support in implementing these actions. In short we have had to implement the basis of an operating system kernel in the foundation to these actions.

This has not been a deterrent to our work, although it has caused some divergence in our team's directions, as we understand that in order for parallel computing to become mainstream, concurrency support, and this implies operating system support, must be built into the processor's instruction set. We are convinced that at the end of the day this will result in simpler, more reliable and more secure computer systems.

5 References

[1] Krste Asanovic, et. al. (2006) The Landscape of Parallel Computing Research: A View from Berkeley, Technical Report No. UCB/EECS-2006-183

- <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>
- [2] D May and R Shepherd (1984) The transputer implementation of occam, Proc. Intl Conf on Fifth-Generation Computer Systems, Tokyo, pp533-541.
 - [3] E lee (2006) The problem with threads, IEEE Computer, 36(5), pp. 33-42.
 - [4] A Bolychevsky, C R Jesshope and V B Muchnick, (1996) Dynamic scheduling in RISC architectures, IEE Trans. E, Computers and Digital Techniques, 143, pp 309-317.
 - [5] C R Jesshope (1992) The f-code abstract machine and its implementation, Proc COMPEURO 92 (IEEE Press) pp 169-174.
 - [6] V B Muchnick and A B Shafarenko (1996) Data Parallel Computing - the Language Dimension, ISBN: 1 85032 179 5, Chapman Hall.
 - [7] Jesshope C. R. (2006) Microthreading a model for distributed instruction-level concurrency, Parallel processing Letters, 16(2), pp209-228, ISSN: 0129-6264.
 - [8] R. Allen and K. Kennedy (2001) Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, ISBN 1-55860-286-0.
 - [9] H Kasahara, M Obata, K Ishizaka, K Kimura, H Kaminaga, H Nakano, K Nagasawa, A Murai, H Itagaki, and J Shirako (2002) Multigrain automatic parallelization in Japanese millennium project IT21 advanced parallelizing compiler, PARELEC '02: Proc. of the Intl. Conf. Parallel Computing in Electrical Engineering (Washington, DC, USA), IEEE Computer Society, p. 105.
 - [10] J W Moore (1983) The HEP Parallel Processor, Los Alamos Science, Fall 1983, pp 72-75. <http://library.lanl.gov/cgi-bin/getfile?09-04.pdf>
 - [11] Alverson, R. et al. (1990) The Tera Computer System, Proc. of the 4th International Conference on Supercomputing, Amsterdam, The Netherlands, 11-15 June, pp. 1-6. ACM Press, New York, NY, USA.
 - [12] J W Berry, B A Hendrickson, S Kahan and P Konecny (2006) Graph Software Development and Performance on the MTA-2 and Eldorado, 48th Cray Users Group Meeting, Switzerland, May 2006.
 - [13] G M Papadopoulos and D E Culler (1990) Monsoon: An Explicit Token Store Architecture, Proc. 17th International Symposium on Computer Architecture, pp82-91
 - [14] G M Papadopoulos and D E Culler (1998) Retrospective: Monsoon: An Explicit Token Store Architecture, In 25 Years of the International Symposia on Computer Architecture: Selected Papers, pp. 74—76.
 - [15] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind (2005) Optimizing Compiler for the CELL Processor, Proc 14th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT), pp 161–172, Washington, DC, USA, 2005. IEEE Computer Society.
 - [16] Richard A. Hankins, Gautham N. Chinya, Jamison D. Collins, Perry H. Wang, Ryan Rakvic, Hong Wang, John P. Shen (2006) Multiple Instruction Stream Processor, Proc 33rd International Symposium on Computer Architecture (ISCA'06), pp. 114-127.
 - [17] R Nikhil Arvind and K Pingali (1989) I-Structures: Data structures for parallel computing (1989) *ACM Transactions on Programming Languages and Systems*, **11**.
 - [18] T.D Vu and C. R. Jesshope (2007) Formalizing SANE virtual processor in thread algebra, in M. Butler, M. G. Hinchley and M. M. Larrondo-Petrie, eds. *ICFEM 2007*, pp 345-365.
 - [19] H L Muller, P W A Stallard, D H D Warren (1995) Hiding Miss Latencies with Multithreading on the Data Diffusion Machine, Proc. 1995 Intl. Conf. on Parallel Processing, ICPP'95, Volume I, ISBN 0-8493-2615-X, pp178-185.
 - [20] G R Gao and V Sarkar (2000) Location consistency-a new memory model and cache consistency protocol Computers, IEEE Trans. Comput. 49 (8), pp798– 813.
 - [21] M. W. van Tol, C. R. Jesshope, M. Lankamp and S. Polstra (2008) An implementation of the SANE Virtual Processor using POSIX threads, submitted to Journal of Systems Architecture, see: http://www.science.uva.nl/~jesshope/papers/SVP_pthread-article.pdf

- [22] Jesshope, C R (2006) μ TC – an intermediate language for programming chip multiprocessors, Proc. Pacific Computer Systems Architecture Conference 2006 - ACSAC06, ISBN 3-540-4005, LNCS 4186, pp147-160.
- [23] G Heiser (1998) The Mungi Single-Adress-Space Operating System, Software: Practice and Experience, 28(9), pp.901-928.
- [24] J S Chase et. al. (1994) Sharing and Protection in a Single-Address-Space Operating System, ACM TOCS, 12,(4), pp.271-307.
- [25] A C Skousen and D S Miller (1998) Operating System Structure and Processor Architecture for a Large Distributed Single Address Space, Proceedings of PDCS'98: 10th International Conference on Parallel and Distributed Computing Systems, pp 631-634.
- [26] R Fabry (1974) Capability-based addressing, *CACM*, 7 no 7, pp 403-412.