Making multi-cores mainstream – from security to scalability

Chris JESSHOPE, Michael HICKS, Mike LANKAMP, Raphael POSS and Li ZHANG
Institute for Informatics, University of Amsterdam, The Netherlands

Abstract. In this paper we will introduce work being supported by the EU in the Apple-CORE project (http://www.apple-core.info). This project is pushing the boundaries of programming and systems development in multi-core architectures in an attempt to make multi-core go mainstream, i.e. continuing the current trends in low-power, multi-core architecture to thousands of cores on chip and supporting this in the context of the next generations of PCs. This work supports dataflow principles but with a conventional programming style. The paper describes the underlying execution model, a core design based on this model and its emulation in software. We also consider system issues that impact security. The major benefits of this approach include asynchrony, i.e. the ability to tolerate long latency operations without impacting performance and binary compatibility. We present results that show very high efficiency and good scalability despite the high memory access latency in the proposed chip architecture.

Keywords. Concurrency model, multi-core, multi-threading, resource deadlock.

Introduction

One of the principle research goals in parallel computing is the ability to write code once, or to take legacy sequential code, and to execute it on any parallel computer with a high efficiency and with scalable speedup. This need only be constrained by the concurrency exposed in the algorithm and the characteristics of the target parallel computer. Problems arise because these characteristics vary significantly and that most developments in tools target a particular architecture rather than a generic parallel execution model. The characteristics that vary include synchronisation and scheduling overhead, which determine the granularity of the units of concurrency that can be executed efficiently and the ratio of computation to communication rates, which determines whether it is worthwhile distributing code at a given level of granularity.

Whether this goal of general-purpose parallel computing can be achieved across a wide range of targets is still an open question but one that we are working towards. Our first steps have focused on the narrower field of achieving genericity of target when programming many-core processors. Here we see the same issues affecting different generations of the same processor or dynamic partitions of a multi-core chip. We want to be able to compile code once and execute it on any number of cores, anywhere on chip and to deal with systems issues such as scalability and security in a multi-user environment. Our execution model, SVP [1], provides concurrent composition by default. This invited paper describes an implementation of that model in the ISA of a conventional, in-order issue RISC core. More details on various aspects of this EU-funded project can be found at (http://www.apple-core.info/).

1. Motivation

That there is a practical urgency in this matter is common knowledge. On the one hand, there is an inescapable requirement to manage power dissipation on chip, which requires many simple cores rather than fewer, more complex ones. On the other hand, a many-core approach requires tools supporting massive explicit concurrency, which are difficult to implement and error prone to use. In embedded and special purpose systems, e.g. picoChips [2], NVIDIA [3, 4], Intel [5, 6, 7] and ClearSpeed [8], this is common. However, here the focus is on a limited set of applications, where skilled effort can be applied to find and map the applications' concurrency. Moore's law still predicts that the number of cores on chip will double every 18 to 24 months (for at least for another decade [9]) and this raises compatibility issues even in a specific processor.

In a more general market, the labour-intensive approach of hand mapping an application is not feasible, as the effort required is large and compounded by the many different applications. A more automated approach from the tool chain is necessary. This investment in the tool chain, in turn, demands an abstract target to avoid these compatibility issues. That target or concurrency model then needs to be implemented on a variety of platforms to give portability, whatever the granularity of that platform.

Our experience suggests that an abstract target should adopt concurrent rather than sequential composition, but admit a well-defined sequential schedule. It must capture locality without specifying explicit communication. Ideally, it should support asynchrony using data-driven scheduling to allow for high latency operations. However, above all, it must provide safe program composition, i.e. guaranteed freedom from deadlock when two concurrent programs are combined.

Our SVP model is designed to meet all of these requirements. Whether it is implemented in the ISA of a conventional core, as described here or encapsulated as a software API will only effect the parameters described above, which in turn will determine at what level of granularity one moves from parallel to sequential execution of the same code. The work presented in this paper describes the execution model, its implementation as an extension to the Alpha ISA and its core compiler that compiles the language μTC, which captures SVP in an architecture neutral form, to a *Microgrid* of SVP-enabled cores. Compilers to this model, emitting μTC, are also being developed from the functional, data-parallel language SAC [10, 11], the high-level coordination language and component technology S-Net [12, 13] as well as an automatically parallelising compiler for legacy C code [14].

2. The Self-adaptive Virtual Processor - SVP

SVP is a hierarchical thread-based model developed in the EU AETHER project (http://www.aether-ist.org/) to support adaptive computing. It provides a complete separation of concerns between the two most important aspects of concurrent programming. The first is the description of an application's functionality expressed concurrently and the second is the mapping of that program onto a set of resources. This separation is achieved by binding processing resources to units of work dynamically using opaque, implementation-defined objects called *places*. In this paper, a place is a ring of SVP-enabled cores but it could just as easily be a conventional core or cluster of cores or even dynamically configured logic (e.g. an FPGA), as was implemented in the AETHER project.

In its resource-neutral form, SVP provides an abstract target for high-level language compilation, which need not be concerned with mapping and scheduling. The code generated is highly concurrent and guaranteed to be free from deadlock [15]. Mapping is performed by the core-compiler (i.e. the μTC compiler) and a run-time system that provides dynamic allocation of places in a manner similar to memory allocation. Scheduling is controlled using synchronising communication. SVP defines *shared* and *global* objects giving pair-wise and collective, one-way synchronisation channels respectively. These are implemented with i-structures [16]. They are written once in one thread and are read-only in one or more other threads. I-structures provide the data-driven scheduling in an SVP implementation. An i-structure suspends a thread attempting to read it in an empty or unwritten state and stores these continuations until data is written, at which point it must reschedule the suspended threads.

Currently SVP is described by the language μTC [17], for which we have a core compiler tool chain based on GCC [18]. This has multiple targets that currently include:

- a sequential implementation for program validation;
- a cycle-accurate multi-core chip emulation, where SVP is implemented directly in the core's ISA [19, 20] - a *Microgrid* of SVP cores; and
- a POSIX-thread-based SVP API [21] for general use, developed in the EU AETHER project.

SVP programs are composed concurrently, at all levels, from the smallest threads (maybe a few instructions) up to complete systems. This means that there is always an appropriate level of granularity that will map to a given target at some point in the concurrency tree. Hence, when a target is selected, the SVP program is transformed to that level of granularity using its sequential schedule. In the Microgrid of SVP cores, no code transformation is required. Places are selected at run time and the hardware provides support for the automatic sequencing of SVP binary code, if too few concurrent contexts are available. This is described in more detail in Section 3.

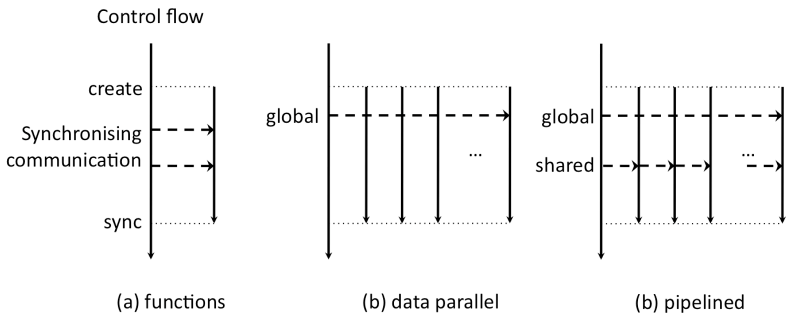

Figure 1. Three variants of an SVP create showing synchronisations: (a) concurrent function execution with synchronising parameters; (b) concurrent loop execution (n.b. each thread is created with a unique index in a specified range); and (c) concurrent loop execution with loop-carried dependencies (linear pipelines).

2.1. SVP concurrency controls

SVP provides concurrency controls to *create* and terminate (*kill* and *break*) a named unit of work. That unit is a family of identical indexed threads and any subordinate families that those threads may create. The index is specified on create by a triple of (*start*, *step*, *limit*) or (*start*, *step*) for unbounded families. Unbounded families must be terminated dynamically with a *break* instruction executed in one of the family's threads.

The parent thread may execute and communicate asynchronously with the family it creates. SVP provides a barrier (*sync*), which signals the completion of a family to the parent thread. Communication between the parent and its children may occur anywhere in the parent from create to sync, using shared and/or global objects.

The create/sync pair is used to compose both functions and loops as concurrently executing threads, including loops with dependencies. This is shown in Figure 1. As can be seen, we allow communication only between parent and first child and between adjacent children defined on the thread's index value. For shared objects, an association is made between a local in the creating thread and the shared object defined in the thread's parameters. A write in the parent thread is only seen by the first child thread and, with more than one thread created in the family, a write to a shared in one thread will be seen by its successor in index sequence. For global objects, a similar association is made between a local in the parent thread and the global object defined in the thread's parameters. Global objects may be read by all threads. The write to a shared object in the last thread in index sequence will update (on sync) the initialising variable in the creating thread. This restriction on communication has a threefold advantage:

- it provides a well defined sequential schedule for any SVP program;
- it guarantees freedom of deadlock in the abstract model, although failure to write to synchronising objects where visible and dealing with finite resource can still cause deadlock; and
- it provides an abstract notion of locality to the high-level compiler, which must transform dependencies to conform to this restriction.

So, what appears as a restriction in the model has advantages. The obvious question that follows is whether the model is still sufficiently general. Work on the C-to-SVP compiler [14] has shown that such transformations are possible in loops for all regular dependencies. Moreover, irregular dependencies can be made regular by a gather and/or scatter stage.

2.2. SVP memory model

We wish to support a very relaxed memory consistency model in SVP that would map naturally onto shared memory but, at the same time, ease any implementation on distributed memory. Consider a branch of the SVP concurrency tree (at any level); then for any memory location used anywhere in that branch and known not to be accessed concurrently by other (unrelated) branches while this branch is running, SVP provides Gao and Sakar's Location Consistency (LC) semantics, but without the synchronising *acquire* and *release* operations described in [22]. Instead, the synchronising operations that establish partial order on memory accesses are SVP's create and sync operations, which have different semantics to LC's acquire and release.

Thus, SVP's concurrency model provides support for non-synchronising, competing shared memory accesses (using the terminology proposed by Mosberger [23]) from different threads in a family, but exposes memory state from one thread to all its subordinate threads. Location consistency is then resolved for a thread on termination of a subordinate concurrency tree. In the hierarchy proposed by Mosberger [23], this model is a hybrid of LC (between parent threads and child families) and a weaker model without any synchronisation (between sibling threads).

Communication via memory is not defined in SVP between sibling threads. The only guaranteed synchronisation is through shared objects, which have different semantics, as described in Section 2.1, and which can be of arbitrary size. Whether these shared objects are supported by a specific SVP implementation using a shared-memory architecture with more constrained consistency semantics, via explicit communication channels or via some other mechanism, is not specified in the abstract SVP model.

In some circumstances it is necessary to provide consistency between global objects used in unrelated threads. We support this through the use of SVP's *exclusive place*. Exclusive place are shared between threads and sequence requests to create families of threads at that place. SVP's exclusive places in effect implement Dijkstra's "secretary" concept [24], where communication can occur between independent sequential processes by means of changing the secretary's local (private) state.

3. The SVP core

We have implemented SVP's concurrency controls and shared object semantics for basic types (integer and floats) as extensions to the ISA of an in-order Alpha core. Support is also provided in the form of memory barriers for arbitrary shared objects using pointers to objects stored in memory. This implementation is a full software emulation of the extended instruction set. It is supported by a set of tools to compile, assemble, link, load and execute μTC programs. This implementation takes account of all internal processor state in each of the six stages of the Alpha pipeline. It also restricts concurrent reads to an implementable number of ports on memory structures and hence provides a cycle-accurate simulation of the execution time of SVP programs.

As an example, consider the register file. This is the largest memory structure in the core and if silicon layout constraints were not taken into account, the core could not be implemented in a reasonable area and with a reasonable clock frequency; the area of a memory cell grows as the square of the number of its ports. Single instruction issue requires two reads and one write to the register file to support the pipeline's operation. However, the register file is also written with a thread's index value by the thread create process (potentially once every cycle). The register file must also be accessed for shared-register communication between threads that are mapped to adjacent processors and for operations that terminate asynchronously (described below). To support all of this, we provide 5 ports to the register file: 3 for pipeline operation and one read and one write port with arbitration for all other uses. Static analysis predicted this to be sufficient [24] and subsequent emulation has shown that while some processes may stall for a few cycles, overall progress is assured.

3.1. Synchronising registers

In SVP (unlike pure dataflow), constraints in a program are captured using two mechanisms, namely program sequence and by capturing dependencies. The latter uses SVP's synchronising objects, as described in Section 2.1. Ideally each should be implemented at the same level of granularity and hence we implement synchronising communication in the register file of the SVP core. By synchronising at this level, threads mapped to the same core can synchronise in a single cycle using the pipeline's bypass bus and between cores in a time not much longer than the pipeline's length.

Each register can be used either as an i-structure or as a conventional register. A state transition diagram for the i-structure is given in Figure 2. It will block any thread attempting to read in the *empty* state (i.e. before it the location has been written), continue to suspend thread continuations while it is *waiting* and reschedule those threads for execution upon being made *full* (i.e. when the location has been written). In the waiting state therefore, a register-file location contains a link to all threads that have attempted to read that location before its value was defined.

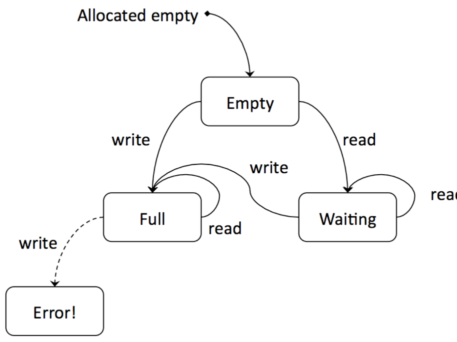

Figure 2. I-structure state-transitions

SVP instructions			
	Family/thread management		**Family parameter setting**
allocate	Takes a place at which a family will execute, allocates a family table entry and returns a family table index - FTid (asynchronous).	*setstart*	Sets a *start* index value for the given family – threads start from this index
create	Takes an FTid and creates threads described by the parameters stored there and returns a termination code, the *sync* (asynchronous).	*setlimit*	Sets a *limit* index value for the given family.
break	Terminates a thread's family and all subordinate families and returns a break value. Only one thread in a family may succeed in breaking its family (asynchronous).	*setstep* *setblock*	Sets a *step* index value for the given family Sets the maximum number of threads created on a given core.
kill	Terminates a family identified by a family table index and all subordinate families.	*setbreak*	Nominates the register that will be used to return the break value

Table 1. SVP instructions.

3.2. Family and thread management

In the SVP core, only a finite number of families and threads may be defined and these are stored in dedicated tables. This information is managed by instructions added to the Alpha ISA, which are listed in Table 1. Family state is stored in the *family table* and thread state is stored in the *thread table*. Both families and threads are identified by their index into these respective tables. Instructions in Table 1: allocate a family table entry, which comes with a default set of parameters; overwrite the default parameters where required; and initiate thread creation. The latter takes a single pipeline cycle to create an unlimited number of threads at a rate of one per cycle until resources are exhausted or the block size has been reached. Kill terminates a family based on its index in the family table and is fully recursive, i.e. all subordinate families are also killed. From a program's perspective only the family table index is visible, however, all instructions executed in an SVP core are tagged with their family and thread indices. This allows us to suspend and resume threads using the i-structures, which maintain linked lists of suspended thread indices.

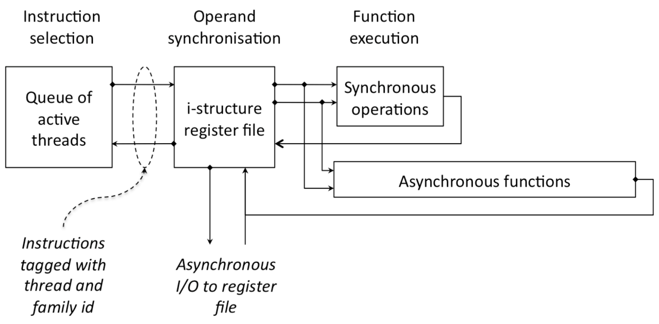

Figure 3. SVP pipeline phases

3.3. Instruction execution

The SVP pipeline is illustrated in Figure 3. It comprises three phases, each of which may comprise multiple pipeline stages. Instructions are issued from the head of the queue of active threads, where threads that can make progress are stored. These threads are not suspended and have their next instruction in the I-cache. Context switching (selecting the next thread from the active list) occurs on branches, when the current program counter increments over a cache-line boundary and, for efficiency, on instructions tagged by the compiler that are dependent on asynchronous instructions. In the latter case, this avoids flushing the pipeline if that instruction finds one of its operands empty at the register-file read. Thus, the core only executes sequences of statically schedulable instructions without context switching and then only when it can be guaranteed that instruction fetch will hit the I-cache. This makes for a very efficient instruction execution. In the limit, threads can context switch on each cycle and thread creation or wakeup can meet this rate.

In the next phase, instructions read their operands from the synchronising register file. Only when both operands are available can the instruction be dispatched for execution. The thread is suspended if either of the instruction's source registers is empty. A suspended thread will be rescheduled and re-execute the same instruction when the register it is suspended on is written to. This differs from dataflow execution where an instruction is only issued when all of its operations are available. The benefit is that statically scheduled instructions from multiple threads can be executed with RISC-like efficiency.

At execution, all instructions write back to the register file in their allocated pipeline slot, however, at this stage, asynchronous instructions simply set the target register's i-structure state to *empty.* Data is written when the operation completes. This may be the completion of a family, i.e. create writing a return code, or other long-latency operation (including memory fetches, floating point operations and any instructions labeled asynchronous in Table 1). In this way, no dependent instruction can execute until the asynchronous operation completes.

3.4. Thread-to-thread communication

Most of the bandwidth for thread-to-thread communication in a Microgrid of SVP cores is provided by the implementation of shared memory on-chip. We adopt an on-chip COMA memory that has already been reported elsewhere [26]. This uses a hierarchy of cache-line-wide ring networks to implement an attraction memory with a large aggregate bandwidth. In this memory, cache lines have no home. They can be copied and invalidated anywhere on chip so that data always migrates to the point of last update. A token-based cache-coherence protocol implements the memory consistency model described in Section 2.2.

An *inter-place* network provides low-latency communication between clusters of cores on a chip (the implementation of SVP's place). The place at which a family is created is defined on allocating its family table entry and if this is neither the core nor the cluster on which the parent thread is executing, then the inter-place network is used to implement the instructions listed in Table 1. The remote execution of a subordinate family on another place is called a *delegation* and requires a proxy family table entry on the creating core, which identifies the remote place. It also requires a family table entry at the remote place that controls thread creation in the normal manner. Parameters that define the family of threads are communicated across this network using these instructions. The Proxy must also manage communication of global and shared parameters between parent and child, which need not have been defined prior to create.

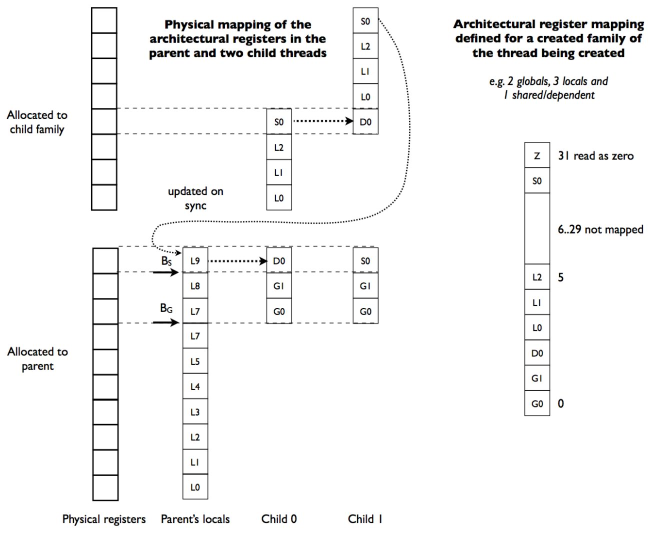
Physical mapping of the architectural registers in the parent and two child threads
Architectural register mapping defined for a created family of the thread being created
e.g. 2 globals, 3 locals and 1 shared/dependent
Allocated to child family
updated on sync
Allocated to parent
B_S
B_G
S0
L2
L1
L0
D0
L9
L8
L7
L7
L5
L4
L3
L2
L1
L0
G1
G0
Z
31 read as zero
6..29 not mapped
5
0
Physical registers
Parent's locals
Child 0
Child 1

Figure 4. Mapping of the overlapping register windows on creating a family of two threads with three local, two global and one shared/dependent defined in its register context. Shared register communication is illustrated with dashed arrows. The base addresses for the mapping of globals (B_G) and shareds (B_S) to the parent's locals is shown. N.b. this picture is repeated for ints and floats in the Alpha architecture.

An *intra-place* network manages communication between cores in a cluster. This includes the distributed implementation of create and sync actions that result in the distribution of the threads in a family to a multi-core place. It also implements a distributed-shared register file between the cores. This network is a word-wide ring network between adjacent cores in a cluster. To understand how this communication is specified it is necessary to understand the mapping of SVP's four classes of variables onto the distributed-shared register file. Register variables are divided into a number of overlapping windows, these are:

- *local* - visible only to one thread;
- *global* - written in a parent thread and read only in all threads in a family;
- *shared* - written once and visible to the next thread in index sequence;
- *dependent* - read-only access to the previous thread's shareds.

When a thread function is compiled, a partition is made of the architectural register context between these classes and this is defined in a header to its binary code, e.g. N_L, N_G, N_S, where: $N_G + N_L + 2 * N_S \leq 31$, n.b. not all registers need be mapped. To create n threads on a single core, n*(N_L+ N_S) registers are dynamically allocated from its register file, where n is determined by the number of threads in a family, a limit on the number of contexts available for a given core or by the block size defined in *setblock* (see Table 1). In Figure 4, eight registers are allocated on the creation of a family of two threads, with N_L=3 and N_S=1.

In order to pass parameters between parent and child threads the creating thread identifies offsets into its local variables to map to the globals (B_G) and shareds (B_S) of the created threads. These registers are written in the parent thread and are visible to all threads for globals and to the first thread only for shareds. Between siblings, a write to a shared in one thread can be read as a dependent in the subsequent thread. In the last thread, the shared write is visible to the parent thread in its locals via B_S, i.e. on sync, the location used as the parent's shared is updated.

The intra-place network implements a distributed-shared register file over the windowing scheme described above, so that the register files of all cores in a place provide a uniform mechanism for reading and writing registers regardless of their location. For efficient communication between cores, the global registers are replicated with a copy in each core's register file. These are allocated in the distributed create operation over the intra-place network. When threads on a core read an empty global, they will be suspended on that core and at most one read request is sent to the parent thread, which eventually responds with a broadcast around the ring, rescheduling any waiting threads. Similarly, when a shared communication is mapped between two cores the shared/dependent registers are also replicated. In this case, a read request is made to the adjacent core, which is eventually satisfied. The latter requires an additional N_S registers to be allocated per family, per core when a family is distributed. Again the dependent thread can be suspended and rescheduled at the core it executes on.

3.5. SVP security

To make multi-core mainstream, we described in the introduction a requirement to execute binary programs on an arbitrary number of cores (i.e. on one or more places of various sizes) by automating whether families of threads execute concurrently or sequentially. However, we also need to guarantee freedom from deadlock when finite resources are applied to an abstract SVP program and to guarantee this in the presence of potentially many different jobs competing for those resources in a multi-user environment. Note that we have to consider the situation where some of those programs may be hostile. It is not only deadlock that is an issue; programs can execute very powerful instructions in an open environment (for example to kill a family of threads and its descendants, see Table 1). We do have solutions to most of these problems although some are not yet implemented in our emulation environment. We deal with each of these issues in turn starting with the latter.

To protect a family from being the subject of an accidental or even malicious kill instruction, we protect families with capabilities. When a family is created, a key of arbitrary entropy is generated, which is stored in the family table and combined with the family table index to comprise a *family identifier*. This can be made arbitrarily secure. In order to issue an asynchronous kill on a family, the thread issuing the kill instruction must provide a family identifier that matches the security key stored in the family table, otherwise the instruction is ignored. In practical terms, this means that it must have been passed the capability by the creator of that family.

To protect a program from resource deadlock we have two strategies. The first is to analyse the resource requirements of a μTC program and to ensure that those resources are exclusively allocated to that program. The issue at hand is not the breadth of the concurrency tree, since a single context on one core is sufficient to execute any family regardless of its breadth. The problem is recursion of creates in the presence of finite concurrency resources. If that can be bounded, then deadlock freedom can be guaranteed by restricting the number of contexts allocated to families using *setblock* and to allocate places at appropriate points in the concurrency tree. At what point in the execution of a family those resources are guaranteed is an issue requiring further research. However, at present we can assume that they are allocated prior to the execution of the program, in which case we have a static solution, although not necessarily the most efficient one. To provide a more dynamic mapping, some guarantee of obtaining minimal resources in a finite time is required.

We must also consider how to ensure that if a program is allocated a place, then no other thread is allowed to create a family at the same place. This could consume those resources required to guarantee freedom from deadlock. This is achieved by including a capability in the *place identifier,* in the same way as described above for securing against kill. If the place identifier used in a create does not match the one-time key stored at that place when it was allocated, then the create will be ignored. Note that the only guarantee we can give on sharing the concurrency resources on a processor is when legacy code is executed. Here a single processor place can be shared between a number of legacy programs, where each is guaranteed to run in a single SVP thread.

Where it is not possible to statically analyse resource usage, we provide a software solution with an instruction that allows the code to determine whether any contexts remain. The procedure is to request a family table entry and then to check whether a context is still available. If so it continues its recursion concurrently. If it has the last context, it is obliged to use it to execute its recursion sequentially using the thread's memory stack. In this way we can guarantee progress, even if every other thread may have suspended in attempting to obtain a new context, as eventually that context will be released and the same procedure will be followed by the other suspended threads. Of course there must be a guarantee that the recursion terminates.

4. Results and analysis

We have configured our Microgrid emulator to implement the following chip design, which will be used in obtaining the results on scalability presented in this paper.

- A 64-bit Alpha core with 1Kbyte, 4-way set associative L1 I- and D-caches, 1024 integer registers, 512 floating-point registers, supporting a maximum of 256 threads. The clock rate is assumed to be 1.6 GHz.
- A pipelined floating point unit shared between two cores with 3, 8 and 10 cycles latency for *add/mult, division* and *sqrt* respectively.
- An on-chip COMA memory with two-levels of ring network and two DDR3 2400 channels off chip. At the top level are four COMA directories each supporting rings of eight 32 Kbyte, 4-way, set-associative L2 caches (i.e. 128 sets of 64-Byte cache lines). This gives a modest 1 MByte of L2 cache on chip.

- 128 cores configured with an inter-place cross-bar network as nine places comprising the following number of cores: {64, 32, 16, 8, 4, 2, 1, 1}.

Figure 5 is a schematic illustration of this chip. Prior work indicates that such a chip is feasible in current technology [20].

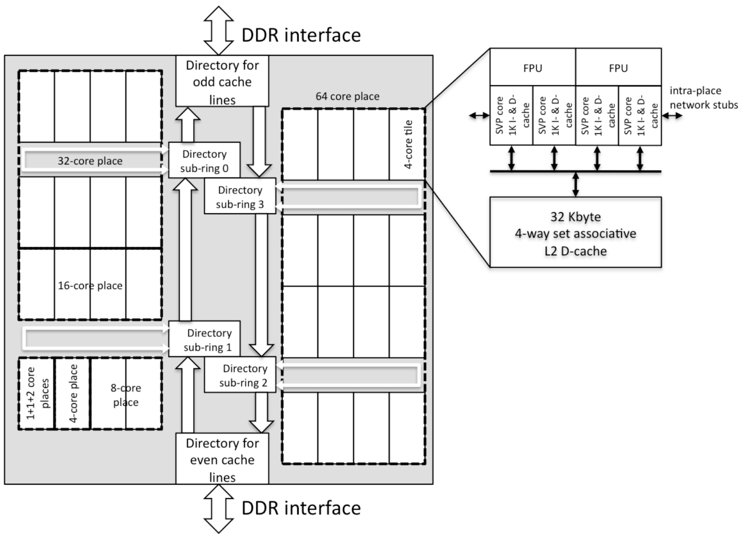

Figure 5. The Microgrid of 128 SVP cores and 32 by 32 KByte L2 caches.

The results presented here use code compiled from μTC versions of the Livermore loop kernels. We have verified this tool chain by comparing the execution of the same μTC code on both the emulator platform and on conventional processors, by applying SVP's sequential schedule. The specific kernels are not chosen to highlight the best results but rather to stress various aspects of the architecture and to illustrate the three different programming patterns found in loop-based code.

Each benchmark creates one or more families of threads on places of size 1 to 64 cores and measures the time to create, execute and synchronise the threads. For each kernel, we execute and time the code twice, the first execution with cold-caches, i.e. all code and data loaded from off-chip memory. The second execution (labeled warm) is run with whatever data remains in the caches and hence we would expect temporal locality when the problem fits into on-chip cache. As the COMA memory injects evicted cache lines into other caches on the same ring, when possible, the maximum cache is 256KBytes for places up to 32 cores and 512KBytes for 64 cores. We evaluate three different problem sizes: n=1K stressing concurrency overheads and limiting virtual concurrency in large places (1K threads is just 16 threads per core at 64-cores); n=8K where at least four arrays of this size map to the on-chip cache; and n=64K where the cache would accommodate at most one array of this size (only on 64 cores).

4.1. Data-parallel loops

The results for the data parallel benchmarks are shown in Figure 6. The *hydro fragment* executes the following simple expression n times, once per thread created.

```
    x[k] = q + y[k]*(r*z[k+10] + t*z[k+11]);
```

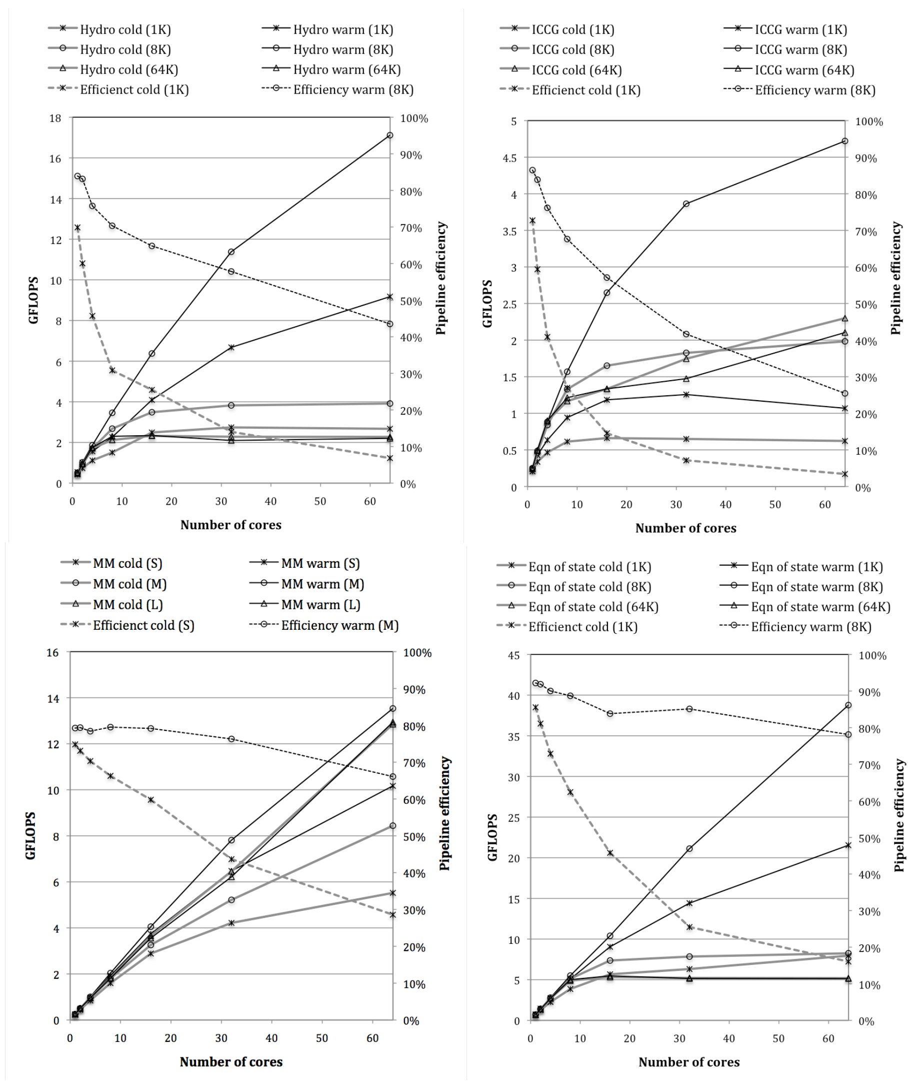

Figure 6. Data-parallel kernels: *hydro fragment* (top left), *ICCG* (top right), *Matrix Multiplication* (bottom left) and *equation of state* (bottom right), showing performance in GFLOPS and upper and lower bounds on pipeline efficiencies averaged over all cores. Execution is on places of size 1 to 64 cores.

The best execution times for the different problem sizes are 0.55µs, 2.4µs and 150µsec on 64 cores. For n=8K warm we get the best speedup, with a factor of 33 over the single core result and an average pipeline efficiency of 42-85%. For n=1K warm, the speedup drops to 17 on 64 cores. Here the total execution time is 893 processor cycles of which 208 are required to execute the 16 threads on one core. The remainder arise from distributing and synchronising the family of threads over a given number of cores and from pipeline stalls due to fewer threads to hide memory access latency. Even so, 1024 threads are created, executed and synchronised across 64 cores in less than one cycle per thread. This demonstrates the efficiency of our heavily overlapped process of thread creation and distribution.

The results for cold caches and for the 64K problem, where the caches are also effectively cold (they will hold the high index array values), we see saturation due to memory-bandwidth limitation between 8 and 32 cores. The peak memory bandwidth is 38.4 GBytes/sec and the peak bandwidth required by the code is 4 GBytes/sec per core at full pipeline efficiency, so these results are not unexpected.

ICCG shows a similar overall pattern but the performance is lower and for n=8K warm the maximum speedup is only 19 on 64 cores. However, ICCG has more steps and less concurrency. A total of $\log_2 n$ families are created, where at each step the number of threads varies from 2 to n/2. Thus, like the smaller problem size above, we have fewer threads and more concurrency-management overhead. Best execution times for ICCG are 2, 3.5 and 57 µsecs for 1K, 8K and 64K respectively.

Matrix multiplication is shown for sizes of n=20 (S), 32 (M), 90 (L). This gives array sizes of 400, 1K and 8K, i.e. n^2 elements, however the algorithm performs $O(n^3)$ operations for n^2 results. The simplest algorithm was implemented, where n^2 threads each compute one element of the result by performing an inner product. It can be seen that the results scale well for both warm and cold caches, due to the amount of computation required in obtaining the result for a single element. This problem stresses the on-chip cache organisation, as although it has temporal on-chip locality, accesses to columns have no spatial locality. This can be seen in the results for the large problem, where both cold and warm performance is reduced due to capacity misses. Maximum speedup is a factor is 53 (warm) and 34 (cold) for n=32 on 64 cores. The best execution times were *1.6, 5 and 90* µsecs, respectively for three problem sizes.

Equation of state is also a single family of n threads, although the thread in this instance are more complex than hydro and give a better overall performance. We have near perfect speedup for n=8K warm, 54 fold speedup on 64 cores. Even allowing for concurrency overheads, the pipelines are still operating at over 78% on 64 cores, i.e. less than 1 bubble in 4 cycles. The best execution times are 1.6, 4.8 and 113 µsecs for the three problem sizes.

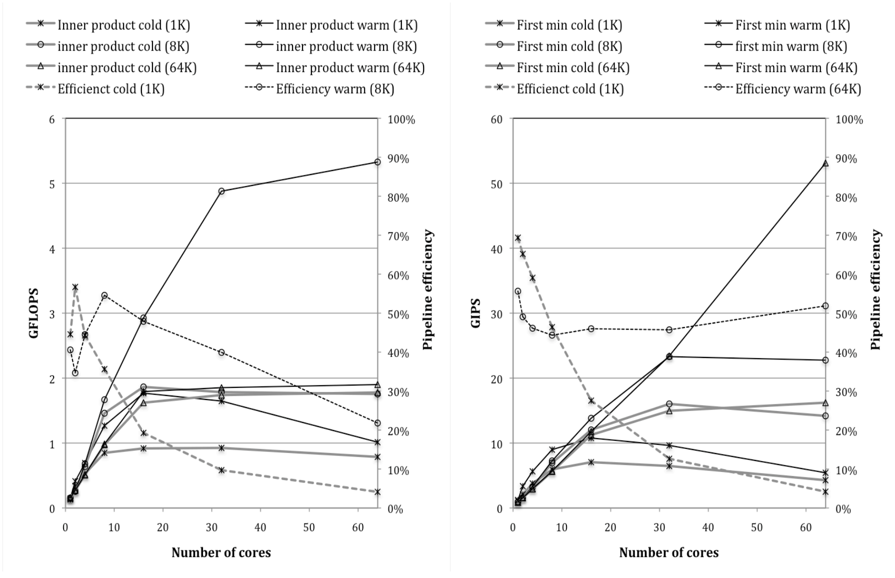

Figure 7. Inner product and first min reductions for 1k, 8K and 64K points.

4.2. Reductions

We implemented two reductions from the Livermore loops, *inner product* and *first min.* The code for both is quite general although they require the system to provide the number of cores, in order to implement four partial reductions on each core before completing the reductions across the cores. As can be seen in Figure 7, we get a similar pattern of performance to the data-parallel loops. Efficiencies overall are lower due to the higher concurrency overheads and the sequential reduction between the cores. For the warm caches we get speedups of 7, 12 and 34 for the different problem sizes.

4.3. Parallel prefix sum

The prefix sum operation is one of the simplest and most useful building blocks for designing parallel algorithms. It appears to be inherently sequential but has an efficient parallel implementation that requires $\log_2 n$ steps. For example, linear recurrences, including many of the sequential Livermore loops can be expressed in parallel using it. Blelloch in [28] lists a range of applications, including parsing, sorting, variable precision arithmetic, regular expression searching, etc. The same algorithm is also used in hardware in most ALUs to perform binary addition (carry look-ahead adders). Parallel prefix sum can be generalised to any binary associative operation and is also known as the scan operation.

Scan takes a binary associative operator ⊕, and an ordered set of n elements:

$$[a_0, a_1, ..., a_{n-1}],$$

and returns the ordered set:

$$[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})].$$

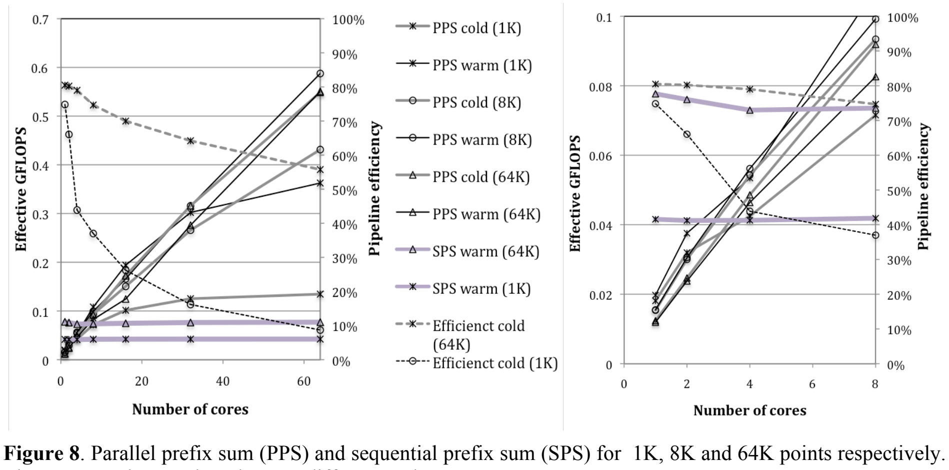

Figure 8. Parallel prefix sum (PPS) and sequential prefix sum (SPS) for 1K, 8K and 64K points respectively. The same results are plotted at two different scales.

Because of its importance, we have investigated the implementation of this algorithm using floating-point addition. We have implemented both parallel (PPS) and sequential (SPS) versions in µTC and compared the results. The sequential version also generates threads but implements the algorithm with a thread-to-thread dependency. The parallel algorithm requires $\log_2 n/2$ more operations than the sequential one, i.e. a factor of 5, 6.5 and 8 for the 1K, 8K and 64K problem sizes. Figure 8 compares the relative performance of both algorithms, i.e. effective GFLOPS are computed using the sequential operation count in both sets of performance curves. The sequential algorithm shows a speedup of 1.1 at 64 cores. The parallel version has a speedup of 18, 38 and 46 on 64 cores (warm caches) compared to its single core performance and 7.2, 8.2. and 8.5 when compared to the performance of the single-core sequential code. Note also, that the cold- and warm-cache performance is very similar due to the locality over the algorithm's $\log_2 n$ stages. The exception is the small problem size, where only 8 threads are created per core, on the 64-core place, which is insufficient to tolerate the latency of off-chip memory accesses.

5. Conclusions

This paper presents a significant amount of work, which spans more than a decade of research and engineering effort. It is exciting to see the fruition of this work, made possible with the support of the EU funded Apple-CORE project, which we gratefully acknowledge. We have demonstrated here results obtained from our core compiler and a realistic emulation of a processor chip that could be implemented in today's technology. We have only just begun to investigate the characteristics of this disruptive approach to many-core computing but these initial results are very encouraging. We have evaluated the performance of a range of common, loop-based programming paradigms and have shown speedup on every one. Although performance saturates due to memory bandwidth constraints in these simple benchmarks, distributing them as concurrent components of a larger application, executing at different places on chip, will minimise this problem. What we have shown is arguably the worst case scenario, where very simple kernels are executed from start to finish where all data and code is sourced off chip.

We still have a significant amount of work to complete in order to demonstrate that this approach is viable in the context of commodity computing. We have started evaluating more complex algorithms and have begun the process of automating the management of resources on chip. Moreover we have shown that solutions exist to the issues of security when using such a chip in an open, many-user environment. Thus the results presented here, we believe, demonstrate a significant first step towards this goal.

Furthermore, the Apple-CORE project has enabled this work to be extended into other research groups working in complimentary areas. Partners in this project are developing high-level compilers from both standard and novel languages targeting this core tool chain. In addition work is almost complete in developing an FPGA prototype based of an SVP core based on the LEON 3 soft core.

References

[1] C.R. Jesshope, A model for the design and programming of multi-cores, in (L. Grandinetti, ed.), *High Performance Computing and Grids in Action, Advances in Parallel Computing*, **16**, IOS Press, 2008, 37–55.

[2] A. Duller, G. Panesar, and D. Towner. Parallel processing—the picoChip way, in (J.F. Broenink and G.H. Hilderink, eds.) *Communicating Process Architectures 2003, Concurrent Systems Engineering Series*, **61**, IOS Press, 2003, 1-14.

[3] D. Kirk, NVIDIA CUDA software and GPU parallel computing architecture, in Proc. *6th international symposium on Memory management, ISMM '07*, ACM, 2007, 103–104.

[4] NVIDIA Corporation, *CUDA Zone - the resource for CUDA developers*, http://www.nvidia.com/cuda, 2009.
[5] J. Held, J. Bautista and S. Koehl, *From a few cores to many: a Tera-scale computing research overview*, Intel Corporation technical report, http://download.intel.com/research/platform/ terascale/terascale_overview_paper.pdf, 2006.
[6] Intel Corporation, *Tera-scale computing research programme*, http:// techresearch.intel.com/articles/Tera-Scale/1421.htm.
[7] Intel Corporation, *Teraflops research chip*. http://techresearch. intel.com/articles/Tera-Scale/1449.htm.
[8] Clearspeed, *CSX Processor Architecture*. Whitepaper, Clearspeed Technology plc, Bristol, UK, 2007.
[9] ITRC, International Technology Roadmap for Semiconductors. http://public.itrs.net, 2007.
[10] C. Grelck and S-B. Scholz. SAC: A functional array language for efficient multithreaded execution, *International Journal of Parallel Programming*, **34**(4), 2006, 383–427.
[11] C. Grelck and S-B. Scholz. SAC: off-the-shelf support for data-parallelism on multicores, in Proc. 2007 workshop on *Declarative aspects of multicore programming, DAMP '07*, ACM, 2007, 25-33.
[12] C. Grelck, S-B Scholz, and A. Shafarenko. Streaming networks for coordinating data-parallel programs, in (I. Virbitskaite and A Voronkov, eds), *Perspectives of System Informatics*, *6th International Andrei Ershov Memorial Conference (PSI'06)*, Novosibirsk, **4378** LNCS, Springer-Verlag, 2007, 441–445.
[13] C. Grelck, S-B Scholz, and A. Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components, *Parallel Processing Letters*, **18**(1), 2008, 221- 237.
[14] D. Saougkos, D. Evgenidou, and G. Manis. Specifying loop transformations for C2μTC source-to-source compiler, in *14th Workshop on Compilers for Parallel Computing (CPC'09),* 2009.
[15] T.D Vu and C. R. Jesshope, Formalizing SANE virtual processor in thread algebra, in (M. Butler, M. G. Hinchley and M. M. Larrondo-Petrie, eds.) *Proc. ICFEM 2007*, **4789** LNCS, Springer-Verlag, 2007, 345-365.
[16] Arvind, R.S. Nikhil, and K.K. Pingali, I-structures: data structures for parallel computing, *ACM Trans. Program. Lang. Syst.,* **11**(4), 1989, 598-632.
[17] C.R. Jesshope, μTC - an intermediate language for programming chip multiprocessors, in *Asia-Pacific Computer Systems Architecture Conference*, **4186** LNCS, 2006, 147-160.
[18] GCC, *the GNU compiler collection*. http://gcc.gnu.org.
[19] 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 (W. Najjar and H. Blume Eds.) Proc. *Intl.Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation, SAMOS-2008*, 2008, 1-9.
[20] K. Bousias, L. Guang, C.R. Jesshope, M. Lankamp, Implementation and Evaluation of a Microthread Architecture*, Journal of Systems Architecture*, **55**(3) 2009, 149-161.
[21] M.W. van Tol, C.R. Jesshope, M. Lankamp and S. Polstra, An implementation of the SANE Virtual Processor using POSIX threads, *Journal of Systems Architecture*, 55(3), 2009,162-169.
[22] G. R. Gao and V. Sarkar, Location consistency – a new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 1998.
[23] D. Mosberger, Memory consistency models. *SIGOPS Oper. Syst. Rev*., **27**(1), 1993, 18–26.
[24] E. W. Dijkstra, Hierarchical ordering of sequential processes. *Acta Informatica*, **1**(2), 1971, 115-138.
[25] K. Bousias, N. M. Hasasneh and Jesshope C R (2006) Instruction-level parallelism through microthreading - a scalable Approach to chip multiprocessors, *Computer Journal*, **49** (2), 211-233.
[26] C. Jesshope, M. Lankamp and L Zhang, Evaluating CMPs and their memory architecture, in (Eds. M Berekovic, C. Muller-Schoer, C. Hochberger and S. Wong) Proc. *Architecture of Computing Systems, ARCS 2009*, **5455** LNCS, 2009, pp246-257.
[27] J. Masters, M. Lankamp, C.R. Jesshope, R. Poss, E. Hielscher, Report on memory protection in microthreaded processors, Apple-CORE deliverable D5.2, http://www.apple-core.info/wp-content/apple-core/2008/12/d52.pdf.
[28] G. E. Blelloch, Prefix Sums and Their Applications, in *Synthesis of Parallel Algorithms*, (J. H. Reif, ed.) Morgan Kaufmann, 1991.