The Computer Journal Advance Access published December 19, 2005

doi:10.1093/comjnl/bxh157

Instruction Level Parallelism through microthreading—A Scalable Approach to Chip Multiprocessors

KOSTAS BOUSIAS[1], NABIL HASASNEH[2] AND CHRIS JESSHOPE[1,*]

[1]*Department of Computer Science, University of Amsterdam, NL*
[2]*Department of Electronic Engineering, University of Hull, UK*
[*]*Corresponding author: Jesshope@science.uva.nl*

Most microprocessor chips today use an out-of-order instruction execution mechanism. This mechanism allows superscalar processors to extract reasonably high levels of instruction level parallelism (ILP). The most significant problem with this approach is a large instruction window and the logic to support instruction issue from it. This includes generating wake-up signals to waiting instructions and a selection mechanism for issuing them. Wide-issue width also requires a large multi-ported register file, so that each instruction can read and write its operands simultaneously. Neither structure scales well with issue width leading to poor performance relative to the gates used. Furthermore, to obtain this ILP, the execution of instructions must proceed speculatively. An alternative, which avoids this complexity in instruction issue and eliminates speculative execution, is the microthreaded model. This model fragments sequential code at compile time and executes the fragments out of order while maintaining in-order execution within the fragments. The only constraints on the execution of fragments are the dependencies between them, which are managed in a distributed and scalable manner using synchronizing registers. The fragments of code are called microthreads and they capture ILP and loop concurrency. Fragments can be interleaved on a single processor to give tolerance to latency in operands or distributed to many processors to achieve speedup. The implementation of this model is fully scalable. It supports distributed instruction issue and a fully scalable register file, which implements a distributed, shared-register model of communication and synchronization between multiple processors on a single chip. This paper introduces the model, compares it with current approaches and presents an analysis of some of the implementation issues. It also presents results showing scalable performance with issue width over several orders of magnitude, from the same binary code.

Keywords: concurrency, CMP, microthreads, code fragments

Received 10 May 2005; revised 24 August 2005

1. INTRODUCTION

For many years now, researchers have been interested in the idea of achieving major increases in the computational power of computers by the use of chip multiprocessors (CMPs). Examples of CMP are the Compaq Piranha [1], Stanford Hydra [2] and Hammond *et al*. [3]. Several architectures have been proposed and some manufacturers have produced commercial designs, such as the IBM Power PC [4] and Sun's MAJC project [5]. Ideally, the performance of such systems should be directly proportional to the number of processors used, i.e. should be scalable. CMPs scale well, with the limit to scalability defined by Moore's law. We calculate that current technology could support hundreds of in-order processors and achieve significant speedup over current architectures that use implicit concurrency and achieve minimal speedup through concurrent instruction issue [6]. One of the major barriers to the use of CMPs is the problem of programming them without using explicit concurrency in the user code. Ideally they should be programmed using legacy sequential code.

In theory, there is no limit to the number of processors that can be used in a CMP provided that the concurrency derived from the sequential code scales with the problem size. The problem is how to split the code into a number of independent threads, schedule these on many processors and to do this with a low and scalable overhead in terms of the control logic and processor efficiency. In fact, on general-purpose code it will

be impossible to eliminate all dependencies between threads and hence synchronization is also required. The goal of this work therefore is to define a feasible architecture for a scalable CMP that is easy to program, that maximizes throughput for a given technology and that minimizes the communication and synchronization overheads between different threads. It should be noted that in this work the term thread is used to describe very small code fragments with minimal context.

Today Intel's Itanium-2 (Madison) microprocessor features over 410 million transistors in a 0.13 μm semiconductor process technology operating at a speed of 1.5 GHz. This is a dual-processor version of the previous Itanium processor (Mckinley), which has an issue width of six. Moore's law would indicate that the billion-transistor chip will become feasible in 65 nm technology within the next 3 or 4 years [7]. Intel expects that the Itanium processor will reach 1.7 billion transistors at the end of 2005. The questions we must ask are where do we go from here and what is the best way to utilize this wealth of transistors, while maximizing performance and minimizing power dissipation?

It can be argued that the current trend in increasing the clock speed and using a large area to marginally improve the instructions executed per cycle (IPC) of the chip is a poor strategy for future generations of microprocessors and does not guarantee better performance. The use of aggressive clock speed as a means of achieving performance only exacerbates the memory wall and requires larger on-chip caches. More importantly this strategy cannot be continued indefinitely as power density is a function of frequency and is becoming a critical design constraint. Using concurrency as a means of increasing performance without increasing power density is a much better strategy so long as models and implementations can be found that are scalable. Conversely, by exploiting concurrency for constant performance, clock frequencies can be reduced, allowing voltages to be scaled down and gaining a quadratic decrease in power density with concurrency. Of course, this assumes completely scalable models and implementations.

Another problem area in future technology is the scaling of wire delays compared with gate delays. As transistor dimensions scale down, the number of gates which are reachable within the scaled clock is at best constant, which means that distributed rather than monolithic architectures need to be exploited [8].

Superscalar processors today issue up to eight instruction per clock cycle but instruction issue is not scalable [9] and a linear increase in parallelism requires at least a quadratic increase in area [10]. The logic required occupies $\sim$30% of the total chip area in a 6-way superscalar processor [11]. In addition, more and more area is being used for on-chip memory. Typically the second level on-chip cache occupies 25–30% of the die area on a modern microprocessors and between 50 and 75% on the recently announced Itanium-2. Moreover, a significant delay and power consumption are seen in high-issue-width processors due to tag matching, wake-up signals to waiting instructions and selection mechanisms for issuing instructions. These delays increase quadratically for most building blocks with the instruction window size [12]. Finally, even with the implementation of a large instruction window, it difficult for processors to find sufficient fine-grain parallelism, which has made most chip manufacturers like Compaq, Intel and Sun look at simultaneous multi-threading (SMT) [13] to expose more instruction level parallelism (ILP) through a combination of coarse- and fine-grain parallelism.

Future microprocessor technology needs efficient new architectures to achieve the demands of many grand-challenge applications, such as weather and environmental modelling, computational physics (on both sub-atomic and galactic scales) and biomolecular simulation. In the near future, it should be possible to integrate thousands of arithmetic units on a single chip [14] but out-of-order execution is not a good candidate, because of the limited concurrency it exposes and the non-linear increase in hardware complexity of instruction issue with issue width. Instruction issue is not the only scaling problem, the complexity of the register file [15] and bypass logic [12] also scale badly giving further barriers to a scalable architecture. Very-long instruction word (VLIW) architectures transfer the task of instruction scheduling from the hardware to the compiler, which avoids the scaling problems in instruction issue. However, in practice, many modern VLIW architectures end up requiring many of the same complex mechanisms as superscalar processors [16], such as branch prediction, speculative loads, pipeline interlocks, and new mechanisms like code compressors.

Multi-threading can expose higher levels of concurrency and can also hide latency by switching to a new thread when one thread stalls. SMT appears to be the most popular form of multi-threading. In this technique, instructions from multiple threads are issued to a single wide-issue processor out of programmed order using the same problematic structures we have already described. It increases the amount of concurrency exposed in out-of-order issue by working on multiple threads simultaneously. The main drawback to SMT is that it complicates the instruction issue stage, which is central for the multiple threads [17]. Scalability in instruction issue is no easier to achieve because of this and the other scalability problems remain unchanged. Thus SMT suffers from the same implementation problems [18] as superscalar processors.

An alternative approach to multi-threading that eliminates speculation and does provide scalable instruction issue is the *microthreaded model*. The threads in this model are small code fragments with an associated program counter. Little other state is required to manage them. The model is able to expose and support much higher levels of concurrency using explicit but dynamic controls. Like VLIW, the concurrency exposed comes from the compiler and expresses loops as well as basic block concurrency. Unlike VLIW, the concurrency is parametric and is *created* dynamically. In pipelines that execute this

model, instructions are issued in-order from any of the threads allocated to it but the schedule of instructions executed is non-deterministic, being determined by data availability. Threads can be deterministically distributed to multiple pipelines based on a simple scheduling algorithm. The allocation of these threads is dynamic, being determined by resource availability, as the concurrency exposed is parametric and not limited by the hardware resources. The instruction issue schedule is also dynamic and requires linear hardware complexity to support it. Instructions can be issued from any microthread already allocated and active. If such an approach could also give linear performance increase with number of pipelines used, then it can provide a solution to both CMP and ILP scalability [19].

The performance of the microthreaded model is compared with a conventional pipeline in [20] for a variety of memory latencies. The results show that the microthreaded microprocessor always provides a superior performance to that of a non-threaded pipeline and consistently shows a height IPC on a single issue pipeline (0.85) even for highly dependent code and in the presence of very high memory latencies. The model naturally requires more concurrency to achieve the same asymptotic performance as memory delays are increased. It will be shown in this paper that applying this model to a CMP can also provide orders of magnitude speedup and IPCs of well over 1000 have been demonstrated. Note that both the instruction issue logic and the size of the register file in this model scale linearly with issue width.

2. CURRENT APPROACHES

2.1. Out-of-order execution

To achieve a higher performance, modern microprocessors use an out-of-order execution mechanism to keep multiple execution units as busy as possible. This is achieved by allowing instructions to be issued and completed out of the original program sequence as a means of exposing concurrency in a sequential instruction stream. More than one instruction can be issued in each cycle, but only independent instructions can be executed in parallel, other instructions must be kept waiting or, under some circumstances, can proceed speculatively.

Speculation refers to executing an instruction before it is known whether the results of the instruction will be used or not, this means that a guess is made as to the outcome of a control or data hazard as a means to continue executing instructions, rather than stalling the pipeline. Register renaming is also used to eliminate the artificial data-dependencies introduced by issuing instructions out of order. This also enables the extension of the architectural register set of the original ISA, which is necessary to support concurrency in instruction execution. Any concurrent execution of a sequential program will require some similar mechanism to extend the synchronization memory available to instructions. Speculative execution and out-of-order issue are used in superscalar processors to expose concurrency from sequential binary code. A reorder buffer or future file of check points and repairs is also required to re-order the completion of instructions before committing their results to the registers specified in the ISA in order to achieve a sequentially consistent state on exceptions.

Control speculation predicts branch targets based on the prior history for the same instruction. Execution continues along this path as if the prediction was correct, so that when the actual target is resolved, a comparison with the predicted target will either match giving a correctly predicted branch or not, in which case there was a misspredict. A misspredict can require many pipeline cycles to clean up and, in a wide-issue pipeline, this can lead to hundreds of instruction slots being unused, or to be more precise, if we focus on power, being used unnecessarily. Both prediction and cleaning up a misspredict require additional hardware, which consumes extra area and also power, often to execute instructions whose results are never used. It can therefore be described as wasteful of chip resources and, moreover, has unpredictable performance characteristics [21]. We will show that it is possible to obtain high performance without speculation and, moreover, save power in doing so.

As already noted in Section 1, as the issue width increases in an out-of-order, superscalar architecture, the size of the instruction window and associated logic increase quadratically, which result in a large percentage of the chip being devoted to instruction issue. The out-of-order execution mechanism therefore prevents concurrency from scaling with technology and will ultimately restrict the performance over time. The only reason for using this approach is that it provides an implicit mechanism to achieve concurrent execution from sequential binary code.

2.2. VLIW

An alternative and explicit approach to concurrency in instruction issue is VLIW, where multiple functional units are used concurrently as specified by a single instruction word. This usually contains a fixed number of operations that are fetched, decoded, issued and executed concurrently. To avoid control or data hazards, VLIW compilers must hoist later independent instructions into the VLIW or if this is not possible, must explicitly add *no-op* instructions instead of relying on hardware to stall the instruction issue until the operands are ready. This can cause two problems, firstly, a stall in one instruction will stall the entire width of the instruction; secondly, adding no-op instructions, increases the program size. In terms of performance, if the program size is large compared to the I-cache or TLB size, it may result in higher miss rates, which in turn degrades the performance of the processor [22].

It is not possible to identify all possible sources of pipeline stalls and their duration at compile time. For example, suppose a memory access causes a cache miss, this leads to a longer than expected stall. Therefore, in instructions with non-deterministic delay like a load instruction to a cache hierarchy,

the number of no-op instructions required is not known and most VLIW compilers will schedule load instructions using the cache-hit latency rather than the maximum latency. This means that the processor will stall on every cache miss. The alternative of scheduling all loads with the cache miss latency is not feasible for most programs because the maximum latency may not be known due to bus contention or memory port delays, and it also requires considerable ILP. This problem with non-determinism in cache access limits VLIW to cacheless architecture unless speculative solutions are embraced. This is a significant problem with modern technology, where processor speeds are significantly higher than memory speeds [19]. Also pure VLIW architectures are not good for general purpose applications, due to their lack of compatibility in binary code [23]. The most significant use of VLIW therefore is in embedded systems, where these constraints are both solved (i.e. single applications using small fast memories). A number of projects described below have attempted to apply speculation to VLIW in order to solve the scheduling problems and one, the Transmeta Crusoe, has applied dynamic binary code translation to solve the backward compatibility problem.

The Sun MAJC 5200 [24] is a CMP based on four-way issue, VLIW pipelines. This architecture provides a set of predicated instructions to support control speculation. The MAJC architecture attempts to use speculative, thread-level parallelism (TLP) to support the multiple processors. This aggressively executes code in parallel that cannot be fully parallelized by the compiler [25, 26]. It requires new hardware mechanisms to eliminate most squashes due to data dependencies [25]. This method of execution is again speculative and can degrade the processor's performance when the speculation fails. MAJC replicates its shared registers in all pipelines to avoid sharing resources. From the implementation point of view, replicating the registers costs significant power and area [27] and also restricts the scalability. Furthermore, the MAJC compiler must know the instructions latencies before it can create a schedule. As described previously, it is not simple to detect all instructions' latencies due to the variety of the hardware communication overheads.

2.3. EPIC

Intel's explicitly parallel instruction computing (EPIC) architecture is another speculative evolution of VLIW, which also solves the forward (although not backward) code compatibility problem. It does this through the run-time binding of instruction words to execution units. The IA-64 [28] architecture supports binary code compatibility across a range of processor widths by utilizing instructions packets that are not determined by issue width. This means a scheduler is required to select instructions for execution on the available hardware from the current instruction packet. This gives more flexibility as well as supporting binary code compatibility across future generations of implementation. The IA 64 also provides architectural support for control and data speculation through predicated instruction execution and binding pre-fetches of data into cache. In this architecture each operation is guarded by one of the predicate registers, each of which stores 1 bit that determines whether the results of the instruction are required or not. Predication is a form of delayed branch control and this bit is set based on a comparison operation. In effect, instructions are executed speculatively but state update is determined by the predicate bit so an operation is completed only if the value of its guard bit is true, otherwise the processor invalidates the operation. This is a form of speculation that executes both arms of some branches concurrently but this action restricts the effective ILP, depending on the density and nesting of branches.

Pre-fetching is achieved in a number of ways. For example, by an instruction identical to a load word instruction that does not perform a load but touches the cache and continues, setting in motion any required transactions and cache misses up the hierarchy. These instructions are hoisted by the compiler up the instructions stream, not just within the same basic block. They can therefore tolerate high latencies in memory, providing the correct loads can be predicted. There are many more explicit controls on caching in the instruction set to attempt to manage the non-deterministic nature of large cache hierarchies. Problems again arise from the speculative nature of the solution. If for some reason the pre-fetch fails, either because of a conflict or insufficient delay between the pre-fetch and genuine load word instruction, then a software interrupt is triggered incurring a large delay and overhead.

EPIC compilers face a major problem in constructing a plan of execution, they cannot predict all conditional branches and know which execution path is taken [29]. To some extent this uncertainty is mitigated by predicated execution but as already indicated, this is wasteful of resources and power and like all speculative approaches can cause unpredictability in performance. Although object code compatibility has been solved to some extent, the forward compatibility is only as good as the compiler's ability to generate good schedules in the absence of dynamic information. Also the code size problem is still a challenge facing the EPIC architecture [29].

2.4. Multiscalar

Another paradigm to extract even more ILP from sequential code is the *multiscalar* architecture. This architecture extends the concept of superscalar processors by splitting one wide processor into multiple superscalar processors. In a superscalar architecture, the program code has no explicit information regarding ILP, only the hardware can be employed to discover the ILP from the program. In multiscalar, the program code is divided into a set of tasks or code fragments, which can be identified statically by a combination of the hardware and software. These tasks are blocks in the control flow graph of the program and are identified by the compiler. The purpose of

this approach is to expose a greater concurrency explicitly by the compiler.

The global control unit used in this architecture distributes the tasks among multiple parallel execution units. Each execution unit can fetch and execute only the instructions belonging to its assigned task. So, when a task missprediction is detected, all execution units between the incorrect speculation point and the later task are squashed [30]. Like superscalar, this can result in many wasted cycles, however, as the depth of speculation is much greater, the unpredictability in performance is correspondingly wider.

The benefit of this architecture over a superscalar architecture is that it provides more scalability. The large instruction window is divided into smaller instruction windows, one per processing unit, and each processing unit searches a smaller instruction window for independent instructions. This mitigates the problems of scaling instruction issue with issue width. The multiple tasks are derived from loops and function calls, allowing the effective size of the instruction window to be extremely large. Note that not all instructions within this wide range are simultaneously being considered for execution [31]. This optimization of the instruction window is offset by a potentially large amount of communication, which may effect the overall system performance.

Communication arises because of dependencies between tasks, examples are loop-carried dependencies, function arguments and results. Results stored to register, that are required by another task, are routed from one processor to another at run-time via a unidirectional ring network. Recovery from misspeculation is achieved by additional hardware that maintains two copies of the registers along with a set of register masks, in each processing unit [32]. In summary then, although the multiscalar approach mitigates against instruction window scaling allowing wider issue width, in practice it requires many of the same complex mechanisms as superscalar and being speculative is unlikely to be able to perform as consistently as a scalable CMP.

2.5. Multi-threading

In order to improve processor performance, modern microprocessors try to exploit TLP through a multi-threading approach even at the same time as they exploit ILP. Multi-threading is a technique that tolerates delays associated with synchronizing, including synchronizing with remote memory accesses, by switching to a new thread, when one thread stalls. Many forms of explicit multi-threading techniques have been described, such as interleaved multi-threading (IMT), blocked multi-threading (BMT) and SMT. A good survey of multi-threading is given in [17].

A number of supercomputers designed by Burton Smith have successfully exploited IMT, these include the Delencor HEP, the Horizon and culminated in the Tera architecture [33]. This approach is perhaps the closest to that of microthreading described in this paper, although the processor was designed as a component of a large multi-computer and not as general purpose chip. The interleaved approach requires a large concurrency in order to maintain efficient pipeline utilization, as it must be filled with instructions from independent threads. Unlike the earlier approaches, Tera avoids this requirement using something called *explicit-dependence lookahead*, which uses an instruction tag of 3 bits that specifies how many instructions can be issued from the same stream before encountering a dependency on it. This minimizes the number of threads required to keep the pipeline running efficiently, which is $\sim$70 in the case of memory accesses. It will be seen that microthreading uses a different approach that maintains full backward compatibility in the ISA, as well as in the pipeline structure.

Unlike IMT, which usually draws concurrency from ILP and loops, BMT usually exploits regular software threads. There have been many BMT proposals, see [17] and even some commercial designs such as the Sun's Niagra processor [34]. However the concurrency exposed in BMT architectures is limited, as resources, such as register files must be duplicated to avoid excessive context switching times. This limits the applicability of BMT to certain classes of applications, such as servers.

SMT, is probably the most popular and commercial form of multi-threading in use today. In this approach, multiple instructions from multiple threads provide ILP for multiple execution units in an out-of-order pipeline. Several recent architectures have either used or proposed SMT, such as the Hyper-Thread Technology in the Intel Xeon processor [35] and the Alpha 21464 [36]. As already described, the main problem with an SMT processor is that it suffers from the same scalability issues as a superscalar processor, i.e. layout blocks and circuit delays grow faster than linearly with issue width. In addition to this, multiple threads share the same level-1 I-cache, which can cause high cache miss rates, all of which provides limits to its ultimate performance [18].

2.6. Recent CMPs

From the above discussion we see that most current techniques for exploiting concurrency suffer from software and/or hardware difficulties, and the focus of research and development activity now seems to be on CMPs. These designs give a more flexible and scalable approach to instruction issue, freeing them to exploit Moore's law though system level concurrency. Some applications can exploit such concurrency through the use of multi-threaded applications. Web and other servers are good examples; however, the big problem is how to program CMPs for general purpose computation and whether performance can ever be achieved from legacy sequential code, either in binary or even source form.

Several recent projects have investigated CMP designs [1, 2, 3, 37]. Typically, the efficiency of a CMP depends on

the degree and characteristics of the parallelism. Executing multiple processes or threads in parallel is the most common way to extract high level of parallelism but this requires concurrency in the source code of an application. Previous research has demonstrated that a CMP with four 2-issue processors will reach a higher utilization than an 8-issue superscalar processor [17]. Also, work described in [3] shows that a CMP with eight 2-issue superscalar processors would occupy the same area as a conventional 12-issue superscalar. The use of CMPs is a very powerful technique to obtain more performance in a power efficient manner [38]. However, using superscalar processors as a basis for CMPs, with their complex issue window, large on-chip memory, large multi-ported register file and speculative execution is not such a good strategy because of the scaling problems already outlined. It would be more efficient to use simpler in-order processors and exploit more concurrency at the CMP level, provided that this can be utilized by a sufficiently wide range of applications. This is an active area of research in the compiler community and until this problem is solved, CMPs based on user-level threads will only be used in applications which match this requirement, such as large server applications, where multiple service requests are managed by threads.

3. REGISTER FILES

The register file is a major design obstacle to scalable systems. All systems that implement concurrency require some form of synchronizing memory. In a dataflow architecture, this is the matching store, in an out-of-order issue processor it is the register file, supported by the instruction window, reservation stations and re-order buffer. To implement more concurrency and higher levels of latency tolerance, this synchronizing memory must be increased in size. This would not be a problem except that in centralized architectures, as issue width increases, the number of ports to this synchronizing memory must also increase. The problem is that the cell size grows quadratically with the number of ports or issue width. If N instructions can be issued in one cycle, then a central register file requires $2N$ read ports and N write ports to handle the worst case scenario. This means that the register cell size grows quadratically with N. Moreover, as the number of registers also increases with the issue width, a typical scaling of register file area is as the cube of N.

The register file in the proposed Alpha 8-way issue 21464 had a single 512 register file with 24 ports in total. It occupied an area of some five times the size of the L1 D-cache of 64 Kbyte [39]. Also, in the Motorola's M. CORE architecture, the register file energy consumption can be 16% of the total processor power and 42% of the data path power [40]. It is clear therefore that the multi-ported register files in modern microprocessors consume significant power and die area. Several projects have investigated the register file problem in terms of reducing the number of registers or minimizing the number of read or write ports [14, 15, 41, 42].

Work done in [41] describes a bypass scheme to reduce the number of register file read ports by avoiding unnecessary register file reads for the cases where values are bypassed. In this scheme an extra bypass hint bit is added to each operand of instructions waiting in the issue window and a wake-up mechanism is issued to reduce register file read ports. As described in [43], this technique has two main problems. First, the scheme is only a prediction, which can be incorrect, requiring several additional repair cycles for recovery on misspredictions. Second, because the bypass hint is not reset on every cycle, the hint is optimistic and can be incorrect if the source instruction has written back to register file before the dependent instruction is issued. Furthermore, an extra pipeline stage is required to determine whether to read data operands from the bypass network or from the register file.

Other approaches include a delayed write-back scheme [42], where a memory structure is used to delay the write-back results for a few cycles to reduce register file ports. The disadvantage of this scheme is that it is necessary to write the results both to the register file and the write-back queue concurrently to avoid consistency problems during register renaming. The authors propose an extension to this scheme to reduce the number of register write ports. However, this extension suffers from an IPC penalty and it degrades the pipeline performance. Furthermore, in this model, any branch misspredictions cause a pipeline stall and insufficient use of the delay write-back queue. In fact most previous schemes for minimizing the multi-ported register file have required changes in the pipeline design and do not enable full scalability. At best they provide a constant remission in the scalability of the register file.

Recently Rixner *et al.* [14] suggested several partitioning schemes for the register file from the perspective of streaming applications, including designs spanning a central register file through to a distributed register file organization. Their results, not surprisingly, show that a centralized register file is costly and scales as $O(N^3)$, while in the distributed scheme, each ALU has its own port to connect to the local register files and another port to access other register files via a fast crossbar switch network. This partitioning proved to use less area and power and caused less delay compared with the purely global scheme, and was also shown to provide a scalable solution. The distributed configuration also has a smaller access time compared with the centralized organization. In this work, a 128 32-bit register file with 16 read ports and 8 write ports is used for a central register file and is compared to 8 local register files of 32 32-bit registers, 2 read ports, 1 write port, and 1 read/write port for external access. The result from their CACTI model showed a 47.8% reduction in access time for the distributed register file organization across all technologies [44].

It is not clear from this work, whether the programming model for the distributed register file model is sufficiently general for most computations. With a distributed register file

and explicitly routed network, operations must be scheduled by the compiler and routing information must also be generated with code for each processor in order to route results from one processor's register file to another. Although it may be possible to program streaming applications using such a model, in general, concurrency and scheduling can not be defined statically.

Other previous work has described a distributed register file configuration [44] where a fully distributed register file organization is used in a superscalar processor. The architecture exploits a local register mapping table and a dedicated register transfer network to implement this configuration. This architecture requires an extra hardware recopy unit to handle the register file dispatch operations. Also, this architecture suffers from a delay penalty as the execution unit of an instruction that requires a value from a remote register file must stall until it is available. The authors have proposed an eager transfer mechanism to reduce this penalty, but this still suffers from an IPC penalty and requires both central issue logic and global renaming.

In our research, it seems that only the microthreaded model provides sufficient information to implement a penalty-free distributed register file organization. Such a proposal is given in [6] where each processor in a CMP has its own register file in a shared register model. Accesses to remote data is described in the binary code and does not require speculative execution or routing. The decoupling is provided by a synchronization mechanism on registers and the routing is decoupled from the operation of the microthreaded pipeline operation, exploiting the same latency tolerance mechanisms as used for main memory access. Section 4.5 explains in more detail how the microthreaded CMP distributes data to the register files and hides the latency during a remote register file access.

4. THE MICROTHREADED MODEL

4.1. Overview

In this section we consider the microthreaded concurrency model in more detail and describe the features that support the implementation of a scalable CMP based on it. This model was first described in [45], and was then extended in [6, 19, 46] to support systems with multiple processors on-chip.

Like the Tera, this model combines the advantages of BMT and IMT but does so by explicitly interleaving microthreads on a cycle-by-cycle basis in a conventional pipeline. This is achieved using an explicit context switch instruction, which is acted upon in the first stage of the pipeline. Context switching is performed when the compiler can not guarantee that data will be available to the current instruction and is used in conjunction with a synchronization mechanism on the register file that suspends the thread until the data becomes available. The context switch control is not strictly necessary, as this can be signalled from the synchronization failure on the register read. However, it significantly increases the efficiency of the pipeline, especially when a large number of thread suspensions occur together, when the model resembles that of an IMT architecture. Only when the compiler can define a static schedule are instructions from the same thread scheduled in BMT mode. Exceptions to this are cache misses, iterative operations and inter-thread communications. There is one other situation where the compiler will flag a context switch and that is following any branch instruction. This allows execution to proceed non-speculatively, eliminates the branch prediction and cleanup logic and fills any control hazard bubbles with instructions from other threads, if any are active.

The model is defined incrementally and can be applied to any RISC or VLIW instruction set. See Table 1 for details of the extensions to an instruction set required to implement the concurrency controls required by this model. The incremental nature of the model allows a minimal backward compatibility, where existing binary code can execute unchanged on the conventional pipeline, although without any of the benefits of the model to be realized.

Microthreading defines ILP in two ways. *Sets* of threads can be specified where those threads generate MIMD concurrency within a basic block. Each thread is defined by a pointer to its first instruction and is terminated by one or more Kill instructions depending on whether it branches or not. Sets of threads provide concurrency on one pipeline and share registers. They provide latency tolerance through explicit context switching for data and control hazards. *Iterators*, on the other hand, define SPMD concurrency by exploiting a variety of loop structures, including for and while loops. Iterators give parametric concurrency by executing iterations in parallel subject to dataflow constraints. Independent loops have no loop-carried dependencies and can execute with minimal overhead on multiple processors. Dependent loops can also execute on multiple processors, exploiting instruction level concurrency but during the execution of dependency chains activity will move from one processor to another and speedup will not be linear. Ideally dependency chains should execute with minimal latency and parameters for the instruction in Table 1 allow dependencies to be bypassed on interactions executed on a single processor giving the minimal latency possible, i.e. one pipeline cycle per link in the chain.

Iterators share code between iterations and use a set of threads to define the loop body. This means that some form of context must be provided to differentiate multiple iterations executing concurrently. This is achieved by allocating registers to iterations dynamically. A *family* of threads then, is defined by an iterator comprising a triple of *start, step and limit* over a set of threads. Information is also required that defines the micro-context associated with an iteration and, as each iteration is created, registers for its microcontext are allocated dynamically. To create a family of threads a single instruction is executed on one processor, which points to a thread control block (TCB) containing the above parameters. Iterations can

TABLE 1. Concurrency-control instructions.

Instruction	Instruction behaviour
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

then be scheduled on one or more processors as required to achieve the desired performance.

Virtual concurrency on a single pipeline defines the latency that can be tolerated and is limited by the size of the local register file or continuation queue in the scheduler. The latter holds the minimal state associated with each thread. Both are related by the two characteristics of the code; the number of registers per micro-context and the cardinality of the set of threads defining the loop body. In this model, all threads are drawn from the same context and the only state manipulated in the architecture is the thread's execution state, its PC and some information about the location of its micro-context in the local register file. This mechanism removes any need to swap register values on a context switch.

Theoretically, physical concurrency is limited only by the silicon available to implement a CMP, as all structures supporting this model are scalable and are related to the amount of the virtual concurrency required for latency tolerance, i.e. register file, continuation queue and register allocation logic. Practically, physical concurrency will be limited by the extent of the loops that the compiler can generate, whether they are independent or contain loop-carried dependencies and ultimately, the overheads in distribution and synchronization that frame the SPMD execution. Note that thread creation proceeds in a two stages. A conceptual schedule is determined algorithmically on each processor following the creation of a family of microthreads but the actual thread creation, i.e. the creation of entries in the continuation queue, occurs over a period of time at the rate of one thread per cycle, keeping up with the maximum context-switch rate. This continues while resources are available.

Figure 1 shows a simple microthreaded pipeline with five stages and the required shared components used in this model. Notice that no additional stages are required for instruction issue, retiring instructions, or in routing data between processors' register files. Short pipelines provide low latency for global operations. Note that more pipeline stages could be used to reduce the clock period, as is the current trend in microprocessor architecture. However, as concurrency provides the most power-efficient solution to performance it is moot whether this is a sound strategy. Two processors can give the same performance as one double speed processor but do so with less power dissipated. Dynamic power dissipation is proportional to frequency f and V^2, but V can be reduced with f giving a quadratic reduction of energy required by exploiting concurrency in a computation over at least over some range of voltage scaling.

The set of shared components used in this model to support the microthreaded CMP is minimal and their use is infrequent. These components are the *broadcast bus*, used to create a family of threads and a ring network for register sharing. In a Cre instruction a pointer to the TCB is distributed to each processor, where the scheduler will use this information to determine the subset of the iterations it will execute. The broadcast bus is also used to replicate global state to each processor's local register file instead of accessing a centralized register file. Both operations are low in frequency and can be amortized over the execution of multiple iterations.

Replication of global variables is one of two mechanisms that allow the register file in microthreaded model to be fully distributed between the multiple processors on a chip. The other is the *shared-register ring network*, which allows communications between pairs of threads to implement loop-carried dependencies. This communication between the *shared* and *dependent* threads is described in more detail below but uses the ring network only if two iterations are allocated to different processors. Note that schedules can be defined to minimize inter-processor communication, more importantly, this communication is totally decoupled from the pipeline's operation through the use of explicit context switching.

Both types of global communication i.e. the broadcast bus and the ring network are able to use asynchronous communications, creating independent clocking domains for each processor. Indeed, the broadcast mechanisms could be implemented at some level by eager propagation within the ring network.

4.2. Concurrency controls

The microthread model is a generic one, as it can be applied to any ISA, so long as its instructions are executed in-order. In addition, the model can be designed to maintain full backward compatibility, allowing existing binary code to run without speedup [6] on a microthreaded pipeline. Binary compatibility with speedup can also be obtained using binary-to-binary translation to identify loops and dependencies and adding instructions to support the concurrent execution of those loops and/or the concurrency within the basic blocks.

Table 1 shows the five instructions required to support this model on an existing ISA. The instructions create a family of threads, explicitly context switch between threads, kill a thread and two instructions provide for global synchronization. One is a barrier synchronization, the other a form of break instruction, which forces a break from a loop executed concurrently. These concurrency controls provide an efficient and flexible method to extract high levels of ILP from existing code. Each instruction will now be described in more detail.

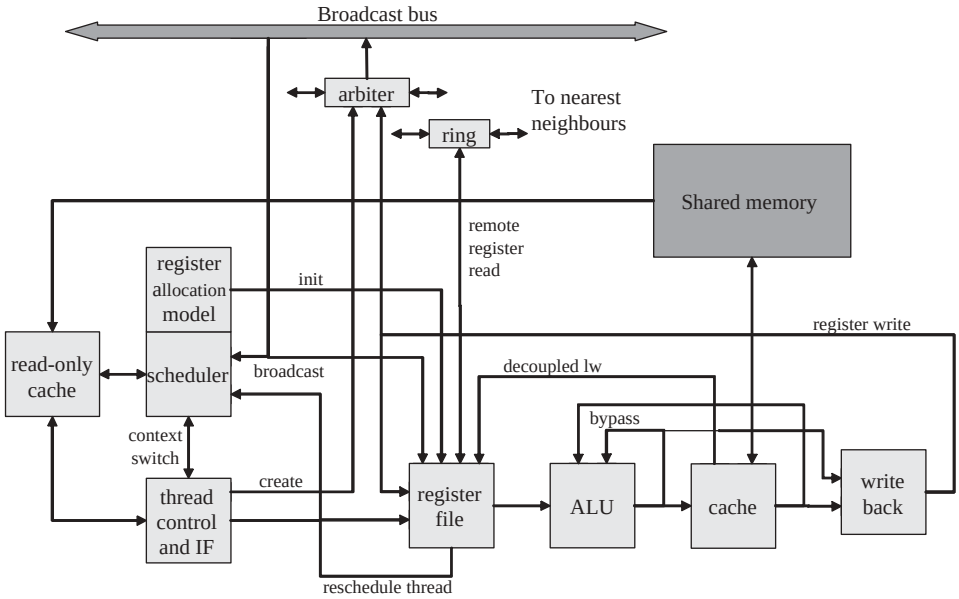

FIGURE 1. Microthreaded microprocessor pipeline.

4.2.1. Thread creation

The microthreaded model defines explicit and parametric concurrency using the *Cre* instruction. This instruction broadcasts a pointer to the TCB and to all processors assigned to the current context; see [47] for details of dynamic processor allocation. The TCB contains parameters, which define a family of threads, i.e the set of threads representing the loop body and the triple defining the loop. It also defines the dynamic resources required by each thread (its microcontext) in terms of local and shared registers. For loops which carry a dependency, it also defines a *dependency distance*, and optionally, pointers to a preamble and/or postamble thread, which can be used to set up and/or terminate a dependency chain. The dependency distance is a constant offset in the index space which defines regular loop-carried dependencies. A family of threads can be created without requiring a pipeline slot, as the create instruction is executed concurrently with a regular instruction in the IF stage of the pipeline. The TCB for our current work on implementation overheads is defined in Table 2.

A global scheduling algorithm determines which iterations will execute on which processors. This algorithm is built into the local scheduler but the parameters in the TCB and the number of processors used to execute the family may be dynamic. The concurrency described by this instruction is therefore parametric and may exceed the resources available in terms of registers and thread slots in the continuation queues. The register allocation unit in each local scheduler maintains the allocation state of all registers in each register file and this controls the creation of threads at a rate of one per pipeline cycle. Once allocated to a processor a thread runs to completion, i.e. until it encounters a Kill instruction and then terminates.

TABLE 2. TCB containing parameters that describe a family of microthreads.

Threads	Cardinality of the set of threads representing an iteration
Dependency	Iteration offset for any loop carried dependencies, e.g. $a[i] := \ldots a[i-d]$
Preambles	Number of iterations using preamble code
Postambles	Number of iterations using postamble code
Start	Start of loop index value
Limit	Limit of loop index value
Step	Step between loop indices
Locals	Number of local registers dynamically allocated per iteration
Shareds	Number of shared registers dynamically allocated per iteration
Pre-pointer	One pointer per thread in set for preamble code
Main-pointer	One pointer per thread in set for main loop–body code
Post-pointer	One pointer per thread in set for postamble code

A terminated thread releases it resources so long as any dependent thread has also terminated. To do so before this may destroy data that has not yet been read by the dependent thread. Note that microthreads are usually (but not exclusively) very short sequences of instructions without internal loops.

4.2.2. Context-switching

The microthreaded context switching mechanism is achieved using the *Swch* instruction, which is acted upon in the first stage of the pipeline, giving a cycle-by-cycle interleaving if

necessary. When a *Swch* instruction is executed, the IF stage reads the next instruction from another ready thread, whose state is passed to the IF stage as a result of the context switch. As this action only requires the IF stage of the pipeline, it can be performed concurrently with an instruction from the base ISA, so long as the *Swch* instruction is pre-fetched with it.

The context switching mechanism is used to manage both control and data dependencies. It is used to eliminate control dependencies by context switching following every transfer of control, in order to keep the pipeline full without any branch prediction. This has the advantage that no instruction is executed speculatively and consequently, power is neither dissipated in making a prediction nor in executing instructions on the wrong dynamic path. Context switching also eliminates bubbles in the pipeline on data dependencies that have non-deterministic timing, such as loads from memory or thread-to-thread communication. Context switching provides an arbitrary large tolerance to latency, determined by the size of the local register file.

4.2.3. Thread synchronization

The only synchronizing memory in the microthreaded model is provided by the registers and this gives an efficient and scalable mechanism for synchronizing data dependencies. The synchronization is performed using two synchronization bits associate with every register, which differentiate between the following states: *full*, *empty*, *waiting-local* and *waiting-remote*.

Registers are allocated to micro-contexts in the *empty* state and a read to an empty register will fail, resulting in a reference to the microthread that issued the instruction being stored in that register. This reference passes down the pipeline with each instruction executed. Using the continuation queue in the scheduler, lists of continuations may be suspended on a register, which is required when multiple threads are dependent on the value to be stored there. All registers therefore implement I-structures in a microthreaded microprocessor. In the *full* state, registers operate normally, providing data upon a register read and, if no synchronization is required, a register can be repeatedly written to without changing its synchronization state to provide backward compatibility. The compiler can easily recognize the potential for a synchronization failure if a schedule for the dependency is not known at compile time. If so, it inserts a context switch on the dependent instruction. Examples include instructions dependent on a prior load word, produced in another thread, or produced in iterative CPU operations.

The register is set to one of the waiting states when it holds a continuation. Two kinds of continuation are distinguished: *waiting-local*, when the register holds the head of a list of continuations to local microthreads and; *waiting-remote*, when the register holds a remote request for data from another processor. The latter enables the micro-context for one iteration to be stored for read-only access on a remote processor when managing loop-carried dependencies. This implements a scalable and distributed shared-register model between processors without using a single, multi-ported register file, which is known to be unscalable.

The use of dataflow synchronization between threads enables a policy of conservative instruction execution to be applied. When no microthreads are active because all are waiting external events, such as load word requests, the pipeline will stall and, if the pipe is flushed completely, the scheduler will stop clocks and power down the processor going into a standby mode, in which it consumes minimal power. This is a major advantage of data-driven models. Conservative instruction execution policies conserve power in contrast to the eager policies used in out-of-order issue pipelines, which have no mechanisms to recognize such a conjunction of schedules. This will have a major impact on power conservation and efficiency.

Context switching and successful synchronization have no overhead in terms of additional pipeline cycles. The context switch interleaves threads in the first stage of the pipeline, if necessary on a cycle by cycle basis. Synchronization occurs at register-read stage and only if it fails will any exceptional action be triggered. On a synchronization failure, control for the instruction is mutated to store a reference to the microthread in the register being read. This means that the only overhead in supporting these explicit concurrency controls is the additional cycle required to reissue the failed instruction when the suspended thread is reactivated by the arrival of the data. Of course there are overheads in hardware but this is true for any model.

The model also provides a barrier synchronization (Bsync) instruction, which suspends the issuing thread until all other threads have completed and a *Brk* instruction, which explicitly kills all other threads leaving only the main thread. These instructions are required to provide bulk synchronization for memory consistency. There is no synchronization on main memory, only the registers are synchronizing. This means that two different microthreads in the same family may not read after write to the same location in memory because the ordering of those operations cannot be guaranteed. It also means that any loop-carried dependencies must be compiled to use register variables. A partitioning of the micro-context supports this mechanism efficiently.

We do not address the design of the shared memory system in this paper, although we are currently investigating several approaches. Indeed the latency tolerance provided by this model makes this design of the memory system somewhat flexible. For example, a large, banked, multi-ported memory would give a solution that would provide all the buffering required for the large number of concurrent requests generated by this model. It is important to note that using in-order processors and a block-based memory consistency model, memory ordering does not pose the same problem as it does in an out-of-order processor.

4.2.4. Thread termination

Thread termination in the microthreaded model is achieved through a *Kill* instruction, which of course causes a context switch as well as updating the microthread's state to killed. The resources of the killed threads are released at this stage, unless there is another thread dependent upon it, in which case its resources will not be released until the dependent thread has also been killed. (Note that this is the most conservative policy and more efficient policies may be implemented that detect when all loop-carried dependencies has been satisfied.)

4.3. Scalable instruction issue

Current microprocessors attempt to extract high levels of ILP by issuing independent instructions out of sequence. They do this most successfully by predicting loop branches and unrolling multiple iterations of a loop within the instruction window. The problem with this approach has already been described; a large instruction window is required in order to find sufficient independent instructions and the logic associated with it grows at least with the square of the issue width.

If we compare this with what is happening in the microthreaded model, we see that almost exactly the same mechanism is being used to extract ILP, with one major difference, a microthreaded microprocessor executes fragments of the sequential programs out-of-order. These fragments (the microthreads) are identified at compile time from loop bodies and conventional ILP and may execute in any order subject only to dataflow constraints. Instructions within fragments however, issue and complete in-order. We have already seen that a context switch suspends a fragment at instructions whose operands have non-deterministic timing. The dependent instruction is issued and stores a pointer to its fragment if a register operand is found to be empty. Any suspended fragments are rescheduled when data is written to the waiting register. Thus only instructions up to the first dependency in each fragment (loop body) are issued and only that instruction will be waiting for the dependency to be resolved; all subsequent instructions in that pipeline will come from other fragments. In an out-of-order issue model the instruction window is filled with all instructions from each loop unrolled by branch prediction because it knows nothing *a priori* about the instruction schedules.

Consider a computation that only ever contains one independent instruction per loop of `l` instructions, then to get `n`-way issue `n` loops must be unrolled and the instruction window will contain `n*l` instructions for each `n` instructions issued. In comparison, the microthreaded model would issue the first `n` independent instructions from `n` threads (iterations), then it would issue the first dependent instructions from the same `n` threads before context switching. The next `n` instructions would then come from the next `n` iterations (threads). Synchronization, instead of taking place in a global structure with $O(n^2)$ complexity, is distributed to `n` registers and has linear complexity. Each thread waits for the dependency to be resolved before being able to issue any new instructions. In effect the instruction window in a microthreaded model is distributed to the whole of the architectural register set and only one link in the dependency graph for each fragment of code is ever exposed simultaneously. Moreover, no speculation is ever required and consequently, if the schedules are such that all processors would become inactive, then this state can be recognized and used to power-down the processors to conserve energy.

Compare this to the execution in an out-of-order processor, where instructions are executed speculatively regardless of whether they are on the correct execution path. Although predictions are generally accurate in determining the execution path in loops, if the code within a loop contains unpredictable, data-dependent branches, this can result in a lot of energy being consumed for no useful work. Researchers now talk about 'breaking the dependency barrier' using data in addition to control speculation but what does this mean? Indices can be predicted readily but these are not true dependencies and do not constrain the microthreaded model; addresses, based on those indices can also be predicted with a reasonable amount of accuracy but again these do not constrain the microthreaded model. This leaves true computational data dependencies, which can only be predicted under very extraordinary circumstances. It seems therefore that there is no justification for consuming power in attempting data speculation.

Out-of-order issue has no global knowledge of concurrency or synchronization. Microthreading, on the other hand, is able to execute conservatively as it does have that global knowledge. Real dependencies are flagged by context switching, concurrency is exposed by dynamically executing parametric *Cre* instructions and the namespace for synchronization spans an entire loop as registers are allocated dynamically. At any instant the physical namespace is determined by the registers that have been allocated to threads.

4.4. Thread state

When a thread is assigned resources by the scheduler, it is initially set to the *waiting* state, as it must wait for its code to be loaded into the I-cache before it can be considered active. A thread will go into a *suspended* state when it has been context switched until either the register synchronization has been completed or the branch target has been defined, when it again goes into the waiting state. The scheduler generates a request to the I-cache to pre-fetch the required code for any thread that enters the waiting state. If the required code is available, then the I-cache acknowledges the scheduler immediately, otherwise not until the required code is in the cache. The thread's state becomes *ready* at this stage. A *killed* state is also required to indicate those threads that have been completed but whose data may still be in use. At any time there is just one thread per processor, which is in the *running* state; on start-up this will be the main thread.

On a context switch or kill, the instruction fetch stage is provided with the state of a new thread if any are active, otherwise the pipeline stalls for a few cycles to resolve the synchronization and if it fails, the pipeline simply stops. This action is simple, requires no additional flush or cleanup logic and most importantly, is conservative in its use of power. Note that by definition, when no local threads are active, the synchronization event has to be asynchronous and hence does not require any local clocks.

The state of a thread also includes its program counter, the base address of its micro-context and the base address and location of any micro-contexts it is dependent upon. The state also includes an implicit slot number, which is the address of the entry in the continuation queue and which uniquely identifies the thread on a given processor. The last field required is a link field, which holds a slot number for building linked lists of threads to identify empty slots, ready queues and an arbitrary number of continuation queues that support multiple continuations on different registers. The slot reference is used as a tag to the I-cache and is also passed through the pipeline and stored in the relevant operand register if a register read fails, where it forms the head of that continuation queue.

4.5. Register file partitioning and distribution

We have already seen that Rixner *et al*. [14] have shown that a distributed register file architecture achieved a better performance compared with a global solution and it also provides superior scaling properties. Their work was based on streaming applications, where register sources and destinations are compiled statically. We will show that such a distributed organization can also be based on extensions to a general-purpose ISA with dynamic scheduling. The concept of a dynamic micro-context associated with parallelizing different iterations has already been introduced and is required in order to manage communications between micro-contexts in a scalable manner. It is necessary for the compiler to partition the micro-context into different windows representing different types of communication and for the hardware to recognize these windows to trap a register read to an appropriate distributed communication mechanism.

A microthreaded compiler must recognize and identify four different types of communication patterns. There are a number of ways in which this partitioning can be encoded, and here we describe a simple and efficient scheme that supports a fully distributed register file based on a conventional RISC ISA, assuming a 5-bit register specifier and hence a 32-register address space per microthread (although, not the same 32 registers for each thread).

The first register window is the global window (represented by \$Gi). These registers are used to store loop invariants or any other data that is shared by all threads. In other models of concurrency these would represent broadcast data, which are written by one and read by many processes. Their access patterns have the characteristics that they are written to infrequently but read from frequently. The address space in a conventional RISC ISA is partitioned so that the lower 16 registers form this global window. These are statically allocated for a given context and every thread can read and/or write to them. Note that the main thread has 32 statically allocated registers, 16 of which are visible to all microthreads as globals and 16 of which are visible only to the main thread. Each thread sees 32 registers. The lower 16 of these are the globals and these are shared by all threads and those in the upper half are local to a given thread.

The upper 16 registers are used to address the microcontext of each iteration in a family of threads. As each iteration shares common code, the address of each micro-context in the register file must be unique to that iteration. As we have seen, the base address of a thread's micro-context forms a part of its state. This immediately gives a means of implementing a distributed, shared-register model. We need to know the processor on which a thread is running and the base address of its microcontext in order to share its data. However, we can further partition the micro-context into a local part and a shared part to avoid too much additional complexity in implementing the pipeline.

Three register windows are mapped to the upper or dynamic half of the address space for each micro-context. These are the local window (\$Li), the shared window (\$Si) and the dependent window (\$Di). Thus the sum of the size of these three windows must be $\leqslant 16$. The local window stores values that are local to a given thread. For example they store values from indexed arrays used only in a single iteration. Reads and writes to the local window are all local to the processor a thread is running on and no distribution of the L window is therefore required. The S and D register windows provide the means of sharing a part of a micro-context between threads. The S window is written by one thread and is read by another thread using its D window.

It should be noted that many different models can be supported by this basic mechanism. In this paper a simple model is described but different models of communication with different constraints and solutions to resource deadlock can be implemented. The mechanism would even support a hierarchy of microcontexts by allowing an iteration in one family of threads to create a subordinate family, where the dynamic part of the address space in the creating family became the static part in the subordinate family. This would support nested multi-dimensional loops as well as breadth first recursion. There are difficulties however, in resolving resource deadlock problems in all but the simplest models and these require further research to resolve.

In this paper we describe a simple model, that supports a single level of loop with communication between iterations being allowed only between iterations that differ by a create-time constant. An example of this type of communication can be found in loop-carried dependencies, where one iteration produces a value, which is used by another iteration. For

example, $A[i] := \ldots A[i-k] \ldots$ where k is invariant of the loop. Such dependencies normally act as a deterrent to loop vectorization or parallelization but this is not so in this model, as the independent instructions in each loop can execute concurrently. This is the same ILP as is extracted from an out-of-order model.

Consider now the implementation of this basic model. It is straightforward to distribute the global register window and its charactistics suggest a broadcast bus as being an appropriate implementation. This requires that all processors executing a family of microthreads be defined prior to any loop invariants being written (or re-written) to the global window. The hardware then traps any writes to the global window and replicates the values using the broadcast bus to the corresponding location in all processors' global windows. As multiple threads may read the values written to the global register window, registers must support arbitrarily large continuation queues, bounded above only by the number of threads that can be active at any time on one processor.

The write to the global window can be from any processor and thus can be used to return a value from an interaction to the global state of a context. The write is also asynchronous and independent of pipeline operation, provided there is local buffering for the data in the event of a conflict on the bus. Contention for this bus should not occur regularly, as writes to globals are generally much less frequent than reads (by a factor proportional to the concurrency of the code). This is analysed later in the paper.

The distribution of S and D windows is a little more complex than the global window. Normally, a producer thread writes to its S window and the consumer reads from its D window, which maps in some sense onto the S window of the producer; we will later return to this. However, there is no restriction on a thread reading a register from its S window so long as data has already been written to it (it would deadlock otherwise). There is also no physical restriction on multiple writes to the S window, although this may introduces non-determinism if a synchronization is pending on it. As far as the hardware is concerned therefore, the S window is identical to the L window, as all reads and writes to it are local and are mapped to the dynamic half of the register-address space. On the other hand, a thread may never write to its D window, which is strictly read-only. The hardware need only recognize reads to the D window in order to implement sharing between two different threads. In order to perform a read from a D window, a processor needs the location (processor id) and base address of the S window of the producer thread. There are two cases to consider in supporting the distribution of register files in the base-level model we have described.

The first and easiest case is when the consumer iteration is scheduled to the same processor as the producer. In this case a read to the D window can be implemented as a normal pipeline read by mapping the D window of the consumer micro-context onto the S window of the producer micro-context. The thread's state must therefore contain the base address of its own micro-context for local reads and also the base address of any micro-context it is dependent upon. In the base-level model we present, only one other micro-context is accessed, at a constant offset in the index space.

In the second case, the producer and consumer iterations are scheduled to different processors. Now, the consumer's read to the D window will generate a remote request to the processor on which the producer iteration is running. Whereas in the first case a micro-context's D window is not physically allocated, in this second case it must be. It is used to cache a local copy of the remote micro-context's S window. It is also used to store the thread continuation locally. The communication is again asynchronous and independent of the pipeline operation. The consumer thread is suspended on its read to the D window location until the data arrives from the remote processor. For this constant strided communication, iteration schedules exist that require only nearest neighbour communication in a ring network to implement the distributed shared-register scheme. Note that a request from the consumer thread may find an empty register, in which case the request gets suspended in the producer's S window until the required data has been produced. Thus a shared-register transaction may involve two continuations, a thread suspended in the D window of the consumer (*waiting-local*) and a remote request suspended in the S window of the producer (*waiting-remote*). As these states are mutually exclusive, the compiler must ensure that the producer thread does not suspend on one of its own S-window locations. This can happen if a load from memory to an S location is also used in the local thread. However, as dependencies are passed via register variables this can only happen in the initialization of a dependency chain. This case can be avoided by loading to a location in the L window when the value is required locally and then copying it to the S window with a deterministic schedule .

The additional complexity required in this distributed register file implementation is 2 bits in each register to encode the four synchronization states: full, empty, waiting-local, waiting-global; a small amount of additional logic to address the dynamically allocated registers using base-displacement addressing; and a simple state machine on each register port to implement the required action based on the synchronization state.

A method has now been described to distribute all classes of communication required in the base-level model. However, we must ensure that this distribution does not require us to implement register files locally that are not scalable. This requires the number of local ports in the register file to be constant. Accesses to L, S and local D windows requires at most two read and one write port for a single-issue pipeline. The G window requires an additional write port independent of the pipeline ports. Finally, reads to a remote D window require one read port and one write port per processor. Contention for this port will depend on the pattern of dependencies, which for the model described is regular and hence evenly distributed with

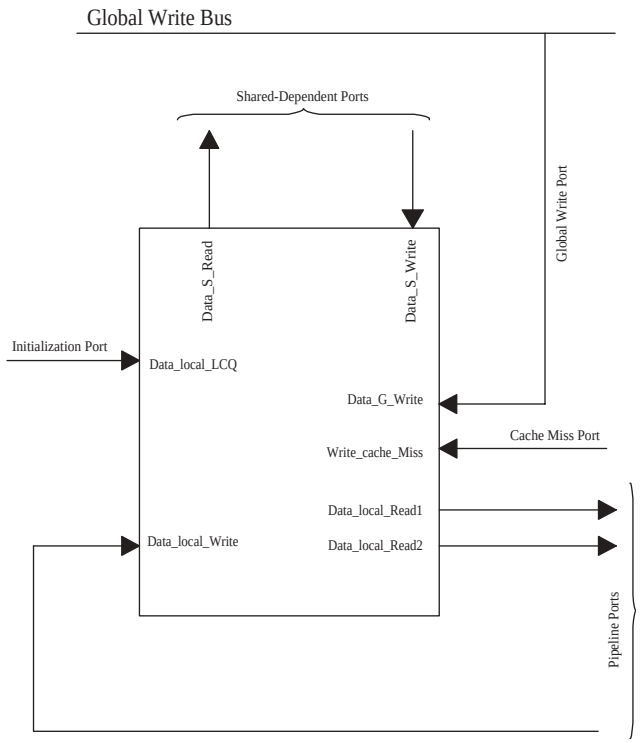

FIGURE 2. Register-file ports analysed.

appropriate scheduling. Each iteration is allocated a separate microcontext in the dynamic half of the register-address space and the first local register ($L0) is initialized by the scheduler to the loop index for that iteration, so this also requires a write port. Finally, a write port is required to support decoupled access to the memory on a cache miss in order to avoid a bubble in the pipeline, when data becomes available.

Figure 2 is a block diagram of the microthreaded register file illustrating these ports. As shown, it has a maximum of eight local ports per processor. The register file could be implemented with just three ports by stalling the pipeline whenever one of the asynchronous reads or writes occurs but this would degrade its performance significantly. An analysis of the accesses to these ports is given in Section 5 below, where we attempt to reduce these eight ports to five using contention, i.e. the three pipeline ports and one additional read and write port for all other cases.

4.5.1. Globally asynchronous locally synchronous communication

Modern synchronous CMP architectures are based on single clock domain with global synchronization and control signals. The control signal distribution must be very carefully designed in order to meet the operation rate on each component used and the larger the chip, the more is the power that is required to distribute these signals. In fact, clock skew, and the large power consumption required to eliminate it, is one of the most significant problems in modern synchronous processor design.

Full asynchronous design is difficult but one promising technique is to use a Globally-Asynchronous, Locally-Synchronous (GALS) clocking scheme [48]. This approach promises to eliminate the global clocking problem and provides a significant power reduction over globally synchronous designs. It divides the system into multiple independent domains, which are independently clocked but which communicate in an asynchronous manner. A GALS system not only mitigates against the clock distribution problem, the problem of clock skew and the resulting power consumption, it can also simplify the reuse of modules as they have asynchronous interfaces that do not require redesign for timing issues when composed [49]. In CMP design, global communication is one of the most significant problems in both current and future systems [6], yet not every system can be decomposed into asynchronously communicating synchronous blocks easily, there must be a clear decoupling of local and remote activity. To achieve this the local activity should not be overly dependent on a remote communication. The model we have described has just this property; each processor is independent and when it does need to communicate with other processors, that communication occurs independently without limiting the local activity. In short, the local processor has tolerance to any latency involved in global communication, as in most circumstances it will have many other independent instructions it can process and, if this is not the case, it will simply switch off its clocks, reduce its voltage levels and wait until it has work to accomplish dissipating minimal power.

The size of the synchronous block in a microthreaded CMP can be from a single processor upwards. The size of this block is a low-level design decision. The issue is that as technology continues to scale this block size will scale down with the problems of signal propagation. Thus the model provides solutions to the end of scaling in silicon CMOS. Compare this with the current approach, which seeks to gain performance by clock speed in a single large wide-issue processor where all strategies are working against the technology.

4.6. Microthreaded CMP architecture

A block diagram of a microthreaded CMP is shown in Figure 3. As shown, N microthreaded pipelines are connected to these two shared communications systems. The first is a broadcast bus, for which there must be contention, although in practice the access to this bus is at a low frequency. It is used by one processor to create a family of threads, it will probably be used by the same processor to distribute any loop invariants and finally, if there is a scalar result, one processor may write values back to global locations. This situation occurs when searching an iteration space—it is the only situation where contention might be required—as a number of processors might find a solution simultaneously and attempt to write to the bus. In

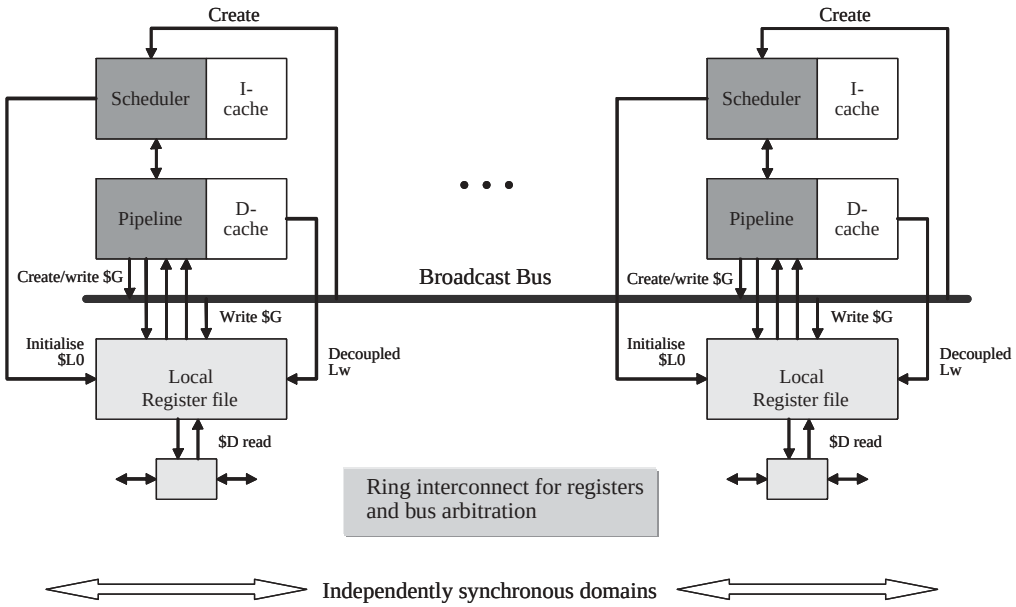

FIGURE 3. Microthreaded CMP architecture, showing communication structures and clocking domains.

this case a break instruction acquires the bus and terminates all other threads allowing the winner to write its results back to the global state of the main context.

The second communication system is the shared-register, ring network, which is used by processors to communicate results along a dependency chain. For the mode described, this requires only local connectivity between independently clocked processors.

All global communication systems are decoupled from the operation of the microthreaded pipeline and thread scheduling provides latency hiding during the remote access. This technique gives a microthreaded CMP a serious advantage as a long-term solution to silicon scaling.

5. ANALYSIS OF REGISTER FILE PORTS

In this section, an analysis of microthreaded register file ports is made in terms of the average number of accesses to each port of the register file in every pipeline cycle. This analysis is based on the hand compilation of a variety of loop kernels. The loops considered included a number of Livermore kernels—some which are independent and some which contain loop-carried dependencies. It also includes both affine and non-affine loops, vector and matrix problems, and a recursive doubling algorithm. We have used loop kernels at this stage as we currently have no compiler to compile complete benchmarks. However, as the model only gains speedup via loops, we have chosen a broad set of representative loops from scientific and other applications. Analysis of complete programs and other standard benchmarks will be undertaken when a compiler we are developing is able to generate microthreaded code.

The results are based on a static analysis of the accesses to various register windows and investigate the average traffic on the microthreaded register file ports. The five types of register file ports are shown in Figure 2 and include, pipeline ports (read-R and write-W), the initialization port (I), the shared-dependent ports (S_d), the broadcast port (B_r) and the write port that is required in the case of a cache miss (W_m). The goal of this analysis is to guide the implementation parameters of such a system. We aim to show that all accesses other than the synchronous pipeline ports can be implemented by a pair of read and write ports, with arbitration between the different sources. In this case a register file with five fixed ports would be sufficient for each of the processors in our CMP design.

The microthreaded pipeline uses three synchronous ports. These ports are used to access three classes of register windows i.e. the $\$L$, $\$S$ and $\$G$ register windows. If we assume that the average number of reads to the pipeline ports in each cycle is R and the average number of writes to the pipeline port in each cycle is W, then these values are defined by the following equations, where N_e is the total number of instructions executed.

$$R = \frac{\sum_{\text{inst.}} (Read(\$L) + Read(\$S) + Read(\$G))}{N_e} \quad (1)$$

$$W = \frac{\sum_{\text{inst.}} (Write(\$L) + Write(\$S) + Write(\$G))}{N_e} \quad (2)$$

The initialization port on other hand is used in register allocation to initialize the $L0 to the loop index. This port is accessed once when each iteration is allocated to a processor and so the average number of accesses to this port is constant and equal to the inverse of the number of instructions executed by the thread

TABLE 3. Average number of accesses to each class of register file port over a range of loop kernels, m = problem size.

Loop	N_e	R	W	I	B_r	S_d
A: Partial Products	$3m$	$\frac{4m-3}{N_e}$	$\frac{2m-1}{N_e}$	0.333	0	$\frac{m-1}{M*N_e}$
B: 2-D SOR	$5m-2$	$\frac{8m-15}{N_e}$	$\frac{4m-4}{N_e}$	0.2	0	$\frac{m-2}{M*N_e}$
L3: Inner Product	$4m+4$	$\frac{5m+3}{N_e}$	$\frac{4m+1}{N_e}$	0.25	0	$\frac{m}{M*N_e}$
L4: Banded Linear Equation	$3m+34$	$\frac{2.4m+37.4}{N_e}$	$\frac{3m+22}{N_e}$	0.2	$\frac{4n}{N_e}$	$\frac{1.8m+1.8}{M*N_e}$
L5: Tri Diagonal Elimination	$5m+3$	$\frac{7m}{N_e}$	$\frac{4m}{N_e}$	0.25	0	$\frac{m-1}{M*N_e}$
L6: General Linear Recurrence	$2.5m+6.5m^{1/2}-5$	$\frac{5.5m+2.5m^{1/2}-5}{N_e}$	$\frac{3m+m^{1/2}-2}{N_e}$	0.1429	$\frac{(m^{1/2}-1)n}{N_e}$	$\frac{0.5m-0.5m^{1/2}}{M*N_e}$
C: Pointer Chasing	$14m+5$	$\frac{9m+3}{N_e}$	$\frac{6m+2}{N_e}$	0.0714	$\frac{n}{N_e}$	$\frac{m}{M*N_e}$
L1: Hydro Fragment	$9m+5$	$\frac{15m}{N_e}$	$\frac{8m+3}{N_e}$	0.1111	$\frac{3n}{N_e}$	0
L2: ICCG	$11m+2\log m-21$	$\frac{17m-5\log m-27}{N_e}$	$\frac{10m-5\log m-12}{N_e}$	0.0909	$\frac{(\log m-1)n}{N_e}$	0
L7: Equation of State Fragment	$26m+5$	$\frac{43m+3}{N_e}$	$\frac{25m+3}{N_e}$	0.0385	$\frac{3n}{N_e}$	0

before it is killed, n_o. Therefore, if I is the average number of accesses to the initialization port per cycle, we can say that:

$$I = \frac{1}{n_o} \tag{3}$$

A dependent read to a remote processor uses a read port on the remote processor and a write port on the local processor, as well as a read to the synchronous pipeline port on the local processor. The average number of accesses to these ports per cycle is dependent on the type of scheduling algorithm used. If we use modulo scheduling, where M consecutive iterations are scheduled to one processor, then interprocessor communication is minimized. An equation for dependent reads and writes is given based on modulo scheduling although we consider it only at the worst case scenario. The average number of accesses per cycle to the dependent window is given below by S_d using the following equation, where M is the number of consecutive threads scheduling to one processor and N_e is the total number of instructions executed. It is clear that the worst case is where $M = 1$, i.e. iterations are distributed one per processor in a modulo manner.

$$S_d = \frac{\sum_{\text{inst.}} Read(\$D)}{M * N_e} \tag{4}$$

The global write port is used to store data from the broadcast bus to the global window in every processor's local register file. If we assume that the average number of accesses per cycle to this port is B_r, then B_r can be obtained from the following equation, where N_e is the total number of instructions executed and n is the number of processors in the system. The result is proportional to the number of processors, as one write instruction will cause a write to every processor in the system.

$$B_r = \frac{\sum_{\text{inst.}} Write(\$G) * n}{N_e} \tag{5}$$

Finally, the frequency of accesses to the port that is required for the deferred register write in the case of a cache miss can also be obtained. It is parameterized by cache miss rate in this static analysis and again we look at the worst case (100% miss rate). The average number of writes per cycle to the cache-miss port is given by W_m, which is given by the formula below where Lw is the number of load instructions in each thread body, n_o is the number of instructions executed per thread body, and C_m is the cache miss rate. Again the average access to this port is constant for a given miss rate.

$$W_m = \frac{\sum_{\text{inst.}} (Lw)}{n_o} * C_m \tag{6}$$

Table 3 shows the average number of accesses to each class of register file port over a range of loop kernels using the above formulae. The first seven kernels are dependent loops, where the dependencies are carried between iterations using registers. The last three are independent loops, where all iterations of the loop are independent of each other.

As described previously, each of the distributed register files has four sources for write accesses in addition to the pipeline

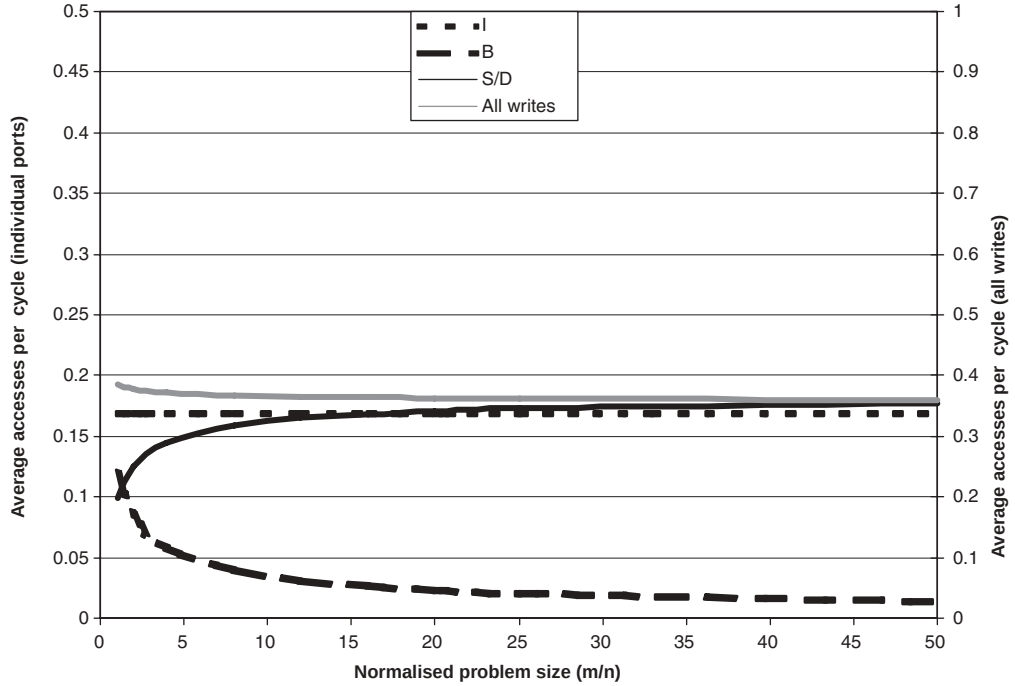

FIGURE 4. Average accesses per cycle on additional ports, $n = 4$ processors.

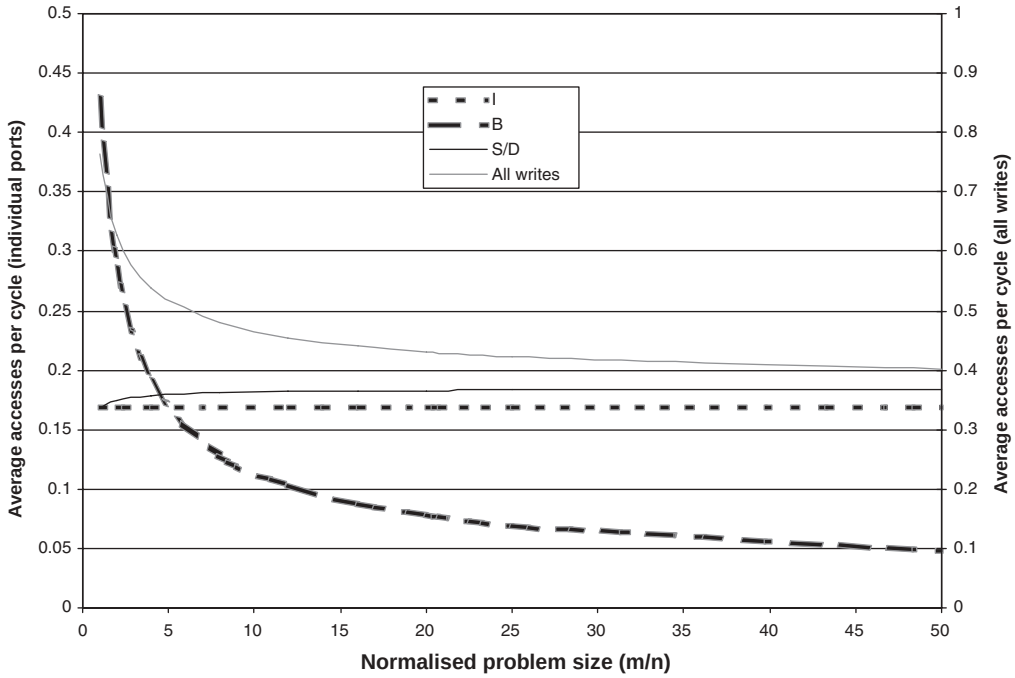

FIGURE 5. Average accesses per cycle on additional ports, $n = 16$ processors.

ports. These are for \$G write, the initialization write, the \$D return data and the write to the port that supports decoupled access to memory on a cache miss. Our analysis shows that the average accesses from these sources is much less than one access per cycle over all analysed loop kernels. This is shown in Figures 4–7 where accesses to initialization (I), broadcast (B_r) and the network ports (S_d, shown as S/D) are given. The four figures illustrate the scalability of the results (from $n = 4$ to

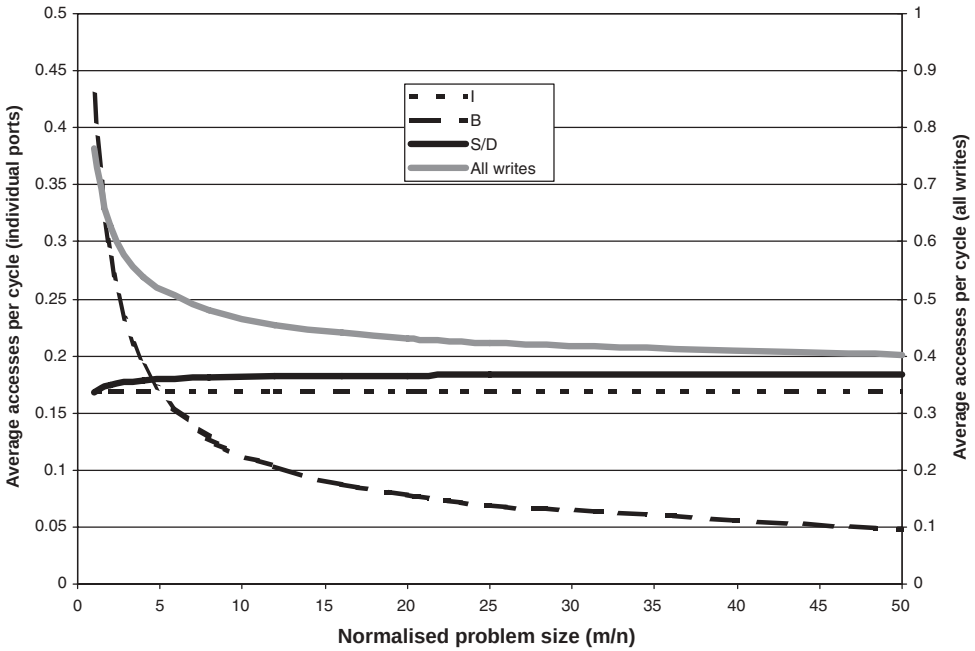

FIGURE 6. Average accesses per cycle on additional ports, $n = 64$ processors.

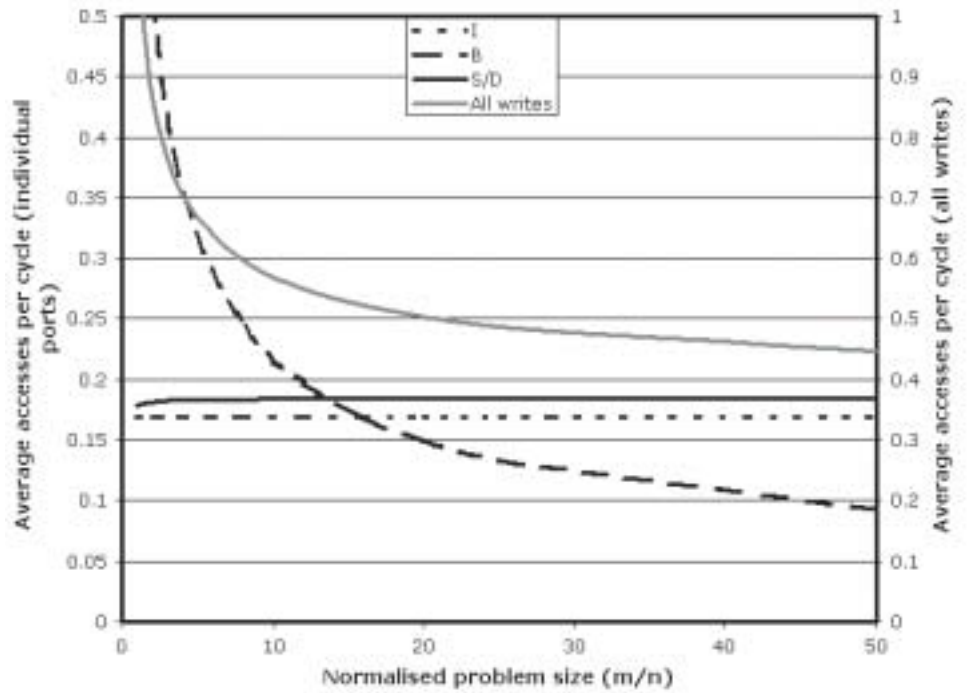

FIGURE 7. Average accesses per cycle on additional ports, $n = 256$ processors.

$n = 256$ processors). Results are plotted against the normalized problem size, where m is the size of the problem in terms of the number of iterations, although not all iterations are executed concurrently in all codes (for example the recursive doubling algorithm has a sequence of concurrent loops varying by powers of 2 from 2 to $m/2$). Normalized problem size is therefore a measure of the number of iterations executed per processor. It can be seen that only accesses from the broadcast bus increase with the number of processors and even this is only significant where few iterations are mapped to each processor. Even in

TABLE 4. Average number of accesses to all additional write ports for different number of processors, $m/n = 8$.

Miss rate (R%)	Average accesses to W_m Port	Average accesses (all write ports) $n = 4$	Average accesses (all write ports) $n = 16$	Average accesses (all write ports) $n = 64$	Average accesses (all write ports) $n = 256$
10	0.038	0.405	0.453	0.517	0.634
20	0.076	0.443	0.491	0.555	0.672
30	0.113	0.480	0.528	0.592	0.709
40	0.151	0.518	0.566	0.630	0.747
50	0.189	0.556	0.604	0.668	0.785
60	0.227	0.594	0.642	0.706	0.823
70	0.265	0.632	0.680	0.744	0.861
80	0.302	0.669	0.717	0.781	0.898
90	0.340	0.707	0.755	0.819	0.936
100	0.378	0.744	0.785	0.857	0.974

the case of 256 processors, providing we schedule more than a few iterations to each processor, the overall number of writes is <50%. Note that a register file of 512 registers supports at least 32 micro-contexts per processor. To put this in perspective, this means that on average a single port sharing all I, B_r and S_d writes would be busy only 50% of the time. There may be peaks in the distribution of writes per cycle, however all of these accesses are asynchronous and they can be queued without stalling the operation of any of the pipelines. This still leaves capacity to include writes from the decoupled-memory accesses.

The analysis of the decoupled-memory port also shows that the average number of accesses per cycle is small. If we assume a cache miss rate of 50%, then the average number of accesses is <20% over all loop kernels. Thus, a single write port would not be fully utilized at this miss rate. For completeness, Table 4 shows the average number of accesses per cycle to all write ports including the W_m port with a variable cache miss rate. This table is compiled for a normalized problem size where the $m/n = 8$, which corresponds to fully utilizing a small register file. Actual cache miss rates for one of the kernels are given in the simulation results presented below.

6. CMP SIMULATION RESULTS

This section gives some preliminary simulation results, which show scalability of performance in executing a single loop kernel. Reference [47] also shows scalability of power for the same simulations. The results are preliminary as they cannot show the relationship between the scalar part of the code and the concurrent part, as a microthreaded compiler is required before we can simulate complete applications. The significant issue however, is that all of these results were obtained from a single binary code that was hand compiled from the Livermore hydro fragment and executed on a microthreaded CMP using from 1 to 2048 processors. The code performs a fixed number of iterations (64K). Clearly this demonstrates the schedule invariance of microthreaded code.

The results are presented as functions of the number of processors on which the code was run and include: the number of cycles to execute the code; the number of instructions executed (note that failed synchronization will cause an increase in the overall number of instructions executed); IPC, which is computed from the previous two; the number of completely inactive cycles, when the pipe is empty and no threads are able to execute; and finally that L1 D-cache hit rate.

All results use identical processors with 1024 registers per processor and a 512 entry continuation queue. The I-cache is also identical in all results, a relatively small 2-Kbyte 8-way set associative cache with 32 byte line size. The first set of results are for a relatively complex level-1 D-cache of 64 Kbyte, with 8-way associativity and 64 byte line size. Figure 8 shows the results. The largest number of instructions executed due to instruction reissue on failed synchronization is <4% and occurred at 256 processors. The maximum IPC was 1613 and represented a speedup of 1602 times the single processor execution and occurred, as expected, with 2048 processors. This is nearly 80% efficient. Indeed the speedup was within 4% of the ideal up to 256 processors. The beginning of saturation in speedup is not unexpected given the fixed problem size. When using 2048 processors, for example, the processor's resources (registers and continuation queue slots) are only partially utilized and hence latency tolerance suffers. Also the fixed overhead of thread creation, including broadcast of the TCB pointer to all processors is amortized over fewer cycles. Note that all simulations are started with cold caches. The overhead is represented by the relatively static 100 or so cycles during which the processors are inactive. As the solution time approaches this, the percentage overhead will increase rapidly. Finally it can be seen that the cache hit rate varies between 75 and 95%. The best hit rate is seen on one processor but the worst occurs on ∼64 processors, where the resources start to become less efficiently used. Note that the local schedules

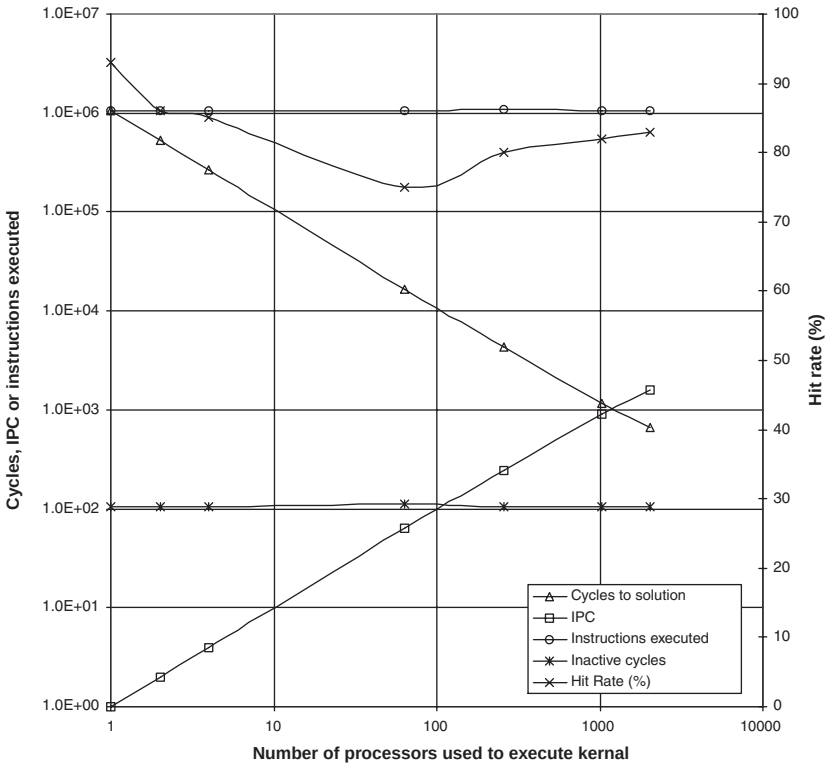

FIGURE 8. Characteristics of microthreaded kernel execution (64K D-cache).

were set to match the cache line size and hence maximize hit rate.

The second set of results was undertaken to illustrate what effect, if any, the D-cache had on performance. In this set, the D-cache parameters were altered drastically to give a residual cache of just 1 Kbyte with direct mapped cache lines (n.b. the register file size is a substantial larger than this at 8 Kbyte). Figure 9 shows these results. It can be seen that number of instructions executed, time to solution and IPC are all virtually unchanged. The only significant change is in the cache hit rate, which is now much worse, varying from 40 up to 88% on a single processor. Indeed the minimal time to solution and maximum IPC by a small amount were observed with 2048 processors using the residual D-cache.

We note that there are some caveats to these results. They are obtained from the execution of a single independent code kernel. Simulations of dependent loops show that speedup saturates, with the maximum speedup being determined by the ratio of independent to dependent instructions in these kernels. For example, a simulation of a naive multiply-accumulate implementation of matrix multiplication (a dependent loop) saturated with a speedup of between 3 and 4 and was achieved using only four processors. This represents the maximum instruction parallelism of the combined iterations. The results also ignore the scalar component of the program, which we cannot effectively evaluate until we have fully developed a microthreaded compiler. Finally, the memory model used in these simulations is non-blocking, as we do not have an realistic model fully simulated yet. The results use a pipelined memory with an 8-cycle start-up for first word and 2-cycles per word to complete a cache line transfer. Note that this limitation is not a major issue with the kernel simulated as data can be distributed in memory according to the local schedule and blocking would be unlikely. In the 64 Kbyte cache only 3% of memory accesses cause a request to second level memory. The remaining cache misses are same-line misses due to the regularity of scheduling. We are currently working on a second level memory architecture and will present full simulation results of this when we have a working compiler. The results presented here, however, show great promise for this approach.

7. CONCLUSIONS

The characteristics of advanced integrated circuits (ICs) will in future require powerful and scalable CMP architectures. However, current techniques like wide-issue, superscalar processors suffer from complexity in instruction issue and in the large multi-ported register file required. The complexity of these components grows at least quadratically with increasing issue width; also, execution of instructions using these techniques must proceed speculatively, which does not always

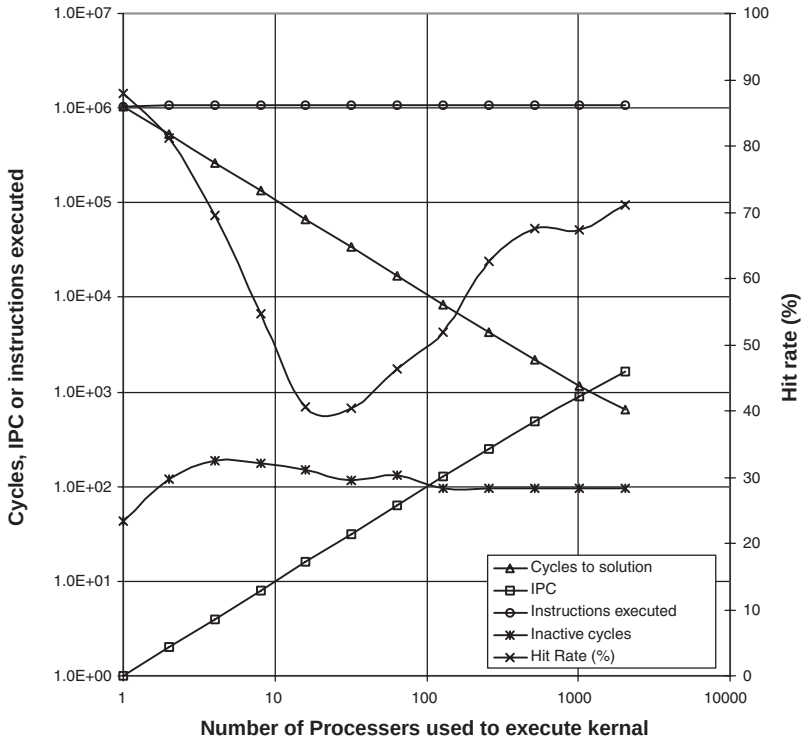

FIGURE 9. Characteristics of microthreaded kernel execution (1K D-cache).

provide results for the power consumed. In addition, more on-chip memory is required in order to ameliorate the effects of the so called 'memory wall'. These obstacles limit the processor's performance, by constraining parallelism or through having large and slow structures. In short, this approach does not provide scalability in a processor's performance, in the on-chip area and power dissipation.

An alternative solution which eliminates this complexity in instruction issue and the global register file, and avoids speculation has been presented in this paper. The model is based on decomposing a sequential program into small fragments of code called microthreads, which are scheduled dynamically and which can communicate and synchronize with each other very efficiently. This process allows sequential code to be compiled for execution on scalable CMPs. Moreover, as the code is schedule invariant, the same code will execute on any number of processors limited only by problem size. The model exploits ILP within basic blocks and across loop bodies. In addition, this approach supports a pre-fetching mechanism that avoids many instruction-cache misses in the pipeline. The fully distributed register file configuration used in this approach has the additional advantage of full scalability in a CMP with the decoupling of all forms of communication from the pipeline's operation. This includes memory accesses and communication between micro-contexts.

The distributed implementation of a microthreaded CMP includes two forms of asynchronous communication. The first is the broadcast bus, used for creating threads and distributing invariants. The second is the shared-register ring network used to perform communication between the register files in the producer and consumer threads. The asynchronous implementation of the bus and switch provides many opportunities for power saving in large CMP systems. The decoupled approach to register-file design avoids a centralized register file organization and, as we have shown, requires a small, fixed number of ports to each processor's register file, regardless of the number of processors in the system.

An analysis of the register-file ports in terms of the frequency of accesses to each logical port is described in this paper. This analysis involved different types of dependent and independent loop kernels. The analysis illustrates a number of interesting issues, which can be summarized as follows:

- A single write port with arbitration between different sources is sufficient to support all non-pipeline writes. This port has an average access rate of $<100\%$ over normal operating conditions. This is true even in the case of a 100% cache-miss rate.
- A second port is required to handle reads to the \$D window. The analysis shows that the average access to this port is $<10\%$ over all analysed loop kernels.
- As a consequence, the distributed register files require only five ports per processor and these ports are fixed regardless of the number of processors in the system. This

provides a scalable and efficient solution for large number of processors on-chip.

- Finally, the average accesses to all write ports does not exceed 100% even in the case of $n = 256$-processor. However, to deal with a large number of processors, the performance would degrade gracefully due to the inherent latency tolerance of the model. Eventually all threads would be suspended waiting for data and in this case the stalled pipeline(s) would free up contention to the non-pipeline write port.

Finally we present results of the simulation of an independent loop kernel, that clearly demonstrate schedule invariance of the binary code and linear speedup characteristics over a wide range of processors on which the kernel is scheduled. Clearly, a microthreaded CMP based on a fully distributed and scalable register file organization and asynchronous global communication buses is a good candidate to future CMP.

REFERENCES

[1] Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, S. and Verghese, B. (2000) Piranha: a scalable architecture based on single-chip multiprocessing. In *Proc. 27th Annual Int. Symp. Computer Architecture*, Vancouver, British Columbia, Canada, June 12–14, pp. 282–293. ACM Press, New York, NY.

[2] Hammond, L., Hubbert, B. A., Siu, M., Prabhu, M. K., Chen, M. and Olukotun, K. (2000) The Stanford Hydra CMP. *IEEE Micro*, **20**, 71–84.

[3] Hammond, L., Nayfah, B. A. and Olukotun, K. (1997) A single-chip multiprocessor. *IEEE Comput. Soc.*, **30**, 79–85.

[4] Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H. and Sinharoy, B. (2002) Power4 System Micro-architecture. *IBM J. Res. Develop.*, **46**, 5–25.

[5] Tremblay, M., Chan, J., Chaudhry, S., Conigliaro, A. W. and Tse, S. S. (2000) The MAJC architecture: a synthesis of parallelism and scalability. *IEEE Micro*, **20**, 12–25.

[6] Jesshope, C. R. (2004) Scalable instruction-level parallelism. In *Proc. Computer Systems: Architectures, Modeling and Simulation, 3rd and 4th Int. Workshops, SAMOS 2004*, Samos, Greece, July 19–21, *LNCS* **3133**, pp. 383–392. Springer.

[7] Bhandarkar, D. (2003) Billion transistor chips in mainstream enterprise platforms of the future. In *Proc. 9th Int. Symp. High-Performance Computer Architecture*, Anaheim, CA, February 8–12, pp. 3. IEEE Computer Society, Washington, DC.

[8] Agarwal, V., Hrishikesh, M. S., Keckler, S. W. and Burger, D. (2000) Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proc. 27th Annual Int. Symp. Computer Architecture*, Vancouver, British, Columbia, Canada, June 10–14, pp. 248–259. ACM Press, New York, NY.

[9] Onder, S. and Gupta, R. (2001) Instruction wake-up in wide issue superscalars. In *Proc. 7th Int. Euro-Par Conf. Manchester on Parallel Processing*, Manchester, UK, August 28–31, pp. 418–427. Springer-Verlag, London, UK.

[10] Onder, S. and Gupta, R. (1998) Superscalar execution with dynamic data forwarding. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, Paris, France, October 12–18, pp. 130–135. IEEE Computer Society, Washington, DC.

[11] Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K. and Chang, K. (1996) The case for a single-chip multiprocessor. In *Proc. Seventh Int. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, Cambridge, MA, October 1–5. Cambridge, MA, September, pp. 2–11. ACM Press, New York, NY.

[12] Palacharla, S., Jouppi, N. P. and Smith, J. (1997) Complexity-effective superscalar processors. In *Proc. 24th Int. Symp. Computer Architecture*, Denver, CO, June 1–4, pp. 206–218. ACM Press, New York, NY.

[13] Tullsen, D. M., Eggersa, S. and Levy, H. M. (1995) Simultaneous multithreading: maximizing on chip parallelism. In *Proc. 22nd Annual Int. Symp. Computer Architecture*, Santa Margherita Ligure, Italy, June 22–24, pp. 392–403. ACM Press, New York, NY.

[14] Rixner, S., Dally, W. J., Khailany, B., Mattson, P. R., Kapasi, U. J. and Owens, J. D. (2000) Register organization for media processing. In *Proc. Int. Symp. High Performance Computer Architecture*, Toulouse, France, January 8–12, pp. 375–386. IEEE CS Press, Los Alamitos, CA.

[15] Balasubramonian, R., Dwarkadas, S. and Albonesi, D. (2001) Reducing the Complexity of the register file in dynamic superscalar processors. In *Proc. 34th Int. Symp. on Micro-architecture*, Austin, TX, December 1–5, pp. 237–248. IEEE Computer Society, Washington, DC.

[16] Diefendorff, K. and Duquesne, Y. (2002) Complex SOCs require new architectures. *EE Times*. Available at http://www.eetimes.com/issue/se/OEG20020911S0076.

[17] Ungerer, T., Robec, B. and Silc, J. (2003) A survey of processors with explicit multithreading. *ACM Comput. Surveys*, **35**, 29–63.

[18] Burns, J. and Gaudiot, J.-L. (2001) Area and system clock effects on SMT/CMP processors. In *Proc. 2001 Int. Conf. Parallel Architectures and Compilation Techniques*, Barcelona, Spain, September 8–12, pp. 211–218. IEEE Computer Society, Washington, DC.

[19] Jesshope, C. R. (2003) Multi-threaded microprocessors evolution or revolution. In *Proc. 8th Asia-Pacific Conf. ACSAC'2003*, Aizu, Japan, September 23–26, *LNCS* **2823**, pp. 21–45. Springer, Berlin, Germany.

[20] Luo, B. and Jesshope, C. R. (2002) Performance of a micro-threaded pipeline. In *Proc. 7th Asia-Pacific Conf. Computer Systems Architecture*, Melbourne, Victoria, Australia, January 28–February 2, pp. 83–90. Australia Computer Society, Inc. Darlinghurst, Australia.

[21] Jesshope, C. R. (2001) Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. In *Proc. ACSAC 2001*, Gold Coast, Queensland, Australia, January 29–30, pp. 80–88. IEEE Computer Society, Los Alamitos, CA.

[22] Zhou, H. and Conte, T. M. (2002) *Code Size Efficiency in Global Scheduling for VLIW/EPIC Style Embedded Processors*. Technical Report, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC.

[23] Hwang, K. (1993) *Advanced Computer Architecture*. MIT and McGraw-Hill, New York, St Louis, San Francisco.

[24] Sudharsanan, S., Sriram, P., Frederickson, H. and Gulati, A. (2000) Image and video processing using MAJC 5200. In *Proc. 2000 IEEE Int. Conf. Image Processing*, Vancouver, BC, Canada, September 10–13, pp. 122–125. IEEE Computer Society, Washington, DC.

[25] Cintra, M. and Torrellas, J. (2002) Eliminating squashes through learning cross-thread violations in speculative parallelisation for multiprocessors. In *Proc. 8th Int. Symp. High-Performance Computer Architecture*, Boston, MA, February 2–6, pp. 43–54. IEEE Computer Society, Washington, DC.

[26] Cintra, M. Martinez, J. S. and Torrellas, J. (2000) Architecture support for scalable speculative parallelization in shared-memory multiprocessors. In *Proc. Int. Symp. Computer Architecture*, Vancouver, Canada, June 10–14, pp. 13–24. ACM Press, New York, NY.

[27] Terechko, A., Thenaff, E. L., Garg, M. J., Van Eijndhoven, J. V. and Corporaal, H. (2003) Inter-cluster communication models for clustered VLIW processors. In *Proc. 9th Int. Symp. High-Performance Computer Architecture*, Anaheim, CA, February 8–12, pp. 354–364. IEEE Computer Society, Washington, DC.

[28] Halfhill, T. (1998) Inside IA-64. *Byte Magaz.*, **23**, 81–88.

[29] Schlansker, M. S. and Rau, B. R. (2000) EPIC: an architecture for instruction-level parallel processors. Compiler and Architecture Research, HPL-1999-111. HP Laboratories, Palo Alto.

[30] Sundararaman, K. and Franklin, M. (1997) Multiscalar execution along a single flow of control. In *Proc. IEEE Int. Conf. Parallel Processing*, Bloomington, IL, August 11–15, pp. 106–113. IEEE Computer Society, Washington, DC.

[31] Sohi, G. S., Breach, S. E. and Vijaykumar, T. N. (1995) Multiscalar processors. In *Proc. 22nd Annual Int. Symp. Computer Architecture*, S. Margherita Ligure, Italy, June 22–24, pp. 414–425. ACM Press, New York, NY.

[32] Breach, S. E., Vijaykumar, T. N. and Sohi, G. S. (1994) The anatomy of the register file in a multiscalar processor. In *Proc. 27th Int. Symp. Microarchitecture*, San Jose, CA, November 30–December 2, pp. 181–190. ACM Press, New York, NY.

[33] Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A. and Smith, B. (1990) The Tera computer system. In *Proc. 4th Int. Conf. Supercomputing*, Amsterdam, The Netherlands, June 11–15, pp. 1–6. ACM Press, New York, NY.

[34] Kongetira, P., Aingaran, K. and Olukotun, K. (2005) Niagara: 32-way multithreaded Sparc processor. *IEEE Comput. Soc.*, **25**, 21–29.

[35] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A. and Upton, M. (2002) Hyper-threading technology architecture and microarchitecture. *Intel Technol. J.*, **6**, 4–15.

[36] Emer, J. (1999) Simultaneous multithreading: multiple Alpha's performance. In *Presentation at the Microprocessor Forum'99*, MicroDesign Resources, San Jose, CA.

[37] Codrescu, L., Wills, D. S. and Meindl, J. D. (2001) Architecture of the Atlas Chip Multiprocessor: dynamically parallelising irregular applications. *IEEE Comput. Soc.*, **50**, 67–82.

[38] Diefendorff, K. (1999) Power4 focuses on memory bandwidth: IBM confronts IA-64, says ISA not important. *Microprocessor Rep.*, **13**, 11–17.

[39] Preston, R. P. *et al*. (2002) Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *Proc. 2002 IEEE Int. Solid-State Circuits Conf.*, San Francisco, CA, February 4–6, pp. 334–335. IEEE Solid-State Circuits, USA.

[40] Scott, L., Lee, L., Arends, J. and Moyer, B. (1998) Designing the low-power M-CORE architecture. In *Proc. IEEE Power Driven Micro Architecture Workshop at ISCA98*, Barcelona, Spain, June 28, pp. 145–150.

[41] Park, I., Powell, M. D. and Vijaykumar, T. N. (2002) Reducing register ports for higher speed and lower energy. In *Proc. 35th Annual ACM/IEEE Int. Symp. Microarchitecture*, Istanbul, Turkey, November 18–22, pp. 171–182. IEEE Computer Society, Los Alamitos, CA.

[42] Kim, N. S. and Mudge, T. (2003) Reducing register ports using delayed write-back queues and operand pre-fetch. In *Proc. 17th Annual Int. Conf. Supercomputing*, San Francisco, CA, June 23–26, pp. 172–182. ACM Press, New York, NY.

[43] Tseng, J. H. and Asanovic, K. (2003) Banked multiported register files for high-frequency superscalar microprocessors. In *Proc. 30th Int. Symp. Computer Architecture*, San Diego, CA, June 9–11, pp. 62–71. ACM Press, New York, NY.

[44] Bunchua, S., Wills, D. S. and Wills, L. M. (2003) Reducing operand transport complexity of superscalar processors using distributed register files. In *Proc. 21st Int. Conf. Computer Design*, San Jose, CA, October 13–15, pp. 532–535. IEEE Computer Society, Los Alamitos, CA.

[45] Bolychevsky, A., Jesshope, C. R. and Muchnick, V. (1996) Dynamic Scheduling in RISC Architectures. *IEE Proc. Comput. Digit. Tech.*, **143**, 309–317.

[46] Jesshope, C. R. (2005) Micro-grids—the exploitation of massive on-chip concurrency. In *Proc. HPC Workshop 2004, Grid Computing: A New Frontier of High Performance Computing*, L. Grandinetti (ed.), Cetraro, Italy, May 31–June 3. Elsevier, Amsterdam.

[47] Bousias, K. and Jesshope, C. R. (2005) The challenges of massive on-chip concurrency. *Tenth Asia-Pacific Computer Systems Architecture Conference*, Singapore, October 24–26. *LNCS* **3740**, pp. 157–170. Springer-Verlag.

[48] Shapiro, D. (1984) Globally Asynchronous Locally Synchronous Circuits. PhD Thesis, Report No. STAN-CS-84-1026, Stanford University.

[49] Shengxian, Z., Li, W., Carlsson, J., Palmkvist, K. and Wanhammar, L. (2002) An asynchronous wrapper with novel handshake circuits for GALS systems. In *Proc. IEEE 2002 Int. Conf. Communication, Circuits and Systems*, Cheungdu, China, June 29–July 1, pp. 1521–1525. IEEE Society, CA.