Towards Scalable I/O on a Many-core Architecture

Michael A. Hicks, Michiel W. van Tol, Chris R. Jesshope
Informatics Institute, University of Amsterdam, The Netherlands
{m.a.hicks,mwvantol,c.r.jesshope}@uva.nl

***Abstract*—The limitations of conventional processor performance scaling mean that general purpose many-core processors are increasingly becoming a reality. Conventional hardware device input/output (I/O), interrupt handling and operating system stacks scale poorly and are inefficient when compared with the parallelism that these architectures provide. Many-core I/O requires a decentralised approach where not every core is directly connected to the I/O infrastructure. As such, this paper discusses a software and hardware model designed to take full advantage of I/O parallelism in the Self-adaptive Virtual Processor (SVP) concurrency model and the Microgrid many-core architecture.**

The generic software I/O stack presented describes a high-level method by which clients and I/O resources can communicate and synchronise in a parallel and decentralised environment. The associated hardware implementation provides a facility to the higher-level interface through the introduction of specialised *I/O Cores* which enable direct high-speed communication between external devices and the bespoke on-chip Cache-Only Memory Architecture (COMA).

***Index Terms*—Many-core, I/O, Parallel I/O, Microgrid, SVP, Operating Systems, Parallel Architectures**

I. Introduction

With processor hardware synthesis approaching physical limitations, many-core processors are no longer the reserve of only scientific and super computing [1], [2]. The implications of silicon scaling on clock frequency and power (thus also, heat) density provide a ceiling for conventional microprocessor designs. One widely approached method of resolving this performance restriction is the design and implementation of many-core processors and associated programming models for massive parallelism. As such, highly parallel, many-core computing is becoming an inevitability. The Self-adaptive Virtual Processor (*SVP* – Section II-A) and *Microgrid* (Section II-C) is one such model [3], and associated hardware implementation, which resolves to relieve this constraint by capturing and implementing maximum concurrency.

This paper presents a design for general purpose device input/output, and associated interrupt system, which works at both the level of the SVP concurrency model, in an *operating system* stack, and also with the associated hardware implementation in the many-core Microgrid.

A. Motivation

The SVP model and Microgrid implementation have been described in great detail in previous publications, notably [4], [5]. As part of the AppleCore project[1], it is proposed not only as a solution for specialised scientific computation but also, importantly, as a system for general purpose computing.

[1]Project funded by the European Union, project grant no. FP7-ICT-215216.

The proposal of a more general purpose Microgrid has initiated great consideration and research into operating system features. Parallel architectures are necessitating a far more decentralised operating system design approach [6], [7], since it is not possible for every core to be directly coupled to the I/O infrastructure. As such it is desirable to have an I/O facility that is not merely an afterthought, but fundamentally engineered to make full use of the parallelism provided by a many-core system, the SVP/Microgrid model, for communication with a diversity of external devices. This operating systems research takes place under the broader framework of the EU funded *AppleCore* project (see Section VI).

B. Contribution

The considerations, proposals and contributions of this paper are chiefly found in the novel methods with which software and hardware *Input/Output* communication and interrupts with external devices can be implemented within the SVP concurrency model and in the Microgrid hardware, but also more generally in many-core architectures and environments. The proposed I/O scheme consists of novel implementations both at the abstract level of concurrency, making use of the features provided by SVP, in an operating system software stack, and also in the Microgrid many-core hardware, introducing the concept of an I/O core and the use of the efficient on-chip COMA memory system for high data-throughput with a variety of devices.

The result is a highly scalable and parallel I/O architecture that, in the hardware implementation, bypasses external memory bus bandwidth limitations and contention, for efficient parallel I/O. The model is completely decentralised from a monolithic operating system model and embraces the idea of parallel services and as such is well placed for integration into a distributed operating system kernel.

Although in this paper the I/O mechanism is applied specifically to the SVP model and Microgrid hardware, the approach is sufficiently generic to be of interest to high-speed I/O in other parallel architectures and operating system environments.

II. Overview of SVP and the Microgrid

The abstract SVP concurrency model is implemented by both a concurrent programming language called μTC [8] (Section II-B) and an associated hardware implementation called the Microgrid [5], [3] (Section II-C), however there exist other more generic implementations such as the Pthreads back-end [9]. The μTC language captures the concurrency expressible in the SVP model and is intended as a system

level language, not an end user language. The SVP model and Microgrid architecture are mature works and currently boast a fully-fledged development and experimental toolchain, including compilers for μTC [10], a cycle accurate processor and memory simulator for the Microgrid and a variety of benchmarks.

A. The SVP Execution Model

The SVP model is designed to allow resource oblivious concurrency to be expressed in a uniform way in programs. Programs are constructed as a series of *thread functions*, which share some similarities with the specification of functions in conventional languages. Under execution, these threads (grouped together as *families*) form a hierarchical concurrency tree which can be dynamically mapped to hardware resources by virtue of several key properties of the model.

1) Threads and Families: A thread is specified in much the same way as function blocks in conventional sequential programming models. The key difference is that, in SVP, a program's execution is dictated by a series of (hierarchical) thread family *create* actions and synchronisations, whereby a thread function is spawned into a family of one or more statically identical threads; each executing thread characterised by a unique *index* value at run-time (similar in nature to a loop iteration counter). This facilitates the capturing of both homogeneous and heterogeneous concurrency behaviour, based on the thread index value. A diagrammatical overview can be seen in Fig.1.

Parameters to thread families and communication between threads are expressed by *global* and *shared* parameters. *Global* parameters are read-only objects passed to a child thread family once at the point of create; each child thread in the created family may read these values. Shared parameter objects specify unidirectional communication between adjacent (by index sequence) pairs of threads only, in the form of an I-structure. A read to a shared object in one thread of a family will block execution of that thread until the previous thread in the family has written its value. These objects capture both general parameter passing and concurrent execution dependencies.

2) Synchronisation: A parent thread that has performed a create action can also, but not necessarily, perform an associated *sync* action on a subordinate family. This action blocks the parent thread until all threads in the subordinate created family have completed execution. A thread can also perform a *kill* action on an identified family which results in the termination of that family's execution.

3) Places: A particular computing resource is encapsulated and abstracted as a *place* in SVP. A place may contain an arbitrary quantity of computational resources. A created family of threads is bound to a particular place at run-time, specified either by the programmer or by a dynamic place allocation scheme. The mapping of the family to the particular resources of the place is decided by the run-time implementation of SVP. The concept of places is used as an abstraction to separate the concerns of exposing concurrency in programs and the mapping of that concurrency to resources; the model requires that only the concurrency be specified, and a mapping will be decided automatically.

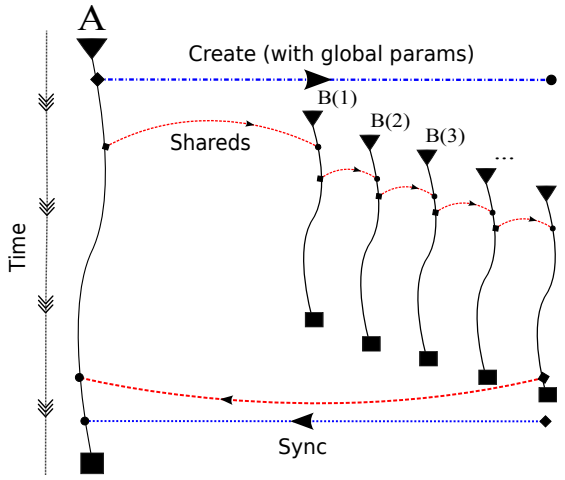

Fig. 1. SVP family creation and execution model. Thread A creates child family B. Execution resumes after *sync* in A when all threads in B have terminated. The Create action passes global parameters to all subordinate threads and shareds are synchronising objects between threads.

An important additional property of places is that they may be specified as *exclusive places*. Only one thread family may be active at any point in time at an exclusive place; it has exclusive resource access during its execution. This makes exclusive places useful for synchronising between families in the weak memory consistency model of SVP, and capable of providing exclusive access to data structures. Multiple SVP create actions to an exclusive place are queued and are not executed concurrently at that place.

4) Summary: SVP defines homogeneous families of threads, hierarchical composition thereof, the synchronisation between them in the form of shared parameters, and abstracts the concept of computational resources into 'places'. Further details on SVP, its specification and memory consistency model can be found in [4], [11].

B. μTC Language

Micro-threaded C (μTC) is a system level programming language designed to capture the concurrency features of the SVP programming model and facilitate their translation into the (potentially hardware) supported features of a particular implementation. In our current research this target is the Microgrid described in Section II-C but in principle the language could be compiled to various architecture implementations.

TABLE I
SVP ACTIONS EXPOSED IN μTC

Actions	Description
create(...)	*Create* a named family of threads of a specified size (number of threads), starting with a particular index value and using a particular index step size
sync(...)	*Synchronise* on a named family of threads, blocking execution in the current thread until all threads in the specified family have completed
kill (...)	Terminate the execution of a named family of threads

μTC is a restricted subset of the C language, augmented with the expressiveness required to implement SVP programs and with minimal intrusion into the conventional assumptions of the C language. Collectively, the keywords and type modifiers exposed in μTC are shown in Table I.

The usage of the constructs shown in Table I is exemplified by the μTC code in Fig.2. The example is simple in nature, implementing the inner product function, but of course μTC programs can be arbitrarily complex.

```
thread ddot(shared double res,  double* x, double* y) {
 index i;
  res = x[i] * y[i] + res;
}

thread main(void) {
  shared x; double u[1000], v[1000];
  create(family_id; place_id ;0;1000;1;) ddot(x = 0,u,v);
  sync(family_id );
}
```

Fig. 2. μTC example: simplified inner product from the BLAS library. The thread 'main' creates a family of 1000 threads of 'ddot'. The family of 'ddot' sums the multiplied values in a shared parameter which defines both communication and synchronisation between adjacent threads in the family; a thread of 'ddot' suspends on a read to the shared until the previous thread in the family has written to it. In thread 'main', execution is suspended at the point of 'sync' until all threads in the created family have completed.

μTC can be discerned from library approaches to concurrency because of its use of language primitives instead of library calls. This reflects both its assertion that concurrency expression should be the norm and also the desire for very fine grained concurrency, supported by an associated hardware implementation (in the case of this study, the Microgrid). However, this does not prevent it from being implemented as a library [9].

C. The Microgrid Architecture

The Microgrid is a massively parallel, many-core processor architecture which implements the SVP model of concurrency. It organises multiple in-order RISC cores into bus-rings, where each ring of cores constitutes an SVP place. Fig.3 shows a Microgrid of 128 cores organised into SVP places. This chip architecture, and its possible physical implementations, has been subjected to realistic area analysis in previously published work [3], [5].

1) SVP Core Pipeline: Each Microgrid core implements the SVP model of concurrency by extending an existing, simple in-order, processor ISA with the handful of required instructions for SVP behaviour, the details of which beyond the scope of this paper.

Potential high-latency operations, such as memory reads and floating point calculations, are trivially flagged by the μTC compiler; at execution time, the Microgrid can near costlessly switch between threads in the local pool at the occurrence of a high latency operation flag. The Microgrid cores facilitate potentially hundreds of threads to be active at any given time, allowing highly efficient interleaving. Thus,

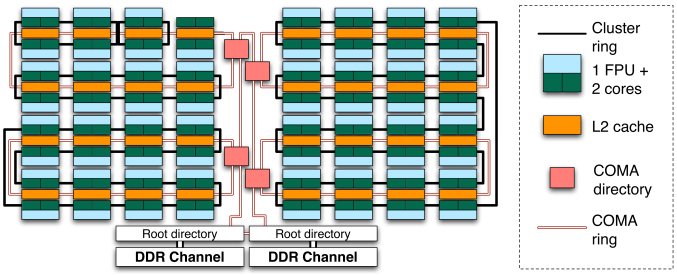

Fig. 3. A typical Microgrid configuration of 128 SVP cores arranged in place-rings, showing the on-chip COMA and external interface to DDR memory

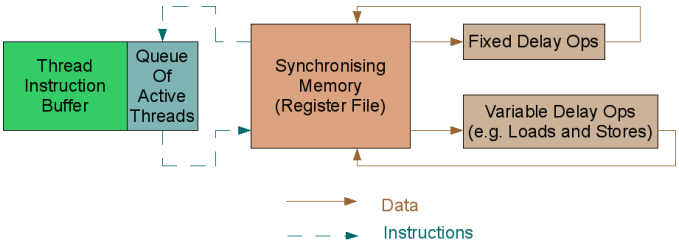

Fig. 4. An abstract representation of the behaviour of an SVP core pipeline. Active threads are queued and interleaved/suspended based on long-latency operations.

a highly important, and demonstrated [3], property of the SVP Microgrid is its tolerance of high latency operations. The general behaviour of the SVP pipeline is summarised in Fig.4.

2) Thread Creation and Delegation: An SVP create process delegates a family of one or more threads, specified and compiled from μTC, to a particular place on the Microgrid, at which point a fixed mapping of threads to cores takes place. Family creation and delegation takes place using an on-chip delegation and synchronisation network. Importantly, the create action is a relatively low-cost process; typically a family of threads can be created in only a few cycles on the Microgrid.

3) Synchronising Register File: Each register in the register file of a Microgrid core is furnished with state-bits which are set based on the particular register's use as the target of an operation, and also to flag whether threads are waiting to read the register. A read operation to a register which is awaiting a write operation will suspend the reading thread until that write takes place. This enables a data-flow-like execution behaviour in hardware and also the simple modelling of the SVP shared objects. Register contexts are efficiently allocated from a large pool of registers at each core. Synchronisation through shared parameters at different cores is achieved through the intra-place ring bus.

4) On-Chip COMA Memory Model: As part of the Microgrid research, a custom Cache Only Memory Architecture (COMA) has been developed and extensively evaluated. The memory system is presented to programs as a single flat address space. Caches in the COMA system are diffused throughout the places on-chip. The number of caches per SVP place depends on the number of cores in that place. Cache lines migrate through the COMA hierarchy to the places at

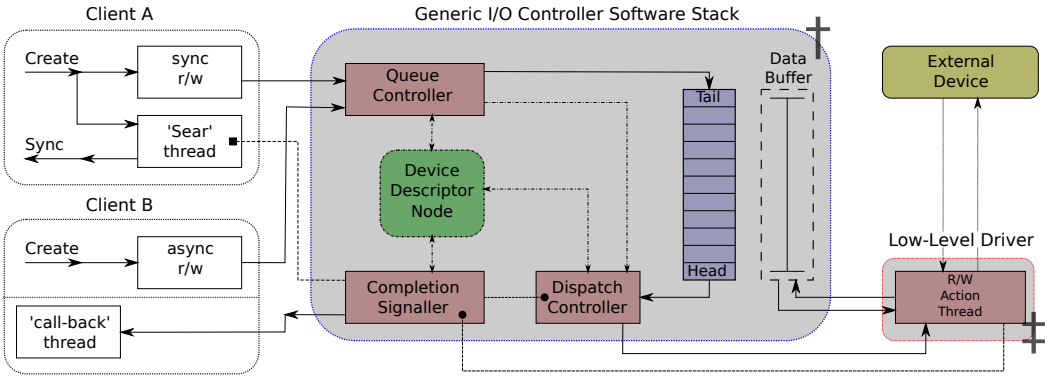

Fig. 5. An overview of the Generic Software I/O Controller software stack which implements generalised I/O events using μTC for SVP. All processes inside the enclosure annotated by the † symbol must take place at the driver controller's designated *exclusive place*. Processes in the low level driver enclosure, annotated with the ‡ symbol, are executed at the designated SVP I/O place. Objects in square-cornered boxes represent threads; round-cornered boxes represent conceptual groupings and privilege domains (where Client A and Client B are applications).

which they are being accessed. Further details of the COMA memory implementation can be seen in [5].

III. High-Level I/O Model and Implementation

When implementing device I/O in the SVP model, and also on the Microgrid architecture, there are two distinct levels at which the problem is addressed. This section is concerned with the implementation of I/O at the software and SVP level, where the traditional requirements of device I/O must be reconciled with the advantages, facilities and restrictions of the SVP programming model. In practical terms, this level will reside in the operating system driver API.

The SVP software level I/O controller described in this section has been implemented in μTC for SVP and functionally verified for behavioural consistency. It is a highly flexible model of I/O which is compatible with all of the implementations of SVP.

The I/O model implemented in SVP is designed to be as noninvasive as possible. This is achieved by providing a familiar software interface to client SVP programs: the standard 'read' and 'write' system/library calls are preserved and additionally the expected synchronous and asynchronous behaviour of I/O is provided. The example instance described in this chapter focuses largely on a device configuration of the 'request-response' form (e.g. a block device), since this is the most typical kind of device use-case, but the model is not limited to those types of devices.

It should also be noted that the model described in this section forms the generic basis of the I/O system. As a result, it is possible to encapsulate calls to this generic stack with further APIs, for instance a file system library which provides an abstraction over the structure of data on a device and would thus issue potentially multiple I/O actions for a single library call.

A. Synchronous and Asynchronous I/O Model

Fig.5 shows an overview of how synchronous and asynchronous read and write calls to the I/O subsystem work. This model is best explained by stepping through each component shown in Fig.5.

1) Client Interface: Client threads A and B issue synchronous and asynchronous read/write requests, respectively, to the I/O API. By specifying the *Device Descriptor Node* for the device, this triggers entry into the generic driver I/O subsystem and a switch to system level privileges (similar to a system call) by the way of an SVP *create* action, which creates an instance of the appropriate library call thread at the *device controller place*. The desired action, a size and pointer to the data in memory are also provided.

The *device descriptor node*, created by initialising a device, is a structure in memory which contains the necessary information about the I/O device, pointers to the relevant data structures and the exclusive device controller place

Client A, issuing a synchronous read/write action, also creates an empty thread referred to as a *sear thread* which itself simply suspends indefinitely. Client A will then wait for this thread to complete by performing an SVP *sync* action. When the synchronous I/O action has eventually been serviced by the software I/O controller, the sear thread will be terminated by the controller issuing an SVP *kill* action. The concept of the sear thread is necessary to allow the synchronisation by a client over the I/O event whilst still fully decoupling the execution of said client from the software I/O controller's context. This is due to a property of the SVP model, and other parallel models, which does not allow the synchronisation of two threads without a parent, child or sibling relationship.

Client B, issuing an asynchronous read/write action, provides the address of a *call-back thread* which will be created by the I/O software controller at the point of read/write service completion. Execution in Client B continues while the I/O operation takes place. It should be noted then that Client B should take care to account for the concurrent execution of its code by the asynchronous callback – the burden lies with the client to ensure synchronisation of its data-flow around this event.

2) Request Queueing: The service executes the *Queue Controller* at the exclusive 'Device Controller' place. At this point, the semantics of the SVP exclusive place ensure that SVP creates are queued and that only one instance of the

controller thread is executed at any given time. The queue controller now checks for space in I/O queue and, assuming there is a slot, adds the read/write request to the back of the queue. The queue controller then invokes the *Dispatch Controller* to notify it of a modification to the queue.

3) Request Dispatch: The *Dispatch Controller* examines the device descriptor node to see if an I/O operation is currently active. If an I/O operation is already active on the device, execution in the *software I/O Controller* terminates. Otherwise, it checks the pending queue of jobs. If the queue is not empty, it performs an SVP *create* on the *R/W Action Thread*, delegating it to the place at which the device exists and updates the device descriptor node to mark the device as active. The r/w action thread is created with the parameters of the particular read/write operation: the size, target, and a channel number. Execution in the *software I/O Controller* terminates and is fully decoupled from the low-level I/O action itself.

4) I/O Action: In the *R/W Action Thread*, a single I/O operation from the queue is served and actual communication with the device takes place. The r/w action thread suspends based on the interrupt of the *External Device* while the operation is serviced by the device. The resulting data is read from or written to the appropriate memory location. Upon completion of the individual r/w, an SVP *create* action is performed on the *Completion Signaller*, to signal completion of the low-level operation. Execution in the *Low-Level Driver* ends.

5) Client Synchronisation Signal: The *Completion Signaller* thread, created by the completed r/w action, is responsible for 'waking up' the client. Based on the information in the record for the particular I/O operation, the completion signaller will either: kill the appropriate *sear thread* in the client (allowing execution to continue in a synchronous I/O action) or perform an SVP *create* action on the specified 'call-back' thread in the client, for asynchronous I/O. The completion signaller also updates the Device Node Descriptor, signifying that the device is now 'free' and triggers the Dispatch Controller so that the next I/O operation can be processed.

6) General Notes: All device controllers in this model have to be initialised with a number of desired behavioural parameters. An important parameter is the location of the buffer for the I/O data. This can either be in the software device controller, where the appropriate I/O data will then be copied into the client's buffer as required, or the device controller can read/write the I/O data directly from/to the client's address space with each I/O action.

The previous example describes the *request-dispatch* device controller behaviour, where I/O requests trigger the dispatch of a low-level operation as required. However, the software I/O controller can also issue a continuously listening low-level thread which fills an internal buffer in the I/O controller stack. Subsequent I/O read requests can then be served from this buffer.

B. Low-Level Driver

The low-level driver thread is required to perform a single I/O action directly with the device on the SVP place at which communication with the device can take place. Additionally, it is the low-level driver that generates the events which drive the higher I/O stack.

The low-level driver implements a standard interface which takes the necessary parameters for an individual device communication (for instance a read or write action, seek and also initialisation activities). Thus, only the low-level driver need be reimplemented to work with new device interfaces. The actual nature of the internals of the low-level driver depend on which SVP implementation is used. On the Microgrid, the low-level driver would need to perform register and bus level communication with the external device itself. However, were the distributed Pthreads SVP implementation in use, this low-level driver could simply be a proxy to the system calls of one of the host environments in which execution is taking place.

C. I/O Places

An I/O place (‡ in Fig.5) is an SVP place at which a particular device exists, in the sense that communication with that device can be locally achieved. An I/O place does not necessarily have to be remote, however in the Microgrid hardware implementation of SVP that I/O place will correlate to the I/O Core, introduced in Section IV. In a more generic implementation of SVP, this I/O place could be a remote environment or machine that provides access to a particular resource. Importantly, more than one device may be associated with a particular I/O place – this is captured by the channel identifier.

D. Parallel I/O

The generic I/O model shown here, with the separation of atomic I/O actions from I/O places, makes parallel I/O relatively straight forward. If a particular device has many interfaces to the I/O infrastructure, where there are multiple places at which a low-level driver thread can communicate with a device, for instance in a RAID style configuration, then the *dispatch controller* can decompose a single I/O operation into multiple segments and distribute these to the array of I/O places. This approach is particularly useful when combined with lower-level implementation support, as is the case in the Microgrid (Section IV-E).

IV. Microgrid I/O Implementation

The SVP I/O subsystem described in Section III requires support from the specific implementation; it must provide the *low-level driver* service through an interface to external hardware. This section describes how such support can be implemented in the hardware of the SVP Microgrid.

The overall hardware scheme for I/O in the Microgrid can be seen in Fig.6, where external devices are connected via a HyperTransport-like bus to dedicated *I/O Cores*. The scheme is based around a model of message signalled interrupts communicated via a bus interface, similar in nature to MSI in the PCI specification [12].

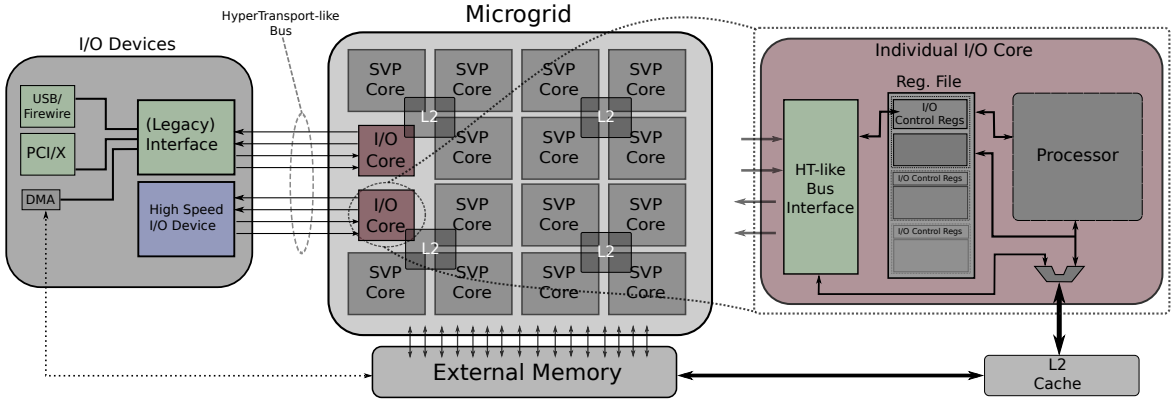

Fig. 6. A schematic overview of the architectural implementation of I/O in the SVP Microgrid. An enlarged individual I/O core is shown to the right. The L2 Cache shown is part of the on-chip COMA. Individual external devices are shown to the left, including a potential legacy DMA controller.

A. I/O Cores

A derivative of the regular SVP core, the *I/O Core* is the hardware realisation of the conceptual *I/O Place* introduced in Section III-C. It is the place to which the *low-level driver* is created for a particular read/write operation. It can be distinguished from other general purpose cores in the Microgrid by the following characteristics:

Bus Interface – the I/O core contains a Bus Interface device which connects the I/O Core to a bus for high-speed communication with external devices. This hardware is only present in I/O cores.

Simplified logic – given the specialisation of the I/O Core as a place only for performing I/O, it does not need to contain floating point logic, and the size of the integer register file can also be much smaller than general purpose cores (given the simplicity of the threads it will be executing).

I/O Instruction – the pipeline of an I/O core responds to a special I/O instruction which allows threads to issue and wait for events on the high-speed bus.

The purpose of the distinct I/O core is to relieve fully-fledged Microgrid cores of the burden of I/O operations and to allow a higher density of parallel I/O to take place in the Microgrid. Specifically, the I/O core allows the rest of the Microgrid to continue executing in parallel while I/O operations are serviced. The reduced complexity of I/O cores, needing only to handle simple atomic I/O operations, means that their footprint in the architecture is relatively small, potentially allowing for more parallel I/O places in a fixed budget.

B. High-Speed Bus

Each I/O Core interfaces with a HyperTransport-like [13] packet/message bus by means of a simple on-chip bus controller. While in principle a variety of packet-based buses could be used, the HyperTransport bus was selected based on its ubiquity in various high-speed I/O applications and the ability to interface with a variety of existing high performance devices. As can be seen in Fig.6, the bus need not be connected directly to a single device, but can itself be connected to another interface which multiplexes between several devices using channel identifiers, including, for example, legacy bus implementations.

The bus is specified as HyperTransport-*like* because it is unlikely that all of the features of the HyperTransport specification would necessarily be implemented; rather, the hardware specifications of HyperTransport serve as a basis for achievable transfer rates in future simulation.

The width of the HyperTransport-like bus is variable and a parameter chosen at implementation time. The current HyperTransport specification (3.1) stipulates a maximum bus width of 32 bits with a unidirectional transfer rate of *25.6 GB/s per second* (this can be seen as double to aggregate bidirectional transfers).

C. Bus Interface

The bus interface is a small piece of logic (comparable to a simple interrupt controller) present at I/O Cores which is responsible for connecting the processor pipeline with bus events and messages. The bus interface performs the following actions:

1) Compose and decompose bus messages appropriately, based on the associated channel information.
2) Deliver channel events into appropriate I/O control registers (Section IV-D2).
3) Perform reads and writes from/to memory of the data payload in I/O events at a specified location (this can be either to the memory subsystem directly, or to the register file).

The bus interface device is controlled entirely through the *ioctl* instruction. As noted in point 2, the bus interface is responsible for triggering changes in the state of the synchronising registers, based on bus events.

D. Device Communication and Interrupt Handling

Communication with external devices connected to an I/O Core take place using a special instruction, the execution of which is only defined at I/O Cores. The *I/O Control Instruction* (see Table II) is used to perform an individual read or write operation on the bus and provides an associated synchronisation with this operation.

TABLE II
I/O CORE 'I/O CONTROL INSTRUCTION'

Mnemonic	Operands (registers)		
ioctl	*control*	*size*	*src/dst*

Operands	Description
control	Specifies mode and channel (device) identifier
size	Requested size of read/write to perform to bus
src/dst	The target or source (buffer or register) of the read/write operation

1) I/O Control Instruction: The low-level driver described in Section III-B essentially implements a small thread containing an appropriate usage of the I/O Control Instruction, associating the dispatched I/O operation's parameters to the operands of the I/O Control Instruction, after performing any required intermediate operations.

The *control* operand specifies a register containing bit-level information stipulating whether a read or write operation should be performed, the channel identifier (which identifies a device) on which to broadcast or listen and whether or not the bus interface should read/write data directly to the COMA memory interface or the register file (the latter negates the use of the *size operand*). This information is contained in one of the standard registers in an I/O Core.

The two modes of operation, Register$\Longleftrightarrow$Bus and COMA$\Longleftrightarrow$Bus, allow for the differences in performance trade-offs; Register$\Longleftrightarrow$Bus communication is useful for low-level control tasks where the control operation can be passed as an thread parameter and accessing memory is an unnecessary performance overhead. High-volume transfers can, in principle, be carried out through registers, however the implementation of the Bus Interface also permits a direct mapping of the processor's address space on reads and writes (see Fig.6 and Section IV-C).

2) Synchronising and Interrupts: Synchronisation is achieved with an elegant modification to the semantics of Microgrid registers at I/O Cores. All registers in an core's register file in the Microgrid have a synchronising behaviour (Section II-C3), achieved through the use of state bits at each register.

When the *ioctl* instruction is issued in the pipeline, the *control* register operand has its synchronising state marked as 'pending'. The low-level driver thread can then perform a read on this control register, at which point its execution will be suspended in the normal way of a thread which reads a register flagged as the target of a write operation. When the appropriate I/O channel's transfer completes, the I/O interface will adjust the state of the control register to 'full', at which point the suspended driver thread will resume execution. As shown in Fig.5, this would consequently trigger the *completion signaller*.

E. Memory Interface

All cores in a Microgrid are connected to the on-chip COMA hierarchy (Section II-C4). Data written to the Microgrid COMA memory system will migrate through the hierarchy to the location at which it is read. This property is exploited to achieve very high-performance I/O in the model described in this paper.

All I/O operations, whether lightweight operations through registers or bulk transfers, will eventually write/read their data to/from the associated buffer in memory. This means that I/O is not bound by the traditional limitations of memory bandwidth, as is the case with existing DMA architectures [14], and is fully decoupled from external memory bus contention by being distributed to potentially several different I/O buses instead.

The organisation of the COMA memory system into rings, unifying separate caches, means that not only can I/O be extremely fast, but also extremely parallel. I/O operations can take place in parallel, at different I/O cores, with different clients, using only the local memory subsystem and avoiding thrashing of the memory hierarchy between conflicting operations. This property introduces the concept of *I/O locality*, where, for the highest performance, a place allocator will delegate a client to a place on the same COMA level as the associated I/O device's I/O Core; i.e. they will share the same L2 cache. A 'smart' placement algorithm would ensure that jobs on Microgrid are created at a place appropriate for the I/O dependencies of that particular job.

1) Comparison to DMA: The current specification [13] of the HyperTransport bus has a unidirectional bandwidth of up to 25.6 GB/s. If this were to be streamed into external memory, as with DMA, the bandwidth would be limited by the bandwidth of the memory. For state-of-the-art DDR3-1600 memory, this bandwidth lies at 12.8 GB/s. However, considering that external memory is shared by all processors on the Microgrid, the effective bandwidth is, in practise, considerably lower.

The COMA memory, with a 1.6 GHz cache-line-wide ring network (in a typical experimental Microgrid) can achieve a bandwidth of up to 102.4 GB/s. This bandwidth is guaranteed between local caches, without interference from the rest of the system. Thus, if the consumer of the data is physically close to the I/O interface, the HyperTransport bandwidth can easily be matched by the COMA system and, furthermore, these transfer rates can take place simultaneously at many rings.

V. RELATED WORK

The use of dedicated programmable processors to handle I/O is not something new, having been first introduced in 1957 when it was implemented in the IBM 709 system [15]. Following this development, the IBM System/360, System/370 and the architectures that superseded them have featured channel processors for high performance I/O [16]. Another system from this period that had even more similarities to the approach described in this paper was the Control Data CDC6600 [17], which had a dedicated I/O processor that shared 10 distinct I/O processor contexts. As these I/O processors were very limited and only supported a simplified instruction set for handling I/O, they are not very different from the programmable DMA controllers found in modern computers. Both approaches serve the same purpose: to prevent the CPU

from being frequently interrupted during dense I/O operations. This is also a problem in real-time embedded systems [18], where it is common that state-of-the-art micro-controllers are equipped with a peripheral control processor which can be used to handle interrupts while the main processor can still meet its real-time obligations.

The Helios OS [7] distributes small satellite kernels to programmable I/O devices in order to offload the execution of programs and system services. It uses affinity meta-data to hint about efficient placement of such programs to put data processing close to its source. As it targets heterogeneous platforms, it uses independent byte-code to represent programs which are then compiled for the specific device. In the I/O system proposed in this paper, there is no such problem of heterogeneity as the I/O cores are similar to the other SVP cores, albeit with a restricted instruction set. The model described in this paper would also benefit from the intelligent placement of components to use the locality of data. Others have also observed this as a problem, and suggest [19] that in order to achieve high throughput I/O, tightly coupled communication between the components, with no global interactions, is highly desirable. This suggestion mirrors the distributed operating system design approach being taken on the AppleCore project.

VI. Evaluation and Ongoing Work

In this paper we have presented a novel combination of methods for performing I/O in a highly parallel environment, in particular SVP and Microgrids. The approach first explored the implementation at the level of the concurrency model; the way in which signalling and interrupts can be represented in the parallel environment of SVP, where conventional interrupt and I/O mechanisms are not applicable due to the decentralised nature of a many-core operating environment. The approach of using listener/writer threads with I/O places obviates the need for a central 'interrupt handler' model. This software level model has been fully implemented in μTC and functionally verified using the Pthreads implementation of SVP.

We also described the implementation of specialised 'I/O Cores' in the Microgrid hardware architecture. These cores are special processing units of reduced complexity which are connected to a high-speed bus for device communication, and they provide a device interface to the higher level I/O stack. The particular advantages of I/O cores are that they allow I/O operations to be fully decoupled from general purpose cores for truly parallel and scalable I/O. The ability of the I/O core to read and write data directly to the local on-chip COMA cache hierarchy allows for very high device-to-client transfer rates and bypasses the limitations of the standard DDR memory buses (which are already under intense pressure in a parallel architecture), when compared to conventional DMA.

Current work is focussed on fully modelling I/O cores in the existing cycle-accurate Microgrid simulator with a view to publish future results regarding the quantitative analysis of the I/O model's performance in specific application circumstances.

Acknowledgements

The development of SVP, the Microgrid architecture and the μTC compiler was initially supported by the NWO *Microgrids* project, then by the EU *Apple-CORE* project. SVP and its implementation is a group effort of the CSA group at the University of Amsterdam. In particular, the authors would like to extend their gratitude to Mike Lankamp and Raphael C. Poss for their contributions to this work.

References

[1] D. Geer, "Industry Trends: Chip Makers Turn to Multicore Processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[2] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, and et al, "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS," feb. 2010, pp. 108 –109.

[3] K. Bousias, L. Guang, C. Jesshope, and M. Lankamp, "Implementation and Evaluation of a Microthread Architecture," *Journal of Systems Architecture*, vol. 55, no. 3, pp. 149–161, 2009.

[4] C. Jesshope, "A model for the design and programming of multi-cores," *Advances in Parallel Computing*, vol. High Performance Computing and Grids in Action, no. 16, pp. 37–55, 2008.

[5] C. Jesshope, M. Lankamp, and L. Zhang, "The Implementation of an SVP Many-core Processor and the Evaluation of its Memory Architecture," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 38–45, 2009.

[6] D. Wentzlaff, C. Gruenwald, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, "A Unified Operating System for Clouds and Manycore: fos," Computer Science and Artificial Intelligence Lab, MIT, Tech. Rep. MIT-CSAIL-TR-2009-059, November 2009.

[7] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: heterogeneous multiprocessing with satellite kernels," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM, 2009, pp. 221–234.

[8] C. R. Jesshope, "muTC - An Intermediate Language for Programming Chip Multiprocessors," in *Asia-Pacific Computer Systems Architecture Conference*, 2006, pp. 147–160.

[9] M. van Tol, C. Jesshope, M. Lankamp, and S. Polstra, "An implementation of the SANE Virtual Processor using POSIX threads," *Journal of Systems Architecture*, vol. 55, no. 3, pp. 162–169, 2009, challenges in self-adaptive computing.

[10] T. Bernard, C. Grelck, and C. Jesshope, "On the Compilation of a Language for General Concurrent Target Architectures," *Parallel Processing Letters*, vol. 20, March 2010.

[11] C. Jesshope, M. Hicks, M. Lankamp, R. Poss, and L. Zhang, "Making multi-cores mainstream - from security to scalability," in *Parallel Computing: from Multi-cores and GPUs to Petascale*. IOS Press, May 2010, pp. 16 – 31.

[12] P. SIG, "PCI Local Bus Specification Revision 2.3 MSI-X ECN," http://www.pcisig.com/specifications/conventional/msi-x_ecn.pdf.

[13] H. T. Consortium, "HyperTransport I/O Link Specification," Technical Document HTC20051222-0046-0026, July 2008.

[14] A. F. Harvey, "DMA Fundamentals on Various PC Platforms," National Instruments, Application Note 011, April 1991.

[15] IBM, "709 data processing system," http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP709.html.

[16] ——, "Ibm system/370 principles of operation," 1974.

[17] J. E. Thornton, "The cdc 6600 project," *Annals of the History of Computing, IEEE*, vol. 2, no. 4, pp. 338 –348, oct.-dec. 1980.

[18] F. Scheler, W. Hofer, B. Oechslein, R. Pfister, W. Schröder-Preikschat, and D. Lohmann, "Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system," in *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM, 2009, pp. 167–174.

[19] M. Burnside and A. D. Keromytis, "High-speed I/O: the operating system as a signalling mechanism," in *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*. New York, NY, USA: ACM, 2003, pp. 220–227.