

Evaluating CMPs and their Memory Architecture

Chris Jesshope, Mike Lankamp, Li Zhang¹

¹ Institute for Informatics, University of Amsterdam
{c.r.jesshope,m.lankamp,l.zhang}@uva.nl}

Abstract. Many-core processor architectures require scalable solutions that reflect the locality and power constraints of future generations of technology. This paper presents a CMP architecture that supports automatic mapping and dynamic scheduling of threads leaving the binary code devoid of any explicit communication. The thrust of this approach is to produce binary code that is divorced from implementation parameters, yet, which still gives good performance over future generations of CMPs. A key component of this abstract processor architecture is the memory system. This paper evaluates the memory architectures, which must maintain performance across a range of targets.

1. Introduction

Although on-chip frequencies have hit a power wall, functional density is still expected to grow exponentially for at least the next decade, maybe more [1], which means explicit concurrency in processor architectures is no longer optional. Hence, attention is being placed on Chip Multiprocessors (CMPs), which appear to be the only way to utilise this surplus of on-chip resources in the future. The issue of programming model now becomes paramount if these are to become truly general purpose. Users of commodity processors have come to expect binary compatibility across generations of the same processor and the ability to program them safely, without dealing with concurrency issues. To translate these advantages to CMPs is a big challenge. In this paper, we use a model that provides these advantages when implemented in the ISA of a regular core. This implementation shows very promising results but the model is disruptive and requires a whole new generation of tools to fully exploit it. However, at the ISA level, we can then exploit concurrency down to the instruction level and we believe that fine-granularity is important in the long term when scaling to thousands or even tens of thousands of cores.

The model defines a Self-adaptive Virtual Processor (SVP) and combines concurrency model with operating system kernel services, e.g. pre-emption and task termination. It was developed in the EU AETHER project (<http://www.aether-ist.org/>) and tools are being developed for it in the EU Apple-CORE project (<http://www.apple-core.info>). We adopt this new model, as those proposed to date do not meet the requirements of future technologies, i.e. concurrent composition of programs without deadlock, locality of communication, low power and fault tolerance.

Transactional Memory (TM) [2] has attracted a lot of interest recently. It provides a lock-free programming model, which divides programs into parallel transactions but

ignores dependencies between them. An implementation then detects dependencies dynamically when transactions commit. Any conflict will cause one or more transactions to roll back and be re-executed. Thus, although TM provides a simple model for writing correct programs, obtaining efficiency is another matter. TM is speculative and this has significant power implications. In contrast, SVP instructions are issued conservatively, i.e. only when their data is available. When an SVP program starves of data, this can be recognised and power-saving measures taken such as stopping the clocks. Normal resumption of activity is then asynchronous, e.g. a decoupled memory requests completing.

Implementing a thread-based model in hardware is not new. For example, Niagara II [7] supports threads in hardware in multi-core chips and supports fine-grained scheduling of threads based on access to shared memory. It does not provide virtualisation or structuring of threads, as found in SVP. Multithreading has also been used to provide virtual concurrency, Niagara II provides fine-grained temporal multithreading with its multi-core implementation. Each core supports eight logical threads with four threads to each pipeline in a single core. The threads in each core are issued in an interleaved manner, where different threads can be selected for execution on each cycle. When one thread stalls, e.g. for a branch or load, further instructions are not issued until the thread is reactivated.

Other related architectures are Cell and Tiler. The Cell [9] adopts a non-threaded approach but provides high performance by incorporating a conventional processor with eight high-performance vector units, which handle the computational workload. This architecture puts a lot of responsibility on the programmer and/or compiler in managing the explicit distributed memory and computation. Tiler is a homogeneous architecture with 64 cores (<http://www.tiler.com>) on one die, where each core has a 3-way VLIW pipeline and where all cores are connected by a packet-routed network. Like the work described here, Tiler is programmed to share register variables. However, whereas SVP restricts communication, there are no such restrictions in Tiler meaning more responsibility on the part of the programmer or compiler again.

2. The SVP model

SVP is based on more than 10 years of research, starting with the dynamic RISC processor [5]. This supported a large number of fine-grain threads (microthreads), to tolerate high memory latency. Based on this initial work, the model was developed and refined and is now quite general. It solves the problem of the interaction between concurrency model and the operating system as SVP model can be considered an operating system kernel, implemented in a core's ISA. The instructions implemented abstract the notion of a processing resource and implement remote job execution to resources. They also provide control and reflection on the concurrency created by fully identifying the units of work created.

SVP is captured by the language μ TC [3], which can be compiled [4] into instructions that capture the model's parameterised concurrency. The instructions and their support structures implement a dynamically scheduled RISC core (DRISC) [5]. Parallelising compilers for higher-level languages such as C and SaC [6] are also being

developed. We do not have space in this paper to deal with the OS aspects of SVP and the reader is referred to [10] for further information.

The most important addition to an SVP core's ISA is an instruction that creates a family of indexed threads at a place (a cluster of cores). There is no mapping or scheduling implied by this instruction and the minimum resource needed is that required for a single thread,

in which case the threads will execute sequentially. However, if there are multiple cores available the threads will be automatically mapped to them. In addition, if there are multiple threads available per core, then many threads will be automatically interleaved on each core to provide latency tolerance. This gives SVP binary code that is independent of where it executes and independent of the number of cores used for its execution. Moreover, existing binary code for the base ISA can be executed on a single core using this instruction.

Threads in SVP are blocking and suspend when waiting on data, such as loads from memory, resources, family termination etc. or when waiting for data from another thread. This dataflow-like communication is constrained in order to provide model-based locality in the binary code. The first thread created may read data from the creating thread and any other thread may read data from its predecessor. This linear dependency chain imposes a constraint on the compiler, which can be satisfied statically when transforming loops to execute concurrently. This then allows threads to be mapped to rings of processors with only local communication. This constraint also guarantees freedom of deadlock under composition, due to its acyclic nature and also provides the bound on resources required in order to avoid resource deadlock.

In addition to this communication between threads, families of threads may communicate using shared memory but there are no guarantees on the timing of this memory. Memory consistency is weak and the data written by a family of threads may only be safely read on the termination of that family. Because SVP offers hierarchical composition, i.e. any thread may create a subordinate family, complex data-flow patterns can be established using a combination of inter-thread synchronisation and bulk synchronisation on family termination.

The aspect of the model that supports self-adaptation is the model's abstraction of a resource, the *place*. The create instruction takes a place parameter, which defines where on chip (or in a larger environment) the family will be created. The threads are statically created on the specified place, (consisting of one or more cores), but scheduled and managed dynamically at run time within that place.

```
thread void si(int * a, shared int sum) {
    index i;
    sum = sum + a[i];
}

thread void main() {
    int * a;
    family fid; place pl; int s_in
    create(fid;pl;0;9;1;;) si(a, s_in);
    ...
    s_in = ...
    ...
    sync(fid);
    ...
}
```

Figure 1. μ TC code generating a family of dependent threads using a shared variable; the si thread sums an

2.1 Creating families of threads

The default mechanism for program composition in SVP is to do so concurrently using the *create* instruction, which replaces both function calls and loop constructs in this model. It defines an indexed family of blocking threads that are created asynchronously subject to available resources (e.g. thread table entries and registers, which are heap allocated). The create instruction takes one pipeline cycle and completes out of order by writing a return code to a register on termination of all its threads.

When creating functions as threads, dataflow concurrency is exploited, as parameters to functions are passed via synchronising registers called *shared variables* in μ TC. This means the thread function can be created before its parameters are defined. Each register implements a dataflow i-store, which is allocated empty, i.e. undefined, blocks a reading thread if read when empty and reschedules a blocked thread when written to. Thus, parameters may be undefined when the family is created as a thread, which blocks until the shared register is set.

Threads in a family are created from a single thread definition, which may contain an *index* to distinguish different instances of it. This allows loops to be defined. The index value is initialised in hardware on thread creation to a value in the range defined by the create parameters and the threads are automatically distributed to the place where they are created (a cluster of cores configured into a ring network).

Blocking on threads is illustrated in figure 1. Note the shared variable, *sum*, in the thread definition. This identifies a value the thread may read from its predecessor's context and which, when set, may be read by its successor thread. In practice, it identifies two registers in a thread's context, one that is read-only and one that is write-once, that the thread must set - if it does not, the program will deadlock! For the first thread, its predecessor is the creating thread and for subsequent threads, it is the prior-thread in index sequence. Note that if two threads are mapped to the same core, a thread and its successor will share a physical register location, in a manner similar to the registers windows in the SPARC architecture. If however, the threads are mapped to adjacent cores (the only other alternative) a register read and/or write to a shared register will initiate a communication between adjacent core's register files. This happens automatically and makes all inter-thread communication implicit in the binary code.

In figure 1, the shared variable is equivalent to a scalar value in the body of the equivalent sequential loop in C. This μ TC code does not capture much real concurrency as the thread comprises only two instructions, one to read *a[i]* and one that adds this value to the prior thread's shared variable and sets its own shared variable. An observant reader may ask "what is gained by this?" The answer is that by specifying this loop concurrently with a shared variable, all of the memory accesses are specified concurrently and may be issued from multiple threads on a single core in any order. This provides a mechanism to tolerate latency in memory access.

In summary then, the create action can be used to implement both independent and dependent loops as concurrent families of threads, and to replace a function call and to capture it concurrently due to the blocking nature of the shared variables used as arguments.

2.2. Places, delegation and mutual exclusion

As illustrated in figure 1, the threads in a family, and its subordinate threads if required, can be executed at a specified remote place (the variable *pl* in figure 1). The place abstraction is implementation dependent and defines a set of processing resources. In the context of this paper, a place is a cluster of processors on the CMP, configured into a ring. Two model-defined places are *local* and *default* (no place specified). Local forces creation on the same processor (i.e. virtual concurrency only) and default distributes the family to the same cluster as the creating thread. All other places must be set by a place-server, in a very similar manner to allocating dynamic memory in C. As memory in SVP is asynchronous and cannot be used for implementing a mutex, we introduce the concept of a mutex place in SVP, which serialises any create requests sent to it. This is required to implement the place server.

The place abstraction solves a number of difficult issues. It provides an address for the delegation of work over the on-chip packet network making communication in SVP dynamically defined and implicit in the binary code. It optionally provides virtualisation of physical places. As an example, up to 32 legacy-code threads may be executed on a single SVP core described here and as described above, it supports mutual exclusion on families created at the same place.

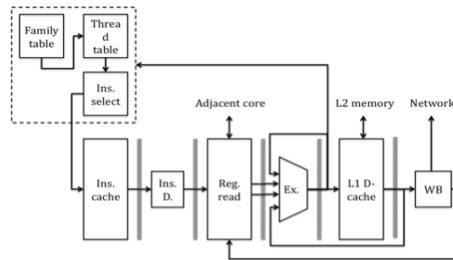


Figure 2. SVP DRISC pipeline showing the major additional components required to support interleaved thread execution.

3. Chip and processor architecture

We have designed a core based on an Alpha, in-order issue pipeline, which is augmented with instructions that support the SVP model. This is illustrated in Figure 2. We also have a reference implementation of a single cluster of such cores. It emulates every interaction in each stage of the pipeline as well as implementing a full memory interface. With this C++ emulator, we can vary parameters such as cache sizes, thread and family table sizes and number of cores configured in the ring network. Reasonable parameters have been chosen for implementation delays based on an evaluation of the silicon implementation [11]. The emulator therefore gives us a very realistic estimation of the processor's expected performance.

In the configuration used in this paper, each core is able to execute up to 256 threads and the thread table is indexed by the thread's identifier (0..255). It contains a program counter and information about the location of the thread's registers in the local register file, i.e. *globals* (accessed by all threads in a family), *locals*, *dependents*

(read only from the predecessor's *shareds*) and *shareds* (write once). It also contains other housekeeping information such as link fields for creating queues of threads, e.g. the queue of active threads, which hold all threads currently able to execute at least one instruction.

Similarly the family table contains 32 entries and the table index is a part of the family identifier, other components of the identifier are processor number and security key. Up to 16 families may be distributed between the cores in a cluster, using a common identifier between cores the other 16 family identifiers are for local creates.

Thread creation is very efficient, requiring only a few processor cycles to acquire a family table entry, initialise it and initiate thread creation, which is independent of pipeline operation and occurs at a rate of one thread per cycle, until either resources or block size are exhausted. It takes zero cycles to context switch and threads may do so every cycle, i.e. execute a single instruction from a thread and yield.

With up to 256 threads, a large register file is required for each thread's context of synchronising variables. In this paper we evaluate a core with 1024 Integer and 512 floating-point registers for their contexts, where registers are allocated in multiples of 32 for a block of threads. Each register in the register file has a fixed number of ports, three for normal pipeline operations and an additional two for asynchronous read and writes. These include operations that complete out of order, sharing registers between cores, initializing the index value, etc. More detail on implementation of the SVP core is given in [11] with area estimates for a core.

3.1 Register file and asynchronous operations

The register file and its synchronisation state need some further explanation. This synchronisation allows threads to be scheduled at the instruction level as when a register is empty, a thread attempting to read the register will suspend by writing its identifier to the empty register. This occurs on the write-back of the failed instruction. Now when data is written to the empty thread, the suspended thread will be reactivated and added to the active queue.

On creation, a thread's context of register variables is set to empty. This is also the case when asynchronous operations are executed, for example on the execution of a family create, the write-back stage will set the register to empty and then the register will be written asynchronously when the family has terminated. Any instruction trying to read the control code returned would suspend on the creating thread until that happens (the sync in figure 1).

3.2 L1 caches

Both the L1 I-cache and D-cache are similar in principal to their counterparts in conventional processors, but with some significant changes to deal with the blocking nature of microthreads and their scheduling. Lines in the I-cache are extended with a reference count and fields to maintain a list of threads waiting on the cache line. This is required as threads are only scheduled when their instructions are present in the cache. Thus on a branch or when the PC increments over a cache line boundary, a

context switch will be forced. Only on a hit to the cache will these threads be re-scheduled. Threads suspended on a register will also touch the cache before being rescheduled.

The L1 D-cache is also modified to keep track of outstanding memory reads on a line. A line is allocated on a cache miss, the request is sent to the next level of memory and the memory read request (which gets stored in the target register) is put on a linked list of requests for that line. When the data returns, these requests are serviced one by one after which the line can be reused. Finally, note that the D-cache can also be much smaller than in a conventional processor since the architecture supports as many outstanding reads as there are registers and the core will not stall as long as there are active threads.

3.3 Memory Architecture

In this paper, we evaluate a number of on-chip memory systems for the SVP CMP. We evaluate only on-chip memory at this stage and assume that some percentage of the chip area will be dedicated to level-2 cache, which must provide the abstraction of a shared memory and achieve this across potentially thousands of cores. Moreover, it must provide scalable throughput. Because SVP cores have good tolerance to latency, of the order of 100s of cycles in the configuration described above, our aim is to evaluate a COMA memory [13]. This has a relatively high latency due to its coherency protocol but which should provide a transparent mechanism to achieve locality of access due to the way in which cache lines are attracted to the cores using the data. This COMA memory is evaluated against a number of alternative designs. These are:

An ideal memory. This provides parallel, conflict-free, multi-ported access to all locations in memory. It takes 10 cycles to read a cache line and each CPU has concurrent, non-arbitrated access to any line.

A multi-bank memory. This assumes a non-blocking, fully connected, multi-stage switch to connect processors and memory banks. Cache lines are interleaved to banks on low-order address bits. Because it is assumed channels from the memory are relatively narrow, the memory has the following characteristics: the time for a request to get to or from memory is $\log_2(\text{number of banks})$ plus one cycle for each 8 bytes of data plus 10 cycles to read the cache line. So a read to 16 banks would take 5 cycles to get to memory, 10 to read a cache line and 13 to return the complete cache line, 28 in total. Conflicts may occur when multiple requests are routed to the same bank. Such requests are buffered at the bank and serviced in order of arrival. Conflicts decrease throughput and increase latency.

A randomised multi-bank memory, with similar network and characteristics but where the bank address is computed using a hash function of the address bits. Such a memory avoids systematic conflict patterns at the expense of losing any locality of access. Both multi-bank systems are non-coherent and distributed across the chip.

The COMA memory is similar to that in the KSR 1 [12], although our protocol has intermediate states to avoid the caches blocking on uncompleted transactions [13]. In this paper a single ring of L2 caches is evaluated where each cache is shared by four cores using a snoopy bus and the ring also has a top-level directory, directing requests

to off-chip backing store. All other on-chip memories evaluated are pre-loaded with data, but this is not possible in the COMA, we therefore implement a low-latency interface to external memory in this evaluation (1 cycle per cache line). This architecture scales up by using a hierarchy of rings but in this paper a single ring is evaluated.

4. Results and evaluation

The results presented here were measured using the cycle-accurate reference implementation of an SVP CMP, based on the specification of the model as described in [11]. It uses the Alpha ISA and each core in the cluster executes Alpha machine code with SVP extensions. Due to the unavailability of the compiler at current moment, the benchmarks were manually compiled into assembly and validated to ensure they generate correct results. Note that for each benchmark, the same binary code is executed unchanged on each memory configuration and for all cluster sizes.

Additional parameters for the results presented here include 1Kbyte, 4-way set associative L1 I and D caches; a 6-stage pipeline with asynchronous floating point operations taking 2, 8 and 10 cycles for add/mult, div and sqrt respectively; and a ring network that broadcasts a create to all cores in $P+k$ cycles for P cores and k parameters and takes 5 cycles for a remote read and 3 for a remote write.

4.1 Inner product

The first benchmark evaluated was an integer inner product (Livermore loop 3). This “sequential” algorithm was so that a distributed sequential reduction of one thread per core locally creates a family of threads to compute a partial reduction. The distributed reduction combines these results. This parallel algorithm has good performance so long as the number of cores is small compared to the number of elements to be reduced.

The benchmark implements a reduction over a 40960-element array on from 1 to 128 cores and the results are given in figure 3. As can be seen, the performance of all memories scales linearly but saturate at between 60 and 100 cores. This is not caused by a lack of threads, as there are more threads than resources available for the largest place evaluated. The saturation is due to the distributed reduction, which requires 10 cycles per core (1 cycle to reschedule a thread, 6 cycles for the pipeline latency and 3 cycles to write to the adjacent register file). This contributes 1 μ sec to a

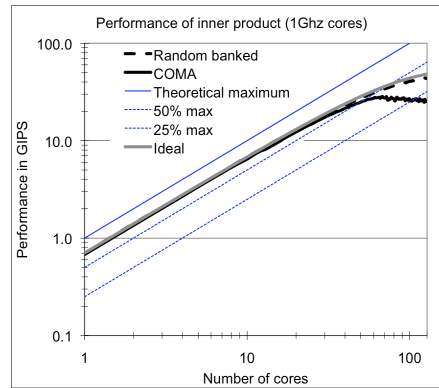


Figure 3. Performance in GIPS for the inner product benchmark, for a 1GHz core. The diagonal bands show the 100%, 50% and 25% efficiency performance for for this problem.

5 μ sec execution time with a 100-core ring. The reduction has a relatively low efficiency as only one thread is active at a time during the reduction operation.

The COMA memory, saturation occurs earlier and is harder than the other two memories and we believe this is due to bandwidth sharing on the cache-coherency ring network (64-core place shares the ring bandwidth between 16 L2 cache banks.) Note that unlike the ideal and random banked memories, the COMA memory sources all data from off chip in this emulation. We do not simulate a hierarchy of rings in this paper, which could limit this saturation in larger CMPs.

4.2 Matrix multiplication

The next results are for integer matrix multiplication. In the reduction, the code only reads data mapped by rows, matrix multiplication on the other hand combines both row and column mapped data in its inner product computation without any optimization. In this code, concurrency is found from the combination of outer and middle loop and the reduction is implemented sequentially. Figure 4 shows results for 64x64 matrices.

Both ideal and COMA memory give an 87.5% efficiency in the use of the pipeline cycles up to about 30 cores. The random banked memory however, is only 50% efficient, although this scales quite well. Given the fact that COMA system is simulated even without any pre-loading, the COMA is the clear winner for small number of cores. The COMA as expected saturates early and by 100 cores the performance of COMA and random banked memories are the same.

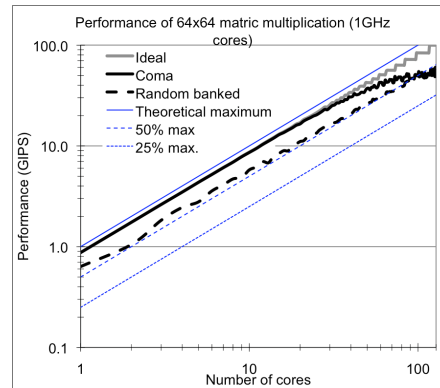


Figure 4. Performance of 64x64 matrix multiplication in GIPS on clusters of from 1 to 128 cores. Again the bands indicate maximum, 50% and 25% efficiency.

4.3 The 256 point FFT

With FFT, the combination of power-of-two data access with concurrent banks seems to result in memory-bank conflict. The 256-point version comprises a sequence of 8 families of 128 threads (using the Cooley Tukey algorithm). Each thread computes a complex butterfly operation using pre-computed roots of unity. No reordering is performed during the FFT execution, so results are produced in bit reverse order.

The results in figure 5 show that for a single core, the banked memory has an efficiency of 90%, the COMA 97.5% and the randomised banked memory 98%. This is based on the number of floating point instructions in the thread code and peak instruction issue rate. This is significant and shows that the core is able to tolerate significant latency through instruction-level scheduling for both floating point and memory operations.

Peaks in performance for the ideal memory occur when the number of cores is a power of 2, i.e. when the 128 threads are distributed perfectly to cores. In this code, there is no surplus of threads and load-balancing issues are visible. However, it can also be seen that the same points give the worst performance for the banked memory. This conflict situation is investigated in figure 6 using the pipeline's statistics.

As the number of cores increases, the performance of all memories saturates due to the limited by the number of threads available. Not surprisingly the ideal memory, which has a low base latency and no queuing latency on conflicts, scales the best and is still within 25% of the theoretical maximum at 64 cores. The COMA on the other hand has the highest latency and this falls below 25% efficiency at around 12 cores. With 128 threads per stage, with more than two cores, the number of threads per core will decrease in inverse proportion to the number of cores (see figure 6 for the average active-queue length, which reflects this). Hence, performance will decrease as the pipelines begin to stall waiting for memory accesses and floating-point operations. Figure 6 also shows the maximum time in cycles for which the pipeline is stalled during the execution, which increases from around 20 for a single core to almost 1000 cycles, with clear peaks at all powers of 2. This clearly indicates memory bank conflict, as it can be seen that the peaks correspond to peaks in execution time, with load balancing only contributing smaller fluctuations.

4.4 The 4096-point FFT

From figure 6 and 7, it is clear that the lack of threads limits performance significantly. Figure 7 shows the performance for the 4096-point FFT. In these results, we compare the ideal, random banked and COMA memory sequence of twelve families of 2048 threads, one means that for 128 cores, a maximum of 16 threads (including the main thread). As expected, the COMA is followed by the randomised banked memory. What

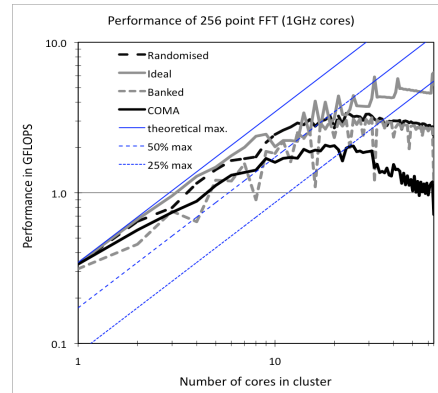


Figure 5. Performance on 256 point FFT for clusters of from 1 to 64 cores.

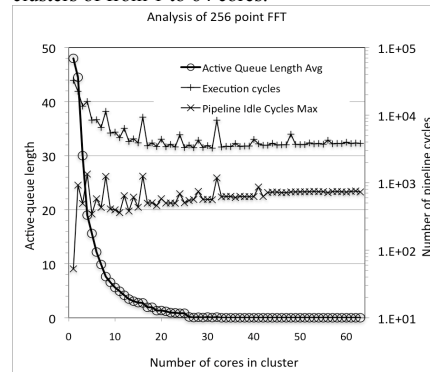


Figure 6. Analysis of the 256-point FFT for different number of cores executing on a banked memory with $M=P$. This shows the average active-queue length for the execution of this code, the number of cycles required for its execution and the maximum length of time the pipeline is stalled waiting for memory.

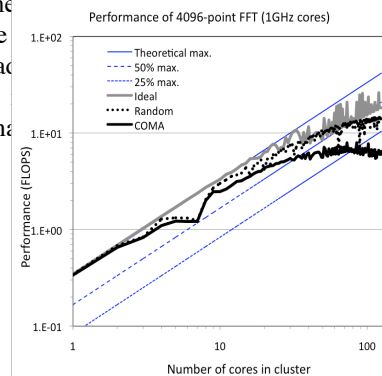


Figure 7. Performance on 4096-point FFT for clusters of SVP cores from 1 to 128. The diagonal bands show the maximum theoretical

sufficient threads, the COMA memory system is able to give a performance that comes very close to the ideal memory and unlike the random banked memory uses only local wires in its implementation, a key issue in any future silicon systems.

The anomalous step between 4 and 8 processors seems to be memory related, as it occurs for both real memory systems although not the ideal memory. We assume this is due to interactions between scheduling and memory delay. This situation is being investigated further.

5. Conclusions

In this paper, we have presented a model of concurrency and its implementation on ring-connected clusters of SVP cores. We have implemented a cycle-accurate software emulator for this MCCMP and have explored the performance of this micro-grid for a variety of memory architectures using a limited number of applications. The applications were chosen to have a range of concurrent memory access patterns. The inner product has entirely-local access patterns to distributed data, whereas the matrix multiplication accesses data by row and by column, so if data is distributed by either row or by column, some form of global access will be required. Finally, the FFT algorithm has access patterns that range from local to offsets of $N/2$ in powers of 2.

Our aim with the microgrid is to provide a concurrent architecture that can be used for a wide range of applications and where systems engineering and compilation issues should be a matter of matching bandwidth rather than relying on the detail of scheduling and execution latencies. This goal, we feel, has been demonstrated by our results. Across a wide range of cluster sizes, we achieve an efficient and consistent performance without changing the binary code.

We have identified a number of areas that contribute to the saturation of results as we scale up problem sizes. The most important issue is the lack of threads providing virtual concurrency, which is essential if an asynchronous schedule is to be achieved. We have also identified thread creation overheads and bandwidth saturation on the coherency network as cluster sizes scale up. The question of which memory system to implement on such a microgrid is also answered to some extent. The COMA memory system has the highest latency and hence requires more virtual concurrency than the other memory systems to maintain its performance. However, this memory system had the most consistent performance in its non-saturated region across the range of benchmarks. In particular, it outperformed the randomised banked memory in matrix multiplication by almost a factor of two. Moreover, it does so with a significantly smaller cost in terms of area and delay when implementing the communications required on chip. Whereas the randomised memory requires a wire whose length is determined by the overall dimension of the switch, all COMA communication on the other hand is in ring networks and good floor planning and layout can keep these wires fast and short.

Acknowledgements

We acknowledge support for this work from NWO in the project Microgrids and from the EU in the project Apple-CORE.

References

1. Semiconductor Industry Association (2006) International technology roadmap for semiconductors update. Technical report.
2. L. Hammond, B.D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun (2004) Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), pp 92–103.
3. C.R. Jesshope (2006) μ TC—an intermediate language for programming chip multiprocessors, *Proc ACSAC06*, LNCS 4186, pp 147–160.
4. T.A.M. Bernard, C.R. Jesshope, and P.M.W. Knijnenburg (2007) Strategies for Compiling μ TC to Novel Chip Multiprocessors, *International Symposium on Systems, Architectures, Modeling and Simulation, SAMOS 2007*, S. Vassiliadis et al. (Eds.), LNCS 4599, pp.127–138.]
5. A. Bolychevsky, C.R. Jesshope, and V.B. Muchnick (1996) Dynamic scheduling in RISC architectures, *IEE Trans. E, Computers and Digital Techniques*, 143, pp309–317.
6. Scholz, S.B. (2003) Single Assignment C - efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13, pp.1005–59.
7. H. McGhan. *Niagara 2 opens the floodgates* (2006) *Microprocessor Report*, 20(11), pp 1–12.
8. A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy (2005) Introduction to the Cell multiprocessor, *IBM Journal of Research and Development*, 49(4), pp 589–604.
9. C. Jesshope (2008) Operating systems in silicon and the dynamic management of resources in many-core chips, *Parallel Processing Letters (PPL)*, 18, (2), pp257 - 274.
10. K. Bousias, L. Guang, C.R. Jesshope and M. Lankamp (2008) Implementation and evaluation of a microthread architecture, *Journal of System Architecture*, <http://dx.doi.org/10.1016/j.sysarc.2008.07.001>.
11. K.S.R. Corporation (1992) KSR1 technical summary, Technical report.
12. L. Zhang and C.R. Jesshope (2007) On-chip COMA cache-coherence protocol for microgrids of microthreaded cores. in Bouge et. al. eds., *Proc EuroPar 2007 Workshops*. LNCS 4854, Springer, pp. 38–48.