Parallel Processing Letters
© World Scientific Publishing Company

ON THE COMPILATION OF A LANGUAGE FOR GENERAL CONCURRENT TARGET ARCHITECTURES

THOMAS A.M. BERNARD[1], CLEMENS GRELCK[1,2], CHRIS R. JESSHOPE[1]
[1] *Informatics Institute, University of Amsterdam, The Netherlands*
{t.bernard,c.grelck,c.r.jesshope}@uva.nl

[2] *Dept. of Computer Science, University of Hertfordshire, United Kingdom*
c.grelck@herts.ac.uk

Received received January 2010
Revised revised January 2010
Communicated by S.G. Akl

ABSTRACT

The challenge of programming many-core architectures efficiently and effectively requires models and methods to co-design chip architectures and their software tool chain, using an approach that is both vertical and general. In this paper, we present compilation schemes for a general model of concurrency captured in a parallel language designed for system-level programming and as a target for higher level compilers. We also expose the challenges of integrating these transformation rules into a sequential-oriented compiler. Moreover, we discuss resource mapping inherent to those challenges. Our aim has been to reuse as much of the existing sequential compiler technology as possible in order to harness decades of prior research in compiling sequential languages.

Keywords: Concurrent execution model, many-core architecture model, compilation transformations, parallel language, compilation challenges, resource mapping, multi-core programming menace.

1. Introduction

The computing industry currently faces a dilemma. It is well understood that power efficiency requires many simple cores in a processor architecture rather than fewer more complex ones [1]. However, exploiting these new architectures requires the exposure of explicit concurrency in the code they execute, in contrast to the implicit concurrency exploited in more complex cores. This in turn requires applications to expose this concurrency, either explicitly by the programmer using some concurrency model or automatically using parallelising compilers, neither of which is easy [2]. This challenge is well acknowledged [3, 4, 5], but the concurrency revolution is happening now and urgently requires new tools and a new ways of thinking [6]. Unfortunately there is no consensus on what the target should be for these tools. The scale of concurrency that needs to be exposed can only be found now in high-performance computing (HPC) and here the common target is MPI, a low-level API designed for distributed memory architecture. This lack of target for general purpose computing and the pitfalls of concurrent programming [7, 8] are the key issues tackled by our group research.

This paper describes work supporting a general abstract concurrent execution model called *Self-adaptive Virtual Processor* (SVP) [9, 10]. This model combines fine-grain threads (concurrent composition is assumed at all levels) with dataflow synchronisation between threads. As such it is a compromise between capturing maximal concurrency (i.e. dataflow) and providing efficient implementation (i.e. threads). It provides deadlock free composition and captures locality and regularity via the model's constraints, which means it is amenable to compiler analysis and code transformation even in the absence of a specific target architecture. Also like sequential and dataflow models it provides determinism of results.

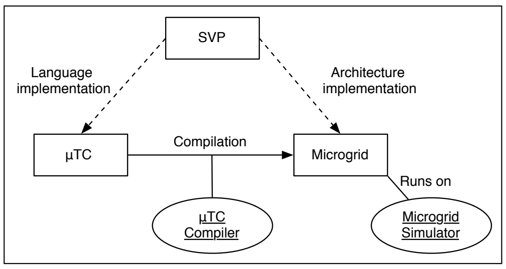

Fig. 1. SVP model has both language and hardware implementations with μTC and the Microgrid. Our μTC compiler transforms programs into Microgrid assembly which can be then run on our Microgrid cycle-accurate simulator.

The execution model supports two forms of synchronisation. Bulk synchronisation is implemented over concurrent regions (hierarchically), which can be used for coarse-grain mapping (at some unspecified level of the hierarchy). In addition to this, dataflow synchronisation captures dependencies in the code and can be used to implement scheduling over multiple threads for a given mapping in order to support asynchronous operations and tolerate latency.

As shown in Figure 1, the SVP execution model has been implemented into a parallel language called μTC and into an architecture called Microgrid. The model is captured in an architecture-neutral manner by adding both forms of synchronisation in a C-based language, μTC, which uses create/sync blocks to define a concurrent region and so-called *shared* and *global* objects to capture dataflow synchronisations. Moreover, we have a many-core simulator [11] of the Microgrid of SVP cores. It provides a target for our μTC compiler with cycle-accurate simulations of programs for our experiments.

The contribution of this paper is the compilation schemes between the language implementation (i.e. μTC) and the architecture implementation (i.e. Microgrid of SVP cores), and then the challenges we encountered to integrate them into our compiler (we have used the GCC framework). Although the SVP is generic (Section 2), we choose to implement this execution model as extensions to the *Instruction Set Architecture* (ISA) of an otherwise conventional in-order core, illustrated in Section 3. One of our research goals has been to reuse existing compiler technology in generating code for this extended ISA from μTC (Section 4). We present our compilation schemes in Section 5. In Section 6, we then identify the challenges of integrating the SVP semantics into a sequentially-oriented compiler.

2. Presentation of the SVP Model

The SVP model [9] offers a uniform means of capturing dynamic concurrency. This is a parameterised *create* action that forks a named family of identical blocking threads as illustrated in Figure 2. This may be applied recursively, and hence the model's bulk synchronisation is by named family (*sync* action in Figure 2). An SVP thread may contain a number of synchronising 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 [12] for formal proofs).

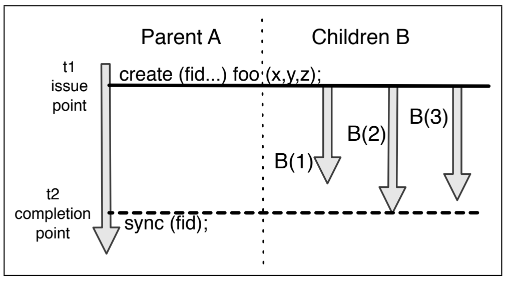

Fig. 2. SVP family creation event with multiple concurrent control flows of threads. The child family is created at the *issue point* in the parent thread. All threads of the child family are terminated after the *completion point.* Note that this figure does not illustrate the inter-thread communications.

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 a *break* action. In principle, unbounded families are possible, however this is constrained by a dynamic space/time trade-off.

SVP uses a relaxed consistency model for memory that is managed by *create* and *sync* events and possibly writes to synchronising objects. It provides a single, flat address space with Location Consistency (LC) [13]; 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 of a synchronising object used as a reference when that object is written. The model requires LC's `acquire` and `release` operations to be implemented on synchronisation actions such as creation and termination of threads and writing to and reading from synchronising objects. There is no way for threads to explicitly synchronise access to any memory location with another, unrelated thread.

SVP defines a binding of unit of work to a set of resources. This set of resources is defined by objects called *places* which are opaque and implementation-defined. A family of threads is bonded to a *place* where it will be executed on. An SVP program is composed of families of threads, where threads (non-creating) are the leaves and families are the nodes of the program's *concurrency tree.*

3. Architecture of the Microgrid of SVP cores

This section introduces the architectural features of the SVP multi-core implementation as far as necessary to understand the constraints on our compiler development. The SVP core implementation extends an existing ISA with extra instructions capturing the model properties. These instructions are shown in Figure 3. In addition to that, the SVP core also implements registers as i-structures [14] , which can suspend and reschedule any number of hardware supported threads (currently 256 per core), implementation details are presented in [10]. Parameterised 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.

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 [15], 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. N.b., the **allocate** instruction identifies the cluster of cores that a **create** instruction will be distributed to and this requires an implementation-dependent place type.

In SVP, inter-thread communication is implemented by reading from and writing to *shared* synchronising objects. This occurs between parent and first child thread and between adjacent threads created in a family, supporting only linear

allocate	gets a family entry in memory and sets the context and the place where the family will be run on.	**setstart**	sets the starting bound of family.
create	action which generates a family of threads.	**setlimit**	sets the limit bound of the family.
swch	marks the use of long-latency-operations results.	**setstep**	sets the step of the family.
end	marks the end of the thread code.	**setblock**	sets the maximum number of threads per code.
break	within a thread code, terminates the current family context and returns the value to parent.	**setbreak**	sets an object in parent's context for collecting family breaking value.

Fig. 3. List of SVP instructions which can be added to an existing ISA.

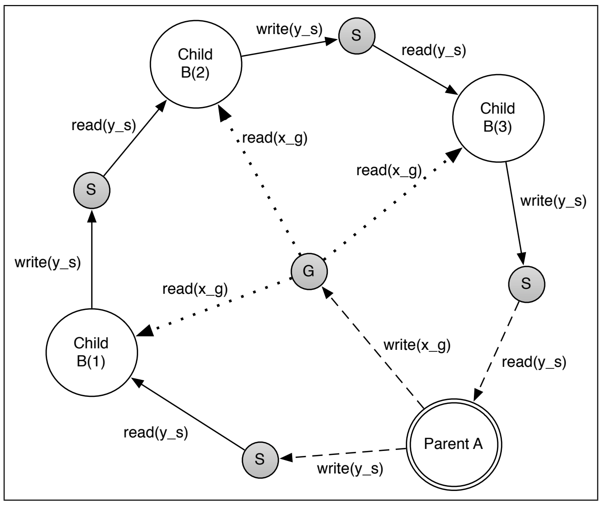

Fig. 4. Example of SVP inter-thread communication. After the *issue point* of an SVP create event in Parent A, this is an example of SVP inter-thread communication using the two different SVP synchronised communication channels, i.e *global* synchronising object *x_g* and *shared* synchronising object *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 synchronising objects: *read(Var)* and *write(Var)* (they are not function calls) where *Var* is a synchronising object. Note that the last *read(y_s)* (between child B(3) and Parent A) is valid in Parent A only after synchronisation (*completion point*).

dependency chains and between the last child thread and the parent. This linearity ensures the model's freedom from deadlock and also provides an abstraction of locality of communication between threads in a family. This inter-thread communication is illustrated in Figure 4. In the Microgrid of SVP cores, for every *shared* synchronisation 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 4, *read(y_s)* and *write(y_s)*). These names are derived from a partitioning of the thread's context into four register classes as presented in Figure 5. This partition is specified by an *assembly directive*. 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*.

Register classes	
locals	These registers are read/write to the local thread only.
globals	These registers are written by the creating thread and are read-only to all threads within a family.
shareds	These registers are for communication between adjacent threads in the family, they are **written** once by the local thread and may be read locally or by the successor thread in the family. The synchronisation event is on the first write.
dependents	These registers are **read** only and access the previous thread's shared registers. The synchronisation event is on the first read.

Fig. 5. List of SVP register classes.

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.

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).

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 for this is much smaller, it requires a single instruction to initialise a register with a stack pointer plus normal stack management overhead.

4. Overview of μTC Programming Paradigm

This section presents the language implementation of the SVP general abstract concurrent execution model into a C-based language. Introduced in [16], the *μTC 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 predefined to a different value when each thread starts;
- *thread creation*, an elementary action of the abstract machine causing the creation of a thread family;
- *synchronising 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 synchronisers*, 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.

Figure 6 illustrates a μTC 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 done through concurrent operations in the thread function *ddot*. The main μTC construct is the implementation of the SVP **create** action which generates a family of threads with the parameters specified in the construct: **create**(*family_id*; *place_id*; *start*; *limit*; *step*; *block*; *break_id*)*thread_function*. 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. Further parameters are explained in Figure 7.

5. Compilation schemes

As shown in Figure 1, our compiler takes the μTC language (Section 4) as input and follows the compilation schemes of this section for generating code for the Microgrid (implementation architecture of SVP model, Section 3). This section uses a simple formalism for expressing these compilation schemes. The transformation rule T is an

```
/* 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)
{
  create(fid;;0;1000;1;;) ddot(x = 0, u, v);
  sync(fid);
}
```

Fig. 6. A μTC 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 μTC **index** construct. The read to 'res' synchronises with the previous thread; the two memory loads can be performed concurrently before synchronisation; the read to 'res' in the next thread completes after the current thread completes its write to 'res'. After synchronisation (i.e. after the μTC **sync** construct),'x' contains the result of ddot in the parent.

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.

Fig. 7. Create parameters which set up the family definition.

abstract representation of the μTC compiler. All the transformation rules described in this section are extensions of the regular C compilation schemes.

A *thread function* is transformed by the μTC compiler into a sequence of instructions and assembler directives, as the compilation scheme in Figure 8 illustrates. On the right-hand side of this figure, the assembler directive *.registers* is responsible of defining the register windows where "gi,si,li" are compiler-calculated numbers of registers for *global*, *shared* and *local* classes for the integer thread context *thread_name* and respectively "gf,sf,lf" for the floating-point thread context. The **end** instruction is the exit point of the concurrent region. If *Body* does not have extra μTC constructs then regular C compilation schemes apply.

Any *thread function* may create a subordinate family. The compilation scheme in Figure 9 shows the SVP **create** action on the left-hand side and the transformed

$$T\left[\!\!\left[\begin{array}{l}\textbf{thread}\ \mathit{break\_type}\\ \qquad \mathit{thread\_name}(\mathit{thread\_args})\\ \{\\ \quad \mathit{Body}\\ \}\end{array}\right]\!\!\right] \Rightarrow \left\{\begin{array}{l}\quad .\mathit{asm}\quad \mathit{thread\_name}\\ \quad .\mathit{registers}\quad gi, si, li\quad gf, sf, lf\\ \mathit{thread\_name}:\\ \quad T[\![\mathit{Body}]\!][\![E[\![\mathit{thread\_args}]\!]]\!]\\ \quad \textbf{end}\end{array}\right.$$

where E is defined as,

$$E\ [\![\ \mathit{type}\ \mathit{id\_name},\ \mathit{Rest}]\!] \Rightarrow \quad E[\![\mathit{Rest}]\!]$$

$$E\ [\![\textbf{shared}\ \mathit{type}\ \mathit{id\_name},\ \mathit{Rest}]\!] \Rightarrow \quad \mathit{id\_name} \cup E[\![\mathit{Rest}]\!]$$

$$E\ [\![\ \mathit{type}\ \mathit{id\_name}]\!] \Rightarrow \quad \emptyset$$

$$E\ [\![\textbf{shared}\ \mathit{type}\ \mathit{id\_name}]\!] \Rightarrow \quad \mathit{id\_name}$$

and F as,

$$T\ [\![\ \mathit{id\_name}_1 = \mathit{id\_name}_2;\quad \mathit{Rest}]\!]\ [\![\mathit{shared\_set}]\!]$$

$$\Rightarrow \left\{\begin{array}{l|l}\mathit{write}(\mathit{id\_name}_1) = \mathit{id\_name}_2; & \mathit{where}\ \mathit{id\_name}_1 \in \{\mathit{shared\_set}\},\\ T[\![\mathit{Rest}]\!][\![\mathit{shared\_set}]\!] & \quad \mathit{id\_name}_2 \notin \{\mathit{shared\_set}\}.\end{array}\right.$$

$$\Rightarrow \left\{\begin{array}{l|l}\mathit{id\_name}_1 = \mathit{read}(\mathit{id\_name}_2); & \mathit{where}\ \mathit{id\_name}_2 \in \{\mathit{shared\_set}\},\\ T[\![\mathit{Rest}]\!][\![\mathit{shared\_set}]\!] & \quad \mathit{id\_name}_1 \notin \{\mathit{shared\_set}\}.\end{array}\right.$$

$$\Rightarrow \left\{\begin{array}{l|l}\mathit{write}(\mathit{id\_name}_1) = \mathit{read}(\mathit{id\_name}_2); & \mathit{where}\ \mathit{id\_name}_1, \mathit{id\_name}_2 \in \{\mathit{shared\_set}\}.\\ T[\![\mathit{Rest}]\!][\![\mathit{shared\_set}]\!] & \end{array}\right.$$

$$\Rightarrow \left\{\begin{array}{l|l}\mathit{id\_name}_1 = \mathit{id\_name}_2; & \\ T[\![\mathit{Rest}]\!][\![\mathit{shared\_set}]\!] & \mathit{otherwise}.\end{array}\right.$$

Fig. 8. Compilation scheme T for a thread function where arguments have to be determined whether or not there are potential synchronising objects declared as **shared** variables in the μTC program (i.e. used as SVP synchronised communication channels). The result of the transformation T is the corresponding assembly procedure (i.e. starting with ".asm thread_name", finishing with "**end**") on the right-hand side. The compilation scheme E determines whether thread arguments are **shared** variables or not and returns a potentially empty set of **shared** variables. This set is then used in combination with the compilation scheme F for dealing with synchronising variables used in the statements of *Body*. The compilation scheme F assumes that the code has been flattened. Moreover, the variable identifiers $\mathit{id\_name}_1$ and $\mathit{id\_name}_2$ can refer to the same variable and are then distinguished by the index number. And, $\mathit{write}(x)$ and $\mathit{read}(x)$ are not function calls but accessor methods on a synchronising object x.

output after compiler transformation on the right-hand side. The first component is the initialization of thread arguments in the parent context via $T[\![\mathit{thread\_args}]\!]$. The **allocate** instruction holds offsets of the parent context for parameter passing to the child context. "$\epsilon(w)$,$\epsilon(x)$" are compiler-calculated offsets in the parent

$$T\left[\!\!\left[\begin{array}{l}\mathbf{create}(\mathit{family\_id};\mathit{place\_id};\\ \quad \mathit{start};\mathit{limit};\mathit{step};\\ \quad \mathit{block};\mathit{break\_id})\\ \quad \mathit{thread\_name}(\mathit{thread\_args});\end{array}\right]\!\!\right] \Rightarrow \left\{\begin{array}{l} T[\![\mathit{thread\_args}]\!]\\ \mathbf{allocate}\ T[\![\mathit{family\_id}]\!], \epsilon(w), \epsilon(x), \epsilon(y), \epsilon(z)\\ \mathbf{setplace}\ T[\![\mathit{family\_id}]\!], T[\![\mathit{place\_id}]\!]\\ \mathbf{setstart}\ T[\![\mathit{family\_id}]\!], T[\![\mathit{start}]\!]\\ \mathbf{setlimit}\ T[\![\mathit{family\_id}]\!], T[\![\mathit{limit}]\!]\\ \mathbf{setstep}\ T[\![\mathit{family\_id}]\!], T[\![\mathit{step}]\!]\\ \mathbf{setblock}\ T[\![\mathit{family\_id}]\!], T[\![\mathit{block}]\!]\\ \mathbf{setbreak}\ T[\![\mathit{family\_id}]\!], T[\![\mathit{break\_id}]\!]\\ \mathbf{create}\ T[\![\mathit{family\_id}]\!], \mathit{thread\_name}\end{array}\right.$$

Fig. 9. Compilation scheme for μTC **create** action. There are three parts to distinguish in the right side: first the initialization of the thread arguments with $T[\![\mathit{thread\_args}]\!]$; second the family settings of the to-be-created family with the **set**-like instructions; and third the **create** instruction.

context for the first passed *global* and first passed *shared* for integer variables and respectively "$\epsilon(y)$,$\epsilon(z)$" for floating-point variables as explained in [10]. Then the **set**-like instructions set up the configuration of the family of threads to be created. The **create** instruction is responsible for the generation of the family of threads *thread_name*.

$$T\,[\![\mathbf{break}(\mathit{expr});]\!] \Rightarrow \quad \mathbf{break} \quad T[\![\mathit{expr}]\!]$$

Fig. 10. Compilation scheme for μTC **break** action. The corresponding thread function is defined as breakable by the presence of a *break_type* other than **void**.

A *thread function* can be defined as breakable by using a *break_type* different from **void** and by using the μTC **break** construct in its *Body* and is transformed as shown on Figure 10. This transformation involves the evaluation of the value of expression *expr*. The type of this value is then compared to the *break_type*. And, this value will be collected in the parent's context as shown in Figure 9 with the object defined with the **setbreak** instruction. Moreover an extra exit edge is added to model a new potential exit point in the execution path of the program.

Figure 8 also illustrates the compilation of a *thread function* where any of its parameters uses a synchronised communication channel declared as **shared** in the μTC program. Therefore all statements held in *Body* which are using variables declared as **shared** are preserved in the transformed program shown on the right-hand side of the figure. There is a distinction made by the compiler when a μTC **shared** variable is accessed: marked in the compilation scheme with accessor methods, *write(id_name)* and *read(id_name)* respectively mapped in *shared* and *dependent* register classes.

The μTC language allows the use of regular C function calls. Nevertheless, the compiler transforms them as shown in Figure 11. On the right side of the figure (i.e. the parent context), instead of conventional caller code, the compiler produces a family of one thread targeting a *call_gate* and generates a family identifier. As shown in Figure 12, the call_container *function_name (function_args)* is generated and wrapped around the conventional sequential generated caller code in a separated context, *call_gate*. This allows us the proper use of sequential functions in thread functions. The callee generated code is conventional sequential code.

$$T\,[\![\mathit{function\_name}(\mathit{function\_args})]\!] \Rightarrow \begin{cases} \textbf{allocate}\ T[\![\mathit{family\_id}]\!], \epsilon(w), \epsilon(x), \epsilon(y), \epsilon(z) \\ \textbf{create}\ T[\![\mathit{family\_id}]\!], \textbf{call\_gate}\_\mathit{function\_name} \\ \textbf{sync}\ T[\![\mathit{family\_id}]\!] \end{cases}$$

Fig. 11. This is a special case of compilation scheme T where a regular C function call is present in the statement list held in *Body*. On the right side, two assembly procedures are the result of the transformation: the *call_gate* and the *thread_name*. The compilation scheme C corresponds to regular C compilation which, in the example, compiles the function call with sequential conventions.

$$\begin{array}{ll} \quad .asm \quad \mathit{call\_gate} & \quad .asm \quad \mathit{thread\_name} \\ \quad .registers \quad gi, si, li \quad gf, sf, lf & \quad .registers \quad gi, si, li \quad gf, sf, lf \\ \mathit{call\_gate\_function\_name}: & \mathit{function\_name}: \\ \quad T[\![\mathit{function\_name}(\mathit{function\_args})]\!] & \quad T[\![\mathit{statements};]\!] \\ \quad \textbf{end} & \end{array}$$

Fig. 12. This is the *call_gate* with the conventional generated caller code introduced by the compiler (on the left-hand side) and the conventional generated callee code (on the right-hand side). Note that the compilation scheme T reflects regular C compilation schemes here.

6. Challenges of integrating compilation schemes in sequential compiler

These compilation schemes have been implemented in our compiler. The challenges encountered to integrate the compilation schemes into an existing compiler framework are discussed in this section. We have used the GCC 4.1 framework for the realisation of this work.

6.1. *General difficulties*

[17] exposes limits and dangers of compiler-driven optimizations when using library directives for exposing concurrency in code. However, integrating SVP concepts na-

tively into an existing compiler infrastructure allows us to reuse existing and adaptable optimizations which have been researched and improved for decades. Thus, it allows us to perform more efficient code generation instead of reimplementing from scratch all existing optimization algorithms. However, in order to achieve this, the compiler must be made aware of the SVP constructs and semantics and must adapt to these in the code as sequential-based and parallel-extended optimizations. Moreover, the compiler must protect concurrent regions in the code (especially synchronising objects in μTC) from aggressive code optimizations.

Besides the well-known issues of compiler code size and compiler code complexity, the major problem encountered is that this sequentially-oriented compiler infrastructure is concurrency-agnostic. Such compilers have been developed over years following an incremental development pattern rather than rebuilding the whole system from scratch when new assumptions or components are inserted. Furthermore, the compiler assumes sequential execution throughout its compilation stages: from optimization algorithms to code generation schemes. Our approach requires concurrency support, not by using plug-ins or external libraries, but by extending the compiler infrastructure, i.e. models presented in [18]. The novelty and hence contribution of this work, in addition to the obvious, i.e. an efficient working compiler for the Microgrid architecture, is in identifying and presenting the challenges encountered while extending a sequentially-oriented compiler.

6.2. *Parallel assumptions inserted*

A bird's eye view of the integration of the SVP constructs would be the extension of the C sequentially-oriented front-end with the new language constructs which then map into extra instructions and special features in the back-end as the compiler transformations in Section 5 illustrate. However looking closer at the compiler infrastructure, it also has to support new assumptions for dealing with the new idioms in order to produce valid code. Therefore, new nodes in the tree representation and new objects in the machine representation are added to support parallel semantics throughout the compilation stages. The extension of the infrastructure with new idioms implies the addition of new semantics in the assumptions made in the compilation schemes. Therefore, the high-level to low-level representation translations are extended to propagate the SVP semantics.

Sequentially-oriented compilers (in our case, the GCC compiler) make the assumption that only a single thread of control is running at any time. In the μTC language, the *thread function* is the smallest composition unit which is seen as a concurrent region with its own context of objects. Thus, a family of threads is a set of concurrent regions which are executed as multiple concurrent control flows running at the same time between the issue point (i.e. **create**) and the completion point (i.e. **sync**) in the parent context, as shown in Figure 2. Furthermore, communication is allowed between concurrent regions using synchronising objects. In passing parameters, communication may occur between the issue point and the completion point in the parent thread and all threads using *globals* and with the first child using *shareds*. Also synchronised communications may occur between sibling threads using only *shared* synchronising objects, as shown in Figure 4. These are expressed in the internal compiler representations with an extra attribute of a given object type. Nevertheless, they cause problems with sequential assumptions in the optimizations if the compiler does not sufficiently distinguish between the synchronising objects and the regular ones.

For example in Figure 6, a conventional C compiler would optimize away the statement computing the reduction in thread function *ddot* during dead-code-elimination since the code resides in a leaf of the control flow graph which never returns anything (i.e. the statement is assumed useless). In SVP, the statement has different semantics since a synchronising object is present. We observe that some optimizations are harmless for sequential C code but dangerous for parallel μTC code.

```
/* thread function definition: 1 shared, 1 global. */
thread void foo(shared double X, double Y)
{
  tmp = X;
  create(fid;;0;1000;1;;) bar();
  sync(fid);
  X = tmp;
}

thread void main(void)
{
/* point of use: create a family of 1000 threads */
  create(fid;;0;1000;1;;) foo(a = 0, b);
  sync(fid);
/* after synchronisation ‘X’ contains the result of foo in the parent. */
}
```

Fig. 13. Example μTC: Shared object used as a token to enforce sequential constraints.

Another example in Figure 13 uses a synchronising object as a token across adjacent threads, with C semantics, the statements $tmp = X$; and $X = tmp$; would be optimized out with the use of various standard optimizations such as copy-propagation, instruction reordering or combining optimizations. With μTC semantics, the optimized program would not be semantically equivalent to the original program because of broken synchronised communication channels. The μTC compiler is aware of those communications and ensures their validity even in the case of aggressive rearrangements in instruction scheduling. Although those optimizations obey 100% the C semantics, they can change the meaning of μTC programs and cause visible effects that can break the μTC semantics. The optimizations have been adapted in order to avoid endangering the semantics of μTC code.

6.3. *Resource mapping*

In the case of μTC, a new kind of object has been added with different access semantics: *shared* synchronising objects whose read and write operations have the i-structure semantics described in [14]. These are intended to directly map to registers with i-structure behaviour in hardware. Here, we observe that the approach taken by the SVP core architecture implementation creates an issue that we believe has not been researched previously: as a way to maximise the number of threads per core within a limited physical register file, threads can be created with *variable register windows* where a variable number of physical registers are mapped in the architectural register window of each thread for either local (exclusive) use, or physically shared as i-structures with other threads. An example is given in Figure 14. This removes the long standing assumption that the size of the architectural register window is a constant for all programs. Also, the compiler cannot reuse physically shared registers via spills: a spill itself would require a read from the register, thus force synchronisation with the previous thread and reduce concurrency; a write of a local temporary value that motivates a spill would unblock any read in the next thread with the temporary value instead of the final desired dependency. Therefore, all architectural registers cannot be reused arbitrarily modulo spilling. Only local registers, exclusive to each thread, can be used to map temporaries and automatic variables.

As a naive route towards reusing existing *register allocators* for the C language, we could arbitrarily select a common fixed number of registers for shared and local use by all programs; these numbers would be part of a standard application binary interface that compilers would adhere to when generating code. However, a conservative selection that would both leave sufficient room for sharing between threads and local use by each thread would cause excessive pressure on the register file, by forcing the hardware SVP processes to allocate a large physical register window for every thread created; this in turn would reduce the local concurrency by capping the number of threads that can be simultaneously allocated on each core. As this has a direct impact on the amount of latency the architecture can tolerate on asynchronous operations, we deem it desirable to adapt the architectural register window size to the requirements of each thread function.

The framework where this issue is addressed can be described as follows:

- the selection of the layout of the visible register window for a given thread function is the *responsibility of the compiler*; it is provided to the assembler along with the generated code for the thread function;
- given a microthreaded architecture, this selection is *constrained by the architectural limits*, namely the maximum number of visible architectural registers per class (e.g. integers/floats) and possible special registers which cannot be shared (e.g. ReadAsZero);

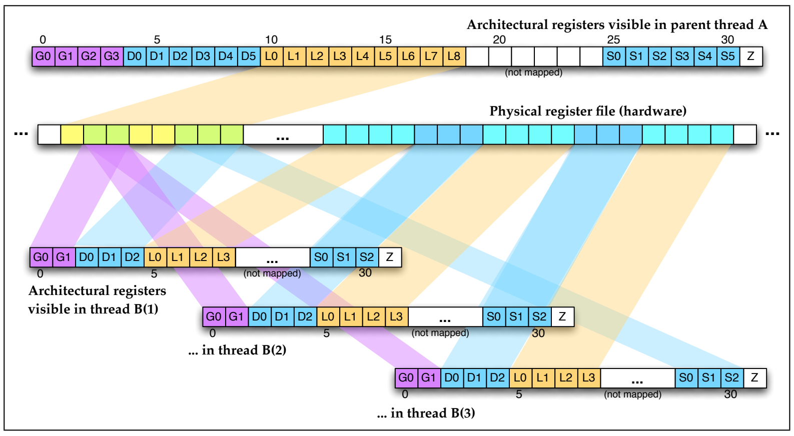
Architectural registers visible in parent thread A
0 5 10 15 20 25 30
G0 G1 G2 G3 D0 D1 D2 D3 D4 D5 L0 L1 L2 L3 L4 L5 L6 L7 L8 S0 S1 S2 S3 S4 S5 Z
(not mapped)
Physical register file (hardware)
Architectural registers visible in thread B(1)
... in thread B(2)
... in thread B(3)
G0 G1 D0 D1 D2 L0 L1 L2 L3 ... S0 S1 S2 Z

Fig. 14. 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.

- due to μTC semantics, the selection is further *constrained by the arbitrary composability of constructs* allowed by C and, by extension, μTC; in particular, the maximum number of registers that can be declared as parameters for a given thread function F to be shared at run-time with its parent cannot be greater than the maximum number of local registers available for sharing by any thread function that can create a family of F.

We formalise as follows: considering a single register class on an SVP architecture implementation where there are at most R visible registers, the compilation of a thread function F_p and any of its potential child F_c is constrained by the following formulae:

$$2S_p + G_p + L_p \leq R \quad (1)$$
$$S_c + G_c + C_p + L_p^* \leq L_p \quad (2)$$

where $S_{c/p}$, $G_{c/p}$ and $L_{c/p}$ are the number of shared, global and local registers in the register window layout declared to the assembler for F_c and F_p, C_p is the number of overhead local registers to perform the create operation, and L_p^* the number of local registers that must be reserved in F_p for other uses (e.g. thread local storage pointer). Formula 1 represents the architectural constraint; the number of shared registers is represented twice since there are sharing regions with both adjacent threads, as illustrated in Figure 14. Formula 2 represents the sharing of the local register window of a thread with its child family. Both formulae imply a recursive constraint on the entire concurrency tree of a program. In addition to these "hard" constraints, the architecture also suggests optimising register file usage, by choosing register windows for all thread functions involved in a program such that the total number of physically required registers is kept as small as possible.

This framework is new since an optimal solution to this formalised constraint system is a code generation algorithm where the number of local registers available for code generation is *both an input and an output* of register allocation. In particular, while the selection of a larger number of local registers in general allows for reduced memory accesses, it is even more important to allow for more concurrency by reducing pressure on the register file.

7. Experimental evaluation

We have utilized the Livermore kernels [19] as a benchmark for testing code correctness and evaluating the code quality of the μTC compiler. Our compiler prototype is based on the GCC 4.1 framework and extended by the presented compilation schemes from the μTC language to an Alpha-based SVP core, for which we have a many-core emulation (Microgrid) which provides a cycle-accurate simulation of execution time [11]. The Livermore kernels gather a set of scientific kernels designed for benchmarking parallel computers (i.e. Hydrodynamics Fragment, HF; Inner Product, IP; Banded Linear Systems Solution, BLSS; Tridiagonal Elimination, TDE; General Linear Recurrence Equations, GLRE; Equation of State Fragment, ESF). Our tool chain allows us to verify the results of compilation and execution on this platform against the execution of the same code on a conventional platform by transforming it into sequential C code, compiling it and executing it on a conventional processor. Figure 15 compares the execution time of compiled Livermore kernels against optimized hand-coded ones. The compiled execution times are normalized against the hand-coded versions for execution on a single core and a cluster of 32 cores.

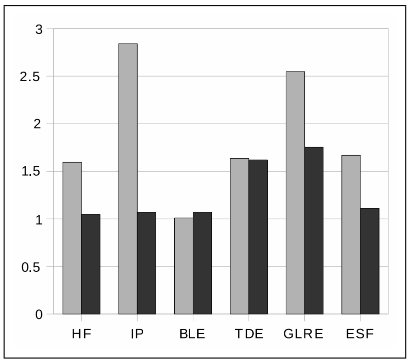

Fig. 15. Comparison of execution cycles between hand-coded Livermore kernels (set as one) versus Livermore kernels compiled with μTC GCC-based compiler with optimizations enabled (normalized) run on the Microgrid simulator. The light-grey bar corresponds to running with single-core execution, and the dark bar with multi-core execution (i.e. 32 cores in the experiments).

We evaluate the compiled code against the hand-coded assembly versions that can be considered as optimal solutions. We would expect the compiled code to be slower than the highly-optimized hand-coded assembly. We would also expect the code to scale with the number of cores and here we see the compiled μTC code scaling much better than the assembly version. For a single core, the compiled code varies from being as good as the hand-coded version (BLE) to a factor of 2.8 times slower (IP). IP performs only one multiplication and one addition in each thread and less efficient address arithmetic can not be masked. The average slowdown is just about 50%. When we execute on 32 cores, however, the smaller hand-compiled kernels execute less efficiently, as there are fewer threads per core and more stalls waiting for memory or synchronisations (e.g. in IP), whereas the longer kernels continue to utilise the pipeline more efficiency. At 32 cores the overhead of compiled code becomes less than 20% and in few cases, HF, IP, BLE and ESF are very close to the hand-coded assembly version.

8. Related work

OpenMP [20] with its pragma approach of exposing concurrency in programs, pressurises compilers with similar parallelism issues as us. To deal with non-sequential OpenMP code, compiler extensions have been incorporated in the GCC compiler 4.4. Nonetheless, it targets in particular SMPs with coarse-grained concurrency and lacks efficient inter-thread synchronisation [21]. In addressing both arbitrary composition and efficient inter-thread synchronisation the SVP model is more closely related to Cilk [22] and UPC [23]. Cilk has a fork-join concurrency approach, but it is not focused on dataflow machines. Moreover, the Cilk run-time scheduler is in charge of mapping concurrency to hardware resources; in contrast μTC constructs map directly to hardware instructions which can implement both efficient delegation and thread interleaving in the pipeline. We observe though that UPC uses a similar approach in their GCC-based compiler extension with new language constructs at the front-end and middle-end levels. Even though UPC has a uniform programming model for both shared and distributed memory hardware, it does not capture properly the data dependencies in the SVP model.

9. Conclusion

Through the SVP compilation schemes, the paper has presented the challenges to integrate explicit concurrency idioms in a sequentially-oriented compiler. We want to emphasize the fact that these challenges are more general than this specific outcome. This is true for our own work, where we intend to reuse this compiler in targeting more general-purpose cores with software implementations of SVP, but also more generally, where synchronisation and consistency issues need to be retrofitted to an existing framework.

As most parallel paradigms are implemented as external libraries and external constructs [18, 24], the compiler is not able to interfere with concurrency regions directly during compilation in the sense that it can only distinguish small concurrent regions instead of treating concurrency as a whole. As a short term vision, using such paradigms permits us to reuse the existing entire sequential tool chain with lower-cost incremental improvements. In the long run though, fully-integrated concurrency in development tools are disruptive from mainstream and would require more investment in order to get efficient and optimized tools. This work favors the latter approach and presents the feasibility of such an approach using the SVP concurrency model. One major achievement of this work is the existence of the compiler per se. [9] has presented results proving the good performance of the SVP general concurrency model. To enable further performance experiments, our advanced working compiler allows us writing non-trivial code for our multi-core architecture implementation.

As future work, we plan to look at compile-time resource management by introducing family definition reconfiguration. This would allow the compiler to redesign the μTC code by changing the concurrency tree family settings, i.e. number of threads per core (*block* parameter), place of computation (*place_id* parameter), for any multi-core configuration of an SVP architecture implementation.

Acknowledgements

The SVP model and its implementation is a group effort and the authors would like to thank all members of the Computer Systems Architecture group for their input to this work: Mike Lankamp, Michiel W. van Tol, Li Zhang, Konstantinos Bousias and Joe Masters. The authors would also like to thank Michael Hicks and Simon Polstra for all the constructive criticisms they made during the elaboration of this work and their contribution to the compiler development. Finally, the authors would like to thank Raphael Poss for his scientific insights on this work. And we must acknowledge the support for this work from NWO in supporting the *Microgrids* project.

References

[1] D. Geer, Industry Trends: Chip Makers Turn to Multicore Processors, in *Computer*, 38(5), **ISSN 0018-9162**, pp. 11–13, 2005.

[2] H. Sutter, The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, in *Dr. Dobb's Journal*, 30(3), pp. 292–210, Mar. 2005.

[3] B. Chapman, The Multicore Programming Challenge, in *APPT '07: Proceedings of Advanced Parallel Processing Technologies*, 4847, **ISBN 978-3-540-76836-4**, pp. 3, 2007.

[4] E. Haritan, T. Hattori, H. Yagi, P. Paulin, W. Wolf, A. Nohl, D. Wingard and M. Muller, Multicore Design is the Challenge! What is the Solution?, in *DAC '08: Proceedings of the 45th annual conference on Design automation*, **ISBN 978-1-60558-115-6**, pp. 128–130, 2008.

[5] S. Amarasinghe, (How) Can Programmers Conquer the Multicore Menace ?, in *PACT '08: Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques*, **ISBN 978-1-60558-282-5**, pp. 133–133, 2008.

[6] H. Sutter and J. Larus, Software and the Concurrency Revolution, in *Queue*, 3(7), **ISSN 1542-7730**, pp. 54–62, 2005.

[7] H. Gabb, T. Mattson and C. Breshears, Thinking in Parallel - Three Engineers' Viewpoints, in *Intel Software Insight Magazine*, 16, pp. 24–26, Feb. 2009.

[8] C.R. Jesshope and A. Shafarenko, Concurrency Engineering, in *ACSAC '08: Proceedings of 13th IEEE Asia-Pacific Computer Systems Architecture Conference*, pp. 1–8, 2008.

[9] T. Bernard, K. Bousias, L. Guang, C. R. Jesshope, M. Lankamp, M. W. van Tol and L. Zhang, A General Model of Concurrency and its Implementation as Many-Core Dynamic RISC Processors, in *: Proceedings of IC-SAMOS '08*, **ISBN 9781424419852**, pp. 1–9, 2008.

[10] K. Bousias, L. Guang, C.R. Jesshope and M. Lankamp, Implementation and Evaluation of a Microthread Architecture, in *Journal of Systems Architecture*, 55(3), pp. 149–161, 2009.

[11] M. Lankamp, Developing a Reference Implementation for a Microgrid of Microthreaded Microprocessors, MSc thesis, University of Amsterdam, The Netherlands, Advisor: C.R. Jesshope, Aug. 2007.

[12] T.D Vu, C. R. Jesshope, Formalizing SANE Virtual Processor in Thread Algebra, in *ICFEM '07: Proceedings of 9th International Conference on Formal Engineering Methods*, **LNCS 4789**, Springer-Verlag, pp. 345–365, 2007.

[13] G. R. Gao and V. Sarkar, Location Consistency—A New Memory Model and Cache Consistency Protocol, in *IEEE Transactions on Computers*, 49(8), pp. 798–813, 2000.

[14] Arvind, Rishiyur S. Nikhil and Keshav K. Pingali, I-structures: Data Structures for Parallel Computing, in *ACM Transactions on Programming Languages and Systems*, 11(4), **ISSN 0164-0925**, pp. 598–632, 1989.

[15] The SPARC Architecture Manual Version 9, http://www.sparc.org/standards/SPARCV9.pdf, 2009.

[16] C.R. Jesshope, μTC - an Intermediate Language for Programming Chip Multiprocessors, in *ACSAC '06: Proceedings of Pacific Computer Systems Architecture Conference*, **ISBN 3-540-4005, LNCS 4186**, pp. 147–160, 2006.

[17] Hans-J. Boehm, Threads Cannot be Implemented as a Library, in HP Internet Systems and Storage Laboratory, HPL-2004-209, Nov. 2004.

[18] H. Kasim, V. March, R. Zhang and S. See, Survey on Parallel Programming Model, in *NPC '08: Proceedings of the IFIP International Conference on Network and Parallel Computing*, **ISBN 978-3-540-88139-1**, pp. 266–275, 2008.

[19] F. McMahon, The Livermore FORTRAN Kernels: A Computer Test of the Numerical Performance Range, in *Technical Report UCRL-53745*, Lawrence Livermore National Lab., USA, Dec. 1986.

[20] OpenMP Architecture Review Board, The OpenMP Specification for Parallel Programming, version 3.0, http://www.openmp.org/mp-documents/spec30.pdf, 2008.

[21] Karl Fürlinger and Michael Gerndt, Analyzing Overheads and Scalability Characteristics of OpenMP Applications, in *High Performance Computing for Computational Science*, 4395, pp. 39–51, 2007.

[22] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall and Yuli Zhou, Cilk: an Efficient Multithreaded Runtime System, in *SIGPLAN Notices*, 30(8), **ISSN 0362-1340**, pp. 207–216, 1995.

[23] T. El-Ghazawi and L. Smith, UPC: Unified Parallel C, in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, **ISBN 0-7695-2700-0**, pp. 27, 2006.

[24] David B Skillicorn and Domenico Talia, Models and Languages for Parallel Computation, in *ACM Computing Surveys*, 30(2), **ISSN 0360-0300**, pp. 123–169, 1998.