On the compilation of a parallel language targeting the self-adaptive virtual processor

Bernard, T.A.M.

Citation for published version (APA):
Chapter 3

Self-Adaptive Virtual Processor Execution Model and its implementations

Moving targets may be fine in hunting, when it comes to generating a compiler this is rather inconvenient to say the least.
Sven-Bodo Scholz, University of Herfordshire, UK

Chapter 2 presented other works in parallel computing and comparisons with our approach described in this chapter. We describe an abstract concurrency execution model named the Self-Adaptive Virtual Processor Execution Model as base of our parallel computing system. The CSA group works to define and to refine this model which is a candidate for the exploitation of concurrent systems. This execution model is the underlying cement of the research undertaken in this thesis; the following chapters use it as a base of the SVP concurrency paradigm.

The contents of this chapter are based on these publications:

We discuss the SVP concurrency model with its concurrency properties such as exposition of parallelism, mechanisms of synchronization, ways of communication, etc. Moreover, we present the other derivations from this execution model such as the architecture and language implementations. We then present other implementations of the SVP model such as high-level simulator (using Pthreads technology). At the end of this chapter, we provide an evaluation (with results) of the SVP parallel computing system comprising a hardware implementation as Microgrids and a set of hand-coded scientific problems.

3.1 Our approach to multicore programming

We, i.e. the CSA group, work on the SVP execution model and its derivations in the hardware, language and compiler implementations. We tackle the “multicore programming menace” by following a three-fold approach illustrated in Figure 3.1 software, compiler and architecture. In this chapter, we present the separation of concerns with this approach which allows us to tweak and distribute responsibilities at different levels in the toolchain.

Figure 3.1: Overview of the SVP parallel computing system where the model is implemented into a language implementation as \(\mu TC\) and into a hardware implementation as the Microgrid. The tools (in rounded containers) depicted here are: the compiler translating from \(\mu TC\) to Microgrid and implementing the SVP paradigm; the simulator of the Microgrid architecture with cycle-accurate emulation.

Our group’s research goal is to provide a whole parallel computing system derived from the SVP execution model and evaluate it with appropriate benchmarks and applications. The software implementation encodes the SVP properties which permit the exposure of explicit parallelism in the source code of an application; however, there is no need to deal with the distribution of concurrency and management of resources. The architecture implementation takes care of solving these issues; it then removes from developers the concerns related to concurrency and resource management. Furthermore, the SVP-aware
compiler preserves and potentially optimizes the concurrent applications targeting the Microgrid platform.

### 3.2 Presentation of the SVP execution model

The SVP execution model [58] offers a uniform means of capturing dynamic concurrency. This is done with a parameterized `create` action that forks a named family of identical blocking threads as illustrated in Figure 3.2. This may be applied recursively; hence, the model’s bulk synchronization is by named family (`sync` action in Figure 3.2). An SVP thread may contain a number of synchronizing objects which provide dependency constraints on the execution of that thread. These objects are set by one thread and read by its adjacent thread, creating dependencies between asynchronously executing threads. These dependency constraints guarantee locality and freedom of deadlock (see article [59] for formal proofs).

![Figure 3.2](image)

**Figure 3.2**: Illustration of an SVP family creation event with multiple concurrent control flows of threads. The child family is created at the issue point in the parent thread. All threads of the child family are terminated after the completion point. Note that this figure does not illustrate the inter-thread communication.

#### 3.2.1 Thread family and creation

All SVP threads are grouped in families as shown in Figure 3.3. A hierarchy appears where a parent thread A is the creator of a family of child threads B. Each child thread is identical in the task they accomplish. In essence, an SVP family is a parameterized group of statically homogeneous, but dynamically heterogeneous threads that are created en masse. The creation of child threads is performed in the order of their index, i.e. B(1), then B(2), and B(3) in Figure 3.3. The threads’ executions are constrained with the dependencies they may contain in the description of their tasks. We focus on these dependencies in Section 3.2.2.
Figure 3.3: An SVP family is a group of child threads created by a parent’s thread. The figure represents a parent A creating a family of 3 child threads B. A family of threads may be bounded to a set of resources. This figure does not illustrate the relationships between threads.

An SVP thread creates a family of threads with a parameterized create action as illustrated in Figure 3.2 in a fork manner at the issue point. At this point, two separate domains are present: the parent and the children. Each SVP thread has its own flow of control and is only bounded in its execution by the exposition of data dependencies on other threads’ execution. The completion point of the family is done with the sync action with the join manner in the parent thread. There, all child threads are terminated; all control flows converge at this point. Every SVP thread can create families of its own; this allows composition and therefore makes the model hierarchical.

Illustrated with Figure 3.3, a family is characterized by the index sequence of the threads to be created, a reference to the thread body and a definition of unidirectional synchronization communication channels from, to and within the family.

3.2.2 Inter-thread communication

The SVP model provides explicit ways to expose the communication which is available during program’s execution. Inter-thread communication is accomplished through synchronized communication channels. In other words, “synchronized” means that a channel uses a dataflow mechanism with blocking reads and non-blocking writes. More precisely, there is a synchronization event linked to a write in a communication channel. The first read from this channel will then consume this synchronization event and then unblock thread’s execution waiting for the data to come through the channel. After the first read though, i.e. the other reads from the same channel, there are no other synchro-
organization events on this channel.

Figure 3.4: Example of an SVP inter-thread communication with a global communication channel. The communication resembles to a star-like pattern where the channel is written in the parent thread. There, the data is distributed over the channels towards the child threads where it will be read. The figure exposes the reads and the writes on the channel. The blocking read property of the SVP model disables the thread’s execution until the data becomes available in the channel. A global communication channel is read-only in the child thread.

In Figure 3.4, we look at the mechanism of the global synchronized communication channel. The parent thread A creates a family of threads and defines only one global channel in the parameters of the create action. This channel is unidirectional from the parent thread to the child threads. More precisely, it is written only once by the parent; it is read-only for the child threads. Each child thread gets a copy of the data propagated in this channel from the parent thread. Therefore, the pattern of this channel resembles a star with as many points as child threads.

Figure 3.5 introduces the shared synchronized communication channel. This channel has two distinct ends in the figure: an incoming end of the channel symbolized with read(y.s) and an outgoing end with write(y.s). Also defined in the parameters of the create action, the dependency chain starts in the parent thread where the outgoing shared channel is written once and is available for reading to the first indexed thread of the family in the incoming shared channel. The shared channel is implemented with a producer-consumer mechanism; it only describes communication between adjacent threads (i.e. thread with indices $i$ and $i+1$), parent and the first indexed thread, the last indexed thread and the parent. The thread writes once in its outgoing shared channel and the
consecutive thread only reads from its incoming shared channel. Consequently, this communication channel looks like a ring where each thread is an edge of the communication pattern. The first read of this channel is synchronized (un-blocking thread’s execution); only the first write to it is taken into account.

![Diagram of SVP inter-thread communication with a shared communication channel](image)

**Figure 3.5:** Example of SVP inter-thread communication with a shared communication channel. The communication resembles a ring-like pattern. With this communication channel, write and read accesses are synchronized and work with a producer-consumer mechanism. The write in the shared channel is made by the producer thread. The data is sent over the channel to the consumer thread (the adjacent thread with an incremented index value $i+1$) where the first read on it is synchronized. The shared communication channel, by definition, is written once; it can be read multiple times.

These two ways of communication expose dependencies between the threads as summarized in Figure 3.6. Because of this explicit exposure of the dependency chains, the SVP model is deadlock free and guarantees locality. The formal proofs of this property is in [59].

### 3.2.3 Thread and family interruption

An SVP family can be interrupted during its execution in two different ways. The first way is with the usage of the `break` action similar to the break semantics in the sequential model. This action is done within one of the threads of the family; when it occurs, the rest of the threads to be executed are stopped. Then, the `break` permits the transfer of an object for the breaking thread back to the parent thread. This is necessary when handling exceptions in thread families.
Figure 3.6: Example of SVP inter-thread communication. After the issue point of an SVP create event in Parent A, this is an example of SVP inter-thread communication using the two different SVP synchronized communication channels, i.e. global synchronizing object $x_g$ and shared synchronizing object $y_s$. Assuming here that parent A generates a family B of three threads with two thread parameters: $x_g$ and $y_s$. The inter-thread communication is exposed with accessor methods to the synchronizing objects: read(Var) and write(Var) (they are not function calls) where Var is a synchronizing object. Note that the last read($y_s$) (between child B(3) and Parent A) is valid in Parent A only after synchronization (completion point).

Furthermore, it is necessary where unbounded loops (e.g. while-loops) are possible in the sequential model; with the SVP model, a family of threads can be allocated in blocks of a given size and terminated in one of its threads by the break action. In principle, unbounded families are possible, however this is constrained by a dynamic space/time trade-off.

The second way of interrupting thread families is possible with the kill action from outside the targeted family of threads. The kill action employs the family identifier and destroys the entire set of threads of the targeted family without preserving any of the temporary results. Typically, this action is useful when an operating system wants to destroy/stop specific thread families when an unexpected event occurs.

The difference between these two actions is that one can be performed within the family of threads (with the break action) and then return an object to the creating environment. The other action is only performed from outside the thread family and targets the entire family, not only one single thread.
3.2.4 Place of computation

The SVP model provides a separation of concerns between the definition of a program’s function and its resource mapping. The latter is achieved by binding a named resource, called a place, to a family on its creation, which can happen at any level in the family hierarchy. The model defines a binding of unit of work to a set of resources with the create action. By definition, a unit of work is a family of threads. This family is then assigned to a set of resources on which to be run on. This set of resources is defined by place objects defined in the parameters of the create action. Places are opaque and implementation-defined. Depending on the implementation, a place can be a virtual resource or a physical one and can have various properties, allowing threads with specialized code to be run on heterogeneous systems via a common mechanism. Regardless of the implementation, there is a special class of place that must be supported, the exclusive place. This class has the property that of all families created at that place, only one will run at a time. Exclusive places provide synchronization between unrelated threads and add non-determinism to the execution model. Typically, they will be used by system code, e.g. in sharing or allocating resources.

3.2.5 Memory consistency model

The SVP model uses a relaxed memory consistency model that is managed by create and sync events and possibly by writes to synchronizing objects. It provides a single, flat address space with a restricted consistency model, such as Location Consistency (LC) [60]. A thread in a family cannot reliably see memory writes made by other threads except for those that its parent could see at the point where the family was created, those that any thread of a subordinate family can see after that family has terminated, and those dependent on a synchronizing object used as a reference when that object is written. The model requires LC’s acquire and release operations to be implemented on synchronization actions such as creation and termination of threads and writing to and reading from synchronizing objects. There is no way for threads to explicitly synchronize access to any memory location with another, unrelated thread.

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

- for references passed through a global channel, threads in a family can only see and use what the parent thread could see at the point of the create, as long as the parent does not change the memory before the family terminates.
• the parent thread cannot see the changes made by the child threads until it has synchronized on the family’s termination;

• a thread cannot see the writes made by a previous thread in the family except for writes to objects which are sent via shared channels.

• in order to share data between unrelated threads, all threads must create a family on the same exclusive place to synchronize consistent access to a shared location.

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

3.2.6 Concurrency tree

Every SVP thread can create families of its own; this makes the model hierarchical. Therefore, an SVP program is represented with a hierarchy of families. This hierarchy resembles a tree of thread families and their organization, and consequently the concurrency present in that SVP program. This representation is called a concurrency tree where each node is a thread which creates a family of threads, called a creating thread. A leaf of the concurrency tree represents a non-creating thread or leaf thread. Consequently, an SVP program is composed of families of creating threads and families of non-creating threads.

![Concurrency Tree Diagram]

Figure 3.7: Representation of a program’s concurrency via a concurrency tree. Each circle is a thread; circles positioned at the same level and with the same direct creator is of the same thread family. The concurrency tree depicts all the thread families of an SVP program including creating threads (i.e. node in the tree) and non-creating threads (i.e. leaf in the tree). Furthermore, the tree contains dependency chains between the threads of a family and with the ascendancy (i.e. parent thread).

Figure 3.7 is an example of a program’s concurrency tree. On the left-hand side, nodes including leaves are exposed with their single ascendancy and their
descendants (only for nodes). We name the ascendant of a family of threads the parent thread. In this figure, each circle is a thread; circles, positioned at the same level and with the same direct parent, are of the same thread family. A parent thread is therefore a creating thread. Moreover, a thread \( A \) can create a family of threads \( B \). These threads \( B \) are named children of thread \( A \). On the right-hand side in Figure 3.7 we focus one thread family that comprises the children seen as leaf threads in the concurrency tree and the parent thread. The concurrency tree contains information regarding to dependencies between parent and children, plus between threads of a same family. These dependency chains are global and shared communication channels.

Relevant information is extracted from the tree related to the program concurrency. For instance, the depth of a program’s concurrency tree, calculated from the first parent at the top of the concurrency down to the lowest of the leaves, reflects the level of nested families which can be a problem with resource mapping and potentially resource starvation when too many thread families are nested. Other information related to the program concurrency, such as the degree of creating families and non-creating families, is useful to developers when exploring concurrency patterns of a program.

3.3 Hardware implementation: Microgrid

A Microgrid is a chip-multiprocessor using microthreaded processors (also referred to as SVP cores) that implement the SVP model directly in hardware [61]. This section explains both the chip architecture and the processor architecture, which is based on a DRISC processor [43] and executes an extended standard instruction set. This section describes the architectural features of the SVP multicore implementation as far as necessary to understand the constraints on our compiler development.

3.3.1 The Microgrid: extension of an existing ISA

The SVP core implementation extends an existing ISA with extra instructions capturing the model properties. These instructions are shown in Table 3.1. In addition to that, the SVP core also implements registers as i-structures [62], which can suspend and reschedule any number of hardware supported threads (currently 256 per core), further implementation details are presented in [61]. Parameterized thread creation is implemented with the ALLOCATE and SET-like instructions that acquire and set a family table entry (i.e. an on-chip memory structure that holds information for thread management at the family level), followed by a CREATE instruction that reads the family table entry and terminates asynchronously by setting a register (the family’s SYNC).

As a side-effect, it iterates the creation of all threads defined by the family table entry (e.g. maximum number of executing threads per core at any time).
This creation is constrained by the resources available or as specified in the family table. A minimum resource set is one thread table entry (i.e. similar to family table but at the level of threads) and one register context on a single core, which yields sequential execution. However, a family of threads created can be distributed, as far as resources are available, to a number of cores for throughput and to a number of threads per core to support latency tolerance.

<table>
<thead>
<tr>
<th>allocate</th>
<th>gets a family entry in memory and sets the context and the place where the family will be run on.</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>action which generates a family of threads.</td>
</tr>
<tr>
<td>swch</td>
<td>marks the use of long-latency-operations results.</td>
</tr>
<tr>
<td>end</td>
<td>marks the end of the thread code.</td>
</tr>
<tr>
<td>break</td>
<td>within a thread code, terminates the current family context and returns the value to parent.</td>
</tr>
<tr>
<td>kill</td>
<td>from outside the targeted family, terminates the current family context.</td>
</tr>
<tr>
<td>setstart</td>
<td>sets the starting bound of family.</td>
</tr>
<tr>
<td>setlimit</td>
<td>sets the limit bound of the family.</td>
</tr>
<tr>
<td>setstep</td>
<td>sets the step of the family.</td>
</tr>
<tr>
<td>setblock</td>
<td>sets the maximum number of threads per code.</td>
</tr>
<tr>
<td>setbreak</td>
<td>sets an object in parent’s context for collecting family breaking value.</td>
</tr>
</tbody>
</table>

**Table 3.1**: List of SVP instructions which can be added to an existing ISA.
3.3.2 Processor overview

A microthreaded processor (i.e. SVP core) uses an in-order issue pipeline with both in-order and out-of-order completion of instructions. The core principle of the processor is to avoid stalls and speculative execution in order to maximize utilization of the pipeline and energy efficiency, respectively. Cache misses, family and thread creation, synchronization and FPU operations are all implemented by issuing the operation without keeping track of outstanding dependencies in the pipeline, and continuing to execute instructions from the same thread. Any instruction that attempts to use the result of an operation which has not yet completed has its thread suspended on the target register and is woken up when it is written. The register file is modified by adding state bits to each register and logic for handling these states on reads and writes. This ability to execute many threads in a single pipeline without stalling gives a microthreaded processor the ability to hide large amounts of latency. To support this number of threads, the processor has data structures to store and manage the families and threads and a relatively large register file to hold all of the threads’ contexts. Dynamic allocation of register contexts to families allows the thread context size to be tailored to its code, optimizing register file usage.

Microthreaded processors employ the following data structures to manage thread families and threads:

- **family table**, on-chip memory structure that holds information for thread family management.
- **thread table**, on-chip memory structure that holds information for thread management.
- **active list**, gathers the list of threads that are currently active in the pipeline of an SVP core.
- **waiting list**, collects the list of threads that are blocked until the data they wait for is delivered.
- **ready list**, lists the threads are just ready to be executed in the pipeline and wait to be allocated.

3.3.3 Microthreaded pipeline

The pipeline in an SVP core is a simple in-order issue RISC pipeline with modifications. When the Read stage determines that one or more of the operands of an instruction are not available, it will change the instruction to a no-op that writes the thread’s ID back to that register and marks it Waiting. The Fetch stage of the pipeline uses the processor’s Active List to get the next thread to execute from. It will read the Thread Table, Family Table and I-Cache and buffer the information while it executes that thread. A switch to another thread is required when:
• the current thread reaches the end of the cache-line,
• the thread executes a jump or branch,
• an instruction in the thread uses a register that is not Full, or
• an instruction has been annotated with a SWCH or END.

In the first two cases, the thread proceeds to a cache-line that may not be present in the I-Cache and to avoid stalling the pipeline, execution switches to the next thread on the Active List. At the end of the pipeline, the thread is put onto the Ready List to fetch its new cache-line. In the third case, the thread needs data that is not present and cannot continue. The thread will suspend on the register and be put on the Ready List when the register is written. The SWCH instruction is used after an operation to mark the use of previous long-latency-operations’ results. In the case of a result that is not ready yet, the thread’s execution is blocked until the data becomes available. In the meantime, another thread’s instructions are issued to the pipeline.

3.3.4 Family and thread management

SVP threads have different states as illustrated in Figure 3.8. Thread life starts when is been allocated; its state goes from empty to waiting. The thread then goes in its ready state as soon as its instruction data is being processed. After that, the thread becomes running when it enters the pipeline. At that state, several cases can happen. First of all, the thread executes its computations completely and it ends in the unused state where it and its context will be deallocated and put in empty. However, if at the running state, data is unavailable the thread will be suspended state until data becomes written and put in as waiting. Consequently, the thread will be put in reactivated in its ready state. Another case can happen when data is there but there is a thread switch tag (SWCH) onto the ongoing operation; then, the thread’s state becomes waiting and then put again in ready state. These two cases trigger a thread switch (SWCH) which activates another thread from the ready state to be put as running, if there is any.

When a thread creates a family, an entry in the Family Table is allocated to store the family’s state. This state includes the number of threads created so far for the family, parameters for creating new threads and a link to the parent thread. An independent hardware process is responsible for creating the threads of created families, up to a program-specified upper bound. All non-register state for the threads created on a processor is stored in the Thread Table. Threads start as soon as they are created and terminate after executing any instruction annotated by the special END annotation. When a thread terminates, its context is reused for the next thread in the family unless all threads in the family have been created. All resources related to a family are released when all threads in that family have terminated and no more references to the family exist. Threads can be in different states besides running and threads in one state.
may need to be moved to another state en masse. To do this efficiently, these states are managed with a linked list with a field in the Thread Table, such that moving threads from one state just means appending one linked list to another. This mechanism is used in the following cases:

- When a register is written that has several threads waiting on it, these threads are woken up by taking the head and tail pointer from the register and appending that list to the processor’s list of threads that should be executed, the Ready List.

- Threads that are waiting on an I-Cache line for their instructions are linked in a list with the head pointer in the cache-line entry itself. This allows the I-Cache, when the line has been read, to immediately make all threads that were waiting on it available to the pipeline via the Active List.

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

3.3.5 Thread context and semantics of objects

Each SVP thread has its own context of registers (register classes are listed in Table 3.2 and explained in Section 3.3.6) and private/shared memory areas. Inside a thread context, there are two types of objects present: conventional
objects which are not synchronized, and the second type is related to the synchronizing objects implementing SVP model communication channels named shared and global channels. The model’s shared objects are identified in the code as a subset of a thread’s register context, while register file address decoding transparently implements inter-context dependencies. This works in a similar manner to windowing in the SPARC architecture [63], except the windows are not fixed, can be written to concurrently and are automatically distributed between the adjacent register files in a cluster of cores when a create instruction is distributed. In particular, the ALLOCATE instruction identifies the cluster of cores that a CREATE instruction will be distributed to and this requires an implementation-dependent place type. Section 3.3.7 explains the layout of the register window and its mapping onto the architectural registers.

### 3.3.6 Thread communication in the Microgrid

SVP communication is implemented with unidirectional synchronized communication channels, previously explained in Section 3.2.2. More precisely, these channels utilize synchronizing objects mapped to registers with states (implemented as i-structures [62]). One of the two ways of inter-thread communication is implemented by reading from and writing to shared synchronizing objects. This occurs between the parent and the first child thread and between adjacent threads created in a family, supporting only linear dependency chains, and between the last child thread and the parent as illustrated in Figure 3.5. This linearity ensures the model’s freedom from deadlock and also provides an abstraction of locality of communication between threads in a family. In the Microgrid of SVP cores, for every shared synchronization required between threads, the compiler must allocate two registers in a thread’s context, one which is read only and which waits for the previous thread’s write and one that is write once that signals to its successor thread in the family. The two registers are called the dependent and shared registers respectively (for example in Figure 3.6, read(y.s) and write(y.s) respectively mapped to dependent and shared). These names are derived from a partitioning of the thread’s context into four register classes as presented in Table 3.2. This partition is specified by an assembly directive (i.e. ALLOCATE, see in Table 3.1) at the beginning of the thread code. This data supplied is used on family creation to define a number of offsets into a thread’s register context that are then used to implement any communication required between distributed contexts, i.e. between a shared register on one core and a dependent register on another. This mechanism gives us a distributed-shared register file for all threads in a family where thread-to-thread communication is restricted to the overlapping windows of one thread’s shareds with the subsequent thread’s dependents.

The other SVP communication channel is implemented over the global registers which are read-only (in the child thread context). These registers are initialized during thread creation from the parent’s context utilizing a ‘pull’ mechanism. During thread creation, thread initialization takes (“pulls”) the created
family’s *globals* and *shareds* directly from the context of the parent thread. Figure 3.4 illustrates this with a single global synchronized communication channel in a family of 3 threads.

<table>
<thead>
<tr>
<th>Register classes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>locals</em></td>
<td>These registers are read/write to the local thread only.</td>
</tr>
<tr>
<td><em>globals</em></td>
<td>These registers are written by the creating thread and are read-only to all threads within a family.</td>
</tr>
<tr>
<td><em>shareds</em></td>
<td>These registers are for communication between adjacent threads in the family, they are <em>written</em> once by the local thread and may be read locally or by the successor thread in the family. The synchronization event is on the first write.</td>
</tr>
<tr>
<td><em>dependents</em></td>
<td>These registers are <em>read</em> only and access the previous thread’s shared registers. The synchronization event is on the first read.</td>
</tr>
</tbody>
</table>

**Table 3.2:** List of SVP register classes.

As the overhead of thread creation and scheduling in our implementation is small (from 2 to 10 cycles for the former and on every clock cycle for the latter), threads may be very fine-grained. It is not uncommon for a thread definition to comprise fewer than ten instructions. Thus, the normal approach of allocating a stack per thread from the heap is not an appropriate solution here. In SVP, where possible, every function is created as a thread of control and it would take as long as threads execution to allocate a stack from the heap (e.g. hundreds of cycles). Section 3.3.7 explains the register layout and their mapping onto the architectural registers.

### 3.3.7 Register window and register mapping

The previous section partly mentions the initialization of thread context at thread creation. In essence, an SVP register window comprises 4 register classes: global, local, dependent, shared as illustrated with Figure 3.9. Each virtual register window is a representation of the context of an SVP thread; this register window is mapped onto architectural registers as Figure 3.10 illustrates.

**Figure 3.9:** Layout of a virtual register window for an SVP thread. Four register classes ordered from the base to the top: Global, Dependent, Local, Shared.

Figure 3.9 depicts the structure of a virtual register window with a size N (e.g. the Alpha instruction set used as experimental target platform, 32 integer
registers and 32 floating-point registers). The global region starts from the base of the register window, then followed respectively by the dependent region, the local region and the shared region. The first two regions (i.e. global and dependent) are initialized at thread creation from the creating environment (i.e. parent’s context) for the first thread. For the following threads, the global region is initialized from the creating environment, whereas the dependent region is initialized from the shared region of the previous thread in the index range. Therefore, the last region (i.e. shared) is bounded with the dependent region of the adjacent thread (the following thread in the index range or back to the parent thread if the current thread is the last indexed thread from the family). The register window is named virtual since the context is specific to the thread it is associated. However, when the mapping, done by the architecture, occurs several cases might happen:

- The entire family is executed on the same SVP core. The globals of the same family then map to the same architectural registers. The shareds and dependents of adjacent threads are mapped onto the same architectural register. The locals have their own segment of architectural hard registers.

- The family is spread onto a cluster of SVP cores. The globals, used by the threads localized onto the same SVP core, are mapped on the same architectural registers. When a couple \{dependent, shared\} of two adjacent threads is executed onto two distinct SVP cores, the Microgrid takes care of distributing the data over the SVP cores. Implementation details are described in [61].

It is important to note that register mapping is completely opaque to the description of the program (i.e. the machine code). In other words, the way of execution of a family does not need to be known before execution-time. The Microgrid architecture allocates the distributed contexts (from the threads of the same family) considering the scheduling of the threads over the SVP cores. Figure 3.10 illustrates the case of the creation of a family of 3 threads localized onto the same place (i.e. same SVP core). At initialization, the parent’s context is required to store in its locals the parameters of the thread function being created. They need to be store in a contiguous frame; they are passed via registers if there is room for all of them. Otherwise, the thread local storage (similar to a conventional stack but with concurrency management) is used. To summarize, for threads with few parameters, these can be passed using shared or global registers, but this requires the compiler to make a register allocation within the necessarily limited context of registers. While this is generally possible at the lower levels of the concurrency tree, at higher levels, the combined number of local objects and parameters to a thread may still require a stack. Given the limited number of hardware threads per chip (256 for each core on a chip) we are able to provide a fixed partition of the virtual address space to act as local memory for a thread (thread local storage). Thus when required, this allows us to spill registers and pass parameters in a conventional manner. The overhead
Figure 3.10: Mapping of hardware registers to architectural registers: example sharing between a parent thread A and a child family B of 3 threads, where all threads are created on the same core. The child family B has been created with 2 global thread parameters, 3 shared thread parameters and 4 locals. The offset for the first local register (available for exclusive use by its thread) in the virtual architectural register window of the parent is 10, whereas it is 5 for the child thread. In each thread both the registers shared with the previous sibling (D i.e. dependent) and next sibling (S i.e. shared) are visible. For this is much smaller, since it only requires a single instruction to initialize a register with a stack pointer plus normal stack management overhead.

3.4 Software implementation: µTC language

The µTC language is introduced in [64]; this section describes the language implementation of the SVP general abstract concurrent execution model into a C-based language. µTC modifies the C programming language with SVP extensions. SVP threads run programs defined by thread functions, expressed with a syntax similar to that of C functions. In every thread, an index variable is automatically predefined according to the index numbering in the family. A create construct triggers SVP family creation. Synchronizing SVP channels are exposed as thread function parameters in the C language (either shared or global) and behave like variables with special read and write semantics. Concurrent family creation is the default method of composing functions and loops in µTC programs.

The extra constructs in µTC are language primitives and can be compared to other approaches [20] where concurrency is available through external libraries or native primitives. µTC uses primitives because concurrency is assumed to be the normal method as program composition. Library interfaces capture a more coarse-grained approach and can induce a significant overhead. Boehm [48] describes the limits and dangers of library-based concurrency that we have tried to avoid in the SVP model.
As required by SVP, $\mu$TC enables resource-agnostic concurrent programs. The developer is only responsible for exposing the concurrency and revealing the dependencies. Mapping and scheduling is resolved by the hardware and the program needs to be compiled only once to be executable on any Microgrid configuration.

### 3.4.1 The principles of the $\mu$TC language

The $\mu$TC abstract machine is defined through the following SVP extensions to the C abstract machine:

- **thread functions**, elementary programs for concurrent threads, defined by syntax similar to C functions;
- **thread families** of concurrent threads running the same thread function, each being distinguished by **thread indices**, integers automatically pre-defined to a different value when each thread starts;
- **thread creation**, an elementary action of the abstract machine causing the creation of a thread family;
- **synchronizing objects**, exposed as data objects in the C language with special read and write semantics; this is further separated into **shared parameters**, shared between adjacent threads in a family, **global parameters** shared by all threads in a family and **termination synchronizers**, which cause a reading thread to wait for termination of an asynchronous operation;
- **asynchronous termination**, another elementary action of the abstract machine, causing the termination of an identified thread family, from within one of the threads with the **break** action.

### 3.4.2 Introduction to $\mu$TC programming

$\mu$TC is a system-level programming language that exposes concurrency explicitly in the definition of tasks. The language embeds in its constructs the SVP properties such as thread family creation with the **create** construct, thread family synchronization barrier, etc. The developer is aware of the concurrency of the problem she/he defines; she/he only exposes the concurrency of a problem per se without having to actually manage the scheduling nor the mapping of the families.

Figure 3.11 illustrates an example $\mu$TC program. The main construct shown is the reflection of SVP’s **create** action. A thread family is identified by **family_id** and its range is bounded by the values of **start**, **limit** and **step**, as in common for loops. The other parameters are the **place identifier**, for the SVP place at which the family should be created, and the **blocking factor** which, if used, limits the number of threads created per core by the run-time system for the family.
/* thread function definition: 1 shared, 2 globals. */
thread void ddot(shared double res, double* x, double* y) {
    index i;
    res = x[i] * y[i] + res;
}

thread void main(void) {
    double a;
    create(fid;;0;1000;1;;) ddot(a = 0, b, c);
    sync(fid);
}

Figure 3.11: A $μ$TC example of a simplified reduction from the BLAS library. It represents a reduction computed by a family of 1000 threads indexed with ‘i’, declared with the $μ$TC index construct. The read to ‘res’ synchronizes with the previous thread; the two memory loads can be performed concurrently before synchronization; the read to ‘res’ in the next thread completes after the current thread completes its write to ‘res’. After synchronization (i.e. after the $μ$TC sync construct), ‘a’ contains the result of ddot in the parent.

3.4.3 New keywords

$μ$TC modifies the C programming language with SVP extensions listed in Table 3.3 and Table 3.4. Each function in $μ$TC is a thread function and employs in its definition the thread keyword to distinguish itself from conventional C. In the thread definition, the parameters are SVP synchronized communication channels: global and shared channels. The use of the shared type differentiates one from the other. The bulk synchronization barrier is encoded in the sync construct that must be within the same scope as the create it refers to with a same family identifier.

| **thread** | defines a thread function. |
| **create** | is the construct for thread family creation. |
| **break** | within a thread family, terminates a family and returns an object to parent thread. |
| **sync** | barrier synchronization targeting a specific family. |
| **kill** | outside a family, terminates the threads of the targeted family. |

Table 3.3: List of $μ$TC constructs.

| **shared** | specifies a shared synchronized communication channel. |
| **index** | specifies an index identifier for threads of a family. |
| **family** | specifies a family identifier, unique per family. |

Table 3.4: List of $μ$TC types.

As SVP pledges, it is possible to preempt the execution of threads via two dis-
tinct constructs. The **break** construct permits the preemption of threads from within one of the threads of the same family. The **breaking** thread can return an object to the parent environment as exception handler mechanism. The second way is with the **kill** construct which allows the external termination of all threads of the target family. Moreover, \( \mu \)TC provides an identification mechanism for each thread family with a specific identifier, before family creation, using the **family** type. Within a family body definition, it is possible to distinguish the indexed threads using the **index** construct.

### 3.4.4 Family creation

Family creation is a major property of the SVP model; it is encoded into the **create** construct in the \( \mu \)TC language. This main SVP **create** construct generates a parameterized group of statically homogeneous, but dynamically heterogeneous threads encapsulated in a thread function \( \text{fun (args...)} \) as illustrated in Formula 3.1.

Figure 3.11 illustrates a \( \mu \)TC implementation of a program from the BLAS library. The program runs a simplified reduction from the BLAS library where the parent thread produces a family of threads. This example also shows how the parallelism can be achieved through concurrent operations in the thread function \( \text{ddot} \). It represents a reduction computed by a family of 1000 threads indexed with ‘i’, declared with the \( \mu \)TC **index** construct. The read to ‘res’ synchronizes with the previous thread; the two memory loads can be performed concurrently before synchronization; the read to ‘res’ in the next thread completes after the current thread completes its write to ‘res’. After synchronization (i.e. after the \( \mu \)TC **sync** construct), ‘x’ contains the result of ddot in the parent.

The main \( \mu \)TC construct is the implementation of the SVP **create** action which generates a family of threads with the parameters specified in the construct:

\[
\text{create(family}_\text{id}; \text{place}_\text{id}; \text{start}; \text{limit}; \text{step}; \text{block}; \text{break}_\text{id}) \text{fun(args...)}. \quad (3.1)
\]

At family creation, a family of threads is identified by a unique family identifier **family_id** and shaped by the **start**, the **limit** and the **step** bounds similar to the **for**-loop statement. It is also possible to define a breakable family; then, there is a need to define an object **break_id**, which will collect the result in the parent thread when one of the threads breaks and returns an object. The developer can also define an area of resources where the family will be executed on within the **place_id** as Section 3.2.4 describes. The create parameters are listed in Table 3.5.

### 3.4.5 Object semantics in \( \mu \)TC

\( \mu \)TC has two distinct types of objects that can be distinguished from the location where they are defined: non-synchronizing and synchronizing objects.
family_id | unique family identifier for the to-be-created family.  
place_id | place of SVP cores where the to-be-created family will be run, default is 0, on the same resource that the parent uses.  
start | start bound of to-be-created family, the default value is 0.  
limit | limit bound, the default value is 1.  
step | step, the default is 1.  
block | block bound which corresponds to the maximum of running SVP threads at any moment per SVP core, default is 0 and is equivalent to maximum of running threads possible (per core) allocatable by the hardware.  
break_id | identifier which would collect a value from the breaking child of the to-be-created family.  

Table 3.5: Create parameters which set up the family definition.

The first type is simply declared within a thread function’s body. This type is named as local; it is used to declare a regular internal function object. This object does not live outside of the scope it is defined. Furthermore, no particular side-effects are encoded in the use of these objects. The second type is declared within the parameters of a thread function’s definition, as illustrated with Figure 3.12. Therewith, these objects have special synchronized properties and are separated into global and shared objects.

```
thread void foo(shared double x, double y)
{
}
```

Figure 3.12: Thread function definition: 1 shared x, 1 global y.

The synchronizing objects have special read-and-write accesses within the body of thread functions. The global synchronizing object is read-only within the scope of the thread function. Moreover, the shared synchronizing object maps, as defined in Section 3.2.2, into a couple of two objects {read,write} respectively mapped onto a couple of {dependent,shared} registers in the corresponding assembly representation. By definition, a shared object cannot be written before it is being read. The incoming shared object (read from dependent register) needs to be read first to consume its synchronization event; then, the outgoing shared (write to shared register) object can be written. Otherwise, the program’s execution will just deadlock.

Between the issue point (at a create construct) and the completion point (at a sync construct), within the same scope, the objects used for parameter passing to the thread function to be created are locked. This is due to architecture restrictions (cf. ‘pull’ mechanism used for thread parameter initialization, in Section 3.3.6). Therefore, these objects are located in a gray area (delimited be-
Between these two points, where thread creation takes place; the paradigm has no means to know when threads will be scheduled at this point. This is illustrated with a code sample in Figure 3.13. Consequently, any change in these objects within this gray area would provoke non-deterministic results upon the family’s execution. At the completion, the gray area stops and there are no locks on these objects.

3.4.6 Concurrency management

The developer exposes concurrent sections of a program and dependencies following the model’s constraints. There is no need to actually expose the schedule nor map this concurrency on resources since the architecture will ensure this. This separation of concerns is an SVP property; it removes from developers this heavy burden which would bind their implementation to specific architecture settings. Furthermore, another important SVP property, embedded in µTC is resource-agnosticism programming, where the configuration of the targeted architecture is not required at the time the program is being composed. The only thing a developer may be asked, is to define an abstract place of computation where a thread family would run.

Developers have constructs to manage behaviors of thread families. With the break constructs, it becomes possible to reproduce WHILE loops from the sequential model. Furthermore, it is also possible to handle exceptions during thread executions. If this occurs, the break construct can return an object to the creating environment. The second construct is kill which allows a family to be terminated if it behaves unexpectedly or beyond a defined limit.

3.5 SVP system performance

This section is an initial evaluation of the SVP computing system comprising a target platform (i.e. the Microgrid simulator in Section 3.5.1) and a set of hand-compiled highly-optimized benchmarks (cf. Section 3.5.2). There is no SVP compiler involved in the experiments; in other words, the following evaluation is performed without knowledge of a working SVP compiler. In Chapter 7,
Section 7.2 presents SVP system performance results with the use of the SVP compiler. For further results of SVP system performance, articles [61] [58] [65] [66] [67] are also available.

### 3.5.1 Target platform: Microgrid simulator

The evaluation, performed in this section, uses the SVP software simulator [68] which is a platform emulation of a Microgrid. The emulation captures state transitions down to the lowest level in the core pipelines. It also provides a fully parameterizable, cycle-accurate simulation of all hardware aspects of the Microgrid: core functional units, interconnects, network-on-chip, memory architecture, and external memory interface. We used parameters suitable for hardware that can be built using current technology [61] [65]. The simulation executes a unit of work (family and any subordinate families) on a variable sized cluster of cores connected in a ring network. The selection of cluster sizes has no impact on the performance of each cluster other than the number of cores. Figure 3.14 provides a schema of this configuration.

![Microgrid tile of 4 clusters of 4 cores (P₀..P₃) sharing an FPU between 2 cores and an L2 cache between 4. It illustrates all on-chip networks, two local rings between cores and L2 caches and routers (R) for the chip-wide delegation network.](image)

**Figure 3.14**: Microgrid tile of 4 clusters of 4 cores (P₀..P₃) sharing an FPU between 2 cores and an L2 cache between 4. It illustrates all on-chip networks, two local rings between cores and L2 caches and routers (R) for the chip-wide delegation network.

### 3.5.2 Methodology: benchmarks and evaluation

Since the SVP compiler is not utilized in this evaluation, the results presented emulate small hand-compiled kernels, which nevertheless are representative of the computation found in many large-scale applications. Our test kernels include several Livermore kernels [69], both independent and dependent ones, an Microgrid assembly version of the sine function using a Taylor series expansion, which is small and very sequential and the fast Fourier transform.
The motivation for this work is both the validation of the model implementation as well as an initial evaluation of its performance, scalability and latency tolerance. The COMA memory system described in [70] is not yet incorporated into this emulation and the results presented here uses two extremes of memory implementation. The first is sequential and the second is an idealized parallel memory that is capable of handling multiple requests in parallel. Both have parameterized latency and the latter a parameterized number of banks. Unless otherwise indicated, each core in the cluster has a family table with 64 entries, a thread table size of 256 entries and a register file with 1024 registers. L1 I- and D-cache sizes were set to 1 KB per core for these results, except where indicated for comparison.

3.5.3 Data-parallel code

The first result set is for the data-parallel Livermore kernels 1 (in Figure 3.15(a)) and 7 (in Figure 3.15(b)) executed over a range of cores. Compiling such loops for the Microgrid is straightforward, the loop is captured by a single create and the thread code captures the loop body, minus the loop-control instructions (increment and branch). This code contains only local and global registers. The results record the execution time (in cycles) required for a problem size of 64K iterations. Figure 3.15 shows the speedup achieved for those kernels relative to its execution on a single processor.

As can be seen, the speedup scales almost linearly with a deviation from the ideal of 20-40% at 128 cores due to the start-up involved. These are the cost of distributing thread parameters between cores and synchronizing between the cores on termination. Kernel 7 has more memory references than kernel 1 and performance saturates earlier in the scaling. Both kernels were re-executed with a 32 KByte cache to show the effect of D-cache size on performance. As can be seen this improves performance, but not significantly and shows the latency tolerance of the Microgrid. Neither experiment is limited by memory bandwidth by design. The stall rate shown in Figure 3.15 measures the percentage of time when the pipeline is stalled due to hazards. However, it does not include the time when the pipeline is completely empty. For high instructions per cycle (IPC), this gives us an indication of the utilization of the pipeline and since the model uses simple in-order pipelines without multiple instruction issue and branch prediction, this metric is very important. It shows that even without those features, the architecture is able to use the pipeline very efficiently by interleaving many threads in the pipeline. We do not count completely idle cycles as this can be detected and the core powered down. For low IPC therefore, this measures energy efficiency.
Figure 3.15: Speedup and stall rate (%) against number of cores in a cluster for Livermore kernels 1 (top) and 7 (bottom). Two sets of results are presented using a 1 KByte and 32 KByte D-cache. The simulation uses a random banked memory.
3.5.4 Families with inter-thread dependencies

This section illustrates how the model captures and exploits dependencies. The code executed here contains one or more thread-to-thread dependencies that require shared registers, which constrain the execution sequence. The test kernels are an Microgrid assembly version of the sine function, implemented as a Taylor expansion of 9 terms (this was implemented with a single memory bank) in Figure 3.16 and the Livermore 3 in Figure 3.17, inner product of 64K iterations. As with the previous experiments, the same Microgrid binary code was executed on a variable sized cluster. Figure 3.16 shows the speedup against number of cores for the sine function. Threads are executed concurrently but dependencies between threads constrain its execution. Concurrency is still exploited locally in tolerating memory latency and do-across schedules exploit a small amount of ILP in the thread code, where speedup is proportional to the number of independent instructions (before or after a dependency). When compared to sequential code, the Microgrid code is some 40% faster on a single processor and this increases to 70% faster on a cluster with 4 cores. This is due to the absence of any loop control overheads and high pipeline utilization from the few local threads.

![Figure 3.16: Speedup of sine function.](image)

Loop-based code generally captures three classes of computation, data parallel where the iterations are independent, recurrence relations like the sine function and reductions. Because of the commutative and associative properties of the latter, these operations can be implemented in any order and concurrency may be exploited. Theoretically, one can obtain $O(N)$ speedup and a latency of $O(\log_2 N)$ for binary operations. In the SVP model, several partial reductions can be performed in parallel with a final reduction giving the required result. The features of the model that allow this are the ability to create nested families...
of threads, the ability to specify where a family will be executed (i.e. a place) and the abstract manner in which this is all captured. For an \( O(P) \) speedup, the Microgrid assembly code can use the number of cores in a cluster, create a single thread per core that implements a distributed reduction on the result of subordinate families that are executed locally performing \( P \) partial reductions, one per processor. When this code is executed, near linear speedup is achieved on up to 128 cores for 64K iterations. This is illustrated in Figure 3.17. It should be noted that this code is also schedule independent and the only dynamic information required is the number of cores in the default cluster.

### 3.5.5 FFT Performance

When there was no SVP compiler, it was difficult to provide performance comparisons for large numbers of benchmarks because each simulation currently requires Microgrid assembly code to be produced by hand. However, in order for performance comparisons to be made against other published results, we have undertaken extensive simulation of the performance of a very common algorithm, namely FFT. Figure 3.18 shows these results. Again the same binary code was executed on a range of clusters from 1 to 64 cores. The results are presented in MFLOPS assuming a 1.5 GHz core and one FPU per core. All FPU operations are assumed to be pipelined with a latency of 3 cycles for \( \text{adds} \) and \( \text{muls} \) and 8 cycles for \( \text{divs} \). Results are given for FFTs of length \( 2^n \), where \( n \) ranges from 8 to 20 by 4.
Results show a maximum performance of 0.5 GFLOPs per processor, with linear scaling out to 64 cores for transform lengths of 64K and above. This maximum performance is limited by the number of FP operations issued per cycle, a static property of the code. The smallest transform of 256 points gives a maximum performance of between 5 and 10 GFLOPS and saturates at using 12 cores, or about 10 threads per processor.

Figure 3.18: Performance of FFT in MFLOPS against cluster size for FFTs of length $2^n$. 
3.6 Discussion and conclusion

This chapter has presented the properties of the SVP execution model and its implementations into a parallel programming language (i.e. \( \mu \)TC) and into a chip multi-processor architecture (i.e. Microgrid of SVP cores). The details of both the software and the hardware implementation are revealed to understand the SVP computing system; the complexity of these implementations is stressed to grasp the effort of providing the efficient SVP compiler discussed in the following chapters. The CSA group has also interests in high-level simulations of the SVP model onto conventional processors and investigates this with a pThreads implementation [71].

The last part of the chapter has presented an overview of potential performance with hand-coded scientific problems onto the SVP computing system (comprising the Microgrid target simulator). The lack of a compiler shows limits on the range of scientific problems to use for evaluation. The \( \mu \)TC parallel programming language presents an easier way to program these benchmarks which will be then compiled by the SVP compiler. While hand-coding benchmarks for an unconventional processor architecture such as the Microgrid is possible, compiling programs directly from a high-level concurrency-oriented language automatically is often much more difficult. The next part of the thesis shows that an extended imperative-language compiler can successfully target the Microgrid architecture. This capability provides further evidence that the Microgrid architecture may be used as foundations for a general-purpose multicore processor and SVP as the underlying paradigm.

Acknowledgments

The SVP model and its implementation is a group effort and the author would like to mention all members of the Computer Systems Architecture group for their input to this work: Mike Lankamp, Michiel W. van Tol, Li Zhang, Konstantinos Bousias, Raphael Poss, Michael Hicks, Simon Polstra, Clemens Grelck and Chris Jesshope. The following former members of the group also contributed to this research Joe Masters, Peter Knijnenburg and Guang Liang. The author would like to thank Raphael Poss for all the constructive criticisms he made during the elaboration of this chapter and his contribution to the evaluation section. The author would also like to thank Mike Lankamp for providing the simulator of the target platform (Microgrid) used in the evaluation section.