

# The Challenges of Massive On-chip Concurrency

Kostas Bousias and Chris Jesshope

Computer Systems Architecture Group  
Instituut voor informatica  
Universiteit van Amsterdam  
 [{jesshope, bousias}@science.uva.nl](mailto:{jesshope, bousias}@science.uva.nl)

**Abstract.** Moore's law describes the growth in on-chip transistor density, which doubles every 18 to 24 months and looks set to continue for at least a decade and possibly longer. This growth poses major problems (and provides opportunities) for computer architecture in this time frame. The problems arise from current architectural approaches, which do not scale well and have used clock speed rather than concurrency to increase performance. This, in turn, causes excessive power dissipation and circuit complexity. This paper takes a long-range position on the future of chip multiprocessors, both from the micro-architecture perspective, as well as from a systems perspective. Concurrency will come from many levels, with instruction and loop-level concurrency managed at the micro-architecture and higher levels by the system. Chip-level multiprocessors exploiting massive concurrency we term *Microgrids*. The directions proposed in this paper provide micro-architectural concurrency with full forward compatibility over orders of magnitude of scaling and also the management of on-chip resources (processors etc.) so as to autonomously configure a system for a variety of goals (e.g. low power, high performance, etc.).

## 1 Introduction

### 1.1 Micro-architecture challenges

Although today's large scale parallel computing systems comprise clusters of commodity processors, discs and networks, future systems will have to address fundamentally new issues as we inevitably move towards large-scale, on-chip parallelism, i.e. from  $10^3$  to  $10^5$  processors, which we call *Microgrids*. Microgrids will also form the basis of mega-scale computing systems, comprising millions of processors, compounding the issues and increasing system-management complexity. To fully exploit such complex systems it is essential to answer some fundamental questions that simplify and give a formal basis for on-chip concurrency models, execution strategies and resource management. Failure to address these problems has delayed the introduction of highly concurrent micro-architecture [1] and consequently, technology advances over the last decade have advanced processor performance through clock speed, using a combination of smaller gate delays and shorter pipeline stages. Performance gains from concurrency have been limited, even though circuit density has grown more rapidly than circuit speed. Instead, increased circuit density is supporting unscalable

execution models, such as out-of-order issue (OoO). Ironically, large on-chip memories are then used to mitigate against the divergence between on-chip clock speeds and memory cycle times, resulting from the aggressive clocking. OoO has a long history going back to Tomasulo's algorithm introduced in the IBM 360/91 [2] and subsequent developments such as reorder buffers [3], which are used extensively in modern microprocessors. However, instruction dispatch [4] and register file [5] implementations have poor scaling properties and this has led to several, recent, high-profile projects being cancelled due to excessive circuit complexity and power dissipation e.g. [6]. This power barrier should have been no surprise, as in 1999 it was predicted that the Alpha 21464 would use a quarter of the chip's power budget on its instruction queue [7]; this chip was also cancelled in 2002.

Using concurrency to obtain performance is a much better strategy, as long as some fundamental questions can be answered. To illustrate this consider the T800 transputer [8], a 0.25M transistor chip with 64-bit floating point capability, designed for concurrent applications. This processor could be replicated 400 times in current technology, giving instruction issue widths a hundred times those found in current OoO processors. Such naïve chip multi-processors (CMPs) are not particularly viable in a general-purpose market, as they require explicit, user-level, concurrency to program them making the migration to such systems slow and difficult. The one and only advantage of the OoO paradigm is that concurrency is extracted and exploited implicitly from legacy binary code. Very-long Instruction words (VLIW) are used increasingly in embedded applications and also have problems in scaling up to massive concurrency. Here scheduling is delegated to the compiler, which although produces lower-power solutions to instruction issue, also generates static schedules that limit scalability. The first challenge we face therefore, is to obtain scalability across a wide range of codes while retaining binary- and source-code backward compatibility.

In [9,10] a number of technological challenges are outlined for future micro-architectures. These include scalability of micro-architectures in area, performance and power dissipation, as well as strategies for chip multiprocessors that address power awareness and power management. Finally, there is the issue of signal propagation, which will force micro-architectures to eliminate global on-chip communication completely. One of the major global communication networks is the clock-distribution network and a more practical approach to future CMP design would be to use a globally-asynchronous, locally-synchronous (GALS) clocking approach but the big question is how to design ILP processors, which naturally synchronise on register variables, with an asynchronous and distributed model of communication. This is the challenge undertaken in this paper.

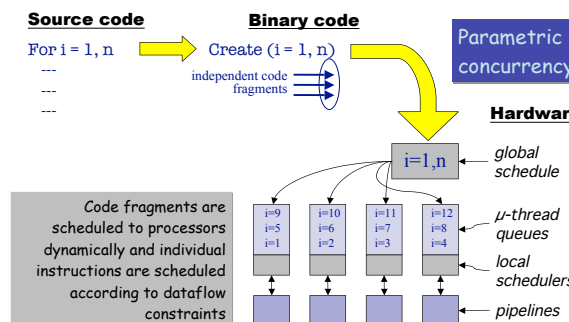
## 1.2 Micro-architecture concurrency

There are two widely-used models of concurrency at the micro-architectural level. The first is implicit and relies on hardware to detect and enforce dependencies when executing instructions out of order. As already indicated this model scales very badly when increasing concurrency. The other model is VLIW, which has better scaling but has compatibility problems. In particular, binary code must be regenerated (as the schedules are static) for each increase in concurrency. EPIC architectures provide

some remission in this area by allowing the binding of instruction to resource to be dynamic. However, it requires many of the structures found in OoO approaches, such as branch predictors and the static schedules limit scalability.

A third way has been explored by a number of groups; it relies on decomposing and managing multiple fragments of code concurrently. The scheduling of these code fragments must be made efficient and this requires the fragments to be exposed within a single context, which differentiates it from most multi-threaded architectures. By interleaving fragments, latency tolerance is achieved and by distributing fragments to different functional units or processors, speedup is obtained. The first published paper on code fragmentation was called *microthreads* and dates back to 1995 [11]. It was proposed as a means by which processors in a distributed system could tolerate high levels of latency. More recently a similar approach called *intrathreads* [12] adopts the same principal but with a different approach to implementation. It uses bounded concurrency and statically-partitioned resources, whereas microthreads describe parametric concurrency where resources are managed dynamically though the concept of micro-contexts. Another difference is that intrathreads separate synchronisation and data storage, where microthreaded processors implement registers as i-structures synchronise between code fragments. In recent papers, microthreading has been extended to support CMPs [13,14] and simulated to evaluate latency tolerance and speed up [15,16].

These models are both incremental and add just a few new instructions to an existing ISA to implement explicit concurrency controls. In microthreading, these instructions define parametric sets of concurrent code fragments, which are scheduled dynamically on multiple processors. In intrathreads, the number of threads is fixed and the implementation is targeted to wide-issue pipelines rather than to a chip multi-processor. A key feature of microthreads is that concurrency is parametric but that schedules are dynamic. The same binary code can therefore be run on an arbitrary number of processors, limited only by the parametric concurrency. This allows for the dynamic management of resources in microgrids. Thus the number of processors can be set dynamically to satisfy constraints on performance or power dissipation without modifying the binary code. It will be demonstrated in this paper that a number of tradeoffs can provide management of power and performance over an extremely wide range of processors using largely linear functions. This makes the model ideal for autonomous configuration.



**Fig. 1.** An illustration of the concept of microthreaded scheduling

Microthreading is applied within a single context and supports a shared-register model of data using blocking reads (code fragments are suspended on registers waiting for data to be written). Memory consistency is managed using a barrier synchronisation instruction, which forces

Microthreading is applied within a single context and supports a shared-register model of data using blocking reads (code fragments are suspended on registers waiting for data to be written). Memory consistency is managed using a barrier synchronisation instruction, which forces

single-threaded, in-order execution. It has been shown [1,17] that implementations of this model are completely scalable and support asynchronous inter-processor communication that is tolerant to communication delay and does not force any pipeline stalls. This approach therefore, solves many of the challenges raised in [9,10]. This model can exploit the full benefit of scaling due to Moore's law to the end of silicon (i.e. an estimated  $10^3$  to  $10^5$  processors per chip).

Microthreaded concurrency is obtained by compiling source code using a microthread-aware compiler (or by a translation from existing binary code). The model requires in-order execution semantics, which means that optimal static schedules can be generated for instruction sequences and deterministic compiler optimisations can be applied. The exception is where data generation is non-deterministic, as in inter-thread communication, cache access or iterative operations. When attempting to read such data, an explicit context switch is signalled to the hardware and another code fragment is executed. Concurrency is captured by instructions that create families of threads to execute loops for all values of their index variable concurrently; basic-block concurrency can also be captured. The former is parametric and the latter is static. There are constraints on the creation of code-fragment instances due to resource availability and also dataflow constraints on the execution of individual instructions. These constraints determine the dynamic schedule for an execution of the code on a given number of processors.

### 1.3 Resource management

Resource management is currently divided between two very different domains. At the processor level, it assumes that computation is performed using a single, powerful processor and a large global memory system. The memory holds images of all current activities, i.e. operating system and user tasks. Resource management is then almost entirely undertaken by sharing processor cycles between these tasks. The large memory is required to store the state of the multiple tasks and is slow. A memory hierarchy is then used to solve this problem by caching the data close to the processor, using implicit data transfers. This is not an optimal solution for several reasons:

- ideally, memory should be distributed to make it faster;
- transfers to cache can be initiated early to achieve tolerance to the high latency memory, interaction between tasks can interfere with implicit and explicit transfers making optimal solutions heuristic rather than deterministic;
- finally, moving state between levels of memory will aggravate bandwidth requirements and cause significant power dissipation.

Solutions to these problems can be found in recent research on processing in memory architectures (PIM) [18] and, as this paper will show, using microthreading combined with novel resource management solutions, while using near-conventional instruction sets with full code compatibility.

The second resource-management domain is on the scale of meta-computers in Grid infrastructures. This assumes the low-level resource management described above and provides on top of that, some measure of service quality by resource reservation and application adaptation. This is usually implemented as a middleware layer on top of one or more conventional operating systems, (e.g. [19,20]). Neither domain

provides solutions to resource management at the micro-architecture level when dealing with large numbers of processors. A middleware solution is too coarse grain and conventional operating systems are more suited to single processor environments.

Recently, much emphasis is being placed on the optimisation of power dissipation. Attempts have been made to manage power dissipation as a function of issue width in speculative processors [21-24]. These solutions rely on code profiling but dynamic concurrency varies significantly leading the use of dynamic hardware profiling [25], which increases power usage and so there are limits on the scope of such techniques. Ideally, compiler-based solutions are preferred, e.g. [26] but such approaches can only effectively control the cache power-performance. Combined compiler and hardware approaches using fetch throttling can also control concurrency [27] but give only marginal impact, i.e. 10 to 20% power savings with similar performance degradation through lower IPC. A second and profound question then is how to design systems of thousands of processors managed from legacy code, while optimising various goals such as performance, power and responsiveness over orders of magnitude?

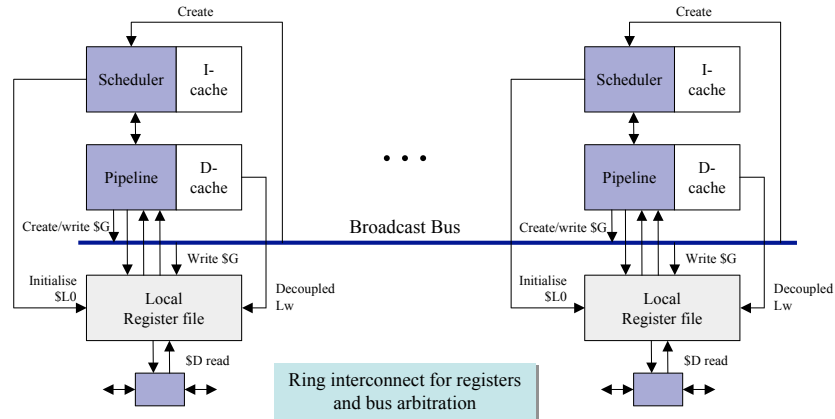
## 2 Microgrids

### 2.1 New processor architectures

Microthreading provides parametric concurrency using a few additional instructions to manage fragments of code efficiently. It adopts a shared-register model of data with synchronisation on all registers. Assume  $ISA\text{-}mt$  are the additional microthreaded instruction, then given a RISC processor whose instructions are defined by  $ISA\text{-}RISC$ , then a new ISA can be defined incrementally by the union of the two, i.e.  $ISA\text{-}RISC + ISA\text{-}mt$ . Similarly we could define a VLIW microthreaded architecture by the union of a different instruction set,  $ISA\text{-}VLIW$  with  $ISA\text{-}mt$ . This paper is concerned only with the issues arising from  $ISA\text{-}mt$ .

$ISA\text{-}\mu t$  has been fully specified in [1]. Using that definition, families of code fragments can be specified using the following two concepts:

- i. *Sets* of code fragments statically define concurrency within a shared register domain. Each code fragment is specified by a pointer to its first instruction  $\{P_i, 0 \leq i \leq n-1\}$  and is terminated by a *Kill* instruction. There is no restriction on communication between sets of code fragments.
- ii. *Iterators* dynamically define concurrency over a set of code fragments. An iterator specifies an integer index variable,  $i$ , using a triple  $\{s, l, t\}$  such that  $\{i \geq s; i \leq l; i = s + kt\}$  where  $k$  is a positive integer. The code defined by the set is shared between iterations by defining a micro-context for each value of  $i$ . A micro-context is a partition of a processor's physical register space allocated to an iteration. The first location of the micro-context is initialised to the index value,  $i$ . In the current model, communication between micro-contexts is restricted and an iteration can access the micro-context of just one other prior iteration, defined by a constant stride in the index space,  $d$ . The number of registers in the micro-context and the value of the stride complete the definition of an iterator.



**Fig. 2.** A chip multi-processor based on a collection of microthreaded pipelines. The ring network and broadcast bus (which may be implemented over the ring) provide asynchronous communication between processors decoupled from the local pipeline operation

Less restrictive models can also be defined to increase the potential concurrency exposed, for example in allowing multiple, constant-strided dependencies or even variable-strided dependencies. However, as concurrency is parametric and unrelated to machine resources, these models may induce resource deadlock, which is not easily resolved. This paper considers only the simplest model above, where conditions under which resource deadlock occur can be easily specified.

## 2.2 New chip architectures

A *Microgrid* is a chip comprising, in our model,  $M$  microthreaded processors, where each processor (or cluster of processors) is implemented in its own, clocked domain and communicates asynchronously with the rest of the chip. Each processor has a local register file, access to a broadcast bus and a ring network for shared-register communications between neighbouring processors, see [1] and figure 2. When executing a single thread of control (a *Context*), a number of processors can be used to execute an iterator, this is called the *Profile* associated with that context. A profile is an ordered subset of processors of cardinality  $P$  selected somehow from the  $M$  processors in the microgrid. This subset is configured to have a broadcast bus and ring network linking only the processors within it.

During the execution of an iterator, each processor's register file will contain the state for the current context plus micro-contexts for iterations scheduled to it. For a given  $P$ , the triple defining an iterator and a processor's position in the profile, it can independently determine the iterations it must execute. These are initialised when it has resources available, in the form of handles to identify its code fragments and registers to allocate to its micro-context. The distribution of iterations to processors is de-

fined by a simple modulo mapping such that iteration  $i: s \leq i \leq l$ , is allocated to processor  $j: 0 \leq j < P$ , where:

$$j = \lfloor i/kd \rfloor_p \quad (2.1)$$

Thus blocks of  $kd$  consecutive iterations are allocated to each processor in turn, where  $d$  is the communication stride and  $k$  is a locality parameter, a natural number limited by the number of registers in the local register file,  $R$ . If the static context requires  $G$  registers and each micro-context requires  $L$  registers, then to avoid resource deadlock the following inequality must be satisfied:

$$kdL + G \leq R$$

Normally  $k$  should be maximised subject to the above constraint, as  $k-1$  is the ratio of local to remote communications in a loop-carried dependency chain. Note that  $k$  also determines locality in the local D-cache if implemented and can be chosen to take full advantage of the line size, data size and access patterns to local data. When executing independent loops, there are no issues with resource deadlock and an arbitrary allocation of iterations to processors is possible as there is no communication between micro-contexts.

### 2.3 Power awareness and low-power operation

The final issue to be considered before a microgrid operating environment can be discussed is that of power models for microthreaded microprocessors. As we have already seen in [21-27], managing power using implicit concurrency is difficult and has limited effectiveness. Implicit ILP relies on speculation and eager instruction execution policies, which are at odds with power conservation. Microthreaded microprocessors, on the other hand, have conservative instruction execution policies that enable power-aware operation and yield power-efficiency. Only a single instruction per code fragment is fetched at a time and branch and data hazards suspend execution of that fragment until the hazard has been resolved; execution can continue from other fragments if any are active. In the case of a branch hazard the fragment is suspended for a few cycles until the branch-target address is computed but with a data hazard the code fragment is suspended indefinitely on a register until the required data has been written (if the data already exists execution is suspended only long enough to discover that fact). Conservative instruction issue policies enhance power efficiency, as:

- no power is dissipated on speculative instruction fetch and execution;
- no area and power are required in making branch or data predictions;
- no area and power are required in managing missprediction cleanup;
- finally, conservative models provides signals to manage power dissipation.

Code fragments are managed by a scheduler, which selects a new code fragment for execution on any instruction from ISA- $\mu t$  that causes a context switch. It also manages the state of all allocated fragments, i.e. whether they are active or suspended. When all fragments are suspended, this could be used to trigger a higher level context switch but as already noted, this will involve significant data movement and power

consumption. Alternatively this state can be actively used to manage local power dissipation. Each processor runs its own clock and that clock can be stopped awaiting data, eliminating any dynamic power dissipation. By definition, as no local threads are active, the wake-up signal must arrive externally, either from the bus, the ring network or from memory. These inputs are managed by an asynchronous interface to the external read/write port of the register file, which can signal the scheduler to restart the local clock when it reschedules the suspended fragment. The same signal can be used to manage a processor's power rails and minimise static power dissipation when idle. Note that such control is not possible in speculative execution policies.

This policy allows us to statically place single contexts on dedicated processors, which become idle while waiting for external events. This distributes the locus of control of many tasks and localizes the use of memory, avoiding excessive and unnecessary migration data between different levels of memory. If further, the computation can be described at a higher level as a communicating collection of components (i.e. a streaming network), the tasks can be data driven and the only movement of data will be due to explicit algorithmic concerns, rather than interference in cache memory from scheduling all tasks to a single monolithic processor.

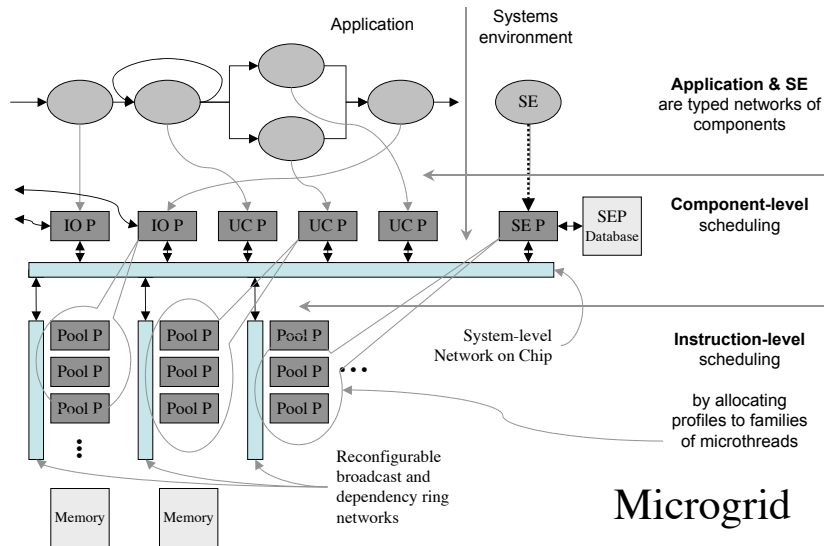
### 3 Microgrid system environment

It is first useful to summarise the properties of microthreaded pipelines before looking at how a system environment can be implemented to manage the processors in a microgrid. The properties relevant to this discussion are that:

- microthreaded binary code captures parametric concurrency and those parameters can be set dynamically;
- microthreaded binary code is schedule invariant and can be executed unchanged on a number of processors up to a limit defined by the parameters;
- instructions are tolerant of high levels of latency in their operands;
- processors have asynchronous interfaces and are independently clocked;
- processors consume minimal power while waiting for external events.

An operating system environment to support the massive on-chip concurrency proposed requires new paradigms to be adopted. Microthreaded ISA extensions allow concurrency to be extracted from legacy code through binary code translation or re-compilation. Because this code is schedule-independent the environment can support the allocation of dynamic profiles to contexts. There is also a need to execute unmodified binary code from the base ISA on a single processor. Thus the system environment must provide support in launching contexts and in adjusting their profiles. This would support all forms of concurrency such as user jobs, multithreaded applications etc. and be flexible enough to support explicitly programmed concurrent applications in new programming paradigms, e.g. [28,29]. A strategy is proposed below for building such a System Environment Process (SEP). It assumes:

- there are a large number of processors on a chip (e.g.  $10^3 - 10^5$ );
- contexts are allocated to processors for their duration;
- contexts communicate using shared memory and/or I/O, managed by microthreads and microcontexts running on dedicated processors;



**Fig. 3.** The microgrid concept showing various levels of scheduling for the on-chip resources. IOP = I/O processor; UCP = user context processor, SEP = system environment process

- one processor runs a “kernel” (the SEP), that manages a model of the system resources and is responsible for launching contexts and configuring profiles.

The profile for a microthreaded context is defined as the number of processors allocated to it, at a given time. A profile is initially a single processor, when the context is launched. Later, if the context exploits ISA- $\mu$ t, then more than one processor can be used to execute the created code fragments. The processors are added to the context dynamically by requests to the SEP. The choice of profile can be used to optimise the chip’s performance according to goals and environmental factors. The times at which a profile may change are during single-(micro)threaded execution, i.e. following a barrier synchronisation and prior to the next create instruction [1]. At these times no micro-contexts exist and the context is fully defined by its state on a single processor.

The simplest form of profile is where a context is allocated a fixed number of secondary processors for the duration of its execution; a more dynamic profile might allocate resources at function boundaries and the most dynamic would be adjusted at the level of individual create instructions (loops). Thus the time constant for reconfiguration would vary from human interaction speeds down to once every 1000 or so processor cycles, i.e. from  $O(1)$  to  $O(10^6)$  times a second, spanning at least 6 orders of magnitude. The lower estimate is based on the number of registers in a processor assuming that each is written at least once during the optimal execution of a loop.

In practice, the smallest time constant in setting a profile will depend on a number of issues. These include the characteristics of the code, the requirements for its execution (e.g. minimum time, maximum throughput, maximum latency tolerance, minimum power etc.) and the time required to configure the profile following a request to the SEP. The latter involves several transactions on an in-memory database

and the configuration of a ring network. Only after the configuration is complete can a Create instruction broadcast a pointer to its parameters to the new profile. Then, each processor autonomously executes its schedule as defined by equation 3.1.

Requests for a profile must be embedded in the compiled code at compilation or binary translation and are remote requests to the SEP, where a global model of all resources is maintained. The SEP also manages the configuration of ring networks. It is important to understand that an execution of the code is unaffected by the number of processors used, except in terms of speed and power dissipated. It is even possible to loosely couple the allocation process and the execution of a subsequent family of microthreads. The simplest protocol would involve:

- i. a context making an SEP request;
- ii. the SEP updating its profile model, configuring a new set of resources but not binding them to the context's current environment;
- iii. both SEP and context competing on the system bus and either:
  - a. the context winning, executing a create to the old profile and releasing the bus – the SEP would re-absorb the unused resources;
  - b. the SEP winning, binding the new resources to the old and releasing the bus – the context requires this to execute the create.

Resource management therefore uses a client-server model and exploits the schedule independence of microthreaded code. Using the above protocol, requests for resources are non-binding and non-blocking and are a part of the compiled code. Non-microthreaded code gets a single processor and benefits from power-awareness without using microthreaded instructions or dynamic profiles.

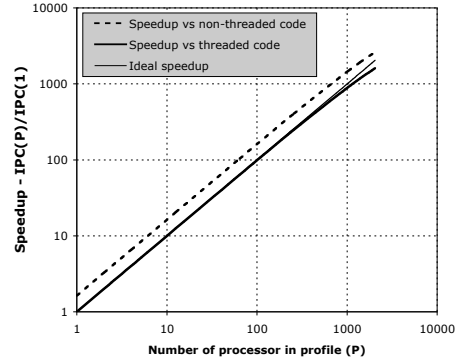
#### **4. Enabling results**

In order to demonstrate the feasibility of this approach, results are presented that demonstrate the scaling characteristics necessary for its operation. They are based on a simulation of a CMP shown in figure 2, using a seven-stage Alpha pipeline, with parameters as defined in table 1. Each processor executes instruction in-order without branch prediction. The CMP is used to evaluate the performance and power scaling assuming fine-grain regulation of dynamic power in the scheduler as described in section 2.3. The code executed is a hand-compiled fragment that implements the Livermoore hydro fragment. The higher-level microgrid architecture is assumed to be ideal, with a relatively slow but non-blocking second level of shared memory. This is a reasonable assumption for regular computation as data can be partitioned according to the a-priori schedules. Scheduling information can also be used to optimize the L1 D-cache hit rate. The scope for optimisation in microgrids, which have simple, regular schedules, is much higher than in a modern superscalar processor, where all aspects of code execution are speculative and heuristic.

The L1 cache controller must quash multiple requests to memory for the same cache line. Association between address and target register is managed by tagging requests with the register specifier and can be buffered anywhere in the memory system. Note that with regular schedules and an 80% cache hit rate, only 2-3% of memory loads cause a request to the second-level memory, which is pipelined and

**Table 1.** Parameters for simulations

System parameters	
Main memory Size	4 MB
Local registers / processor	1024
LCQ entries / processor	512
I-cache parameters	
Line Size	32bytes
Associativity	8
No Of Sets	8
Buffer entries	2
D-cache parameters	
Line Size	64bytes
Associativity	8
No Of Sets	128
Buffer entries	512



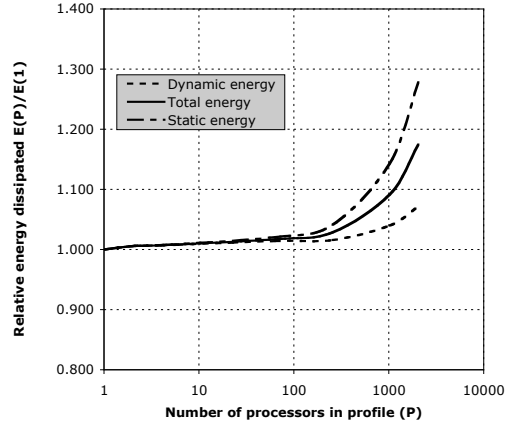
**Fig. 4.** Speedup of hydro-fragment against number of processors in microgrid profile

provides a line of 64 bytes in 24cycles. Transfer is in 8-byte words and the requested word is returned first in 10 cycles.

The results presented in Figure 4 which shows the speedup for the Livermore hydro fragment over a wide range of profile sizes executing the same binary code. The simulation is performed with cold caches and includes the overhead of thread creation and barrier synchronization following the execution of the loop. The schedule maps 16 consecutive iterations per processor, in order to optimise the L1 D-cache hit rate. The speedup is within 3% of the ideal for up to 256 processors and is still within 20% of the ideal for a profile of 2048 processors. The loss of efficiency for larger profiles is due to the amortisation of start-up overheads over fewer cycles and less latency tolerance resulting from the fixed problem size of 64K iterations. This gives only 32 iterations per processor for the largest profile or a 20% utilisation of the processors' resources. Figure 4 also shows speedup against non-microthreaded, single-processor execution and here the speedup is super-linear for all profiles. Note that microthreaded code executes about 20% less instructions than non-microthreaded code to achieve the same result, as index and loop control operations are "executed" in the scheduler. Management of control and data hazards also contributes to the superior single-processor performance of microthreaded code, although but other architectures will have different solutions to this that are not simulated here. Nevertheless the single processor profile achieves an IPC of 99.8% on this code even in the presence of cache misses and a slow second-level memory.

The simulator also models power dissipated and figure 5 shows the results, which assume the processors' clocks are enabled on a cycle by cycle basis depending on whether the scheduler has active threads to execute or not. It also assumes that all processors in a profile dissipate static power for the duration of the computation. The results are presented as the relative energy consumed by the hydro-fragment kernel as a function of the number of processors in its profile. The total energy assumes that a processor consumes the same amount of energy from dynamic power and static power dissipation. These results also assume that processors not in a profile consume no power.

Regardless of the balance between static and dynamic power, the envelope between static and dynamic energy in figure 4 will contain the total energy. Again it can be seen that over two and a half orders of magnitude the power dissipation remains within 3% of that consumed by executing the computation on a single processor. At 2048 processors the efficiency of the computation decreases due to inefficient use of the pipeline and a 17.5% overhead in energy consumed is seen. This is a very significant result as it opens up a mechanism for very low-power computation.



**Fig. 5.** Relative energy dissipated for the same computation using different numbers of processors in a profile

Figure 4 shows power dissipated for scaled performance using a large number of processors. When using multiple processors for constant performance, their frequency can be reduced by a factor equal to the number of processors used. Scaling the voltage with frequency would reduce the total energy required by a factor close to the square of the processors used!

## 5. Conclusions

In this paper the microthreaded model of concurrency has been summarised and the concept of a microgrid, based on massively concurrent and asynchronous collections of microthreaded processors has been proposed. It is argued that microgrids need new concepts of operating environments and that this approach can exploit Moore's law to the end of silicon. The proposed approach exploits massive concurrency by statically placing user and systems contexts rather than time-sharing them. This is combined with dynamic profiles for contexts that use microthreaded instructions. The schedulers in each processor can exploit the model's parametric concurrency by autonomously organising the computation across all processors in the profile. Higher-level control of this mechanism can be used to optimise a number of system goals over a wide range of parameters. The results presented show a linear performance scaling on independent loops that spans profiles ranging over two and a half orders of magnitude for a fixed sized problem. The performance over this range deviates from ideal scaling by only 2% and the energy required does not grow by more than 3% despite the additional performance.

In a microgrid, different contexts would draw dynamically from a pool of processor resources rather than being time-sliced on a single powerful processor. All contexts thus retain a minimal profile of one processor, even though that processor may be idle for much of the time. With  $10^5$  processors available, there will no lack of proc-

essors. Also many low-frequency (system) tasks can be scheduled as collections of microcontexts on a single processor, responding efficiently to external events, such as timers or I/O and not wasting system resources. Even if these root processors are idle, this strategy still makes sense, so long as the idle processors do not consume power. In general, a good systems management strategy must minimise the critical resource usage and in a microgrid, this is not processor cycles! Rather it will be power dissipated, memory bandwidth, chip I/O and perhaps other characteristics. This static allocation of minimal profile plus dynamic allocation of additional processors can be based entirely on compiled user-code and most importantly the requests for resources are both non-binding and non-blocking.

There are many research questions still to be answered in providing a foundation of tools and interfaces for the control and optimisation of these critical resources but this paper demonstrates the scalability that underpins this approach. Clearly it introduces massive design space optimisation issues, however, as the tradeoffs are largely linear, the exploration can be simplified and even made dynamic and embedded into the systems environment model.

## References

1. C Jesshope (2004) Micro-grids - the exploitation of massive on-chip concurrency, invited paper, Cetraro HPC workshop 2004, Cetraro, Italy, to be published in *Advances in Parallel Computing*, 2005 (<http://staff.science.uva.nl/~jesshope/Papers/HPC-paper.pdf>)
2. D W Anderson, F J Sparacio, R M Tomasulo (1967) The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling, *IBM J. Res. Dev.*, **11**, Number 1, p8.
3. J Smith and A Pleszkun (1985) Implementation of Precise Interrupts in Pipelined Processors, in *Proc. of Int'l. Symposium on Computer Architecture*, pp.36–44.
4. G Kucuk, D Ponomarev, K Ghose and P M Kogge (2001) Energy-Efficient Instruction Dispatch Buffer Design, in *Int'l. Symp. on Low Power Electronics and Design (ISLPED'01)*, August 2001.
5. R Balasubramonian, S Dwarkadas and D Albonesi (2001) Reducing the Complexity of the Register File in Dynamic Superscalar Processor, in *Proc. of the 34th Int'l. Symposium on Microarchitecture (MICRO'01)*, p37.
6. A Shilov (2004) Intel to Cancel NetBurst, Pentium 4, Xeon Evolution, <http://www.xbitlabs.com/news/cpu/display/20040507000306.html>, accessed 7/1/2005.
7. K Wilcox and S Manne (1999) Alpha processors: A history of power issues and a look to the future, Cool-Chips Tutorial, November 1999. Held in conjunction with MICRO-32.
8. M Homewood, D May, D Shepherd and R Shepherd (1987) The IMS T800 Transputer IEEE Micro, October 1987, pp10-26.
9. R H J M Otten and P Stravers (2000) Challenges in physical chip design, *Proc. ICCAD '00*, p84, San Jose, California, IEEE Computer society press.
10. R Ronen, A Mendelson, K Lai, F Pollack and J Shen (2001) Coming challenges in Microarchitecture and architecture. *Proc IEEE*, 89 (3), pp325-40.
11. 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
12. A Gontmakher and A Schuster (2002) Intrathreads: Techniques for Parallelizing Sequential Code, *6th Workshop on Multithreaded Execution, Architecture, and Compilation (MTEAC-6)*, November 2002, Istanbul (in conjunction with Micro-35).

13. C. R. Jesshope and B. Luo (2000) Micro-threading: A New Approach to Future RISC, *Proc ACAC 2000*, pp34-41, ISBN 0-7695-0512-0 (IEEE Computer Society press), Canberra Jan 2000
14. Jesshope, C. R. (2001) Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines, *Proc. ACSAC 2001, Australia Computer Science Communications*, **23**, No 4., pp80-88, IEEE Computer Society (Los Alimitos, CA, USA), ISBN 0-7695-0954-1.
15. Luo B. and Jesshope C. (2002) Performance of a Micro-threaded Pipeline, *in Proc. 7th Asia-Pacific conference on Computer systems architecture*, **6**, ( Feipei Lai and John Morris Eds.) Australian Computer Society, Inc. Darlinghurst, Australia, ISBN ~ ISSN:1445-1336 , 0-909925-84-4 , pp83-90.
16. Jesshope C. R. (2003) Multithreaded microprocessors – evolution or revolution (Keynote paper), *Proc. ACSAC 2003: Advances in Computer Systems Architecture*, Omondo and Sedukhin (Eds.), pp 21-45, Springer, LNCS 2823 (Berlin, Germany), ISSN0302-9743, Aizu, Japan, 22-26 Sept 2003.
17. Jesshope C. R. (2004) Scalable Instruction-level Parallelism, *In Computer Systems: Architectures, Modelling and Simulation, Proc 3<sup>rd</sup> and 4<sup>th</sup> Int'l. Workshhops, SAMOS 2003, SAMOS 2004*, (LNCS 3133, Springer), ISBN 3-540-22377-0, pp383-392, presented Samos, Greece, July 2004.
18. Brockman J. B., Thoziyoor S., Kuntz S. K. and Kogge, P. M. (2004) A low cost, multi-threaded processing-in-memory system, *Proc. 3rd workshop on Memory performance issues (ISCA 31)*, ISBN:1-59593-040-X, pp16 – 22.
19. I Foster, A Roy and V Sander (2000) A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation, *Proc. 8th International Workshop on Quality of Service (IWQOS 2000)*, Pittsburgh, USA.
20. R Buyya and M Murshed, (2002) GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing, *Journal of Concurrency and Computation: Practice and Experience (CCPE)*, **14**(13-15), Wiley Press, Nov.-Dec., 2002
21. S Manne, A Klauser, and D Grunwald, (1998) Pipeline Gating: Speculation Control for Energy Reduction, *Proc Intl Symp. On ComputerArchitecture (ISCA)*.
22. D Marculescu (2000) Profile driven Code Execution for Low Power Dissipation, *Proc. Intl. Symp. Low Power Electronics and Design*.
23. V Zyuban and P Kogge (2000) Optimization of High-Performance Superscalar Architectures for Energy-Delay Product, *Proc Intl. Symposium on Low Power Electronics and Design*.
24. S Ghiasi, J Casmira and D Grunwald, (2000) Using IPC Variation in Workloads with Externally Specified Rates to Reduce Power Consumption, *Workshop on Complexity Effective Design. ISCA27*.
25. A Iyer and D Marculescu (2001) Power aware microarchitecture resource scaling, *Proc Design, automation and test in Europe, Munich, Germany*, pp190-6, ISBN:0-7695-0993-2.
26. R Sree, A Settle, I Bratt and D A Connors (2003) Compiler-directed resource management for active code region, *Proc. 7th Workshop on Interaction between Compilers and Computer Architecture*, February 2003, 5.5.
27. S Chheda, O Unsal, I. Koren, C M Krishna and C A Moritz (2004) Combining compiler and runtime IPC predictions to reduce energy in next generation architectures, *Proc. 1<sup>st</sup> Conf. On Computing Frontiers archive*, Ischia, Italy, pp 240-54 , ISBN:1-58113-741-9.
28. A Shafarenko and S-B Scholz (2005) General Homomorphic overloading. Accepted for publication in LCNS (2005). A preliminary version was submitted to IFL'2004 in Luebeck.
29. A Shafarenko (2003) Stream Processing on the Grid: an Array Stream Transforming Language. *SNPD 2003*: 268-276.