

A Model for the Design and Programming of Multi-cores

Chris JESSHOPE

Informatics Institute, University of Amsterdam.

Abstract. This paper describes a machine/programming model for the era of multi-core chips. It is derived from the sequential model but replaces sequential composition with concurrent composition at all levels in the program except at the level where the compiler is able to make deterministic decisions on scheduling instructions. These residual sequences of instructions are called microthreads and they are small code fragments that have blocking semantics. Dependencies that would normally be captured by sequential programming are captured in this model using dataflow synchronisation on variables in the contexts of these microthreads. The resulting model provides a foundation for significant advances in computer architecture as well as operating systems and compiler development. The paper takes a high-level perspective on the field of asynchronous distributed systems and comes to the conclusion that dynamic and concurrent models are the only viable solution but that these should not necessarily be visible to the users of the system.

Keywords. Programming models, Concurrency, Multi-cores

Introduction

This paper presents the results of more than ten years of research that started in the mid 1990s and whose goal has been to produce an architectural model that can provide solutions for effectively programming distributed multiprocessor systems. Initially, the goal was to support a programming model based on the data-parallel abstraction [1,2]. Today, the goal posts have moved and our aim is to support the programming of distributed multiprocessor systems using sequential languages, while also supporting other forms of concurrent software engineering, such as data-parallel, streaming and threaded, etc. The architectural landscape has also changed and a whole new generation of engineers and scientists are tackling the challenges of a new era of multi-core chips. The fact is however, that the problems remain unchanged, at least in principal issues, with only a minor shift in a few of the underlying parameters.

The work described here offers systematic solutions that can be applied across design spaces represented by large parametric ranges. This makes it applicable both to multi-cores as well as to so-called Grids. Indeed, in reflection on the period described above, one could argue that it is the Grid bandwagon, an utterly pragmatic diversion of resources that has been

responsible for the lack of funding in systematic research in the general area of concurrent computing.

Systematic solutions, that span the entire range parameters from multi-cores to distributed systems, are rooted in the design of processor architectures that are both efficient and can tolerate a large latency in responding to external events, such as accessing distributed memory or in the communication between distributed processes or threads. To achieve this requires the asynchronous scheduling of some unit of concurrency so that the processor is able to dynamically schedule work in order to hide latency. This use of concurrency to hide latency is called *parallel slackness* [3] and the concept predates this reference by probably another decade.

The first direction our work has focused on was therefore in reducing latency in communication [4], as there is always a cost in supporting parallel slackness. To increase the amount of latency that can be tolerated means providing additional synchronising memory for scheduling the units of concurrency. The more latency to be tolerated the more synchronising memory is required for a given granularity. Synchronising memory in its most general form detects an enabling event (e.g. arrival of remote data) and schedules the unit of concurrency dependent on it.

The second research direction was in being able to manage the finest level of granularity in terms of the frequency and size of the unit of concurrency that was scheduled, as this also has an impact on efficiency and can be best explained by analogy. Were scheduling to use regular units of concurrency with statically known parameters, we could use the analogy of building a wall. Whatever the size of brick, our wall would always contain the same amount of matter as we can arrange regular units without any gaps (this analogy ignores the mortar). However, scheduling is dynamic and the units of concurrency are irregular. A better analogy therefore, is a bucket of sand vs. a bucket of stones. Because they are irregular, the larger the unit, the less matter can we pack into a bucket.

This led us to look at instruction-level scheduling of small units of concurrency and resulted in a proposal for a dynamically scheduled RISC processor [5]. This model of scheduling microthreads in a DRISC processor has been developed over the intervening years and the recent resurgence of interest in architecture has accelerated these results following the rather sudden and recent realisation that superscalar architectures must give way to multi-cores. The result for us is a machine/programming model that gives all of the advantages of our current sequential one, yet can be used to program multi-cores through to the end of silicon scaling with similar properties in terms of binary-code compatibility.

1. Influences on this work

There are a number of significant results that have influenced this work. By far the earliest of these was the pioneering work by Burton Smith. A number of processors he designed have successfully managed units of concurrency at the instruction-level. These included the Delencor HEP [6] in the early 1980s, the Horizon, and culminated in the Tera architecture [7].

The HEP supported multiple blocking processes interleaved on a cycle-by-cycle basis in an eight-stage pipeline. This required at least eight active processes to keep the pipeline full. The HEP used parallel slackness to tolerate latency in accessing its distributed memory, by using a queue of suspended processes that were rescheduled by the arrival of data or a write acknowledgment from memory. The concurrency required in a single processor for accessing memory was estimated to be about 20 processes per processor for a 4-processor system [6]. The HEP 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 suspend the process until the operation completed. By providing both normal and blocking operations to memory, various programming models could be supported. Delencor exposed this model in HEP-FORTRAN, which added asynchronous variables, which defined synchronisation between processes, and a CREATE construct that was used for creating processes and which mapped onto the machine instruction that implemented it.

The Tera, which later became the basis for Cray's MTA product (The Tera Computer Company bought the remains of the Cray Research division of Silicon Graphics in 2000 and renamed itself Cray Inc.) incrementally improved on the earlier architectures. It could support up to 512 processors and a similar number of I/O and cache processors connected by a sparse 4096-node indirect network implemented as a three-dimensional torroid. Memory was randomised and with the larger number of processors, the latency of memory access in Tera increased to around 70 processor cycles on average [7]. Tera also supported the issue of multiple instructions from a single instruction stream reducing the number of threads required to tolerate the latency by allowing the compiler to statically schedule instructions with known delays. Tera processors still supported 128 streams each with 32 GP registers and 8 branch target registers per stream, giving 4096 GP registers and 1024 target registers per processor. Given that Tera had a mildly wide instruction issue (3 operations per cycle), the 4096 by 9-port GP register file would have been a major implementation issue.

Both HEP and Tera were significant architectural achievements but neither was a commercial success. The problem in both cases was twofold. Firstly Delencore forced a new programming model on the user, when at the same time, vector supercomputers could be programmed using a sequential model, even though code had to be tweaked in order to obtain optimal performance. The advantage of the vector platform was that reasonable performance could be obtained immediately and then by incrementally transforming the code, the vectorising compiler could begin to approach the peak performance. With HEP it was parallel or nothing, as sequential code ran at 1/8th of a single processor's peak performance. This fact was exacerbated by the technology used, for example, GaAs logic was used in implementing the HEP, which although fast, was only implemented in small scale integration leading to cost and reliability problems. Only in 1999 did Tera switch to CMOS technology. Indeed, technology seems to have a history of killing off innovative computer solutions, as it was also partly responsible for the demise of the next milestone in this area, the transputer.

The transputer [8] was developed by INMOS and, like the HEP and Tera, supported multiple processes explicitly at the ISA level. In the transputer, concurrency was supported by two instructions in its ISA to start and end a process, as well as the implementation of a read and write instruction on a channel, which could be virtualised. Whereas the HEP

supported parallel slackness and scheduling through instructions reading and writing memory, the transputer supported it by reading and writing a channel. Virtualisation was introduced using a static, link-time mapping of processes to processors. If communicating processes are mapped to the same processor, the communication channel is implemented by a memory word rather than a physical link (each transputer had only four physical links). Communication over channels was implemented by two further instructions that implicitly managed scheduling and process suspension was achieved by holding the suspended process identifier in the channel word. Channels therefore provided synchronising communication between processes implemented in a CSP style of programming defined by a language called occam [9].

Similar work was also undertaken in the dataflow community, especially work by Papadopoulos and Culler on the Monsoon architecture [14], which had many similarities with the work by Burton Smith. The difference between their work was that in Monsoon, programs were implemented in the context of dataflow graphs rather than the explicit management of threads of conventional instructions (i.e. instructions target other instructions rather than named locations in the register file). The authors later realised that evaluating simple arithmetic expressions suffered in this implicit model [15]. The work reported here also commenced with a study of dataflow techniques and we too were convinced that exposing concurrency through concurrency management in a conventional RISC processor was the most fruitful direction for efficiency in implementation.

2. Machine/programming models

If we consider machine/programming models generally, we can identify three kinds of model. The first is the *sequential model*, which has been around for some 50 or so years and which has changed very little in that time; the second might be called *ad-hoc parallel*, where a range of limited concurrency primitives are grafted onto the sequential model and finally we have *fully parallel models*, where concurrency is treated as a first-class concept. Perhaps the best way to differentiate between the latter two is to ask whether the model is used to expose the concurrency of the machine or used to capture the concurrency of the problem being solved?

The MTA and transputer provide good examples of these different views. From the Cray web site we can read: “Each MTA processor has **up to** 128 RISC-like hardware threads” (http://www.cray.com/products/programs/mta_2/). Thus, even though virtualisation of the processor supported parallel slackness, the concurrency described was the hardware, a bounded concurrency of 128 threads/processor. The transputer on the other hand described the concurrency of the application, as occam allows the user to create an arbitrary number of processes only weakly bounded, through limitations on memory size.

2.1. Sequential model

Before we discard the sequential model completely, there is much we can learn from it and its many advantages. Perhaps the first is that programmers find it easy to understand the

model and to write correct programs. The main issue here is one of determinism. Sequential programs are deterministic and give the same results however long it takes to execute its component parts. Concurrent programs on the other hand must introduce synchronisation. Too much synchronisation induces deadlock and too little introduces non-determinism in the results. The issue of non-determinism in programming models is explored more deeply in a recent paper by Lee [10].

Sequential programs are also safely composable under certain reasonable constraints (e.g. encapsulation of data, pure functions etc.) through the well-understood concept of hierarchy. This is not the case for unconstrained parallel models and it is one of the biggest pitfalls to CSP-based languages, such as occam, where deadlock could be easily induced through composition. It is possible to construct programs from networks of processes that are deadlock free by design but these must conform to certain restricted patterns [11]. When patterns of networks are provably equivalent to such design patterns it is also possible to design programs in a hierarchical fashion [12]. Ideally however, the constraints should be imposed on the machine or at least the programming model in order to preclude deadlock entirely.

There are a number of other practical advantages of the sequential model. Source code is universally compatible and can be compiled to any target without modification; one cannot say the same for many ad-hoc concurrent models. Moreover, over the last decade, binary-code compatibility has been one of the biggest marketing forces in the computer industry. Enormous resources have been directed at keeping binary code compatible in a sequential ISA, over a range of machine generations, each of which increased the concurrency of execution marginally. There was however, an enormous cost in hardware to support this.

The goal of the work described here is to obtain the same benefits of the sequential model, as described above, but from the perspective of a concurrent programming model. That is a deterministic model that is free from deadlock under concurrent composition and one that is binary compatible across a range of implementations from a single processor to the highest level of concurrency a particular application can support. Finally, the Holy Grail is that such binary code should be relatively easy to derive from existing sequential programs, which is not such a mystery. Prior work on parallelising compilers has had the dual problem of exposing concurrency and then scheduling it. Removing the requirement for the latter by providing a true concurrent model that abstracts scheduling removes the most difficult task from that goal.

2.2. The SVP model

The SVP model described here was developed in the AETHER project to define a SANE virtual processor (<http://www.aether-ist.org/>), where SANE refers to a *self-adaptive network entity*. The goal of the project is to explore self-adaptivity in complex computing systems, which in turn requires dynamic concurrency. The model was based on a substantial amount of prior research that started with DRISC [5] and developed through various ad-hoc concurrency models based on microthreading [13]. SVP however, puts the concurrency at its core and provides a full range of concurrency control mechanisms. In doing so, it subsumes various facets of both programming model and operating system functionality.

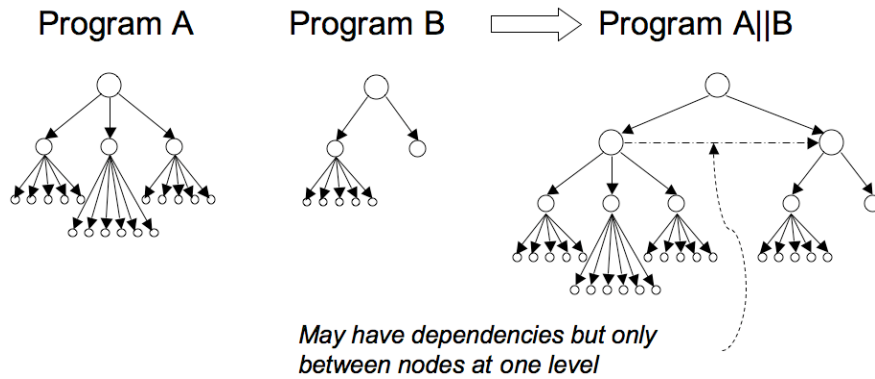


Figure 1. Concurrent composition in SVP

2.3. Composition in the model

In SVP, composition is concurrent and also dynamic. The result of this is that a program in the model can be represented by a dynamically evolving concurrency tree, a snapshot of which is shown in Figure 1. Concurrent composition replaces sequential composition, which is the key to flexibility in terms of implementation granularity. The concurrent composition is based on the sequential model and maps to the constructs found there. Serialisation is therefore trivial and is the mechanism for controlling the granularity of implementation. It is a tacit assumption in what follows, that the leaf nodes are microthreads, i.e. just a few machine instructions sharing a small context of registers. In Figure 1 then, the nodes represent *microthreads* and branching at a node represents concurrent composition. Sequence is introduced into the model in one of two distinct and separate ways:

1. the microthreads comprise sequences of one or more machine instructions. In an implementation, complexity of the nodes at the lowest levels depends on the parameters of the target architecture. In a microthreaded microprocessor, this may only be a couple of instructions, due to its efficient management of concurrency. It should be noted however, that a leaf node could also be a complete binary program that has no understanding of the model at all.
2. explicit dependencies are allowed between microthreads but in a very restricted manner. The dependencies allowed are between the creating thread and its subordinate threads and between all subordinate threads at one level. These are defined as an acyclic dependency chain from the creating thread through each thread created in some defined order.

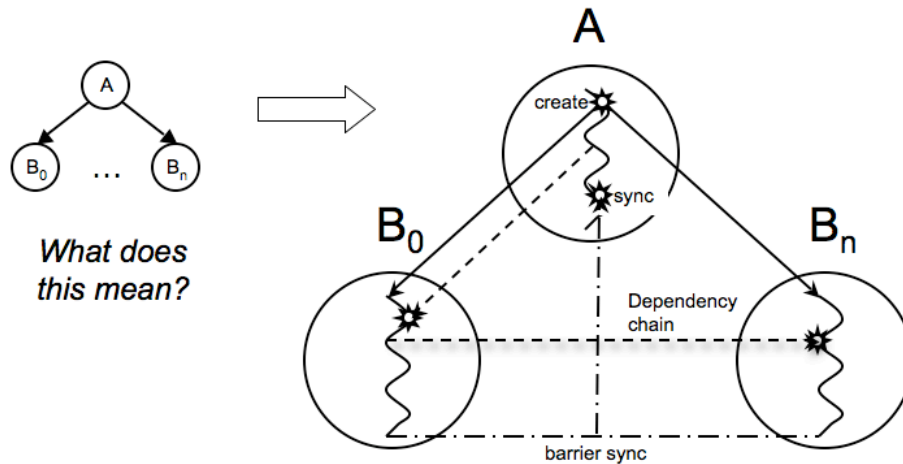


Figure 2. The events in creating a family of threads

Concurrency is introduced in the model through the dynamic creation of a parameterised *family* of microthreads; defined on a single thread definition, see Figure 2. The creating thread A executes a *create* action, which causes an ordered set of threads to be created subject to resource constraints, i.e. not all of the threads need be created simultaneously although they may be. The set may also be semi-infinite, so that an unbounded number of threads may be defined, executed on finite set of resources and terminated dynamically. The second event in the creating thread A is when all threads in the family have terminated and is identified by a *sync* action. A proceeds asynchronously with B_i but must wait on the *sync* action associated with any family of threads that it creates. This is not optional in most circumstances, as the family B_i cannot reliably communicate any information back to the creating thread without this action. The exception is where the created family is a continuation of the current thread and in this case it must be a family comprising a single thread. Synchronisation is described in more detail in Section 4.

Create is a concurrent analogue to both function invocation and loop structure in the sequential model. The parameterisation of the family captures both static and dynamic loop bounds. As described above, a family is defined as an ordered set and this set is defined over a sequence of index values captured by three parameters: $\{start, limit, step\}$. Each thread in the set has available to it, through the implementation of *create*, a unique value from this implied sequence. Dynamic loop bounds are obtained by setting limit to infinity (or as close to this as is possible in a finite machine) and terminating the family from any of the threads created using a *break* action. Note that if SVP is serialised by executing each thread to completion in index sequence, the analogue above is exact.

The use of the family in creating concurrency is in order to amortise the implementation overhead. As already indicated, at the lowest level in the concurrency tree, threads will

comprise just a few basic machine instructions and creating them one at a time would cause a significant overhead.

2.4. Communication in the model

Threads in SVP are blocking and can participate in synchronising communication. This communication is deliberately restricted and is implemented as a blocking read or dataflow-like synchronisation. In future silicon systems, communication and synchronisation should be localised and must be exposed to the compiler in some way, as communication costs across chip will be expensive. For this reason we assume that synchronising communication within a family of threads is implemented locally in the machine’s registers (or some form of synchronising memory close to the processor).

SVP communication is fine grain (i.e. on scalar values) and where required, defines a set of events that cascade through the set of threads in a created family. The creating thread, A in Figure 2, may write data only to the first thread created in the family, B₀, which will block on the first action requiring that data. Thread B₀ can in turn write to the next thread in index sequence and so on, so that each thread is able to write to the next in index sequence until we get to B_n. To provide closure to this sequence of communications, corresponding data written by B_n is available to the creating thread, A, but this is not a synchronising communication. To maintain symmetry and to enable concurrency controls to be implemented efficiently, this data overwrites the same variable that initiated the dependency chain and is only defined on the *sync* for the family.

It has been suggested that this restricted communication, equivalent to a loop with a loop-carried dependency from one iteration to the next, is too restricted. To put this criticism in perspective however, any regular dependency over an iteration space with a skip distance of greater than one can be transformed by nesting families, so that an independent family of extent equal to the skip distance is based on a thread that creates a subordinate family containing a local dependency. This is illustrated in Figure 3 in terms of adjacency; it is a transformation on the index space from one to two dimensions.

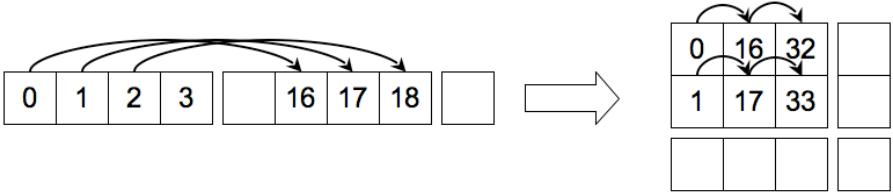


Figure 3. Transforming an index space to achieve local communication of dependencies.

Another technique that can be used to transform non-local dependencies to local ones is static routing between threads in a family. This is achieved by defining scalar buffers in each thread and copying values from one buffer to the next so that each buffer represents a skip distance from one up to the number of buffers used. In a fine grain model, this compiles to a register-to-register copy and is very efficient. The technique is illustrated in Figure 4. Finally, dynamic, index-set transformations can be achieved by data copying in memory,

exactly as would be achieved in a vector machine, i.e. gather and scatter operations. However this could be inefficient in some distributed implementations.

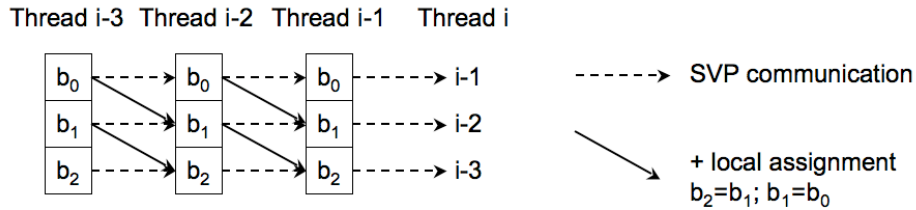


Figure 4. Static routing between buffers in a thread to achieve non-local dependencies

There are very good reasons for this restriction on communication in SVP. It guarantees freedom from communication deadlock within a family. It 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.

There are two other forms of communication in the SVP model. These occur at a higher level of abstraction (and of granularity) and are implicit rather than explicit. 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. For example, coherence protocols in a distributed-shared memory or direct communication between distributed memories. The final form of communication is in the implementation of binding a unit of work to a resource, which is described in more detail in section 6.

2.5. Concurrency control

To provide closure on concurrent composition within the model, some form of concurrency management is required. This has far reaching implication in an implementation but is a necessity in the context of implementing self-adaptation in the AETHER project. In a retrospective on the Monsoon architecture, Papadopoulos and Culler first reiterated the need for concurrency at the machine level: *“Threads are a key agent within all modern machines, and yet there are no operations defined on them at the machine level. Thread operations, such as create and terminate are implemented in software by the operating system...”* and also explored this further: *“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 I do not believe the solution they propose is at an appropriate level of abstraction for the explicit management of concurrency, the notion of reflection on the state of a concurrent computation must be introduced. In SVP, both creation and reflection are supported as explicit actions in the model, which in turn are implemented in the ISA of a microprocessor.

We have already encountered the *create* action and normal termination identified collectively by the *sync* action, possibly initiated by a *break* action in one of the threads. To this we add two further reflective actions on families of threads, namely *kill* and *squeeze*. *Kill* terminates a complete family of threads losing all the state of that computation, whereas *squeeze* terminates a family by identifying and capturing an intermediate state, so that the

family can be recreated and completed, probably on different resources. These two actions are executed, not in the creating thread or any of the threads it creates, but in an independent and asynchronous control thread that is monitoring some aspect of the computation or its environment (see Figure 5). These actions require that families must be identified within the machine. It may seem that this is also required for the *sync* action but this action is internal to the creating thread and can be implemented by allocating a local register to receive a return code from the *create* action, which is blocking.

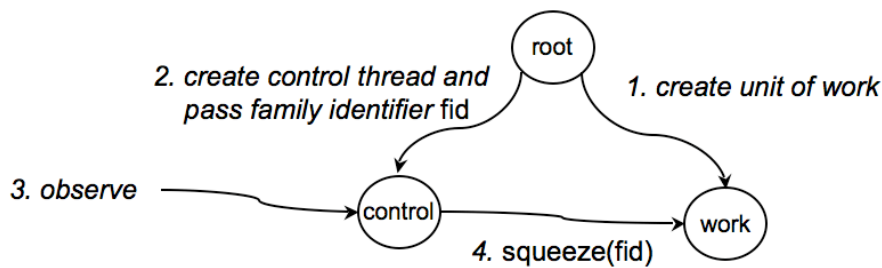


Figure 5. Reflection in the model by named families of threads and concurrency control

Managing concurrency asynchronously requires that any reflective action be observed. Typically this would be in the creating thread, for example *root* in Figure 5. In SVP this is achieved by providing a return code and possibly a return value to the creating thread via the *sync* action. The return code identifies how a family was terminated, it may be one of the following codes: {*normal*, *break*, *kill*, *squeeze*}, where *normal* is the result of the execution to completion of all threads specified in the *create*, and *break*, *squeeze* and *kill* indicate that the family was terminated by the action with the same name. A return value is set only by *break* and *squeeze*. In the former it is set by the *break* action in the thread that succeeds in executing its *break* and in the latter it captures a unique squeeze point in the index sequence and is described in more detail below.

One of the far-reaching implications of reflection is the fact that such actions are extremely powerful, especially when implemented at the level of instructions in an ISA. Typically these would be controlled by some user identity in an operating system. I.e. only the user of *root* is allowed to terminate a job in typical operating systems. Here we have a self-similar model and the capability to implement these actions must be granted to a thread. Only in this way can they be implemented in a manner that can be guaranteed to be secure. In SVP, the family identifier contains a capability, for example a random token generated on creation, which is passed securely to a controlling thread and which must be matched before *kill* and *squeeze* actions are executed. In SVP, a secure channel may be implemented using memory and appropriate memory protection domains or using a synchronised communication, which passes a value by a shared register accessible only to the creating and control threads.

Another issue that must be considered in implementing these reflective actions is whether to implement them as deep or shallow actions, i.e. whether they are propagated down the concurrency tree illustrated in Figure 1 or not. For *kill* it does not make sense to

implement the action shallowly, so a *kill* should be propagated from one family to its subordinate families. For *squeeze* the appropriate implementation is not clear. For threads that perform computation it may not be efficient to preempt them, as this may cause as much work in preserving the partial state as in completing the computation itself. However, for control threads that *create* a significant amount of work in their sub trees, some depth of action may be required in order to minimise the latency of the action.

The solution adopted in SVP is to allow for a program-controlled depth of propagation of the *squeeze* action. Threads may take on the attribute *squeezable*, which implies that a family defined on this thread is able to propagate any *squeeze* action it receives to its subordinate families. It is the responsibility of the thread to capture any state required to re-create the subordinate thread when re-executed itself. Because of the restriction on communication, the state is limited to one or more scalar values communicated between threads so long as the *squeeze* identifies a partition on the family, the *squeeze point*, which must also be captured as an index value.

Squeeze returns this index value via the *sync* action in the creating thread. The value is defined as the first thread in index sequence not yet executed to completion. This is the first thread not to have been allocated resources if the thread is not *squeezable*. Alternatively, if it is *squeezable*, it will have created subordinate threads, which were also squeezed. Thus any thread that terminates with a *squeeze* return code will have to be re-executed. The result on a family receiving the *squeeze* action is to cease the creation of new threads and allow all currently active threads to complete. In addition, the family will send a *squeeze* action to any subordinate families created by its threads but only if the thread is labeled as being *squeezable*.

The algorithm to determine the index value therefore depends on whether the thread is *squeezable* or not. If the thread is *squeezable*, the index value captured by *sync* is defined as the index of the first thread in index sequence to have been allocated a context of synchronising memory but which has not yet terminated normally. This index partitions the family into those threads that have completed and those threads must be re-executed because they created a subordinate family and that family was squeezed. A *squeezable* thread must update the start index of the *create* parameters of its subordinate family with the return value at *sync* so that on re-execution it will *create* only those subordinate threads that had not completed.

Figure 6 illustrates three partitions over the ordered set of threads in a family on receiving a *squeeze* action. The first, identifies those threads that have all completed, these are not re-executed. The second, those threads that have been allocated contexts and which may already have completed, these are allowed to complete, if necessary by squeezing their subordinate families, but they will all be re-executed when the family is re-created; this partition is identified by at least one thread (the first) that has not yet completed. Finally, there are those threads in the family not yet allocated a context of synchronising memory, which will of course need re-execution when the family is re-created.

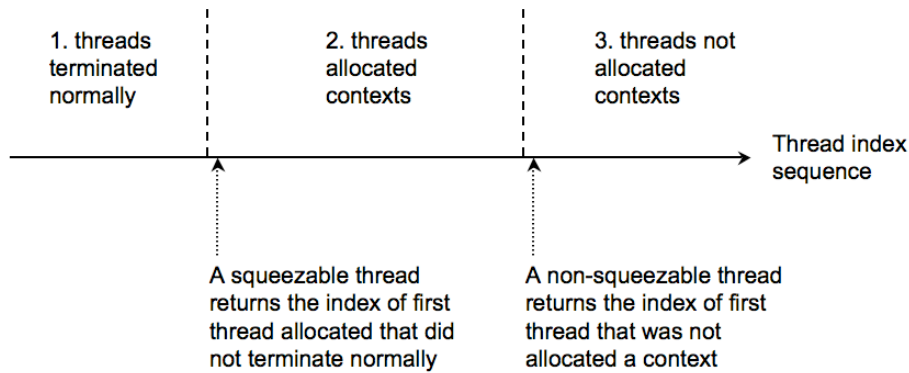


Figure 6. Determining the squeeze point in an executing family of threads

If the family is not squeezable it will not pass the *squeeze* action onto any subordinate family and all threads in partition 2 in Figure 5 will eventually terminate normally and the return value at *sync* is set to the index of the first thread not to have been allocated a context of synchronising memory.

On synchronisation, the value of any dependency at the *squeeze* return index is automatically saved to the variable that initialised the dependency chain in the creating thread, again allowing the family to be re-created from the *squeeze* index position to complete it.

Squeezing a family of threads allows the task defined by that family to be preempted and restarted at a different *place* (set of processors). By using the concept of a squeezable thread a disciplined approach to the rapid preemption of a concurrently executing program can be provided in the model. This concept is at the heart of the model and provides one of the most important features for implementing self-adaptivity.

2.6. Summary of the model

The SVP model provides a number of actions that initiate and control concurrency. These actions are summarised below and captured schematically in Figure 7.

- *create* - defines a parameterised family of microthreads, which are allocated contexts for execution as resources become available. Microthreads may themselves *create* subordinate families of microthreads;
- *sync* - identifies the termination of all threads in a particular family. Using a *create/sync* pair allows the creating thread to continue asynchronously with the threads it creates. *Create* returns a code via its *sync*, which defines how the family terminated. If termination was by a *break* or *squeeze* action the *sync* also provides a return value;
- *break* - terminates a family of threads on a condition being met within one of its threads, the thread successfully executing a *break* will set the return value;

- *kill* - terminates an identified family of threads so that no state set by that family can be guaranteed. The *kill* action is propagated to all subordinate families;
- *squeeze* - terminates a family of threads so that its state can be captured and the family can be re-executed. *Squeeze* is preemption over a concurrency tree and returns an index value of the first thread to be executed when the family is recreated. A *squeeze* action is only propagated to a subordinate family of threads if those threads are defined as being *squeezable*.

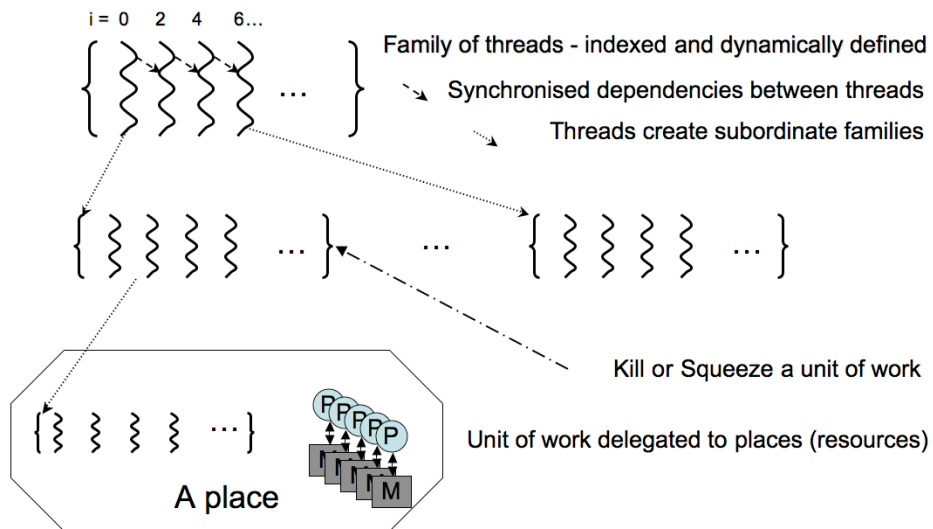


Figure 7. A schematic summary of the SVP model

It should be emphasised that SVP is a dynamic model that creates threads dynamically on resource availability. Both the extent of the family and the resources that the family are executed on are defined when the *create* action is executed. The *create* action captures all parameterisation within the model; it defines:

- the threads' code - possibly in many different forms (e.g. specialised implementations on reconfigurable logic);
- the family's parameters, i.e. extent and index range;
- whether the family has dependencies between threads;
- the extent of concurrency of a unit of work - can be inferred from the above two facets applied recursively;
- where the unit of work is to execute - i.e. the delegation of a family to some place or resource set;
- whether the *create* requires mutual exclusion or not where it is executed;
- the notion of a timer, if required for partial failure;
- any other meta-data required, e.g. real-time constraints, power constraints etc.

3. SVP memory model

The state of the SVP model at any given time is defined by two abstractions, the first and most persistent is an asynchronous shared memory, which comprises a number of addressed locations, each of which may be read and written to by a thread but subject to certain constraints. The constraints are due to the asynchronous and concurrent update of the memory by multiple threads. No guarantees can be given about the access time to this memory. The second abstraction is a context of synchronising memory associated with each thread, which provides the operands to all arithmetic or logical operations in the thread. It provides synchronisation on scalar values produced by other threads in the same family or from values input from shared memory.

3.1. Shared memory

SVP is based on units of work, which comprise families of threads (and their subordinate threads). A family has inputs and outputs, which are defined by the thread used to *create* it. These inputs and outputs are the locations in shared memory that the threads read and write. The output can only be defined on the *sync* for a family, i.e. the memory is bulk synchronous. Prior to this event, because we are dealing with a potentially asynchronous concurrent system, there will always be a subset of locations whose state cannot be determined. This means that in order to give deterministic programs, no two concurrent threads may read after write to the same location of memory and no two concurrent threads may write to the same location.

Note that this abstraction of shared memory may be implemented in any manner including using distributed memories and message passing between them. In the latter case, before a family of threads is created at a place, a synchronising communication must provide the place with the data that the family will read as input and before the family can be deemed to have synchronised, all of its output must be returned. Figures 8 and 9 show the shared and distributed view of family creation.

3.2. Synchronising memory

Synchronising memory provides the mechanism by which threads within a family can synchronise with each other and their creating thread; it also provides the mechanism for suspending and rescheduling threads. All operands must be loaded into this memory before the processor can perform the corresponding operations. It therefore provides the mechanism by which the processors can synchronise with asynchronous-shared memory.

Synchronising memory is dynamic and transient. An implementation dynamically provides a context of scalar variables for a thread, which are allocated empty, written once identifying a synchronising event and which are discarded when the thread completes. Each location provides a blocking read or dataflow-like synchronisation and dependencies between operations in different threads are enforced by this synchronising memory. It is assumed that this memory is fast, distributed and “close” to the processor or logic cells implementing the thread.

Note that threads may be sequentialised on their index order if synchronising memory is limited. So, during the execution of a family, there may only be a subset of the family's threads active and contributing to this synchronising state. This constrains thread creation to be performed in index order, at least for dependent families of threads. The amount of synchronising memory allocated will limit the parallel slackness and hence extent of latency that can be tolerated.

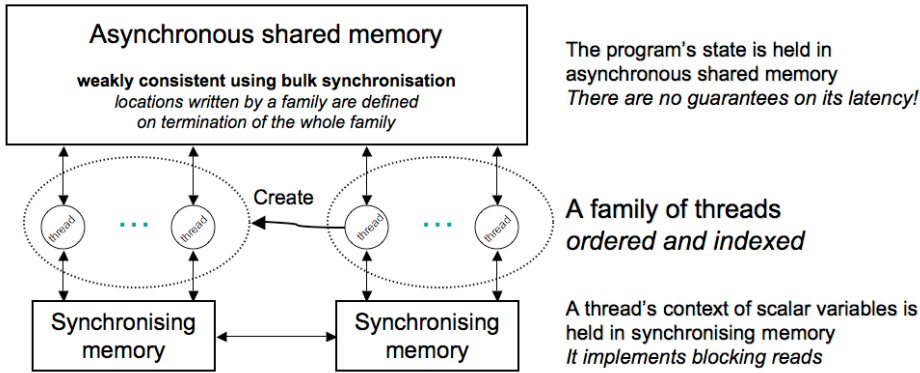


Figure 8. Family creation in a shared-memory environment one thread creates a new family of threads.

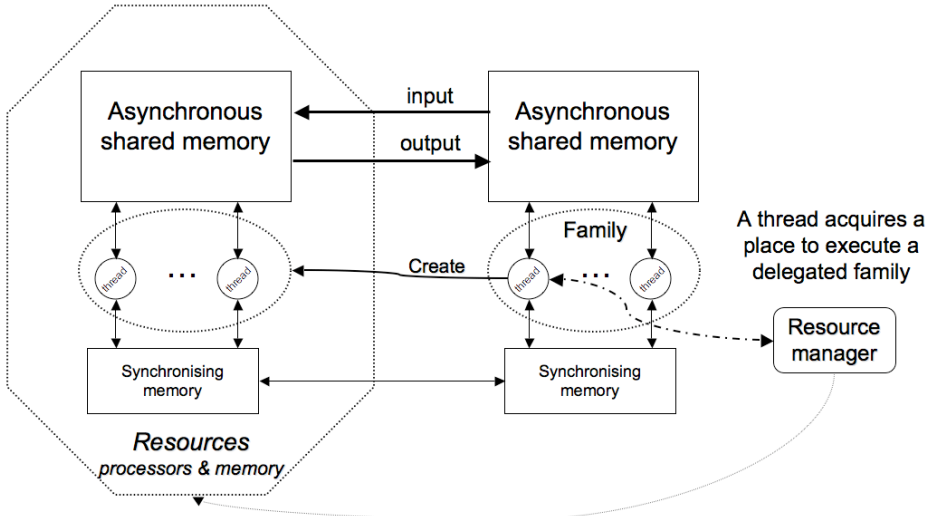


Figure 9. Family creation showing the dynamic resource management required with SVP. Again one thread creates a new family but this time it acquires a new place to execute the family. Here a place is a set of processors and associated distributed memory.

4. Resources in the model

Programming models are abstract but when implemented they have to deal with physical entities, i.e. resources such as processors and memory, access to hardware such as I/O registers and security issues such as access controls. SVP binds resources dynamically by creating a *family* at a *place* where the threads will be executed. Typically a place will be one or more processors, an accelerator, FPGA cells etc. This process of binding a family to a place is called *delegation* and is like a remote-procedure call, where the “procedure” is the *unit of work* defined by the family being created and any subordinate families that it may *create*. Like the model itself, delegation is hierarchical and a unit of work can delegate one of its subordinate families to yet another place.

To make the model amenable to static analysis, regardless of the fact that the number of processors and extent of the family may be dynamic, the distribution of threads to processors on family creation is deterministic. This will be implementation dependent and will typically be a cyclic or block-cyclic distribution. Thus, if each processor at a place has access to:

- the number of processors in the ordered set of processor allocated for a delegation;
- its own location in the set;
- the parameters defining the set of threads, $\{start, limit, step\}$;

then each processor can independently allocate only those threads that comprise its distribution.

One of the key parameters in family creation is the number of threads created at once on a single processor, called the *block size*. This parameter can be used to manage resources and avoid resource deadlock at a place, at least this is possible with bounded recursion of *creates*.

Note that the concept of place must deal with more than processor resources; it must also deal with security and system services. A place will be identified by some namespace (address) through which the creating thread can communicate parameters to the processors for thread creation. For security, there must be a capability associated with a place, so that an arbitrary thread cannot interfere with any legitimate families executing. Again this can be a random token given to the place and creating thread on resource allocation, which is then matched on delegation.

Creating at the *default* place executes the family on the same resources that are being used to execute the creating family of threads. Another identified place is *local*, which forces a family to execute all of its threads on the same processor as the creating thread. This allows for control of locality. A local family may only create parallel slackness.

More generally, a place is an abstraction, which gives a union of:

- a communication address defining a physical place to which the family’s parameters are sent on delegation;
- a capability to execute the family at that place;
- optionally - a service provided at a particular place, which requires exclusion.

4.1. Resource deadlock

Although the model is designed to be free of communication deadlock by design, like all data-driven models, it is still susceptible to resource deadlock. However, resource deadlock is easier to manage than communication deadlock and can be analysed and controlled.

Resource deadlock occurs through the unfettered exposure of concurrency with dependencies on a finite set of resources. As SVP restricts dependencies between threads, i.e. communication between adjacent threads in index (creation) order, then resource usage in a family, i.e. size of the contexts of synchronising memory required to avoid deadlock can be statically analysed; the minimum resource required is the sum of one producer's and one consumer's contexts.

However, in SVP there are also dependencies between a thread and any families of threads that it creates. With finite resources and unbounded recursion of creates resource deadlock cannot be avoided. This is illustrated in Figure 10. When synchronising memory is exhausted, no family can be created and any failed *create* will block the creating thread from completing and so on back up to the root thread.

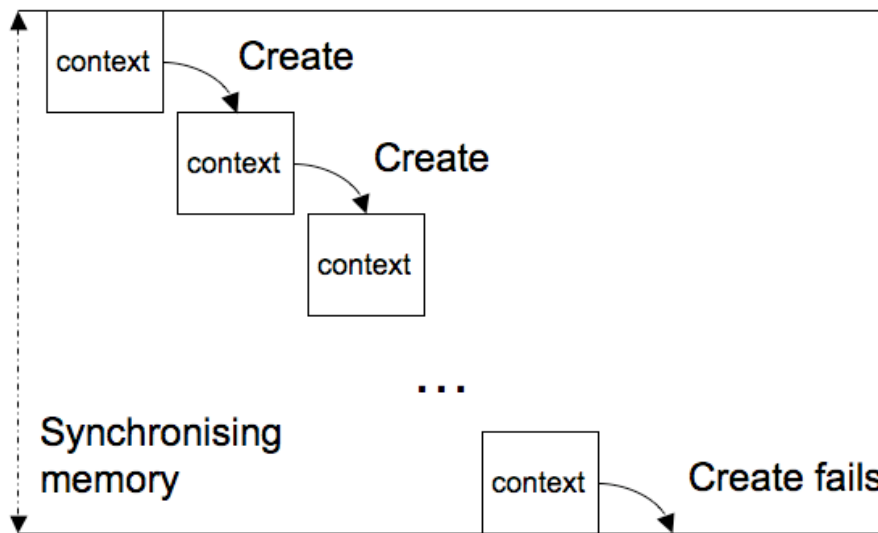


Figure 10. Resource deadlock caused by the limitation of synchronising memory with unbounded recursion of creates. No thread can release its resources until its subordinate families have completed. When synchronising memory runs out no thread can complete.

This situation can always be identified although it does not always indicate deadlock. However, with constraints on the way units of work are allocated, deadlock can be detected, e.g. the example in Figure 10. Even more generally, given unbounded recursion, the weaker signal of possible deadlock can be used to trigger resource acquisition and to resolve the potential problem by delegation. Note that this is conceptually no different to stack overflow

in the sequential model. The difference is that here, it is synchronising memory rather than main memory that is the limitation.

5. Implementation of the model

SVP can be implemented at a number of different levels of granularity. The key issue is at what level in the concurrency tree are units of work distributed and synchronisations achieved. For example a coarse-grain implementation could be implemented on distributed systems of conventional processors, provided that any units of work below the distribution level are serialised in order to amortise the communication and synchronisation overheads.

At the University of Amsterdam, we are interested in very fine-grain implementations, where the model is captured in the ISA of a microthreaded processor. We have implemented prior microthreaded processors that supported a single level family via software emulation [16,17], as well publishing estimates of the area required for the support structures required in a silicon implementation [18]. During the last year we have implemented a further software emulation of the model described here based on the Alpha ISA. The only additional support structures required for the hierarchical model is a table to store family-related information. However, the implementation also requires some re-partitioning of thread-related information, such as queues of active threads, waiting threads etc, as well as requiring family identifiers to be propagated through to the load/store unit in the pipeline for synchronisation purposes. Results on this work will be published soon.

The model has also been captured as an extension to the C language called μ TC and we are developing a compiler from this intermediate language to target the emulation of the Alpha-based DRISC microprocessor [20]. Another development we have started is the compilation of C to μ TC. Although many parallelising compilers have been attempted, we believe that the lack of success in generating commercial interest in this approach is not the exposure of concurrency in the compiler. Rather it is the requirement to statically schedule the concurrency exposed. With SVP this is no longer a requirement, as the parallelising compiler can simply express all concurrency exposed and leave the scheduling and resource management to run-time systems, in the case of a DRISC implementation, implemented in the instruction set.

6. Conclusions

This paper has summarised the results of a significant research effort carried out over the last ten years. The culmination of this research is a hierarchical model of concurrency that has dynamic data-driven scheduling as well as dynamic management of resources. It can be argued that as systems become more and more complex, i.e. with the introduction of more and more processors servicing a user task, then it is no longer possible to statically analyse and schedule the execution of that task.

The model has all of the attributes of the sequential model. It is deterministic; its is deadlock free by construction and most importantly, when code is compiled to processors

that support this model at the instruction level, then that binary code is binary compatible across any number of processors up the limits of concurrency dictated by the application itself. This is a significant property and one that should provide serious consideration of this model.

Finally we believe that it is possible when using this model to be able to compile from sequential code and thus realise the holy grail of being able to take existing sequential code bases and distribute these across distributed systems, be they fine-grain implementations based on DRISC processor or coarse-grain implementations based on existing distributed environments, such as we have in today's globally networked world.

We do not underestimate the difficulties in achieving these goals. The model is disruptive in its efforts to push the management of concurrency into the lowest levels of granularity in processing systems, i.e. at the level of basic machine instructions. These instructions usurp operating system functionality and turn the current perception of operating systems on their head. Currently operating systems apply the principle that processor cycles are expensive and share these cycles between user tasks. Of course concurrency is required on a single processor in the form of parallel slackness but this concurrency should only be used to tolerate latency and should be derived from single user task. It is OK for a processor to be idle, in the world of multi-cores, what is not acceptable is for idle processors to be drawing power. Data-driven rather than speculative models also provide the necessary mechanisms to recognise this property. For example the DRISC processor measures work not by the number of created processes but by the number of threads in its active queue. The threads will still exist in synchronising memory but may be inactive waiting for long latency events.

In a modern operating system, on-chip storage and locality of communication are far more important issues and the operating system should be optimising these rather than doling out processor cycles. SVP therefore will have an impact on compilers, operating systems and indeed middle-ware in current distributed systems. These are indeed exciting times to be working in the area of computer systems architecture!

7. Acknowledgments

The work described in this paper is currently being supported by two projects. The first, *Microgrids*, is a national project in the Netherlands funded by NWO and the second, *AETHER*, is a collaborative project funded by the European Community. We gratefully acknowledge the financial support received from both projects, without which the significant progress made over the last year in the development of the SVP model would not have been possible.

References

- [1] V B Muchnick and A B Shafarenko (1996) *Data Parallel Computing - the Language Dimension*, ISBN: 1 85032 179 5, Chapman Hall.

- [2] C R Jesshope (1992) The f-code abstract machine and its implementation, *Proc COMPEURO 92* (IEEE Press) pp 169-174
- [3] G V Wilson (1993) A Glossary of Parallel Computing Terminology, *IEEE Parallel & Distributed Technology*, **1**(1), pp 52-67
- [4] C R Jesshope and C Izu (1993) The MPI network chip and its application to parallel computers, *The Computer Journal*, **36**, pp763-777
- [5] 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.
- [6] 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>
- [7] 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
- [8] D May and R Shepherd (1984) The transputer implementation of occam, *Proc. Intl Conf on Fifth-Generation Computer Systems*, Tokyo, pp533-541.
- [9] INMOS (1984) occam Programming Manual, Prentice-Hall, ISBN 0-13-629296-8
- [10] E lee (2006) The problem with threads, *IEEE Computer*, **36**(5), pp. 33-42
- [11] P H Welch, G R R Justo and C J Willcock (1993) Higher-Level Paradigms for Deadlock-Free High-Performance Systems, *Transputer Applications and Systems '93*, Proceedings of the 1993 World Transputer Congress, **2**, pp 981-1004, ISBN 90-5199-140-1.
- [12] J.M.R. Martin (1996) *The Design and Construction of Deadlock-Free Concurrent Systems*, PhD Thesis, University of Buckingham, UK, 1996 <http://wotug.ukc.ac.uk/parallel/theory/formal/csp/jeremy-martin/>
- [13] Jesshope C. R. (2006) Microthreading a model for distributed instruction-level concurrency, *Parallel processing Letters*, **16**(2), pp209-228, ISSN: 0129-6264
- [14] G M Papadopoulos and D E Culler (1990) Monsoon: An Explicit Token Store Architecture, *Proc. 17th International Symposium on Computer Architecture*, pp82-91
- [15] 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
- [16] Kostas Bousias and Chris Jesshope(2005) The Challenges of Massive On-Chip Concurrency, *Lecture Notes in Computer Science*, **Volume 3740**, pp157 - 170
- [17] Bousias, K, Hasasneh N M and Jesshope C R (2006) Instruction-level parallelism through microthreading - a scalable Approach to chip multiprocessors, *BCS Computer Journal*, **49** (2), pp 211-233
- [18] Bell, I, Hasasneh, N and Jesshope C R (2006) Supporting Microthread Scheduling and Synchronisation in CMPs. *Intl. J Parallel Processing*, Jan 2006, pp1-9, DOI
- [19] 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
- [20] T. Bernard, K. Bousias, B. de Geus, M. Lankamp, L. Zhang, A. Pimentel, P.M.W. Knijnenburg, and C.R. Jesshope (2006) A Microthreaded Architecture and its Compiler, *Proc. 12th International Workshop on Compilers for Parallel Computers (CPC)*, M. Arenez, R. Doallo, B.B. Fraguera, and J. Tourino (eds.), pp 326-340