Supporting microthread scheduling and synchronisation in CMPs

Dr. Ian Bell & Mr. Nabil Hasasneh

Department of Electronic Engineering
University of Hull,
Cottingham road,
Hull, HU6 7RX, UK
I.M.Bell, N.Hasasna@ hull.ac.uk

Prof. dr. Chris Jesshope

Institute for Informatics,
University of Amsterdam,
Kruislaan 403,
1098 SJ Amsterdam, NL
jesshope@science.uva.

Abstract

Chip multiprocessors hold great promise for achieving scalability in future systems. Microthreaded chip multiprocessors add a means of exploiting legacy code in such systems. Using this model, compilers generate parametric concurrency from sequential source code, which can be used to optimise a range of operational parameters such as power and performance over many orders of magnitude, given a scalable implementation. This paper shows scalability in performance, power and most importantly, in silicon implementation, the main contribution of this paper. The microthread model requires dynamic register allocation and a hardware scheduler, which must support hundreds of microthreads per processor. The scheduler must support thread creation, context switching and thread rescheduling on every machine cycle to fully support this model, which is a significant challenge. Scalable implementations of such support structures are given and the feasibility of large-scale CMPs is investigated by giving detailed area estimate of these structures.

Keywords: Microgrids, Microthreads, CMPs, Schedulers, Register files.

1. Motivation

Chip multi-processors (*CMPs*) are a very promising solution for future high-performance computing and we anticipate that many new microprocessor designs will be based on such an approach. Several projects have already investigated CMPs, such as the Compaq Piranha[1], Stanford Hydra[2] and Hammond et. al.[3], and manufacturers are beginning to produce commercial designs, such as the IBM Power PC[4], Sun Niagara[5] and Intel Montecito[6].

The appeal of CMP architectures comes from factors that limit the scalability of multiple-instruction issue in conventional processors[7], such as the superscalar paradigm, which continue to use more silicon and power for very little improvement in instruction-level parallelism (ILP). Evidence of this is provided by Intel's cancellation of its 4GHz Pentium4[8], which has effectively reached a limit in both performance and scalability. Scaling up concurrency in these processors gives very large circuit structures and this is exacerbated by lengthy global communication arising from the increasing problems of wire delay in technology scaling. These require excessive chip area and increased power consumption respectively. For example, the logic required for out-of-order issue does not scale with issue width[9] and will eventually dominate the chip area and power consumption.

The Semiconductor Industry Association (*SIA*) roadmap indicates that by 2018 the number of transistors on a single chip will reach 4 to 25 billion depending on the circuit type[10]. How to gain performance from this level of integration within acceptable power budgets is a major problem. Performance cannot be achieved by simply increasing the speed of conventional processors or by squandering a large number of transistors on unscalable support structures, as used in out-of-order issue. Instead, we have to embrace explicit parallelism, but systematic solutions to parallel programming and parallel architectures have yet to emerge, even with small-scale concurrency. In the near future, we will be able to integrate thousands of arithmetic units on a single chip[11]. Note that a 32-bit integer ALU occupies less than 0.05mm^2 in an 180nm CMOS process technology and typical chip sizes are between 100 to 400mm^2. However, before such chips can be utilised, we need programming paradigms for generating this level of concurrency and support structures for scheduling and managing this concurrency, which are fully scalable.

Today's small-scale CMPs are based on the same complex processor designs that preceded them and use high-level or software-based concurrency (e.g. threads). These threads may be scheduled in software or hardware and even used to extend the pool of instructions to support out-of-order issue. The latter, in particular, suffers from major problems, which limit performance and prevent overall-system scalability. These problems are summarised in[12] and systems based on this approach scale badly and are unable to exploit Moore's law to the full.

CMPs based on scheduling high-level threads, either in hardware or software, also fail to follow the exponential growth in circuit density but here the problems are in generating and managing the high levels of concurrency required. Ideally models should exploit the lowest levels of concurrency and synchronisation, such as instruction-level and loop-level parallelism (ILP, LLP), rather than the thread-level parallelism (TLP) typically exploited in the software-based approach. Synchronisation must also be implemented close to the processor, i.e. in the registers rather than in memory, if scalability is to be achieved.

In general, there are only a few requirements for the design of efficient and powerful general-purpose CMPs, these are: scalability of performance, area and power with issue width, and programmability from legacy sequential code. Issue width is defined here as the number of instructions issued on chip simultaneously, whether in a single processor or in multiple processors and no distinction is made here. To meet these requirements a number of problems must be solved, including the extraction of instruction-level parallelism (ILP) from legacy code, managing locality, minimising global communication, latency tolerance, power-efficient instruction execution strategies (i.e. avoiding speculation), effective power management, workload balancing, and finally, the decoupling of remote and local activity to allow for an asynchronous composition of synchronous processors. Most CMPs address only some of these issues as they attempt to reuse elements of existing processor designs, ignoring the fact that these are suitable only for chips with relatively few cores.

In this paper a CMP is evaluated, that is based on microthreading, which addresses either directly or indirectly, all of the above issues and, theoretically, provides the ability to scale systems to very large number of processors[13]. It will be shown that such CMPs use hardware scheduling and synchronisation and have structures to support this that are distributed, fully scalable and have locality in communication wherever possible. This is achieved with distributed schedulers that jointly manage large parametric families of threads and a distributed register file that provides synchronisation and sharing of data between them.

These structures provide support for a shared-register, instruction-level model of concurrency in which synchronisation occurs between instructions and in the registers.

The model requires instructions in the ISA to specify and manage this concurrency, but this is achieved by adding just a few additional instructions to a conventional ISA. The result, is that concurrency can be captured in an abstract and parameterized way in the binary code, rather than by calls to an operating system. This is a large advantage in exploiting efficient execution of concurrency in CMPs. This concurrency provides both speedup and latency tolerance in a single processor.

2. Related Work

As described in section 1 above, CMP architectures must overcome multiple problems if they are to deliver their full potential. In this section, some existing models of concurrency are evaluated against these problems. This evaluation is used as a comparison to position the microthreaded against related work before it is introduced in detail.

2.1 Scalability and Performance Improvement

To keep multiple execution units as busy as possible in the presence of significant latency in obtaining operands, modern processors use an aggressive out-of-order (OOO) instruction-execution. This allows instructions to be issued and completed out of the original program sequence, thereby exposing concurrency in the legacy, sequential instruction stream. OOO execution increases the performance of a superscalar processor by reducing the number of stall cycles in the pipeline. Synchronisation is managed by the instruction-issue logic, which keeps track of resources required by an instruction and any dependencies on the results of other instructions, which may not yet have been scheduled or completed. The instruction window maintains the set of decoded instructions on the currently predicted execution path that have not yet been issued. Its logic triggers those instructions for execution but requires an area that is quadratic in issue width, i.e. the number of instructions that can be issued simultansously[14]. Other support for renaming registers and retiring instructions adds to this cost. The key problem is that the mechanism for synchronisation is centralised.

Monolithic processors (i.e. wide-issue from a single instruction stream) have other structures that do not scale, these are the register file[15] and bypass logic[16], which are also centralised.

Finally, the concurrency exposed in OOO is limited due to the inefficient use of the instruction window. In practice its size is limited by scalability constraints but its use is required for all instructions, independent of whether those instructions can be statically scheduled or not.

Simultaneous Multi-threading (*SMT*)[17] is an attempt to make more efficient use of OOO scheduling by fetching instructions into the instruction window from a number of independent threads, thus guaranteeing fewer dependencies between the instructions found there and hence allowing more efficient use of the wide instruction issue. However, it does not address any of the issues outlined above and suffers from the same scalability problems as conventional OOO processor, i.e. layout blocks and circuit delays grow faster than linearly with issue width, and synchronizing memory is used inefficiently. Indeed it introduces other problems, such as multiple threads that share the same level-1 I-cache, which can cause high cache miss rates, all of which limit the ultimate performance[18].

The latency across a memory hierarchy may require hundred of cycles, which can significantly impact performance. The only way to avoid an impact on a processor's performance is to provide instruction-level concurrency, in addition to wide instruction issue, to provide tolerance to this latency. That, by definition, requires hundreds of independent instructions per processor. With CMPs comprising thousands of processors, this means providing synchronising memories capable of supporting hundreds of thousands of synchronizations will be required in future CMPs and they must be designed with this in mind.

An alternative approach to on-chip concurrency is to exploit user-level threads rather than dynamically extracting concurrency from legacy binary code. Sun has proposed a commercial, 32-way threaded version of the SPARC architecture in its Niagara device. The chip has eight cores, each able to handle the contexts of four threads. Each core has it own L1 cache and all cores share a 3MB L2 cache. Key problems with the Niagara approach are the significant resource consumption for the aggressive speculative techniques used, and the significant time wasted waiting for off-chip misses to complete, see[19]. Also, the basic implementation of this SPARC chip is a superscalar processor and, as already described, the superscalar approach provides diminishing returns in performance for increasing issue width. The performance of a 6-issue OOO processor will achieve only 1.2 to 2.3 IPC, compared with 0.6 to 1.4 IPC in a2-issue processor[20].

It should be noted that Niagara is better suited to server rather than general-purpose workloads, as a profusion of high-level threads are available in server applications, e.g. where a server's users are each managed by a concurrent thread. However, for general-purpose workloads, typical programs are not so heavily threaded and unless an automatic means of generating them can be found, this will severely limit the software-thread approach.

2.2 Concurrency and Programmability

Exposure and management of concurrency then are the key issues in supporting CMP design and implementation. This is the case for distributed systems as well chip-level systems, but in the latter situation, the constraints and opportunities dictate a different approach that is able to minimise the overheads of managing that concurrency. Concurrency has the ability to increase overall system performance as well as provide power savings in obtaining a given performance, by scaling frequency and voltage.

The use of OOO to expose and manage concurrency is ideal in one respect and one respect only. That is the ability to obtain concurrency from legacy code, without the programmer having to be aware of it. This has great commercial appeal. However, the model has no tacit knowledge of concurrency and synchronisation and this must be extracted dynamically in hardware, using complex support structures, not all of which scale with issue width. This is wasteful of chip resources, does not have predictable performance and is not able to conserve power in the execution of instructions. If concurrency were explicitly described in the instruction stream, some of these unscalable structures could be avoided.

User-level concurrency based on threaded applications is one alternative solution exemplified by the Niagara described above, but not all codes contain thread-level concurrency and therefore tools are required to extract threads from sequential programs. One example of such tools is the use of speculative, pre-execution threads to provide latency tolerance in memory access. This can be performed statically by a compiler, dynamically in the hardware, or indeed by a hybrid of the two[21]. However, as its name suggests, the model is speculative, which can again result in unpredictable performance and, like all speculative methods, is not conservative in its use of energy, e.g. when the speculation fails.

An alternative approach to extracting threads from user-level sequential code is described in[22], which compiles legacy applications for a multithreaded architecture. The most important goal of this work is to create a sufficiently large number of threads so that there is sufficient parallelism to hide communication latency. A second goal is to create threads of a sufficient granularity so that the context switching cost is relatively small compared with the cost of the actual computation. These goals are contradictory but can be achieved by distributing remote data dependencies between different threads and using these dependencies to schedule the thread when data dependencies are resolved, i.e. by using non-blocking threads. The approach described here, microthreading, has extremely efficient context switching and consequently does not require threads to be non-blocking.

Most approaches to extracting concurrency use the well-known fact that most computation is performed in loops and that loop iterations can often be performed concurrently, LLP. Compilation can extract software concurrency, as[22], or provide instruction-level concurrency as in the case of microthreading, which has an ISA extension for the compiler to target; this instruction dynamically creates a whole families of threads. Alternatively, in conjunction with control speculation, loops facilitate the concurrency exposed in OOO by using branch prediction.

However, not all loops are independent and concurrency is often limited by data dependencies, which may arise between different loop iterations when executed concurrently. The vector computers of the 1970s and 1980s were unable to deal with LLP that involved dependencies. OOO, on the other hand, manages these dependencies, which are often regular, just like any other irregular dependency. It has no contextual data to optimize and structure such management. There are other explicit approaches to manage loops containing data dependencies[2, 17] but in these, loop-carried dependencies are expressed as concurrently executing threads that share memory. This is bad as it induces high latencies in the dependency chains. In contrast, microthreads, synchronise in registers rather than in memory but this requires large register files as well as large support structures. This can only be achieved using distributed structures and in microthreading, unlike monolithic wide-issue approaches, synchronisation and scheduling are managed by distributed register files and schedulers even though the concurrency is specified and managed at the instruction level.

The requirements on these support structures are severe; they must support a context switch on every cycle, as the compiler identifies context switch points in the code and can flag any instruction to context switch. They must also support thread creation on every cycle, as thread creation occurs concurrently with instruction execution and must keep pace with the rate at which context switches can occur. Finally, they must support thread rescheduling at one thread per cycle, as when all threads are created, rescheduling must also keep up with context switch rates.

2.3 Scaling Processor Support Structures

In superscalar processors, the logic necessary to handle OOO instruction issue typically occupies 20-30% of the chip area[20] and the issue logic in processors that support speculation can be responsible for 46% of the total power[23]. On-chip caches are another critical challenge in modern processors, occupying large die areas, consuming significant power, and in some cases restricting system performance and scalability. Large cache bandwidth requirements and slow global wires will sharply diminish the effective performance of processors sharing a large monolithic cache as advances in fabrication processes effectively decrease global propagations times.

The alternative is to build thousands of processing elements on a die and surround each with a small amount of fast storage. Compare this to Intel's Montecito processor where cache memory occupies some 70% of the total die area or the equivalent to 32,000 32-bit integer ALUs (0.18um technology). Huh et. al. [24] address this issue by comparing the architectural trade-offs between in-order and OOO issue processors for serial applications. Their study demonstrated that if no L-2 cache area were required, then it is possible to integrate 556 2-way in-order processors on a single chip, or 201 4-way OOO processors with a maximum area of 400mm^2 in 35nm technology.

Clearly, the use of ever larger hierarchical memory systems does not serve scalability and does not guarantee better performance. Instead, as argued above, there is a requirement for large, fast and distributed synchronisation memory to support very wide instruction issue as well as latency tolerance. Ideally a deterministic distribution of instruction execution and data mapping is required in order to explicitly manage locality and to eliminate, as far as is possible, slow and power-hungry global communication. This goal is not served by using a large and monolithic processors connected to a large and monolithic on-chip menmory. In short, some form of distribution becomes essential and without a deterministic distribution of data and computation on chip, very wide-issue CMPs are just not feasible.

Recently, Rixner et. al.[11] analyzed register file area, delay and power dissipation for streaming applications. The analysis showed that for a central register file, area and power dissipation grows as N^3 and delay grows as $N^{3/2}$. Examples of the effects of this scaling can be found in the proposed Alpha 8-way issue 21464, which used a 512 location register file requiring 24 ports to serve the wide-issue processor. Even with a clustering, the 4Kbytes of register file occupied an area some 5 times larger than that used by the L1 D-cache[25] (64KB plus tags). That is a per-bit, density ratio of 100:1 and graohically illustrates Rixner's results. Power is also an issue in such large structures and in Motorola's M.CORE architecture, the register file energy consumption is 16% of the total processor power and 42% of the data path power[26]. These examples, which support only modestly-wide instruction issue confirm that multi-ported register files in modern microprocessors are not the way to proceed in future CMPs.

ILP processors communicate and synchronise using a namespace interpreted at the instruction level, i.e. the register specifiers. This is typically limited to 5 or 6 bits and the question that must be asked is how can the large a and distributed synchronization memory be addressed with such small addresses? OOO processors use register renaming for subsequent uses of the same register specifier and thus expand the namespace dynamically. (This also removes the artificial dependencies introduced by executing instructions out of program sequence). Of course, additional hardware is now required to perform this mapping and to re-establish the mapping back to the original binary code to give the illusion of sequentially executed instructions.

Microthreading on the other hand executes loops as concurrent code fragments and in order to share code for a loop body, each iteration must have its own registers, which are unique. In contrast to renaming, this is achieved by addressing a register file relative to some unique offset, so that the same instruction will access a different location in the register file for different iterations. Those offsets are a part of the microthread's state. This mechanism extends the ISA's namespace so that it is limited only by the parametric concurrency expressed in the creation of the microthreads that execute the loop.

The disadvantage of this approach is that registers must now be allocated dynamically and state, in addition to its PC, must be maintained for each microthread. To allocate registers dynamically requires additional logic and with many concurrent threads, any additional thread state can lead to significant storage in the scheduler. The contribution of this paper is in the design and analysis of these support structures.

2.4 Power Dissipation

Another challenge in modern processors is power consumption and heat dissipation, which is already a serious problem and can only become worse in future [27]. For example, Intel's Madison consumes up to 130W, the alpha 21364 EV-7 consumes 155W and the International Technology Roadmap for Semiconductors expects that power consumption in processors will reach close to 300W by 2015[10]. This 300W does not follow the past exponential growth in power dissipated and recognizes this as a major conatraint on processor design. This problem is exacerbated as in future process technologies, the leakage power will also become a significant percentage of the overall power dissipated[28].

Several papers have considered power reduction in CMPs[27, 28] but these techniques can not hope to find significant principle solutions as branch prediction and out-of-order issue do not provide a significant performance improvement relative to the area and power consumed and doe not execute instructions conservatively with respect to power dissipation. Indeed, the only solution is to remove these features to save power[29]. Another current trend that highlights this problem is the current practice of increasing the number of pipeline stages in order to reduce the clock period and hence increase performance. This also can not continue, as it is simply not feasible to continue to extract exponentially growing amounts energy from a chip as heat, as the result of power dissipated. Indeed, there is a case for the trend to higher and higher clock frequencies to be stopped or even reversed and instead to use of concurrency as a means of providing performance improvements without excessive power consumption.

Concurrency can also provide power reduction for a given performance. With scalable processor, two acting concurrently should give the same overall performance as one at double the speed. The scalability required is performance with issue width, logic or area with issue width and power dissipated with instructions issued per cycle (IPC). Although the above comparison breaks even in power dissipated, power can be saved by scaling supply voltage with frequency. As power dissipated is proportional to V_{DD}^2, this gives a quadratic reduction in power for a given performance, over the linear portion of frequency-voltage scaling.

The use of IPC rather than issue width as a base for power scaling assumes that when a processor is inactive it can be powered down. As a result, speculation or eager execution must be avoided, as by definition an eager processor can never determine when there is nothing to do!

Microthreading uses simple in-order instruction issue without branch prediction and has explicit control of instruction scheduling, it can therefore provide all the hooks required to support conservative instruction issue and hence take advantage of this power scaling[13]. Processors with no active threads are aware that instructions cannot be scheduled and can therefore go into standby mode dissipating minimal power. This power usage can be scaled with IPC rather than issue width.

This conservative scheduling also provides an insight into asynchronous partitioning of a CMP. By definition, if a processor has all of its threads inactive, then any event triggering further computation must either be external to the processor (asynchronous) or the processor must be deadlocked. A microthreaded CMP can therefore use a local clocking with asynchronous communication between processors, further reducing power requirements. This fact, together with the processor's inherent latency tolerance provides all the hooks required to implement a globally-asynchronous, locally-synchronous (GALS) implementation. Additional power savings come from not requiring such powerful drivers in distributing the clocks to the entire chip.

3. Microthreads, Microcontexts and Microgrids

3.1 Introduction

First it is necessary to define the terms used in this section heading:

i. *Microthread* – (not hyphenated to distinguish it from other uses of the same combination of terms), refers specifically to code fragments managed by the model described in this and previous, related papers. Microthreads are small sequences of code (as short as a single, executable instruction) that are created dynamically and execute concurrently. Creation is by an instruction added to the ISA for that purpose. A family of microthreads can be distributed to more than one processor and both the number of processors used and the number of microthreads created is parametric and not bound by the resources available on those processors. The *create* instruction specifies a family of related microthreads, which are created as resources become available and at the same time as previously created microthreads are being executed. All microthreads follow an execution path which ends in the execution of an instruction which terminates that thread, at which point its state is lost and its resources are released.

ii. *Microcontext* – refers to the private state associated with a microthread. This includes a microthread's program counter and an offset into the register file, which locates its private register variables. The contents and synchronisation state of the registers is also a part of its microcontext. The microcontext is stored in two structures, the local register file and the local scheduler of the processor on which the microthread is executing. Using a 5-bit register specifier, this state is bounded above by 32 register variables and one slot in the scheduler's tables. The microthread and its microcontext are identified uniquely by its address in the scheduler's tables and this is called its slot number. It should be noted that a family of microthreads will share all memory variables in the scope of a given higher-level context and may also share a number of register variables.

iii. *Microgrid* – refers to a CMP where all processors have a microthreaded scheduler and a synchronising, distributed shared register file. A microgrid will have a inter-processor network to support the sharing of microcontexts between microthreads in a family of microthreads, mapped to different processors. The network also supports the broadcast of shared-register variables and the parameters defining the creation of a families of microthreads. A microgrid may also have a systems environment processor that manages the allocation of processors to families of threads dynamically and configures the network accordingly. Microgrids are described further in[13].

Microthreaded code is not backward compatible. It must be recompiled from the original source code, although this can be legacy, sequential source code. The parallelisation of the source is primarily, but not exclusively, obtained by translating loops into families of microthreads that execute concurrently. Techniques have been used to parallelise both *for* and *while* loops, as well as loops with and without loop-carried dependencies. The type of dependency supported is a function of the detailed implementation of the processor and network. In this paper, we consider only constant-strided dependencies, which are supported by a simple ring network.

Figure 1 gives an overview of such a microgrid, showing the networks required and the datapaths between the major components within a processor. These are an in-order pipeline, a scheduler, a large register file and a local I-cache. The processor may also have a local D-cache but latency tolerant access to data means this is not a necessity. In a *profile* of processors, a subset of the microgrid, any processor can create a family of microthreads for execution on that subset. This requires the distribution to each processor of the address of a data block in memory. This is called the thread-control block (*TCB*), which contains all of the parameters that define the family microthreads.

This is the only global communication required in the execution of a family of microthreads, apart from those defined by memory accesses in the code. Each processor receiving the address of the TCB will execute a deterministic subset of that family, based on the parameters in the TCB, the number of processors in the profile and its position in the profile.

Microthreads created on a processor are queued in its scheduler for execution and a microthread is removed from this queue and passed to the instruction fetch stage of the pipeline on a context switch. The active microthread will continue to execute until either it completes or is context switched itself, because of a blocking read to a register. This may occur on instructions dependent on memory loads or data produced by other microthreads. These are recognized by the compiler and flagged as context-switch points. These instructions may or may not suspend on reading their operands and the explicit context switch merely enables the scheduler to eliminate bubbles in the pipeline in the event that the instruction does block. In this case, the slot number of the suspended thread is stored in the empty register until the data arrives, at which point that thread is rescheduled and added to the scheduler's active queue again. Swapping execution between threads when data is unavailable keeps the processor's utilisation high and hides communication or memory-access latency.

During execution, any data exchange between concurrent microthreads is achieved using register variables. Concurrent microthreads may not communicate using shared memory, as no guarantee can be made about their order of execution. Memory consistency is achieved therefore using bulk synchronisation, either using knowledge of the termination of a dependency chain in a dependent family of microthreads or by the use of a barrier synchronisation in an independent family of microthreads.

3.2 The Microthreaded ISA

Microthreading is a general model that can be applied to an arbitrary ISA with the addition of a small number of instructions to provide the concurrency controls. These instructions are shown in table 1. The model provides concurrency-control instructions to create families of threads (*Cre*), to explicitly context switch between threads (*Swch*) and to kill a thread (*Kill*). Two global synchronization instructions are also provided, one is a barrier synchronisation (*Bsync*), the other is a form of a break instruction (*Brk*), which forces a break from a loop executed concurrently. Note that all of these instructions can be completed in the first stage of a pipelined as they only control the action of the scheduler. Because of this, these additional instructions do not require a pipeline cycle so long as they are fetched concurrently with executable instructions. This allows concurrency controls in the model to be very efficiently implemented.

As already described, microthreading exploits LLP by executing the same loop body for multiple instances of an index variable. It is also able to capture ILP within basic blocks. LLP is specified parametrically using loop bounds, with multiple iterations sharing the same code but using different micro-contexts; this is SPMD concurrency. MIMD concurrency can also be specified using pointers to multiple code blocks but is static in extent, as the compiler must make the partition of the basic block and generate code fragments accordingly. Both are captured through the control instruction *Cre*, which initiates the creation of threads on all processors defined in a given profile. Each scheduler will continue to create threads, until its distribution of iterations has been exhausted. It may then continue to create threads from other families, whose Cre instructions may have been queued in the scheduler.

The process of thread creation requires the following actions (refer to figure 3):

a. A slot number is obtained to address the scheduler's tables (called continuation queue – CQ, in figure 3) from the scheduler's empty queue and the empty queue is updated.

b. The register allocation unit (RAU) reserves the required number of registers for the microthread's context and returns a base address in the register file.

c. The code pointer, the base address of the microcontext, and the base address and slot number of the microcontext it is dependent upon are all stored in the CQ slot.

d. Finally, the index value associated with the microthread is written into the first local register variable of its microcontext and all other variables are initialised to empty.

e. The slot number is then passed to the I-cache to prefetch the first instruction, only after it has been prefetched, will the slot number be added to the active queue of the scheduler, where it is available for execution.

The RAU maintains the allocation state of all registers in the local register file. If no registers are available or there are no empty CQ slots, the thread cannot be created until some resources have been released. It will be shown in this paper, that the local scheduler's tables and the RAU both scale with the number of local registers and are both similar in area to the register file, which gives an scalable with concurrency, be it local concurrency used for tolerating latency or concurrency of instruction issue.

3.3 In-order Pipeline

Figure 2 shows a microthreaded, in-order pipeline with its five stages and the communication interfaces required to implement this model in a distributed manner. The pipeline stages are: thread control/instruction fetch, instruction decode/register read/reschedule, execute, memory (if implemented) and write back. Notice that instructions normally complete in order but that in circumstances where the execution time is non-deterministic, such as a cache miss, data is written asynchronously to the register file on a port dedicated to this purpose. In this situation, instruction issue stops in a thread as soon as an instruction attempts to read a register, that is empty. Note that all registers have synchronisation bits associated with them defining their state: {*full, empty, waiting local, waiting remote*}. No additional pipeline stages are required for instruction issue, retiring instructions, or for routing data between different processors' register files. Short pipelines provide low latency for global operations but a short pipeline can be super-pipelined if required, to increase clock frequency (but note the comments in section 2 above).

Context switching is determined explicitly by the *Swch* instruction, which can follow/precede any executable instruction and causes control to be transferred to another microthread on the fetching of that instruction. Whether it follows or precedes the instruction it flags is an implementation detail. In this paper we assume it follows at no loss of generality. In this case a Swch instruction is fetched concurrently with an executing instruction and causes a context switch in the same cycle.

As well as managing data dependencies, context switching is also used to manage control dependencies in the pipeline, as all transfers of control are also flagged to context switch and only rescheduled when the execution path has been determined. A context switch is also used as a pre-fetching mechanism in the instruction cache. A context switch is forced when the PC increments over a cache-line boundary. This makes a potential cache miss to be a part of the scheduling process rather then the instruction-execution process. Indeed, it provides a unified mechanism for cache pre-fetching as any thread will not be scheduled for execution unless its current PC is guaranteed to be in the I-cache.

Contexts switches or Kills must be planted by the compiler on all branches of control and on instruction that might stall on reading data. The latter occurs when communicating with other threads, or following a load or long operation. Note that a Swch instruction will always update the value of the PC in the thread’s state, and this update occurs after the register-read stage. This is obvious in the case of a branch but not so obvious following a data dependency, where the state of the register will determine whether the instruction will be re-executed or not. If a register reads fails, the instruction reading the register must be re-issued, when the data is available. On the other hand, if the register read succeeds, the next instruction must be executed, which may be the next executable instruction or the one at the branch target location, thus the action at the register read stage determines the value of the thread’s PC for all programmed context switches.

Each register in a microgrid therefore acts as a synchroniser, which can control the issue of instructions from the thread or threads that access it. A reference to the thread’s slot number is stored in the register on a synchronisation failure and that thread is rescheduled only when data is written to the register. This mechanism is distributed and scalable, requiring only two additional bits per register together with state machines on each of the registers file’s ports. This is in stark contrast to an OOO processor’s instruction window.

The mechanism of thread suspension and activation provides latency hiding during long or non-deterministic delays when obtaining data. The maximum latency tolerated is related to the size of the CQ or the size of the register file, which can both restrict the number of local threads active at any time. Of course, the latency is also related to the average number of statically scheduled instructions between context switches. Only if a processor has no active threads, will the pipeline stall on attempting to read an empty register.

This means of scheduling instructions is similar in complexity to that of a conventional, single-issue, in-order processor. The only additional overhead is the larger than normal register file, the maintenance of the CQ and the RAU, which are investigated in detail in this paper. However, as they are both scalable with local concurrency they can both be tuned in size at design time in order to provide a given amount of latency tolerance.

3.4. The Scheduler

A global scheduling algorithm determines the order in which a group of related threads is distributed to the processor profile. This algorithm is built into the local schedulers and is parameterized by data from the TCB as well as the number of processors in the profile. Within each processor, the local scheduler manages the execution of all microthreads currently allocated to that processor. The schedulers in different processors are independent and each manages a local model of their own resource utilisation for the subsets of the families of threads that it must execute.

Figure 3 shows more detail of a local scheduler, its connections with the I-cache and the processor pipeline. The register allocation unit (RAU) within each scheduler models the allocation of micro-contexts to the local register file and determines when new microthreads may be allocated. If registers are available it will provide the base address of a micro-context, which is used by the create process to create an entry in the CQ. Similarly, when the threads associated with a micro-context have been killed, its registers will be relinquished and the RAU will update its allocation model. In this way, the scheduler can manage concurrency that is parametric and which exceeds the statically available resources.

The process of dynamically allocating micro-contexts and thread creation is fully decoupled from the pipeline execution, allowing the pipeline to work at full utilisation without any extra pipeline stages for allocation. Furthermore, the distributed organisation of both the global schedule and register allocation provides a scalable solution to concurrent instruction issue and keeps both control and communication signals local, with the exception of the broadcast of the TCB pointer to each processor, which is the only global overhead in this model.

Thread distribution

One algorithm that can be used to distribute an iteration space to the array of processors is to use the following equation, where iteration i, is mapped to processor q, using a profile of P processors and a block of b consecutive iterations allocated to each processor:

$$q = |i/b|_{\mathrm{mod}\ P}$$

In this schedule, b can be chosen to minimise inter-processor communication and ensure that regular inter-micro-context communication can be mapped to a point-to-point network, more specifically a ring network.

The process of thread creation and code generation will be illustrated using the following dependent loop:

```
for (i=0;i<n;i++)
      Q = Q + A[i]*B[i]
```

The loop has a dependency in the add operation between Q in the current iteration and Q' from the previous iteration. The compiler generates code to carry this dependency between iterations using a register shared between two microthreads. This is specified by a dependency distance of 1 in the TCB, which is used to link dependent threads in the scheduler. The registers in a microcontext are divided into a local part $Li, a shared part, $Si, and a dependent part, $Di, where the shared part of one micro-context maps to the dependent part of the iteration that is dependent upon it. Thus the assembly code below uses $S0/$D0 to carry this dependency between iterations, where $S0 is written by the producer thread and $D0 is read by by the consumer thread. The dependency chain is initialised and terminated in the main thread. In the assembly code, three parts can be identified. The first is the TCB; the second is the code for the main thread, which creates and synchronises this family; and the third part is the code for the loop body. Note that n iterations of this body execute concurrently between the *mv* and *sw* instructions in the main thread.

```
           Thread Control Block
           .data
loop:      .word 1              #threads per iteration
           .word 1              #dependency distance
           .word 1              #loop start
           .word n              #loop limit
           .word 1              #loop step
           .word 2              #number of local registers
           .word 1              #number of shared registers
           .word 0              #number of global variables
           .word body           #pointer to code fragment

            Code for main thread
main:       cre loop            #create family of threads
            mv $G0 $S0          #initialise dependency chain
            sw $D0 Q            #store result and synchronise

            Code For loop body
body:       lw  $L1,A($L0)      #load A[i] from memory
            lw  $L2,B($L0)      #load B[i] from memory
            mul $L1,$L1,$L2     #A[i]*B[i]
            swch 2,0            #context switch
            add $S0,$D0,L1      #Q':=Q + A[i]*B[i]
            kill 1,1            #terminate thread
```

Looking at the concurrency in this code, it can be seen that all loads and multiplications can proceed concurrently but that the accumulation of Q in $S0/$D0 is constrained to execute in sequence and may be mapped to different processors. During the execution of this dependency chain, only one processor will be active while the result is accumulated. This constraint will limit speedup, but during the execution of the dependency chain, only the processor currently executing will be active and consuming power as all other processors will recognize an empty active queue. This situation is easily detected and can be used for power management. When executing multiple iterations on one processor, the chain can be executed at one addition operation per cycle using the bypass network as a mechanism has been developed to reschedule threads in dependency chains predictively, i.e. where we know one thread must reschedule the next, e.g. on the *add* instruction for all threads. Information to detect this situation is given by the compiler in parameters to the switch and kill instructions. Those parameters signal the number of non-deterministic operands and whether to predictively reschedule the next interation in a dependency chain.

3.5 Sharing Registers

Several sub-models of microthreading can be defined that differentiate the manner in which micro-contexts are shared between microthreads, the work in this paper considers a model that supports data sharing between micro-contexts defined by a constant-strided loop-carried dependency, which is specified at create time via the TCB. This model covers a wide range of loops and is broader by far than the vector model, however not so broad as OOO. The advantage of this model is in the static definition of the relationship between created microthreads, which allows for very efficient register sharing over ring networks. The implementation requires the micro-context to be partitioned into different windows representing different types of communication and for the processor to recognise these windows in order to initiate a shared-register transfer when necessary.

A micro-context in a conventional RISC ISA comprises a maximum of 32 registers and is divided into four classes of variables, namely G *globals*, S *shareds*, S *dependents* and L *locals*. This partitioning is specified by including the parameters G, S and L in the TCB. The G globals are mapped to a subset of the locals in the thread issuing the Cre instruction and those values are replicated to all processors used in executing that family and are shared by all threads in that family on a given processor.

The local class is private to a thread, unless it creates a subordinate family of microthreads, when these may be shared as globals to the subordinate threads. The shared and dependent classes provide for the communication between threads to support loop-carried dependencies. The dependency chain originates in the main or creating thread, passes through all iteration threads and terminates in the main thread again. In this chain, the shared class of the producer thread is mapped to the dependent class of the consumer thread. In the main thread, both shared and dependent classes are mapped over its local registers. In created threads, when the producer of a dependent class executes on the same processor as the consumer, only shared and local classes need to be allocated by the RAU. Otherwise, when the producer of the dependency is executing on another processor, dependent, shared and local registers must all be allocated. In the latter case, the dependent class acts as a local cache for the remote shared class but otherwise shared and dependent classes from producer and consumer both map onto the same S locations in that processor's register file. This is illustrated in Figure 4, for a family of four threads mapped to two processors.

It is important to note that remote register reads are completely decoupled from the pipeline's operation through the use of explicit context switching. The consumer thread reading a remote value suspends locally in one of its $D locations, and then sends a request to the corresponding $S location on the remote processor. This remote request may also suspend in the consumer's micro-context if the data is not yet available but eventually will return a value which will reschedule the suspended thread. Both types of continuation are identified in the register's synchronization state.

4. Implementation and Area Estimates for Support Structures

4.1 Register File

The register file is partitioned and distributed across multiple processors and is therefore scalable if it can be implemented on each processor with a fixed number of ports. This has already been demonstrated in a prior publication[31], which shows that only five ports per processor are required in an implementation to obtain full pipeline utilisation. Here we estimate the area required for the local register file using procedures from prior work[33, 34]. We calculate the area and compare this with that of the Alpha 21264 for a given technoplogy. Figure 5 shows the estimated area for a local register file for different numbers of registers (note that this size determines the amount of latency tolerance). It can be seen that the area of 1024 32-bit registers is less than 0.6mm^2 in 0.07 micron technology.

The Alpha 21264 splits its integer file into two that contain duplicates of the 80-entry register file. Two pipes and a single register file form a cluster, and the two clusters are combined to support 4-way integer instruction execution. The architecture also has two floating-point execution pipes organised in a single cluster with a single 72-entry register file. Figure 6 compares the area of a microthreaded CMP register file and the Alpha register file. The area of the microthreaded register file is less than the area of alpha 21264 for the same technology for all sizes up to 512 64-bit registers. Note that the 21264 only provides 152 registers to the microarchitecture.

4.2 Register Allocation Unit

In this section, a mechanism is described for implementing dynamic register allocation. This hardware block uses information provided by the compiler through the TCB to define the allocation requirements and uses a set of 1-bit flags to model the allocation state of the registers. Allocation can be made to any contiguous set of free registers that meet the requirements of the microcontext. In an allocation cycle, the RAU provides the base address of that microcontext and the allocated registers are marked as allocated. Initially, only the base-architecture's registers are allocated, i.e. registers 0-31 on the processor running the main thread, all others are free. The architectural registers provide backwards compatibility for non-microthreaded code on a single processor.

It is important to perform the allocation process in a minimum number of cycles. Our allocation scheme allocates one micro-context per cycle, which is the fastest rate necessary to support thread creation at a rate of one per cycle. Design tradeoffs can be made to allocate registers in blocks of from 1 to 32 registers, which provides a tradeoff between speed and area of the RAU and occupancy of the register file. However, for a given block size, the compiler can still optimize register file occupancy in code generation using a number of techniques, such as loop unrolling.

The design for the RAU has been implemented in VHDL. It uses an area proportional to the product of number of allocation bocks in a given register file and the number of bits in the register specifier. For a given ISA, or number of bits in the register specifier, the allocation has a constant (worse-case) time delay. An analysis of 10 Livermore loop kernels, including both independent and dependent loops, gave an average number of registers required per micro-context of 6, and a minimum number of 2 (one for the loop index and one other). Thus, even in the smallest microcontext, area can be saved by allocating in blocks of greater than one.

Figure 7 shows the Top-level design of the RAU and its interaction with the thread-create process and hence, the rest of the scheduler. It shows that the RAU comprises an iterative array of allocation slices, one slice per register-allocation block. Information on the action required (no op., allocate or release), the required block size (for allocation), and the base address (for release) is supplied to each slice from the scheduler. Each slice maintains a flag, which indicates whether the corresponding block of the register file is allocated or free. Note that in cycles when no action (allocate or release) is being performed, the RAU still calculates the next base address ready for allocation, so that it is available before an allocation is actually required.

In figure 7 data ripples through the allocation slices from bottom to top, corresponding to increasing register-file address. The output from the final slice identifies the base address in the register file of the first free contiguous block that meets the current block size requirement for allocation (if one exists). The scheduler uses this to determine whether the current allocation round can proceed and to set the base address in the CQ for the thread associated with this micro-context.

The ripple inputs to the first slice are not hardwired, but held in a register to facilitate test and adaptability. The base address input to the first slice is set to the address of the first register that can be allocated in the register file, i.e. 32 in the processor running the main thread and 0 in all other processors.

Information propagated from slice to slice includes whether a free block has been found, the base address of largest free block, the size of the largest free block, the base address and size of the current free block. An error flag is also propagated which indicates if inappropriate inputs have been applied to the RAU. The data manipulated and propagated between slices is listed in full in table 2 and illustrated in figure 8. As already explained, the number of registers per micro-context is between 1 and 32 and so it is possible to limit the size field to 5 bits, which can significantly reduce the propagation time within a slice and hence the time to perform the allocation update process. Note that each slice performs an increment on size and this will saturate at 32. The state of the allocator is held entirely in a set of flags, one per slice, which indicates if the associated n-register block is available for allocation or not.

The RAU always provides the next allocation, to be used in a creation cycle. The scheduler initiates the create process, which updates the allocation state and allows the RAU to settle into a new allocation state. This update process, may take longer than one cycle, depending on the size of the register file and allocation block size (n), which define the RAU's ripple-through time. The rate of performing allocations will, on average, be less than one per cycle, although it is desirable for the RAU to recover in a single cycle to manage periods of peak context switch rates during thread creation.

When threads are killed, the scheduler also causes the allocation model in the RAU to be updated to reflect this, by providing the base address of the micro-context being released and its size. This information propagates through the slices to determine which flags to reset.

The allocation scheme is relatively simple and allocating registers in blocks of n provides both area and propagation-delay reduction in this scheme. If we assume that the size of the register file is R, and the number of registers allocated in a unit of allocation is n, then the complexity of the allocation scheme is O(R/n)

The allocation scheme has been modeled in VHDL and the generate statement has been used to create the logic for the allocation scheme, with the registers per allocation unit being parameterised as a constant in the top-level of the model and passed to lower level components. We have also estimated the area of the allocation scheme and compared it with the area of the register file. Figure 9 shows an area comparison between the register allocation scheme and the register file for 2- and 4-register allocation units. The allocation scheme uses less area than the register file in both cases.

4.3 Scheduler

Within the local scheduler, the continuation queue (CQ) manages the state of all currently allocated threads; the components of this state are shown in table 3. This includes the program counter (PC), the base address of its micro-context (l-base) and the base address of a dependent micro-context if used (d-base), which includes a flag to specify whether this is local or on an adjacent processor. Two additional field are used to hold pointers to other slots in the table. The first of these is used to build queues, for example, the empty-slot queue and the active-thread queue. The other is used to identify a thread's producer in the dependency chain. This is required in releasing a thread's resources, as in a dependent loop, physical registers are shared between two different microcontexts and the producer's registers can not be released until the consumer has read them. This is implemented conservatively by releasing registers only when the consumer has been terminated. Thus the Kill instruction must backtrack one place down the dependency chain to release that thread's resources. The table is initialised into a state where all slots are in the empty queue except for the main thread, if it exists on a processor, which occupies slot 0. For a 32-bit PC, a 512-location register file and a 256-entry CQ, each entry in the continuation queue requires 67 bits.

The structure of the continuation queue can be decomposed into three different components, each of which has a different number of read and write ports :

i. The first part holds the PC (32 bits) and is written on two ports, one when a thread is created and the other when a thread is rescheduled. Both may occur at a high frequency, so two ports are assumed so as to be able to reschedule and create in the same cycle. There are also two read ports, one to access the head of the active queue to provide a PC on a context switch and a second to obtain the PC of a suspended thread when it is rescheduled after suspension in a register. This must be sent to the I-cache to pre-fetch its code before that thread can be placed in the active queue again. Again both can occur frequently and two ports are assumed to perform both in the same cycle;

ii. The second part (27 bits) holds the micro-context state (base addresses and producer) and requires two ports, one of which is written to when the thread is created and the other is used to access the head of the active queue on a context switch or kill;

iii. The last part is used to organise the thread slots into various queues and this is the link field (8 bits). This is accessed in each cycle to maintain various mutually exclusive queues, which are linked using this field. They are discussed in more detail below.

All queues are maintained using the link field to indicate the *next* slot number in the structure and two registers are used to maintain pointers to the *head* and *tail* of the corresponding queue. This is illustrated in figure 10. The head, tail and link fields are used to address all memories defined above.

Figure 10 shows the four processes involved in managing the thread state, i.e. thread creation, pre-fetching, building continuation queues on registers and context switching. These are as described below in more detail:

- *Create* - when a thread is created, a read port is used to update the head of the *empty queue*. The slot number from the old head is the one allocated to the thread and this is passed to the cache along with the thread's PC to initiate a prefetch.

- *Prefetch* - when the PC address is known to hit the cache, the new thread is added to the tail of the *active queue*, which supplies new threads to the pipeline on a context switch.

- *Context switch/kill* - when a context switch or kill occurs at the IF stage of the pipeline, a read is required to update the head of the active queue. Also, but only on a kill, a write is required to update the tail of the empty queue. This requires two ports, as the read and write are to different addresses in the CQ.

- *Rescheduling* - this process is required to manage the *continuation queues* of multiple threads suspended on a given register (Ri in the diagram). This will either write to the link field to update the tail of the Ri queue, when a new thread is added or read the link field to update the head off the Ri queue, when rescheduling a thread from it. This requires a single read/write port.

In total therefore, this link part of the CQ requires the most ports and the sum of the above gives 5.

The size of the CQ is related to the size of the register file through two parameters. The first is the number of registers required per micro-context (Rmc) and the second is the number of threads that share a micro-context (Tmc), i.e. though global registers. The more registers per micro-context the smaller the CQ in comparison to the register file. The more threads that share a micro-context, the larger the CQ in comparison to the register file. We have already shown that over a sample of the Livermore loop kernels, the average number of registers per micro-context was 6. The number of threads sharing some or all of a context is more difficult to ascertain. For this reason, figure 11 shows an area comparison between the continuation queue and the register file for 1, 2, and 4 registers per slot in the continuation (i.e. Rmc/Tmc = 1, 2 or 4) all of which are very conservative (i.e. require large continuation queues.)

5. Chip Architecture and Estimated Core Area

In order to demonstrate the feasibility of the microgrid CMP, this section provides an overview of the chip architecture, gives an estimated area of the microthreaded processor core and estimates the number of processors feasible on a die in emerging technologies.

5.1 COMA vs Multibanking

The Microgrid CMP is capable of supporting a large number of processors on-chip, but such a design requires a similar number of memory banks to satisfy parallel access. The ratio of memory banks to processors required is dependent on the cache hit rate and the access pattern to these banks. Two possible memory organisations are being considered. The first uses a processor with an L1 D-cache per processor, supported by a cache-only memory architecture (*COMA*). In such a memory, data is automatically migrated or replicated to where it is being used by the processors. The second memory architecture eliminates the L1 D-cache completely and makes use of latency tolerance to access a flat multi-banked memory structure. Simulations[31] have shown that such an organisation is entirely feasible. These previously published simulation results are reproduced in figure 12. This shows a number of parameters for the execution of the same binary code across profiles of from 1 to 2048 processors. Scalability is clearly illustrated. The other issue is that performance results (except for cache hit rate) is completely independent of the cache implemented in the processor. The top graph if for an 8-way associative 64Kbyte cache and the lower graph is for a direct-mapped 1Kbyte cache. The only difference can be seen in the number of inactive cycles, which changes marginally and the cache hit rate, which is drastically reduced. The change in cache does not affect performance and the IPCs are almost identical. In fact, by a small margin, the maximum IPC was observed with the smaller cache.

The advantage of the COMA structure is that it requires fewer memory banks, as each bank can have multiple, independent cache-line buffers for each processor in a cluster, see figure 13, where a cluster of processors share a single banked COMA node. Access to the COMA node is by cache line and access by the processor is by word. This allows a number of processors, equal to the number of words in a cache line, to share a port into the COMA node without conflict, so long as there is full cache locality. Note that the deterministic distribution of threads to processors in the micro-threaded model allows data accesses to be organised in such a way as to maximise the cache hit rate and minimise accesses to the COMA nodes. The simulations from [31] shown in figure 12, using the 64Kbyte cache produced an 80% cache hit rate with only 2-3% of memory loads causing a request to the COMA node. However, not all algorithms can be regularly mapped and some require global- rather than local-communication patterns. For example, matrix multiplication accesses data using both row and column strides through memory structures and such an algorithm would generate cache misses and bank conflicts on at least one of these strides, unless the algorithm was coded in a block structure, where the blocks matched the cache line size.

An alternate memory architecture that uses a word-wide memory bank per processor with no L1 D-cache in the pipeline is shown in figure 14. In this case, all memory accesses incur a delay, dependent on location of data on chip. However, the microthreaded processors can be designed to tolerate any latency by scaling register file size and support structures to give the required local concurrency. This memory structure would still suffer from bank conflicts in the example given above, unless some form of randomisation was employed in mapping the address space to the memory structures, see figure 14. The advantage of this scheme is that the complexity of the processor is reduced, by omitting the L1 D-cache. It also supports arbitrary access patterns to data, although this comes at a cost, as there is more load put on the on-chip network and more energy dissipated in moving data around the chip, as locality is ignored in randomizing memory accesses. These issues have yet to be explored in depth, using our simulators. It is clear that the choice of memory structure for the CMP is a complex one and is likely to be application specific. For this reason, in this section, we simply assume that half of the chip area is given over to processors and the remaining half to memory structures such as memory banks and the network to access them.

5.2 Estimated Core Area

The major advantage of the microgrid CMP is its scalability in terms of performance, power and area with instruction-issue width. The first two issues are demonstrated in[13] and in this paper we have analyzed the area scaling. We have shown that the area of the support structures for a microthreaded microgrid are scalable in instruction-issue width, as they are distributed to the processors, but we have also shown that the structures are scalable in the virtual concurrency supported on a local processor, which determines the amount of latency tolerance. Because of this, performance, power and latency tolerance can all be managed, the latter in the microgrid processor design and the former two in the dynamic management of concurrency in a microgrid.

In this final section an estimate is given of the number of microthreaded processors that can be integrated onto a single chip using emerging technology (0.07 micron CMOS). We assume that each core in the microgrid CMP is a 32-bit RISC processor with a dedicated, 64-bit, floating-point unit (*FPU*). We consider two possible architectures, which correspond to the memory organisations briefly described above, i.e. with and without D caches. To estimate the microgrid-core area, we have used CACTI to estimate the area of the L1 caches and we use[34, 35] to estimate the area of the other core components. Note that both I-cache and D-cache are single port memory structures.

Table 4 gives the estimated area of a microthreaded processor core including an L1 D-cache. In this table, we assume that the processor has a direct-mapped L1 I-cache of 8KB and a two-way set-associative L1 D-cache of 64KB. It can be seen that the L1 D-cache consumes about 47% of the core area and that the register file of 512 registers consumes 12%. Based on the work presented in this paper, we assume that the RAU allocates registers in units of 2 and that the size of the CQ is 256-entries. This gives support structures for the microthreaded model that consume 7% and 12% of the core area respectively, giving a total area for the processor core including the FPU of 2.43mm^2.

Results for the alternative configuration without the L1 D-cache use similar parameters, with the only difference being that the L1 I-cache is reduced to 4KB. The results are shown in table 5. In this configuration, the support structures begin to dominate the core area, with 23% going on the register file, 25% on the CQ and 14% on the RAU. However, the new estimated core area is now only 1.19mm^2, which less than half the area of the previous configuration. It should be noted that with 512 registers and 256 microthread slots, we have chosen to characterise a generous configuration that would tolerate 100s of cycles of latency from a memory system.

To put these estimates in perspective, using the model without the L1 D-cache, if we assume that half of the die area is given to memory structures, a 128-processor chip with 64 thousand registers would require 305mm^2, which is significantly less than the area of Intel's Montecito chip.

Recently, Kumar et.al. [36] estimated the die area of chip multiprocessor with eight cores sharing a 4MB L2 cache. In this work, each core is a 4-issue in-order processor (Alpha 21164) and has 64KB L1 caches (I/D). The total chip area was 127.76mm^2. Our estimation methodology is similar to their work and we have used the same feature size. Using the same die size and the same amount of shared memory, we could support about 50 microthreaded processors each with an FPU with a combined register file size of 25 thousand registers and able to support over 10 thousand active threads. Sharing an FPU, as proposed in that paper, is quite feasible in a microthreaded processor design and this would further increase the number of processors in the same area. A co-joined dual processor single FPU processor design would require approximately 2mm^2, allowing 64 processors with 32 thousand registers to be integrated in the same die area.

6. Conclusion

The Microgrid CMP based on microthreaded microprocessors is a promising new approach for scalability. This approach allows concurrency to be extracted from sequential code, exploiting different types of parallelism. ILP and LLP are both detected by the recompilation of legacy, sequential source code or indeed, could be obtained from the translation of existing legacy binary code. The microgrid also supports TLP by assigning application threads to groups of processors in the CMP. Moreover using microthreading it is possible to dynamically adapt the number of processors used in each group at the loop level[13].

In this paper, we have investigated the overhead of the support structures for a microthreaded microprocessor implementation, these are the CQ and RAU, as well as a larger than normal register file. All three structures are related to the local concurrency supported and hence the latency tolerated by the processor. We have described in detail a register allocation scheme, which dynamically allocates registers to micro-contexts. It is shown that for a given ISA, the scheme has an area proportional to the register file size. Moreover, the area required is tunable by choosing the unit of allocation, at the cost of some loss of efficiency in the use of the register file.

Our results show that the area of both the allocation scheme and the scheduler queue are less than that of the register file, given reasonable assumptions about the size of each. The paper also estimates the microgrid area for different configurations of memory and cache using an 0.07μm technology. This shows the feasibility of 128-way CMPs using this emerging technology and with a generous latency tolerance capability, i.e. tolerating many 100s of cycles of latency on memory or external I/O.

7. Acknowledgements

We gratefully acknowledge the contribution to this work from the Government of Palestine for supporting Mr Hasasneh's scholarship, we also acknowledge NWO for their support of the *Microgrid* project, which has been funded in their GLANCE program, project number: 600.643.000.05N07.

8. References

1. Barroso, L. A. et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing", Proc. of 27th Annual International Symposium on Computer Architecture, Vancouver, British Columbia, Canada, pp. 282-293, June 2000.

2. Hammond, L., Hubbert, B. A., Siu, M., Prabhu, M. K., Chen, M. and Olukolun, K., "The Stanford Hydra CMP", IEEE Micro, vol. 20, pp. 71-84, March-April 2000.

3. Hammond, L., Nayfah, B. A. and Olukotun, K., "A Single-Chip Multiprocessor", IEEE Computer Society, vol. 30, no. 9, pp. 79-85, September 1997.

4. Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., and Sinharoy, B., "Power4 System Micro-architecture", IBM Journal of Research and Development, vol. 46, no. 1, pp. 5-25, 2002.

5. Kongetira, P., Aingaran, K. and Olukotun, K., "Niagara: 32-way Multithreaded Sparc Processor", IEEE Computer Society, vol. 25, no.2, pp. 21-29, March-April 2005.

6. McNairy, C. and Bhatia, R., "Montecito: A Dual-Core, Dual-Thread Itanium Processor", IEEE Computer Society, vol. 25, no. 2, pp. 10-20, March-April 2005.

7. Agarwal, V., Hrishikesh, M. S., Keckler, S. W. and Burger, D., "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures", Proc. of the 27th Annual International Symposium on Computer Architecture, Vancouver, British, Columbia, Canada, pp. 248-259, June 2000.

8. shilov, A., "Intel to Cancel NetBurst Pentum 4 Xeon Evolution", http:/www.xbitlabs.com/news/cpu/display/20040507000306.html, (Accessed 7/1/2005), 2004.

9. Lipasti, M. H. and Shen, J. P., "Superspeculative Microarchitecture for Beyond AD 2000", IEEE Computer Society, vol. 30, no. 9, pp. 59-66, September 1997.

10. International Technology Roadmap for Semiconductors, http://public.itrs.net, 2003, Accessed 20/4/2005.

11. Rixner, S. et al., "Register Organisation for Media Processing", International Symposium on High Performance Computer Architecture, Toulouse, France, pp. 375-386, January 2000.

12. Ronen, R. et al., "Coming Challenges in Microarchitecture and Architecture", Proc. IEEE, vol. 89, no. 3, pp. 325-340, March 2001.

13. Bousias, K. and Jesshope, C. R., "The Challenges of Massive On-chip Concurrency", to be published Proceedings ACSAC 2005, Springer, Singapore, (http://staff.science.uva.nl/ ~jesshope/Papers/ACSAC05.pdf), 2005.

14. Onder, S. and Gupta, R., “Superscalar Execution with Dynamic Data Forwarding”, Proc. of the International Conference on Parallel Architectures and Compilation Techniques, Paris, France, pp. 130-135, October 1998.

15. Balasubramonian, R., Dwarkadas, S. and Albonesi, D., “Reducing the Complexity of the Register File in Dynamic Superscalar Processors”, In Proc. of the 34th International Symposium on Micro-architecture, Austin, Texas, pp. 237-248, December 2001.

16. Palacharla, S., Jouppi, N. P. and Smith, J., “Complexity-effective Superscalar Processors”, In Proc. of the 24th International Symposium on Computer Architecture, Denver, Colorado, United States, pp. 206-218, June 1997.

17. Tullsen, D. M., Eggersa, S. and Levy, H. M., “Simultaneous Multithreading: Maximizing on Chip Parallelism”, Proc. of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, pp. 392-403, June 1995.

18. Burns, J. and Gaudiot, J-L., “Area and System Clock Effects on SMT/CMP Processors”, Proc. of the 2001 International Conference on Parallel Architectures and Compilation Techniques, Barcelona, Spain, pp. 211-218, September 2001.

19. Spracklen, L., and Abraham, S.G., “Chip Multithreading: Opportunities and Challenges”, Proc. of the 11th Intel's Symposium on High performance Computer Architecture (HPCA-11 2005), San Francisco, CA, USA, pp. 248-252, February 2005.

20. Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K. and Chang, K., “The Case for a Single-Chip Multiprocessor”, In Proc. of the Seventh International Symposium, Cambridge, MA, pp. 2-11, October 1996.

21. Ro, W., and Gaudiot, J-L, “SPEAR: A Hybrid Model for Speculative Pre-Execution”, Proc. of 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), Eldorado Hotel, Santa Fe, New Mexico, pp.26-30, April 2004.

22. Zoppetti, G. M., Agrawal, G., Pollock, L., Amaral, J. N., Tang, X., and Gao, G. R., “Automatic Compiler Techniques for Thread Coarsening for Multithreaded Architectures”, Proc. of the 14th International Conference on Supercomputing, Santa Fe, New Maxico, USA, pp. 306-315, May 2000.

23. Wilcox, K., and Manne, S., “Alpha Processor: A history of Power issues and a look to the Future”, In Cool-chips Tutorial, Held in conjunction with MICRO-32, Dec. 1999.

24. Huh, J., Burger D., and Keckler, S.W., “Exploring the Design Space of Future CMPs”, In Proc. Of International Conference on Parallel Architectures and

Compilation Techniques, Barcelona, Spain, pp. 199-210, September 2001.

25. Preston, R. P. et al., “Design of an 8-wide Superscalar RISC microprocessor with Simultaneous Multithreading”, 2002 IEEE International Solid-State Circuits Conference, San Francisco, CA, pp. 334-335, February 2002.

26. Scott, J., “Designing the Low-Power M-CORE Architecture”, Proc. IEEE Power Driven Micro Architecture Workshop at ISCA98, Barcelona, Spain, pp. 145-150, June 1998.

27. Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M., “Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction”, Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, San Diego, CA, USA, pp. 81, December 2003.

28. Yingmin, L., Brooks, D., Zhigang, H. and Skadron, K., “Performance, Energy, and Thermal Considerations for SMT and CMP Architectures”, Proc. of the 11th IEEE International Symposium on high performance computer architecture (HPCA), San Francisco, CA, USA, pp. 71-82, February 2005.

29. Kiemb, M. and Choi, K., “Memory and Architecture Exploration with Thread Shifting for Multithreaded Processors in Embedded Systems”, Proc. of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, Washington DC, USA, pp. 230-237, September 2004.

30. Jesshope, C. R., “Scalable Instruction-level Parallelism”, In Computer Systems: Architectures, Modeling and Simulation, 3rd and 4th International Workshops, SAMOS 2004, Samos, Greece, pp. 383-392, July 2004.

31. Bousias, K, Hasasneh N M and Jesshope C R (2005) Instruction-level parallelism through Microthreading - a scalable Approach to chip multiprocessors, an electronic version of an article to be published in the BCS Computer Journal. Online access: http://comjnl.oxfordjournals.org/cgi/rapidpdf/bxh157?ijkey=EoSzke60tdKdUYz&keytype=ref

32. Jesshope C. R. (2005) Micro-grids - the exploitation of massive on-chip concurrency, pp 203-223 (Invited paper, HPC 2004Cetraro, June 2004), In Grid Computing: A New Frontier of High Performance Computing, 14, pp203-223, (ed. L. Grandinetti, Elsevier, Amsterdam, 2005) pp203-223.

33. Silberman, J. et al., “A 1.0 GHz single issue 64b PowerPC integer processor”, ISSCC, Department of Computer Sciences, IBM Austin Research Lab., Austin, Tx, pp. 230, 1998.

34. Gupta, S., Keckler, S. W., and Burger, D.C., “Technology Independent Area and Delay Estimates for Microprocessor Building Blocks”, Tech. Report TR2000-05, Department of Computer Sciences, the University of Texas at Austin, pp. 1-27, May 2000.

35. Lopez, D., Llosa, J., Valero, M. and Ayguade, E., "Resource Widening versus Replication: Limits and Performance-Cost Trade-Off", 12th International Conference on supercomputing (ICS-12), Melbourne, Australia, pp. 441-448, 1998.

36. Kumar, R., Jouppi, N.P., and Tullsen, D.M., "Conjoined-Core Chip Multiprocessing", Proc. of the 37th annual International Symposium on Microarchitecture (MICRO-37 2004), Portland, Oregon, pp. 195-206 December 2004.

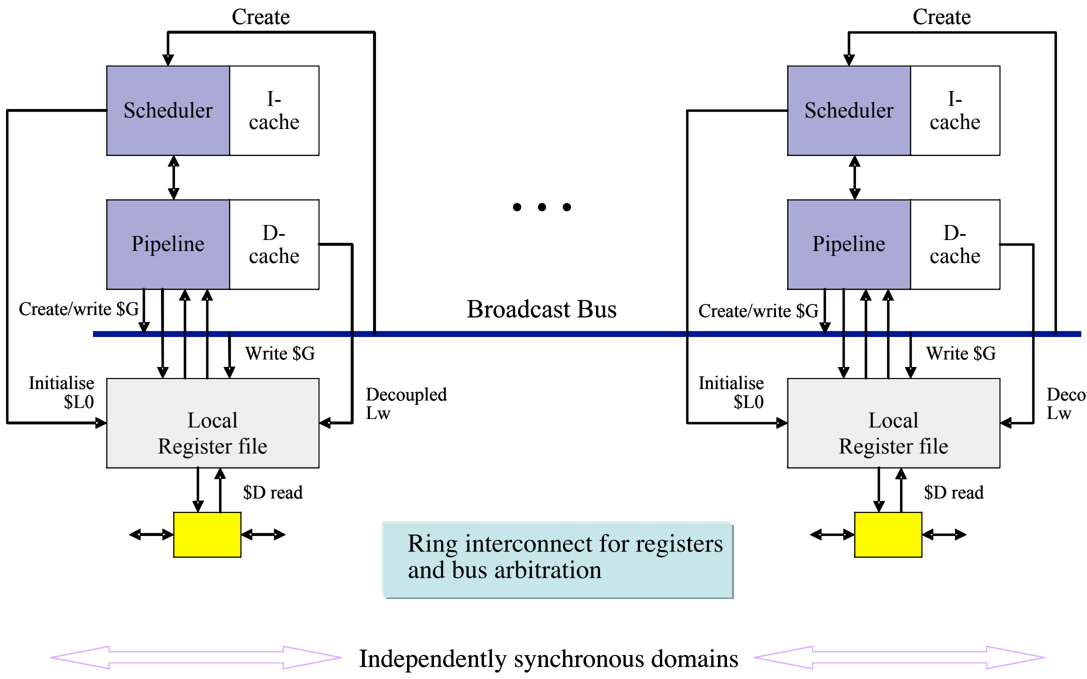


Figure 1. Microgrid chip multiprocessor based on asynchronous collection of microthreaded pipelines communicating with a broadcast bus and ring network.

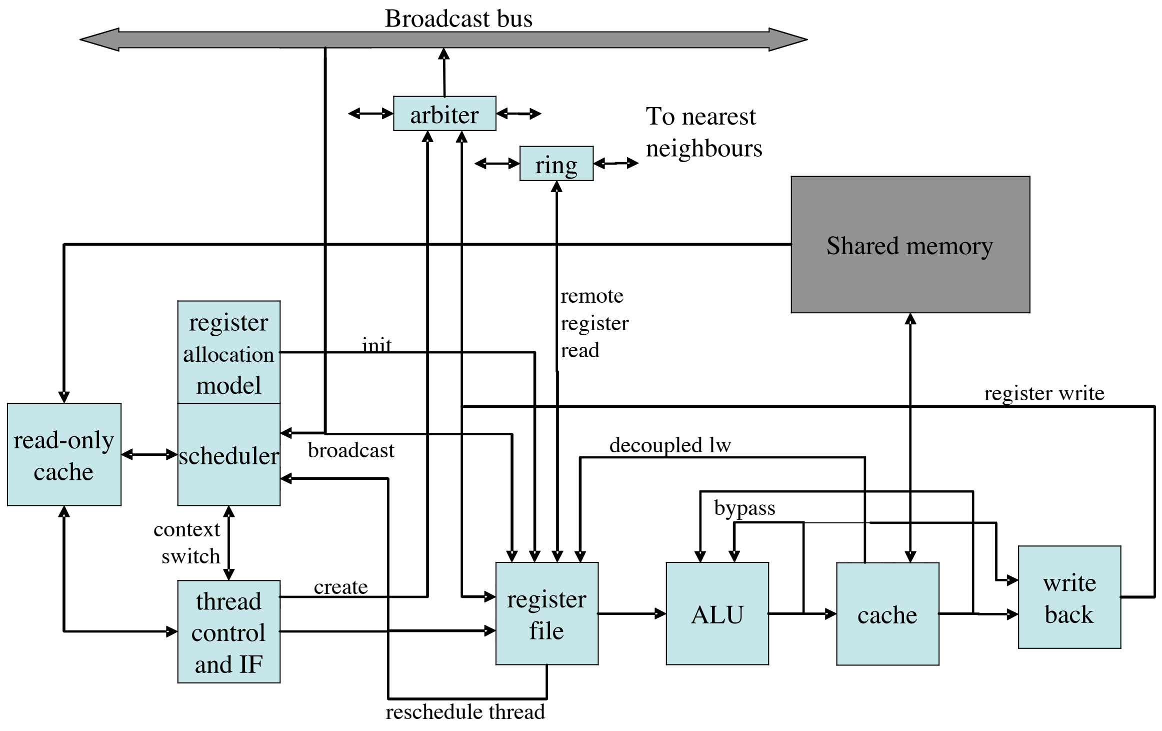


Figure 2. The pipeline of a microthreaded microprocessor, showing five stages including an L1 D cache

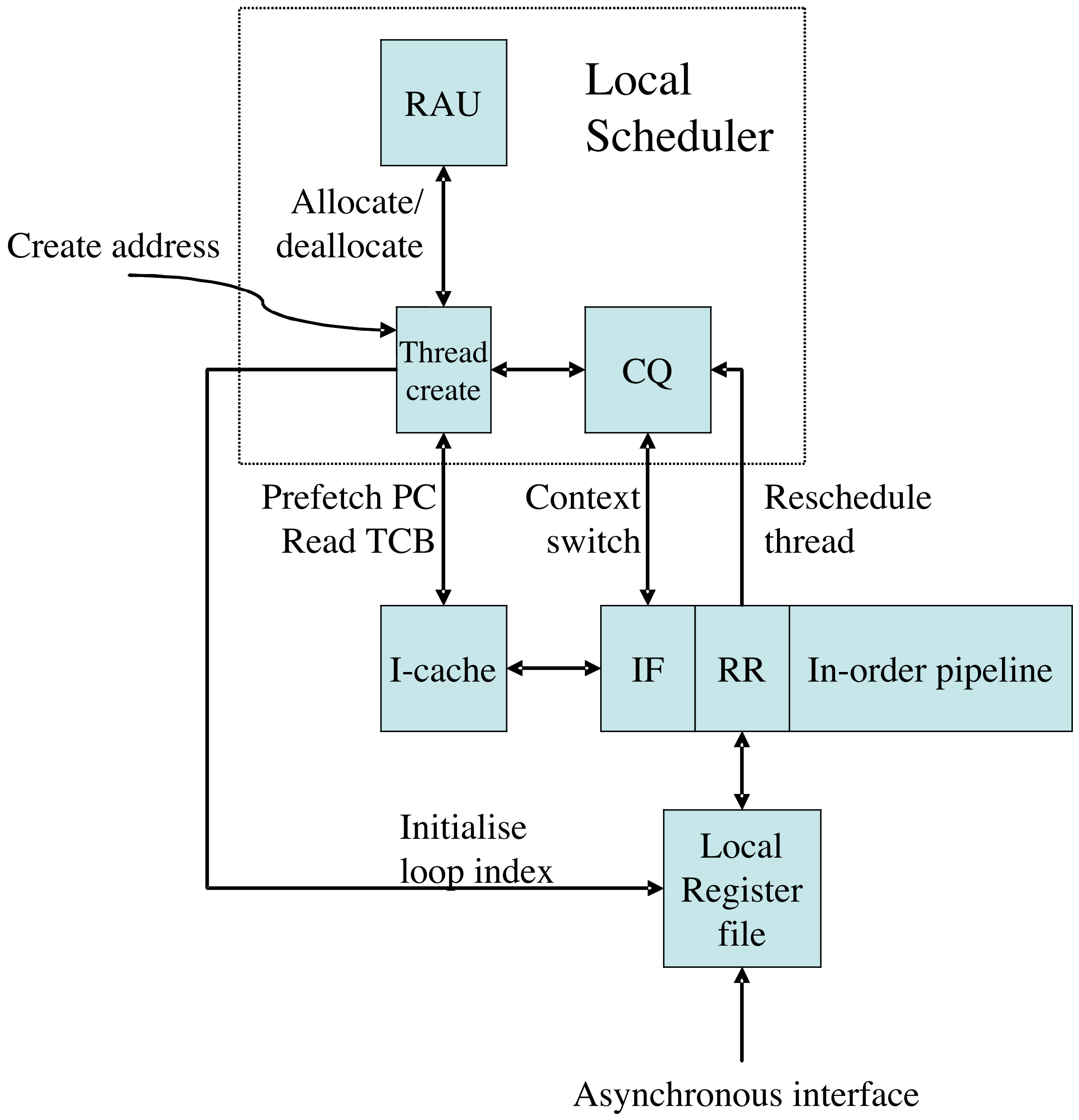


Figure 3. Detail of the local scheduler showing its main components and the data paths between it and other stages of the pipeline.

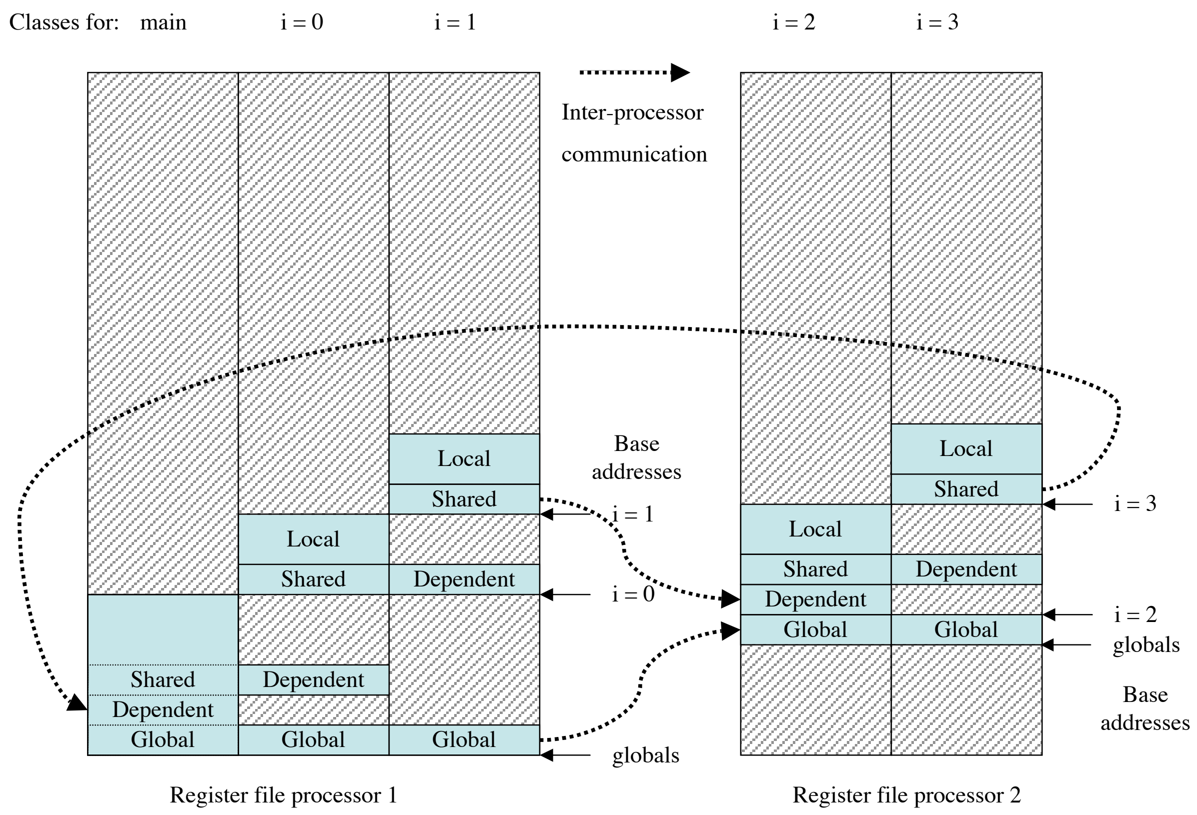


Figure 4. Diagram showing the mapping of microcontexts for a family of 4 threads (i=0 to i=3) mapped to two processors. The diagram shows the four register classes: globals, shareds, dependents and locals for the main thread and for each microthread created. A location is addressed by a base addresses plus a regular register specifier, whose value determines the class. In any thread addressing requires a base address for that processor and a base address for itself and any thread it is dependent upon. The latter two are stored in the CQ slot for each thread, with one bit indicating whether the shared class is mapped to the same or an adjacent processor.

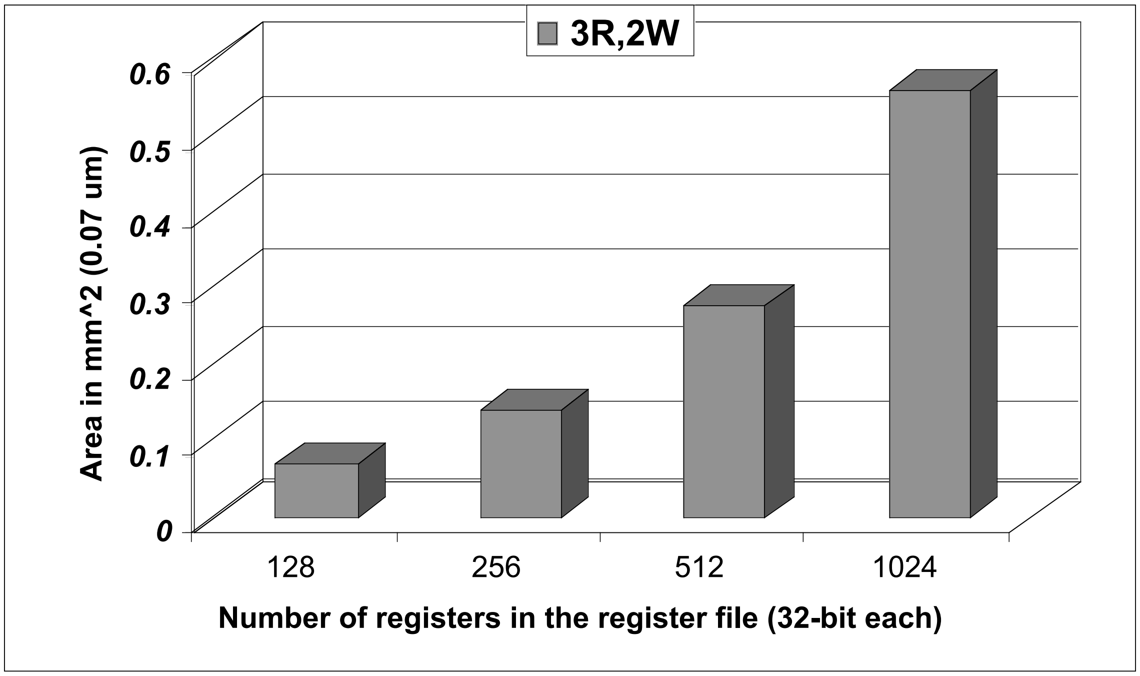


Figure 5. Estimated area of one processor's partition of a distributed register file comprising 5 ports per processor. The area estimate is for 0.07μm technology.

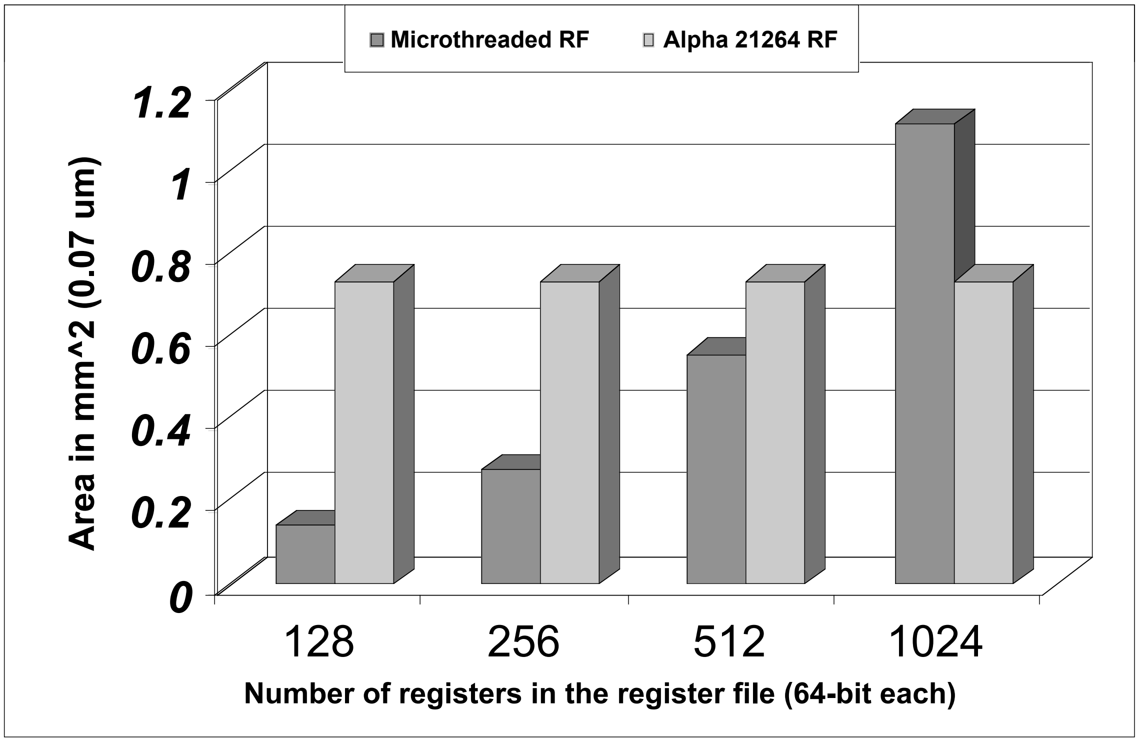


Figure 6. Area comparison between different sizes of a microthreaded register file and for the alpha 21264's register file (both estimates are for 0.07µm technology). Note that the 21264 provides only 152 registers to the microarchitecture.

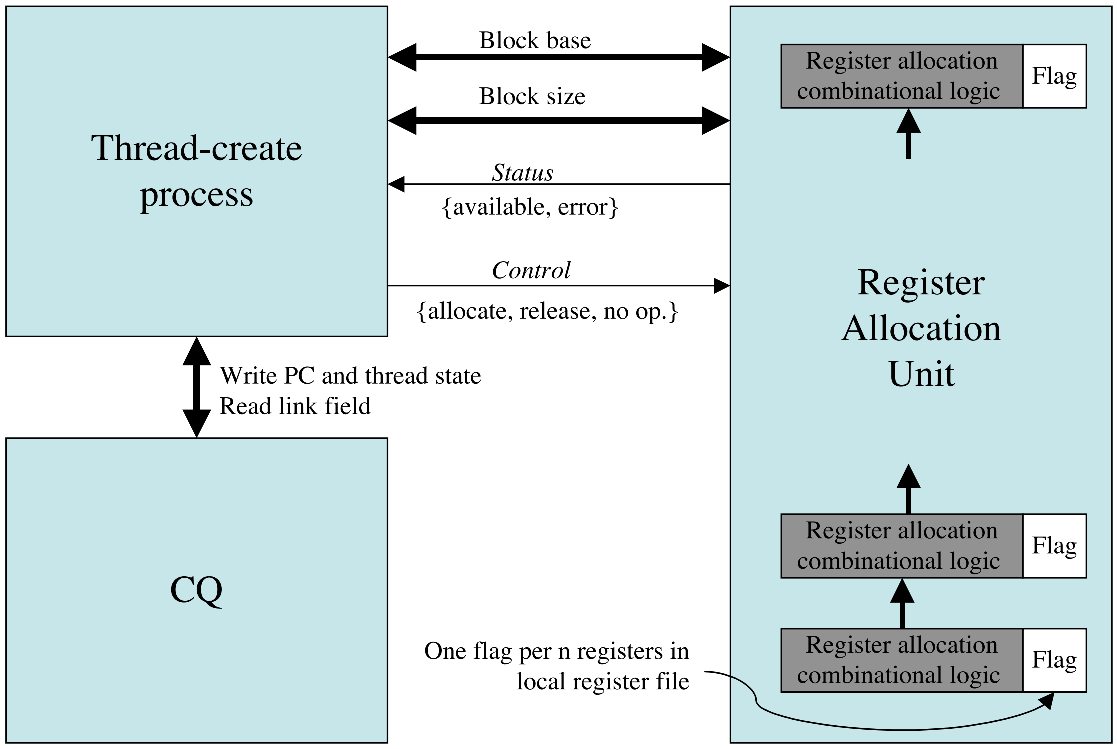

Thread-create process
Block base
Block size
Status
{available, error}
Control
{allocate, release, no op.}
Register Allocation Unit
Register allocation combinational logic
Flag
Write PC and thread state
Read link field
CQ
One flag per n registers in local register file

Figure 7. Diagram of the RAU and its interaction with the thread-create process. The RAU has one logic slice for every block of n registers in the local register file.

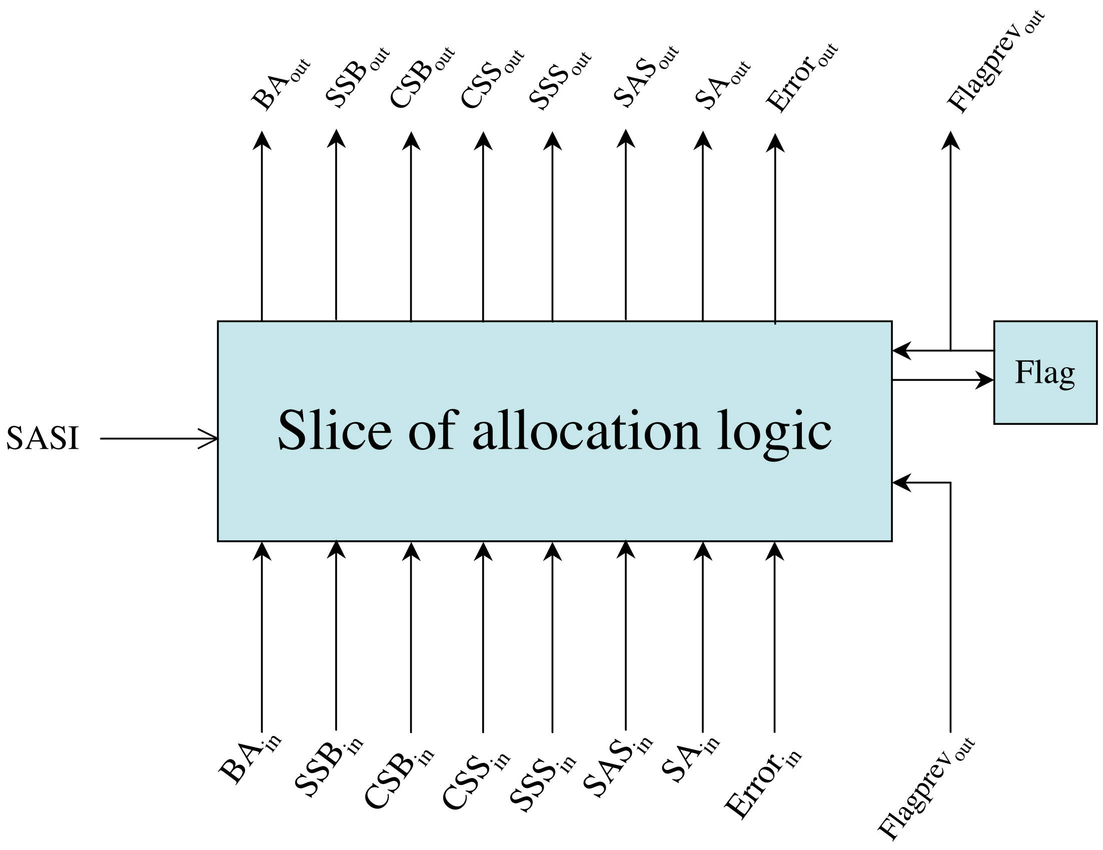


Figure 8. Register allocation unit’s combinational logic slice

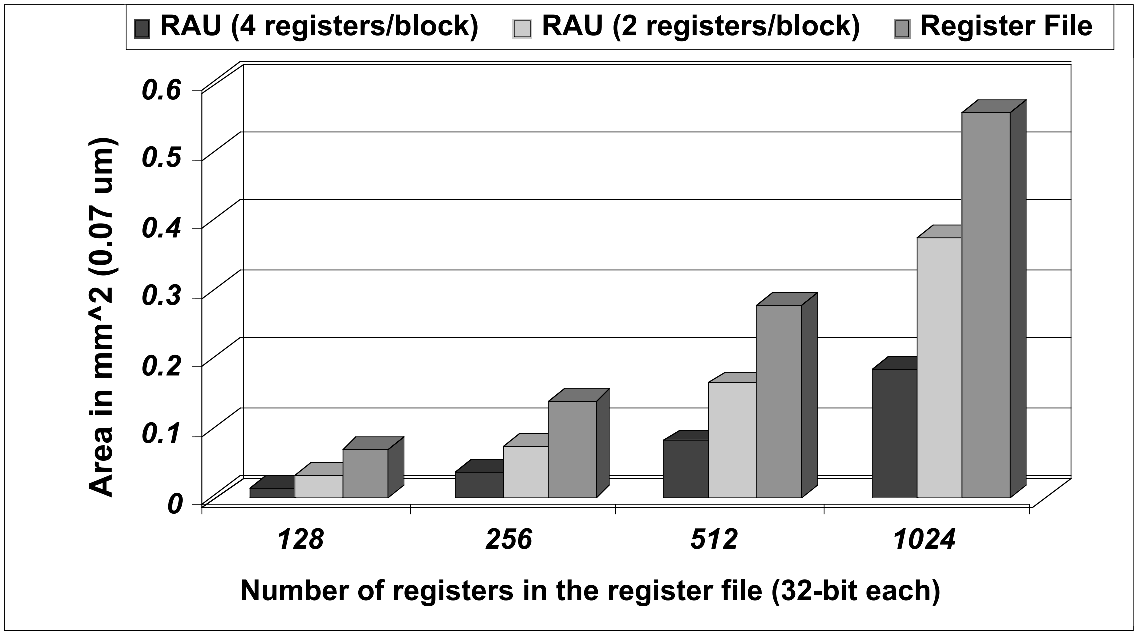


Figure 9. Area comparison between the register allocation unit and the register file for 2- and 4-register allocation blocks and for various sizes of register file.

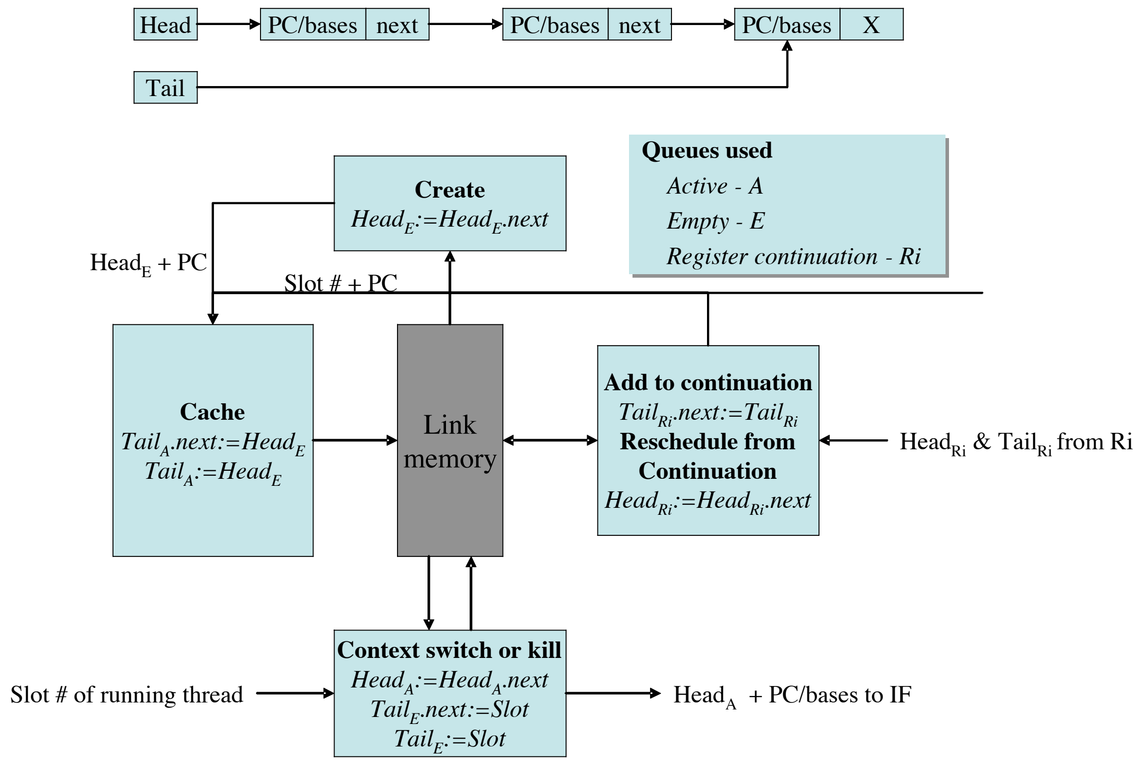


Figure 10: Block diagram of the interactions within the CQ, which use its link field to build: a) a queue for empty slots, b) a queue containing active slots and c) Any continuation queues for threads suspended in the register file.

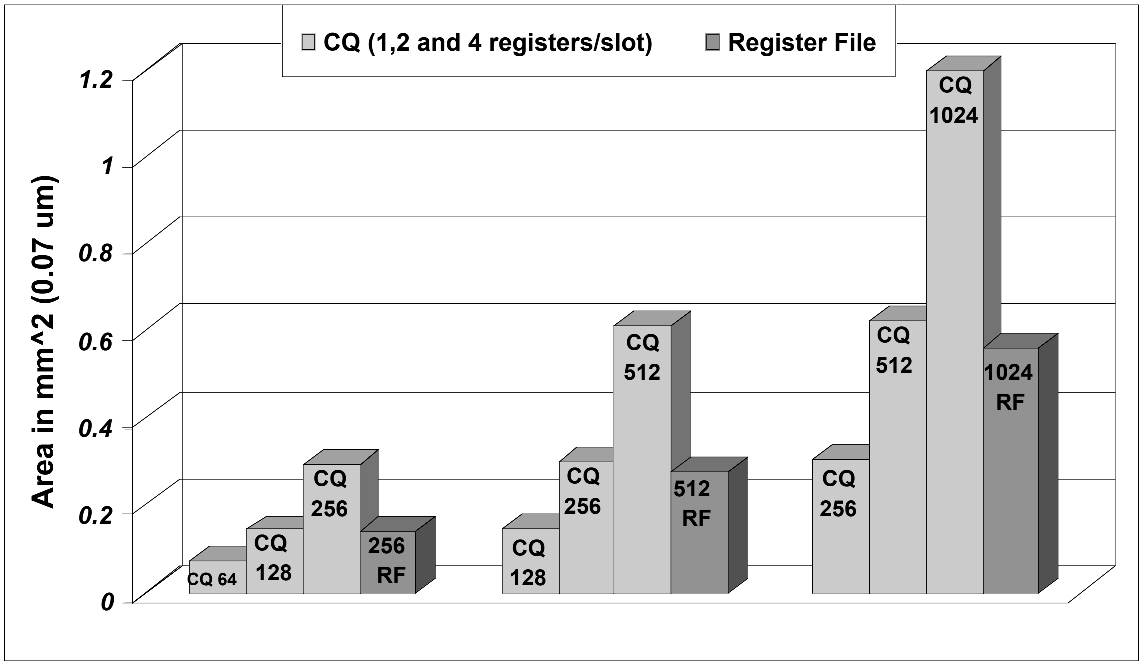
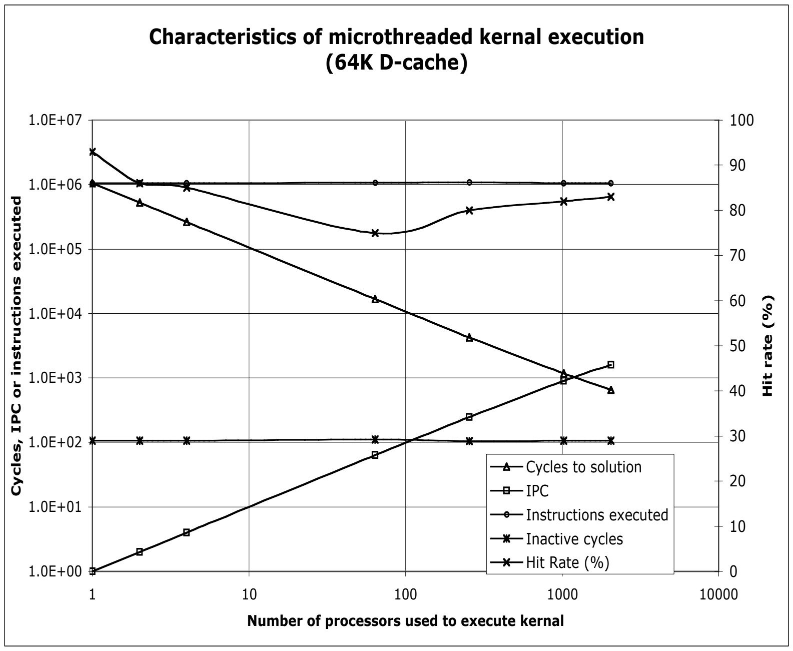


Figure 11. Area of the continuation queue compared with the register file for 1, 2 and 4-registers per slot in the continuation queue.



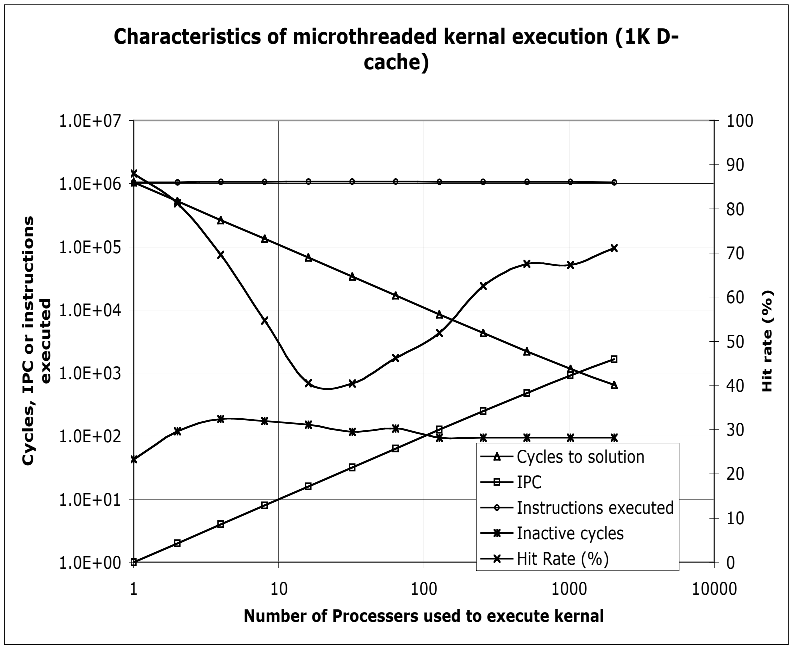


Figure 12. Simulation results of an independent loop mapped to profiles of from 1 to 2048 processors. These results show scalability of performance (both figures) as well well as complete invariance to cache size and mapping strategy – compare IPC for both cases.

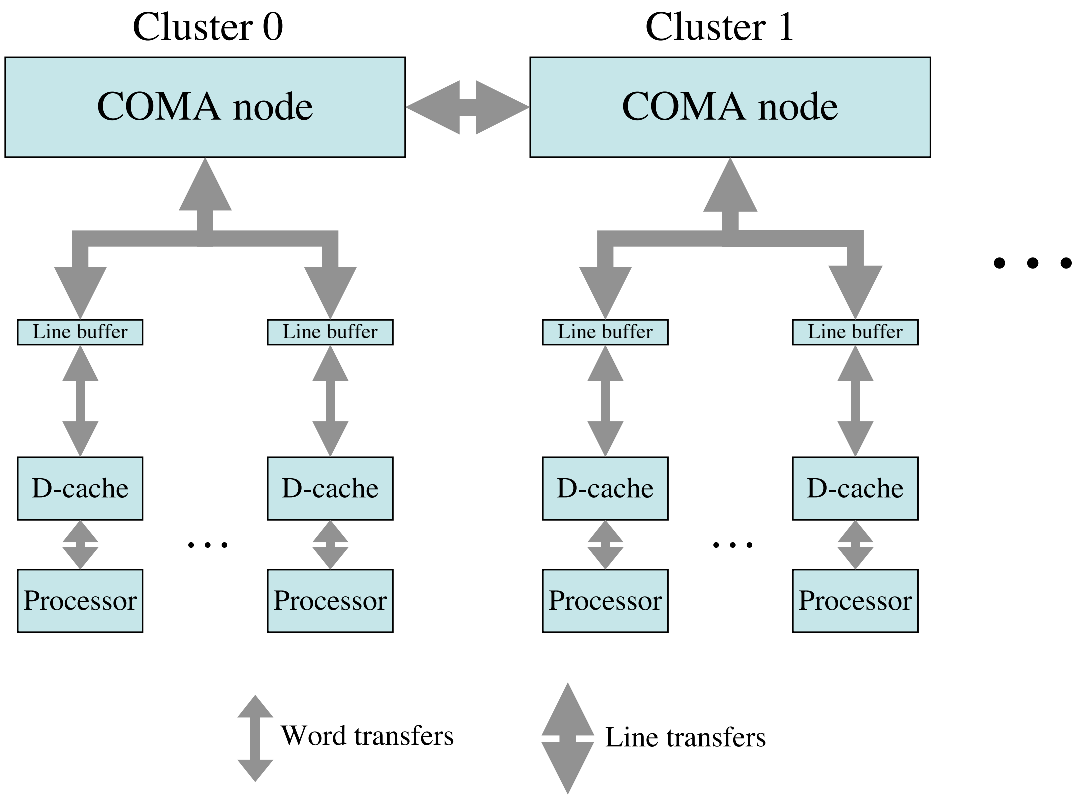


Figure 13: Memory architecture using COMA nodes and clusters of processors.

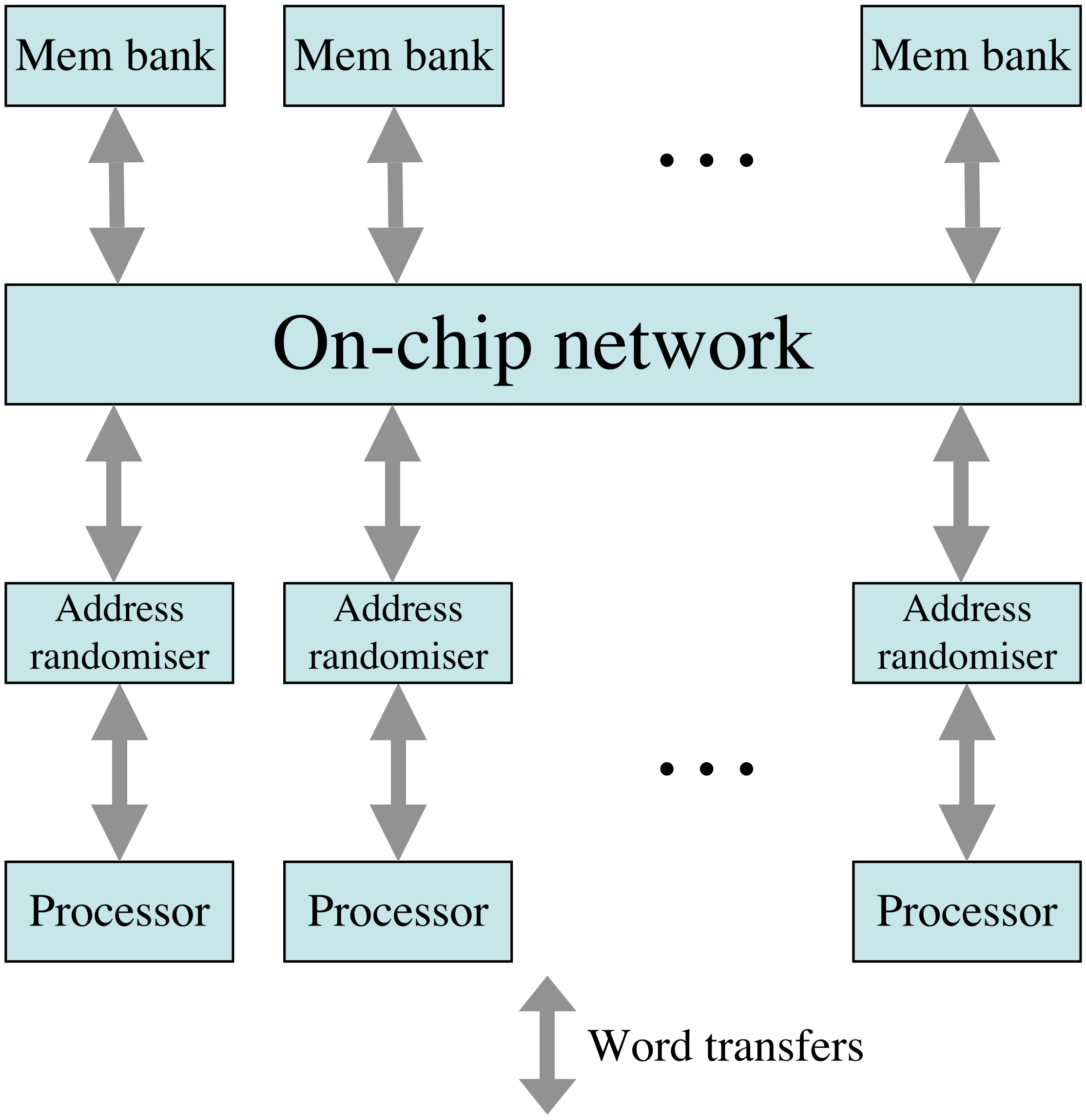


Figure 14: Memory architecture using a flat structure of multiple banks with address randomisation. Such an organisation would not use an L1 D-cache.

Table 1: Concurrency-control instructions

Instruction	**Instruction Behavior**
Cre	Creates a new family of threads
Swch	Causes a context switch to occur
Kill	Terminates the thread being executed
Bsync	Waits for all other threads to terminate
Brk	Terminates all other threads

Table 2: Allocation logic parameters

Abbreviated Name	Description
BA	Base Address
SSB	Selected Slice Base
CSB	Current Slice Base
CSS	Current Slice Size
SSS	Selected Slice Size
SAS	Set Allocate Size
SA	Slice Available
Error	Error Signal
Flagprev	Previous Flag State
Flagout	New Flag State
Flagin	Current Flag State
SASI	Set Allocate Size In
Reg	Register

Table 3: Thread entry format in the continuation queue for 256-entry CQ and 512 entry register file

Field name	Number of bits
Program counter	32
Local base	9
Dependent base	10
Producer	8
Pointer	8

Table 4: Microgrid-Core estimate area using 0.07μm technology

Functional Block	Size	Area in mm^2 (0.07μm)	%Core
L1 I-cache	8KB, Direct map	0.178	7%
L1 D-cache	64KB, 2-Way	1.15	47%
Register file	512 (32-bit each)	0.279	12%
RAU	Allocate block of 2	0.167	7%
CQ	256-entry (67-bit each)	0.299	12%
FPU	64-bit	0.356	15%
Total Core Area mm^2		**2.43**	**100%**

Table 5: Microgrid-Core estimate area without L1 D-cache using 0.07μm technology

Functional Block	Size	Area in mm^2 (0.07μm)	%Core
L1 I-cache	4KB, Direct map	0.08927	8%
Register file	512 (32-bit each)	0.279	23%
RAU	Allocate block of 2	0.167	14%
CQ	256-entry (67-bit each)	0.299	25%
FPU	64-bit	0.356	30%
Total Core Area mm^2		**1.19**	**100%**