Int J Parallel Prog (2008) 36:184–205
DOI 10.1007/s10766-007-0062-1

Dynamic Instruction Scheduling in a Trace-based Multi-threaded Architecture

Peter A. Rounce · Alberto F. De Souza

Received: 1 November 2006 / Accepted: 2 April 2007 / Published online: 24 January 2008
© Springer Science+Business Media, LLC 2008

Abstract Simulation results are presented using the hardware-implemented, trace-based dynamic instruction scheduler of our single process DTSVLIW architecture to schedule instructions from several processes into multiple streams of VLIW instructions for execution by a wide-issue, simultaneous multi-threading (SMT) execution engine. The scheduling process involves single instruction execution of each process, dynamically scheduling executed instructions into blocks of VLIW instructions cached for subsequent SMT execution: SMT provides a mechanism to reduce the impact of horizontal and vertical waste, and variable memory latencies, seen in the DTSVLIW. Preliminary experiments explore this extended model. Results achieve PE utilization of up to 87% on a 4-thread, 1-scalar, 8 PE design, with speed-ups of up to 6.3 that of a single processor. Noticeably it only needs a single scalar process to be scheduled at any time, with main memory fetches being 1–4% that of a single processor.

Keywords Simultaneous multi-threading · Dynamic instruction scheduling · Wide issue architectures · VLIW

1 Introduction

A key issue for wide issue processors, such as superscalar and very Long Instruction word (VLIW) architectures, is keeping their many processing elements (PEs) fully

P. A. Rounce (✉)
Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK
e-mail: p.rounce@cs.ucl.ac.uk

A. F. De Souza
Departamento de Informática, Universidade Federal do Espírito Santo, Av. Fernando Ferrari, 514, Vitoria 29075-910, ES, Brazil
e-mail: albertodesouza@gmail.com

supplied with instructions to execute. Superscalar architectures have been very successful, but have required complex speculation and branch prediction hardware to use any significant proportion of their PEs. A superscalar processor parallelizes the code on route from the fetch logic to the processing elements. This offers benefits in that it allows for adjustment of the execution sequence to immediate execution-time effects (long latency operations, resource constraints), but makes the fetch-issue logic very complex. The achievement of good performance despite frequent conditional branches, that would otherwise stall the fetch-issue logic and reduce parallelism, requires speculative execution across these branches with complex branch prediction to reduce the impact of branch miss-predictions. The last seriously impacts on performance through vertical wastage (processor cycles where no instructions can be issued) from the long fetch-issue pipeline. Thus, the simultaneous multithreading of superscalars is no longer seriously considered because of their complexity [1], although they are of interest as single-threaded processors within Chip Multiprocessors [2] (CMPs). In pure VLIW systems, the compiler has complete responsibility for creating a package of operations that can be simultaneously issued to the VLIWs many PEs. Their processors do not dynamically make any decisions about multiple operation issue, and thus they are simple and fast. Much work has been done on compiler technology so as to try and generate highly parallel code that will keep the PEs of a pure VLIW busy. Even so, it is very difficult to prevent horizontal wastage, i.e. empty slots in the VLIW long instruction provided to the PEs. These architectures are also unable to adjust to variable latencies in instruction execution that differ from those presumed by the compiler, e.g. misses in their D-caches, leading to processor stalls (*vertical wastage*). Unlike superscalars, pure VLIW architectures also suffer from *the VLIW object-code compatibility problem*: code generated for a 4-functional unit VLIW processor cannot run on a 3-functional unit VLIW processor without recompilation.

A range of modified VLIW architectures have been developed to deal with the different problems of VLIW architectures. The EPIC architecture [3] is a more elaborate form of VLIW architecture where, although the responsibility of detecting and extracting the ILP lies with the compiler, improvements to the hardware enable better exploitation of the available ILP. In EPIC systems, the compiler essentially uses a model of unlimited parallelism that builds groups of instructions with an unlimited number of instructions that can be executed in parallel. The instructions of these groups are allocated into *bundles* with a fixed number of instructions: a bundle may contain instructions from more than one group and has extra bits that identify the boundaries between concurrently executable instructions. The EPIC processor reads bundles sequentially from memory and uses the boundary information to identify instructions that can be issued concurrently. The bundling and unbundling process allows the compilation width and the instruction fetch width to be decoupled from the VLIW processing engine width, so different hardware can use the same program code without recompilation. The Weld architectural model [4,5] is a modified VLIW model that executes statically compiled code (compiler created), but code in which the compiler has identified sub-threads that can be speculatively executed by the processor in parallel to the main thread. When the hardware identifies that such a sub-thread can be executed, a thread is spawned off to execute concurrently. This sub-thread is merged back into the main thread, if the main thread reaches the address of the

sub-thread otherwise the results are quashed. The Weld model is attempting to use more of the available PEs by providing multiple threads to supply more instructions to the processor engine. Weld is a single process, simultaneous multi-threading architecture (SMT) [6,7] with each thread having its own register file, that aims to reduce the process's execution time. The results presented in the Weld Papers [4,5] indicate that in many cases most of the performance increase is achieved with just 2 threads, and having more threads is not cost effective: the 2001 paper [4] gives an average speedup of 27% over a single-threaded VLIW architecture. Weld dynamically adjusts the instruction execution sequence supplied to the PEs depending on the program execution sequence, but using statically defined code sequences. The DSVLIW [8] (named the dynamically scheduled VLIW by Rau) also executed statically compiled code, but split instructions into two parts with the first part generating the result value placed in a renaming register, that is later copied into the target destination by the 2nd part. Both parts are issued to reservations stations so that out-of-order execution can occur and the architecture adjust to variable latencies.

A number of architectures perform dynamic code scheduling of the input code stream to identify concurrent code sequences, as "a schedule created at run-time is often better than one created at compile time"[9]. Thus DIF [10], DTSVLIW [11–16], and rePLay [9] architectures are all single threaded ones that do dynamic code scheduling on a single process. The DIF and DTSVLIW architectures both schedule a scalar instruction stream into VLIW long words using essentially the same concept, while rePLay uses a static scheduler to produce the source binary code, which is then optimized by a scheduler during code execution. We believe the DTSVLIW architecture to be simpler than that of DIF and easier to implement. Work on the DIF appears to have ceased, while it is not clear how rePLay could be extended to multi-threading: the rePLay paper [9] states that the scheduler can be hardware or software based, and the hardware scheduler takes 10 clock cycles for each instruction scheduled, which indicates that the scheduler does not lie in the main execution path. In the DTSVLIW, scheduling occurs in a scalar mode of execution with the scheduler designed to process 1 instruction per cycle, although this may not be achieved consistently because of delays in the arrival of instructions at the scheduler due to latency issues elsewhere. In the rest of the paper we concentrate on the DTSVLIW architecture and its multi-threading extension, the mDTSVLIW [8].

The mDTSVLIW extension to the DTSVLIW architecture uses the scheduling logic to schedule instructions from several processes to produce parallelized code for several threads, and the code of these threads is later combined to produce a single simultaneous multithreaded instruction stream for a wide issue processor. A key feature of the design is that it does not depend on a large instruction bandwidth from the main memory.

In the mDTSVLIW and the rest of this paper, the execution engine is VLIW, but the instruction stream scheduling logic is not bound to VLIW architectures: it could as well provide a parallel instruction stream to the issue logic and execution engine of a superscalar.

2 Single-threaded DTSVLIW Architecture

The idea behind the DTSVLIW architecture is to execute programs in two phases: one sequential, the other parallel. The first phase occurs when a fragment of code is first seen during execution, the second when the same fragment of code needs to be re-executed. In the sequential phase, instructions are fetched from the instruction cache and executed by a simple pipelined processor. Concurrently, they are scheduled into VLIW long instructions and saved in blocks in a VLIW instruction cache. In the parallel phase, the scheduled VLIW instructions are fetched from this VLIW cache and executed by a VLIW processor.

Figure 1 shows a block diagram of the DTSVLIW architecture. The Primary Processor is a simple pipelined processor, which fetches and executes instructions from the I-Cache to produce the trace for the Scheduler Unit. This schedules the trace into blocks of long instructions, which are stored in the VLIW Cache. On each execute cycle the address of the instruction to be executed is used to look-up the VLIW cache. On a hit, the block hit is executed by the VLIW engine. At the end of a block, or on a branch out of the block, further hits may occur on the VLIW cache, so that control may stay within the VLIW Engine. At this time, the primary processor is unused and no accesses are made to the I-Cache. The Scheduling Unit essentially performs superblock scheduling [17,18]. In the DTSVLIW, the Scheduler Engine provides object-code compatibility, and the VLIW Engine provides VLIW performance and simplicity. To execute code in two distinct modes, one sequential and one parallel, results in four positive characteristics. First, it gives code compatibility between different machine generations. Second, complex instructions can be dealt with in sequential mode by the Primary Engine or it is possible to decompose them into several simpler operations and to schedule them into VLIW code. Third, the task of finding parallelism is simplified as the Scheduler Unit receives no more than one instruction per cycle and, therefore, can have a simple and fast hardware implementation. Fourth, instruction exceptions can be dealt with in sequential mode: on an exception during parallel execution, a DTSVLIW machine switches to sequential mode to deal with it. However, to take

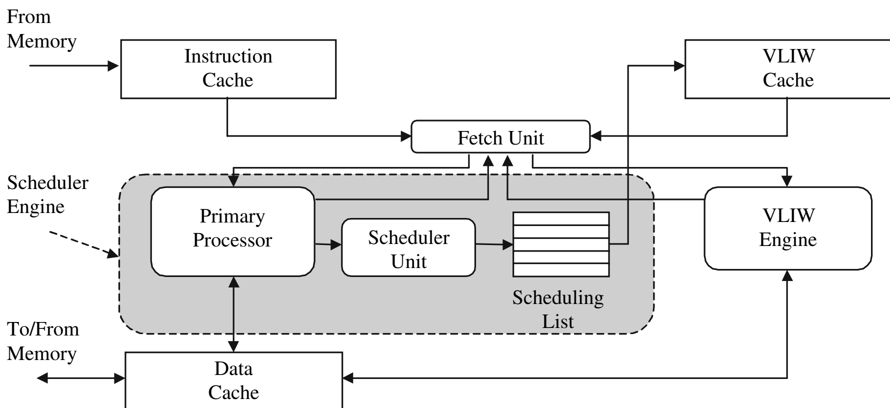

Fig. 1 The Dynamically Trace Scheduled VLIW (DTSVLIW) architecture

advantage of these characteristics, a DTSVLIW machine has to reuse the blocks of VLIW instructions saved in the VLIW Cache many times.

3 Research Motivation

Previous simulation work by the authors and others have demonstrated that the DTSVLIW can provide parallel performance similar to other enhanced VLIW architectures and superscalar architectures. Figure 2 shows results presented in our 2000 paper [12] of the effect of varying the number of instructions per long instruction and the number of long instructions per cache block on the DTSVLIW performance, measured as the average number of instructions executed per cycle for programs from the SPEC95 benchmark suite (www.spec.org). The VLIW cache size is held constant. In the legend of Fig. 2, "4 8" means 4 instructions per VLIW instruction and 8 long instructions per block. The research presented in the 2000 paper, besides demonstrating that the DTSVLIW works, also shows a common problem of processors that exploit ILP, both VLIW-derived and superscalar-derived ones, that inefficient use is made of the available parallelism. The average ILP found across the benchmarks in Fig. 2 varies from about two, with four processing elements (PEs), to 4.5, with 16 PEs. Only one benchmark approaches an average 50% utilization of the PEs on the 16 PE-wide architecture.

As discussed earlier, VLIW architectures can be badly affected by horizontal and vertical wastage. The DTSVLIW is no different in this respect, as the scheduler unit determines instruction scheduling prior to storage in the cache. The only response to an increase in latency during VLIW-mode execution is to stall the VLIW Engine. Superscalar architectures are more resilient to changes in latency, as their instruction scheduling is done at the execution unit. Another critical issue for the DTSVLIW is the size of the VLIW cache, which has to be large enough to hold sufficient VLIW instructions to keep the VLIW Engine busy and not require too many returns to the Primary Processor.

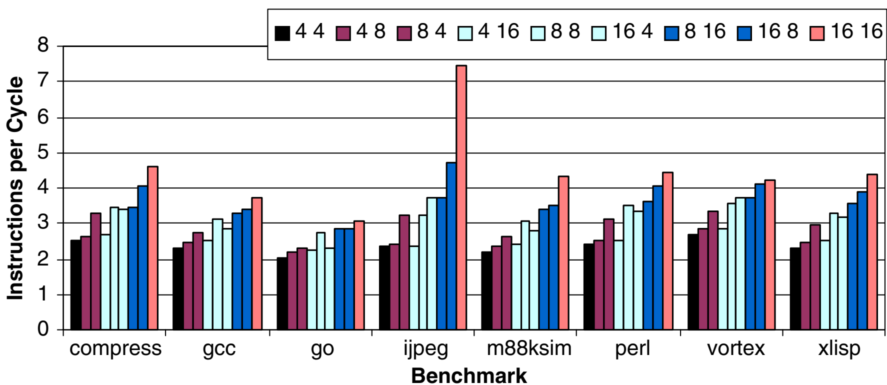

Fig. 2 Variation of parallelism with the block size and geometry

Making more use of the available parallelism, reducing the impact of memory latency variations, and reducing the on-chip size of the VLIW cache are the motivations of our work. The first two of these are well known reasons for moving to a simultaneous multi-threaded architecture [7], and this paper presents work on these issues. With simultaneous multi-threading (SMT), instructions from several threads are issued for execution on each cycle, reducing both vertical waste (if a thread is stalled due to memory access latencies, instructions from other threads may be available for execution) and horizontal waste (unused PEs due to a lack of available parallelism in the thread being executed) [7].

4 Rationale

The idea for the multi-threaded design arose from consideration of why we get improved performance from the DTSVLIW when the Primary Processor only reads one instruction at a time from the program memory: execution occurs mainly in the VLIW Engine, leaving the Primary Processor and I-Cache unused. Figure 3 demonstrates this: the upper bound of the trace shows when the VLIW Engine is executing; the lower bound when the primary processor is executing. Excursions between these bounds mark the transitions from one to the other. The simulator in this instance implements the SPARC architecture [19] and most of the many short transitions to the primary are to execute complex operations: mostly SPARC "save" and "restore" instructions. The longer period in the primary engine at 600 cycles is a period of VLIW block building by the scheduler: the snapshot is early in the execution of the vortex benchmark with a 16-PE VLIW Engine. The small amount of block building activity shows that the VLIW Cache has a working set of blocks. Typically, a DTSVLIW process will spend only 5–10% of its cycles in primary mode operation.

The primary processor is an under-used resource as is the I-Cache. Its increased utilization would seem to offer a route to making more use of the available PEs by using them to schedule blocks for other processes, and extending the VLIW Engine to SMT operation. This last requires that the issuing logic issue instructions from available long instructions of different threads into the VLIW Engine on each cycle: these instructions can be executed in parallel because they are either from the same long instruction or from different threads. There is of course a need for multiple register sets. SMT increases the utilization of the PEs but with some increase in run-time of individual threads: there is competition between threads for the PEs, and a process no longer has full use of the resources. This differs from the Weld architecture [4] where

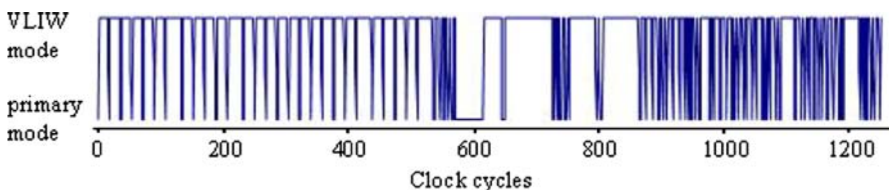

Fig. 3 Snapshot of 1200 clock cycles of DTSVLIW activity during execution of a single process

the single-threaded design reduces the execution time of an individual process: our SMT design overlaps the execution of processes to improve total process throughput. SMT also provides a mechanism by which the VLIW Engine can keep on processing when a memory access for a particular process fails to complete in the predicted time: instruction issue for this process is stalled, but the issuing logic can select and issue instructions from other processes. Of course, there are negative impacts to this multi-threading. One is the potential increase in memory traffic from the multi-threading. Another is the potential for issuing stalls when there are insufficient specialized PEs for the instructions requesting them, e.g. for load/store operations. This is a problem with all ILP architectures, both statically (compiled-based) and dynamically scheduled, but it is exacerbated by the multi-threading. All these are well-known advantages and disadvantages of multi-threading, as has been well documented elsewhere [2,6,7]. A second area of concern is the increase in the number of VLIW blocks that need to be stored and thus a pressure for an increase in the capacity of this cache. A solution to this VLIW cache size problem might seem to be to move the bulk of this off-chip. However, results from these experiments indicate that increasing the D-cache capacity might be a better approach to deal with both concerns: reducing D-cache misses has the positive side-effect of reducing the VLIW cache requirement for a similar level of performance.

A further possible benefit of multi-threading is that it allows the width of the scheduling unit logic to be decoupled from the width of the VLIW Engine, as is the case with EPIC architectures. In the standard DTSVLIW, these widths are the same, but now the scheduler width can be increased to allow the possibility for more parallelism. This will increase the number of empty slots in the long instructions created (the number of instructions per long instruction does not increase as rapidly as the number of instruction slots), but these can be discarded: a mechanism inspired by the bundle mechanism of EPIC architectures [3] has been proposed previously for the standard DTSVLIW [14]. In the event that the number of available instructions in a long instruction is greater than the number of PEs in the VLIW Engine, the extra instructions can be delayed one cycle and executed with instructions from other processes. This cannot be done in the standard VLIW architecture as the delayed instructions would execute on their own with low utilization of the PEs.

5 Multi-threaded Architectural Model

To implement a multithreaded DTSVLIW the changes to the standard DTSVLIW architecture are rather small in complexity, with the major changes being:

(a) multiple sets of registers with one register set for each thread,
(b) a new logic block for the issuing logic,
(c) a modified VLIW cache logic that allows a cache lookup on each thread in parallel,
(d) multiple sets of scheduling logic as required.

Increasing the instruction width of the scheduler list is not a change as such from the standard DTSVLIW, although the decoupling of the width from the width of the VLIW engine is, and the stripping of the empty slots from the long instructions has

been proposed before for the DTSVLIW [14], again to increase the effective utilization of the DTSVLIW cache.

There are a number of essentially minor changes to the standard DTSVLIW due to the need to identify which register set to use for each instruction executed, and to identify the VLIW blocks from different threads in the VLIW cache. Thus an identifier per thread is needed for both these purposes: the identifier would be used as part of the VLIW cache tag along with the address. This identifier is retrieved with each long instruction from the VLIW cache and issued with each sub-instruction to the PEs: the PEs concatenate the identifier with the register references in the instructions to identify the correct set and registers. A process ID is also used along with the thread ID for cache lookup to avoid the need to flush read caches on thread switches and process changes.

5.1 Issuing Logic

The issuing logic adds extra complexity to the VLIW fetch-execution path. It selects instructions from the available (one at most) long instruction for each thread, and it has to allocate the selected instructions across the PEs of the VLIW Engine. In a pure VLIW, the compiler schedules instructions into particular slots in a long instruction and a particular slot feeds a particular PE in the VLIW processor; thus, the compiler allocates instructions to PEs and there is no allocation required from the hardware. Thus, the standard DTSVLIW, which has a standard VLIW engine at its core, does not need an allocation process, as the scheduler performs this operation as its builds the long instructions in a block. However, it is something that EPIC architectures must do [3]: their unbundled long instructions can have more instructions than available PEs and can have more specialized instructions than PEs to process them.

The new issuing logic now has the role of allocating instructions to PEs. The issuing logic essentially holds blocks of VLIW instructions, one per thread, but pre-fetching of following blocks is feasible, since a VLIW block holds the address of the following block. Long instructions from a VLIW block are accessed sequentially, and the issuing logic selects from active long instructions of threads executing in VLIW mode to fill the available slots and resources in the VLIW Engine. Execution of a long instruction for a thread only completes when all its instructions have completed and their results have been committed to registers, i.e. execution and memory access latencies have elapsed. The issuing logic is thus similar to that of superscalars, but less complex since instruction ordering and renaming has already been done by the scheduler(s) and the instructions available to it are executable concurrently.

6 mDTSVLIW Architecture

The basic mDTSVLIW architecture tested is shown in Fig. 4. There is one set of PEs of which a sub-set operates as primary engines when needed, otherwise operating as part of the VLIW engine. Thus the "width" of the VLIW Engine is variable. Each primary processor has its own scheduler, its own I-Cache, fetch and decode units. When a thread needs executing in primary mode, the VLIW Engine width is reduced

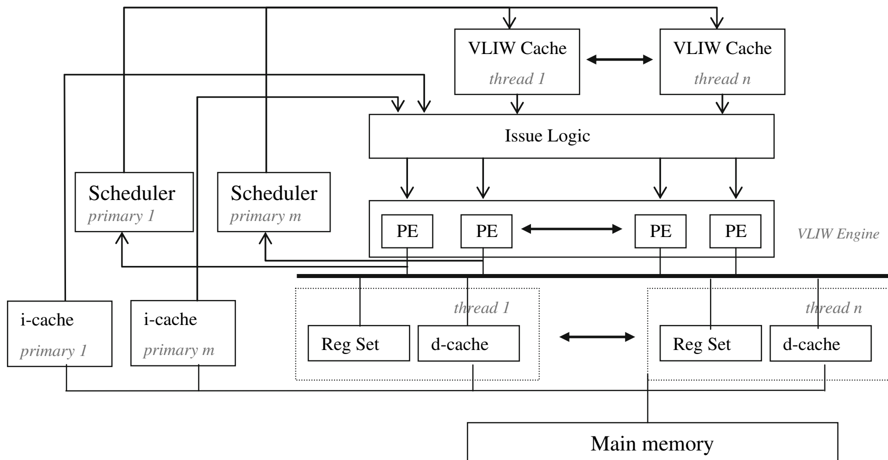

Fig. 4 mDTSVLIW architecture

by one and the PE released is allocated as the Primary Processor for this thread. It should be possible to move this PE on a per cycle basis between primary mode and VLIW mode use, but this has not been attempted in these experiments. Thus, PEs used as primary processors are stalled during I-cache misses.

There are multiple register sets, one per thread, each holding the registers of the modeled ISA, the SPARC ISA for this work, and a set of renaming registers. There is a separate L1 D-cache and associated aliasing unit [12] for each thread: this removed the need to implement, at this stage, a more complex single D-cache and aliasing unit, and allowed existing implementations from the DTSVLIW simulator to be re-used. A shared D-cache allows greater sharing of cache capacity [7]. Similarly, there are separate private L1 I-caches: one per primary unit. The process identifier is used as part of the address tag, so that switching the thread being executed by the primary processor does not require caches to be flushed. Logically, there is a single L2 cache, which is assumed to be large enough to hold all active process code. Access to this is sequential, so misses at the L1 caches are queued to access it. There is no address translation, but there is no code or data sharing between processes.

The PEs and the register sets operate as in the DTSVLIW, except that the target register set for a PE depends on the thread from which the executing instruction is taken, and can change on each cycle. The mode of writing to the registers during VLIW-mode execution had to be changed. A DTSVLIW long instruction is constructed so that its instructions execute in parallel and commit all their results at the same time, allowing the results of its instructions to write the sources of other instructions within the long instruction. Executing the instructions of a single long instruction across multiple cycles and committing their results as they complete breaks this model. To correct this, the results of these instructions are temporarily stored between the PEs and the registers sets, and are only committed into the main registers at the end of the cycle

Table 1 Cache sizes and latencies

Unit	D-Cache	I-Cache	VLIW cache
Quantity	1/Thread	1/Primary processor	1/Thread
L1 total capacity	64 or 128 kB shared between threads 256 kB in some cases	16 or 64 kB shared between primaries	384, 768 or 1534 kB shared between threads
Access time	1 cycle		1 cycle
Miss penalty	Selectable: 7, 11, 15 cycles used with 49 cycles used in a small number of cases	Thread moves to primary on miss	

in which the last remaining instructions from a long instruction are executed, and any load or stores to D-cache have completed.

In the baseline architecture of Fig. 4, the VLIW cache is wholly internal to the processor, and is just an L1 cache. There is a VLIW cache per thread for the same reasons as for the D-cache. This allows parallel lookup by threads of their associated VLIW caches. However, although possible, this doesn't occur on every cycle: the DTSVLIW model is to cache a VLIW block of long instructions as a single cache element, so a cache lookup is only performed by a thread when a new block has to be fetched. With multiple long instructions per block and multiple cycles per long instruction, the incidence of all threads making concurrent VLIW cache lookups is reduced.

The access times to the L1 and L2 caches are variable, but the same for both L1-D and L1-I caches. The total L1 cache capacities are variable and the sizes detailed in Table 1 were used in the experiments. These total capacities are shared between the cache instances. There are no constraints on the number of accesses made by a thread to its L1 D-cache. There is no miss penalty as such for the VLIW cache, but a miss in this cache stops the process from executing in the VLIW Engine and moves it to the primary processor queue. As for the other caches, the VLIW cache capacity (see Table 1) is shared between threads: a capacity of 384 kB is 1 kB blocks with 8 long instructions per block with 8 instructions per long instruction, although the full capacity is never used as not all long instructions are full. A scheduling list is twice the maximum VLIW block size (blocks may have as few as 1 long instruction); up to two blocks can be held in the scheduling list for output to the VLIW cache whilst continuing to construct a further block. Each of these blocks can be from a different thread.

There are no constraints on the PEs: each can execute any instruction with no limit on the number of load and stores executed by the VLIW Engine in a cycle. There are 32 renaming registers per thread.

7 Experimental System

The mDTSVLIW architecture was implemented in a C++ simulator. This allowed the concept to be proved and much of an existing C++ DTSVLIW simulation software

system to be re-used and for elements to be easily replicated. The simulator emulates the architecture at the instruction and pipeline level, and endeavours to replicate the clocking structure of a feasible hardware implementation. The simulation loads the benchmark programs into the simulator and executes them instruction by instruction: system calls are not executed, but are serviced by the underlying operating system. Program traces are not used in the experiments reported here.

8 Results of Experiments on Baseline Architecture

The experiments presented here have been performed as proofs-of-concept, and not to give definitive answers on performance. Thus, all experiments execute the same set of eight processes, each a different SPECInt95 program: compress, ijpeg, gcc, go, lisp, m668sim, perl and vortex (www.spec.org). All experiments have an 8 × 8 × 8 configuration: 8 instructions per long instruction 8 long instructions per VLIW block, and 8 PEs. There is no change in the number of instructions per long instruction between the scheduler, the VLIW cache and execution engine, postulated as a possible means of extracting increased parallelism. All caches are 4-way set associative. The majority of experiments have been with a 4-thread architecture, but there are results for 2-thread and 8-thread architectures.

Figure 5 plots the execution activity as the 8 SPEC95 processes execute on a 4 thread, 2 primary 8 × 8 × 8 mDTSVLIW, with I and D-cache miss penalties of 7: the first 4 threads all start at time 0. The x-axis is cycle count and the vertical access is either instructions executed per cycle or process count per cycle depending on the curve. The two smooth curves that run between six and seven are the running averages from cycle 0 of the number of VLIW instructions executed per cycle and of all instructions executed per cycle (VLIW instructions and primary instructions). The small difference between them is the average number of instructions executed by the primary processors, showing that most processing is being done by the VLIW Engine. The jagged curve behind these two curves show the average number of VLIW instructions per cycle calculated over 10,000 cycle periods. The variability in this reflects the underlying variability in the VLIW mode instructions executed each cycle from 0 to 8. The vertical lines mark new processes being loaded as earlier ones complete.

The middle plot, oscillating just below 4, is the number of threads running in VLIW mode per cycle averaged over 10,000 cycle periods. The corresponding plot

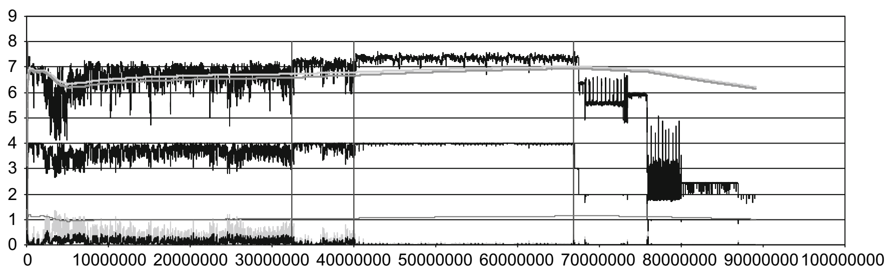

Fig. 5 Activity plot for a 4 thread, 2 primary mDTSVLIW

for threads being executed in primary mode is the upper of the two jagged plots (in grey) at the bottom of the figure: this generally stays below one. The bottom (black) plot is the average number of threads per cycle queued waiting for a primary processor to become free, again averaged over 10,000 cycle periods. The under-usage of the primary processors led to the VLIW Engine PEs being used in this role. Much of the time a dedicated primary processor has nothing to do. When primary mode is active, the VLIW Engine has a lower load and does not much miss a PE being allocated to primary mode.

The remaining plot that runs fairly consistently around one is the running average from cycle 0 of the number of copy operations executed per cycle. These are additional instructions to the original process code arising from moving instructions across conditional branch boundaries: copy instructions are left behind to move results from renaming registers into the target destinations (register or memory locations) [11]. These copy instructions are included in the running instruction averages at the top of the graph. Although increasing the ILP, they can add to the execution time and use resources: in the experimental architecture, these copy instructions are allocated to any functional unit despite the fact that the copy instruction functional requirements are rather small—just moving one or two values from renaming registers to designated units. On the right, the curves drop away as processes complete: the "threads in VLIW mode" plot shows this clearly.

Table 2 shows the results for a number of experimental runs on an $8 \times 8 \times 8$ architecture with a range of configurations, shown in the "config" column as "threads × primaries × miss penalty", e.g., $4 \times 1 \times 7$. The speedup compares the time for a single scalar processor (SSP) operating to execute the same workload to the time the mDTSVLIW took: these times run to the completion of the last process even though the mDTSVLIW has a reducing numbers of threads running near the end as seen in Fig. 5. [The SSP has the same total cache resources for I and D Caches, but it has single instances of these: the SSP is just a single thread, single primary mDTSVLIW with the VLIW Engine and Scheduler switched off.] The ILP values are calculated by dividing the total number of instructions executed by the SSP by the number of mDTSVLIW clock cycles for the same work load, and doesn't take account of SSP memory or other latencies, which accounts for the lower performance by this measure. The instruction averages are per cycle calculated from time 0 until the number of active threads drops below 4, when this value start to drop away as the mDTSVLIW in under-utilized. The "All" column is the average of all instructions both VLIW and primary executed

Table 2 Results with different thread, primaries and cache latency combinations for the test set

Config	Speedup	ILP	Instruction averages			In primary (%)	iC, dC, vC sizes (kB)	iC_acc/hit rate (%)	dC_acc/hit rate (%)
			All	VLIW	Copy				
$4 \times 1 \times 7$	6.3	5.9	7.0	6.9	1.1	12	16/128/1536	1/93	86/98
$4 \times 1 \times 7$	6.1	5.8	6.9	6.6	1.1	28	64/128/384	4/98	86/98
$4 \times 2 \times 7$	6.0	5.8	6.9	6.6	1.1	34	64/128/384	4/96	86/98
$4 \times 1 \times 11$	5.8	5.5	6.6	6.3	1.0	31	64/128/384	4/98	86/98
$4 \times 1 \times 15$	5.5	5.1	6.2	6.0	1.0	35	64/128/384	4/98	86/98

per cycle, while the "VLIW" column does not count in the instructions executed in primary mode. The "Copy" column shows the average number of Copy instructions per cycle. These three columns show that the PEs can be kept more fully active, with most activity in VLIW-mode, but that copy instructions are using resources not required in primary-mode (but see above). The "In primary" column is calculated by adding one to a counter for every PE active in primary mode in a cycle, and shows the percentage of 1 PE used in primary mode: this value increases as the VLIW Cache size decreases and its miss rate increases, and also with increasing memory latency when primary mode PEs are stalled.

The last two columns show the I- and D-cache access totals as percentages of those of the SSP and their hit rates. The mDTSVLIW makes very few I-cache accesses (1–4%) compared to the SSP, but its hit rate is marginally worse: the SSP's is ∼99% for a 64 k I-cache. But, the much lower number of I-cache accesses means that its miss total is much less than that of the SSP. The mDTSVLIW makes fewer D-cache accesses than the SSP, with the missing accesses being translated into renaming registers accesses.

The primary result for examining the performance of the mDTSVLIW is the speedup: the time to completion of execution of all the processes in the workload compared to the time for the SSP to do the same. Also of interest is the utilization of its PEs shown by the instruction execution average (the instructions per cycle average, the IPC) at the last time that all threads are active. The variation of these figures against the different cache sizes and miss latencies are shown in the plots of Figs. 6–15: both I and D caches have identical miss latencies in the L1 caches. There are speed up and IPC figures for 5 thread and primary configurations: 4×1, 4×2, 4×4, 2×1 and 8×1—the first value is the number of threads, the second the number of primaries. The results are grouped by miss penalty, and each member of a group is for a specific configuration of caches: 16/128/384 indicates total cache capacities

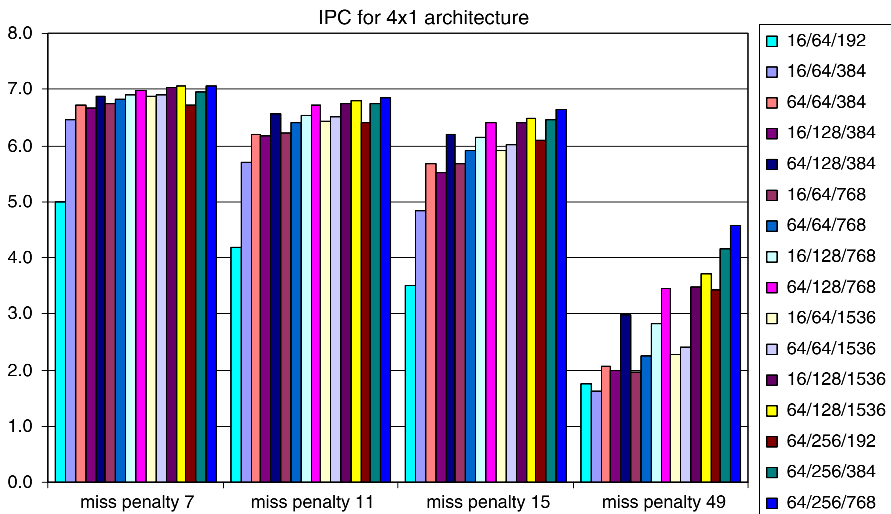

Fig. 6 IPC averages for 4×1 architecture

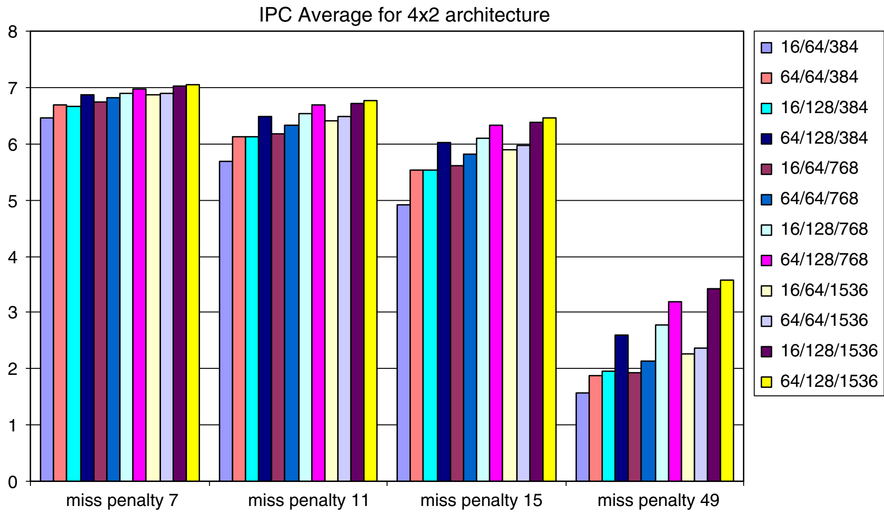

Fig. 7 IPC averages for 4 × 2 architecture

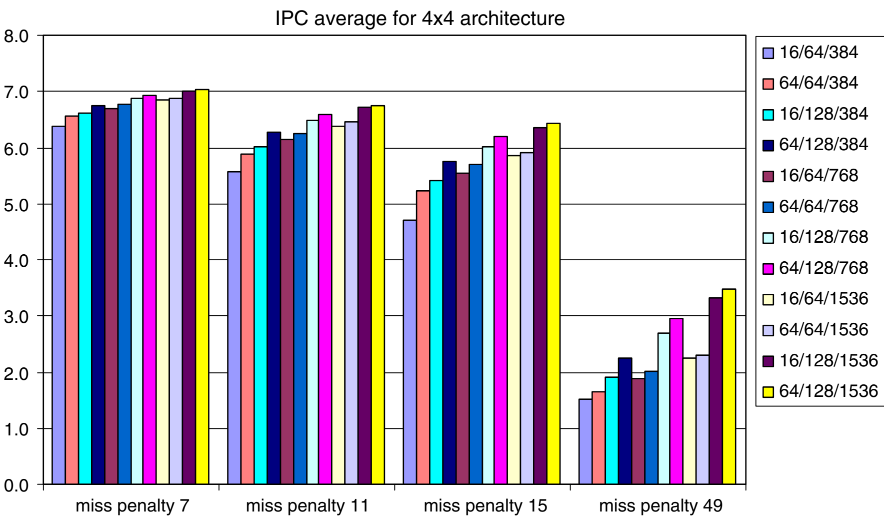

Fig. 8 IPC averages for 4 × 4 architecture

as 16 kB for the primary mode I-caches, 128 kB for the thread D-caches, 384 kB for the thread VLIW L1 caches. Note that the 4 × 1 plots have an extra configuration 16/64/192 (a VLIW cache capacity of 512 blocks in total or just 128 blocks per thread with 4 threads—64 blocks with 8 threads) to demonstrate the drop off in performance with a small VLIW cache, and also three extra configurations with a larger D-cache (64/256/192, 64/256/384, 64/256/768). The 4 thread plots also have results for a miss penalty of 49 to give an idea of performance with very long external memory access delays.

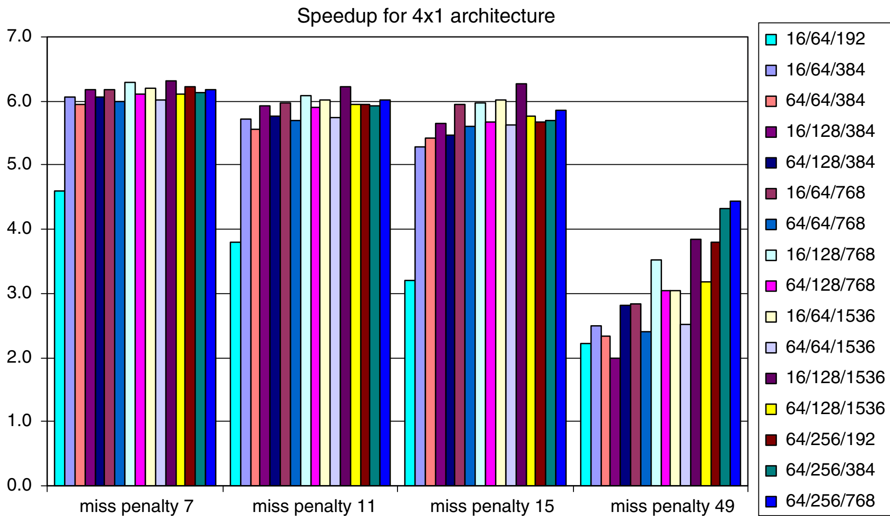

Fig. 9 Speedups for 4 × 1 architecture

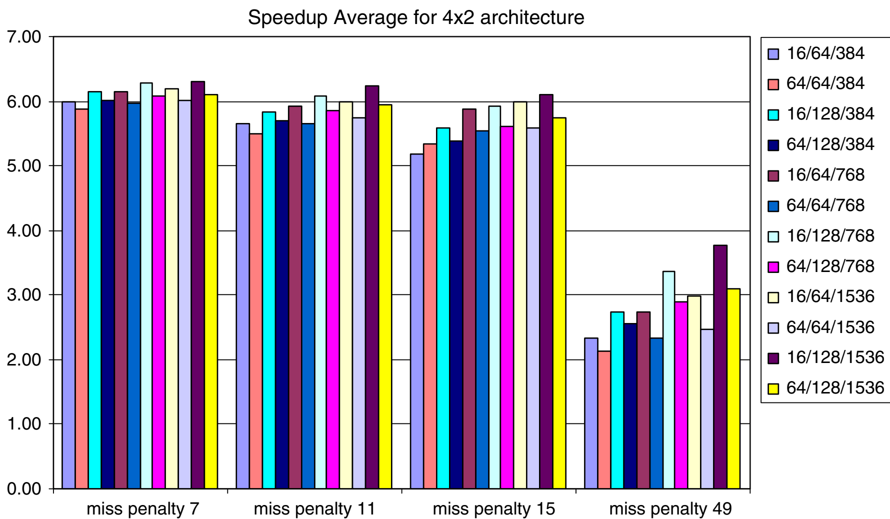

Fig. 10 Speedups for 4 × 2 architecture

Examining the IPC plots for the 4-thread architectures, Figs. 6–8, it can be seen that PE utilization is very good with miss penalties of 7 and 11, and that there is only a small variation with VLIW cache size within a group: the increased incidence of primary mode execution from the extra misses in the VLIW cache with smaller size does not impact too heavily because of the low miss penalty on the I-caches. The miss penalty does of course impact as can be seen in the general decrease in IPC with increasing miss penalty. One feature of the IPC results that stands out is the

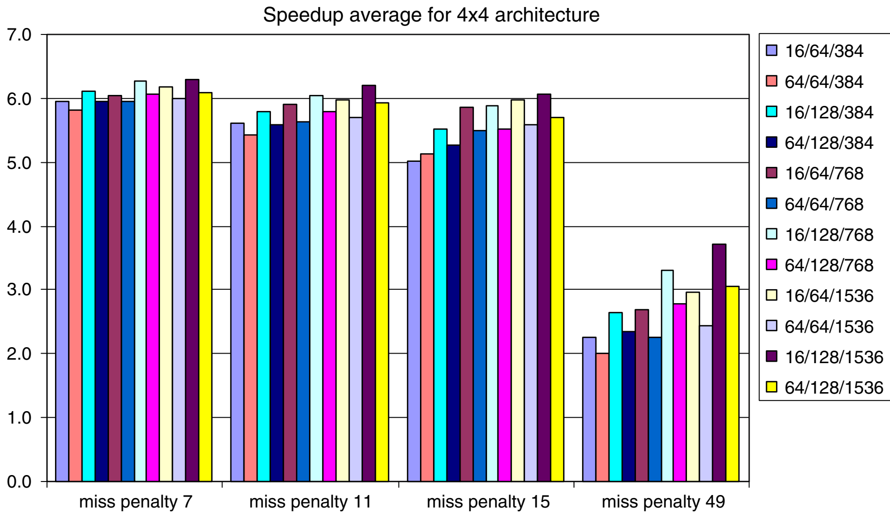

Fig. 11 Speedups for 4 × 4 architecture

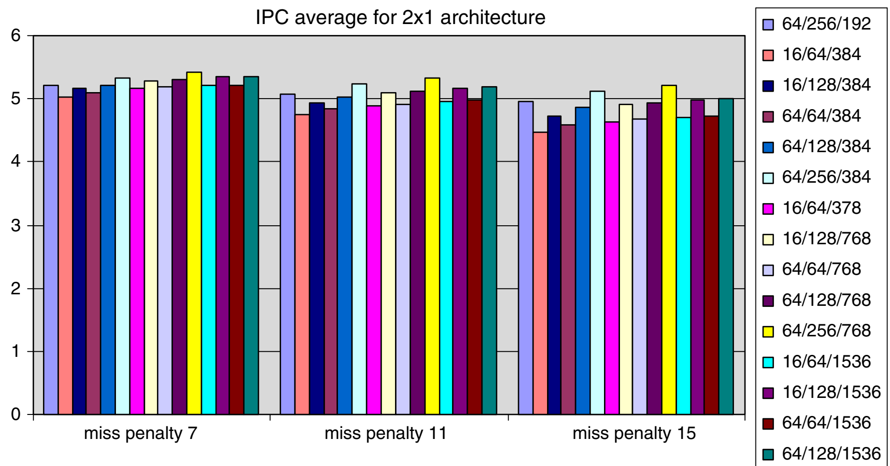

Fig. 12 IPC averages for 2 × 1 architecture

impact of the D-cache size at greater miss penalties. Within a group of four results for the same total VLIW cache size for all but some of the 384 kB results, the IPC increases with total D-cache capacity with the total I-cache size playing a lesser role. This reflects the use of these caches by the mDTSVLIW: the I-cache has a much lower utilization than in a conventional processor designs as pointed out earlier, but there is no such reduction in the utilization of the D-Cache. The per-cycle utilization of the D-Cache is much increased over that of a single-threaded processor because of the multi-threading. Of course, because of the low I-cache utilization and the reduction in total misses, there is much easier access to the external memories for data misses

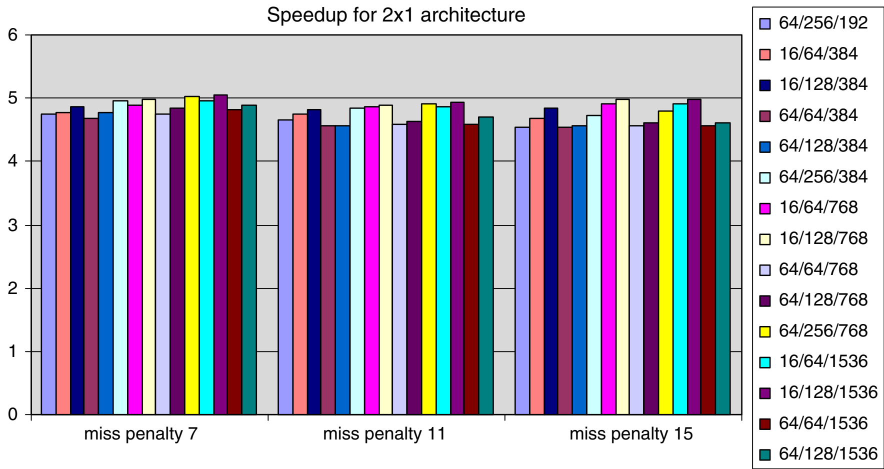

Fig. 13 Speedups for 2 × 1 architecture

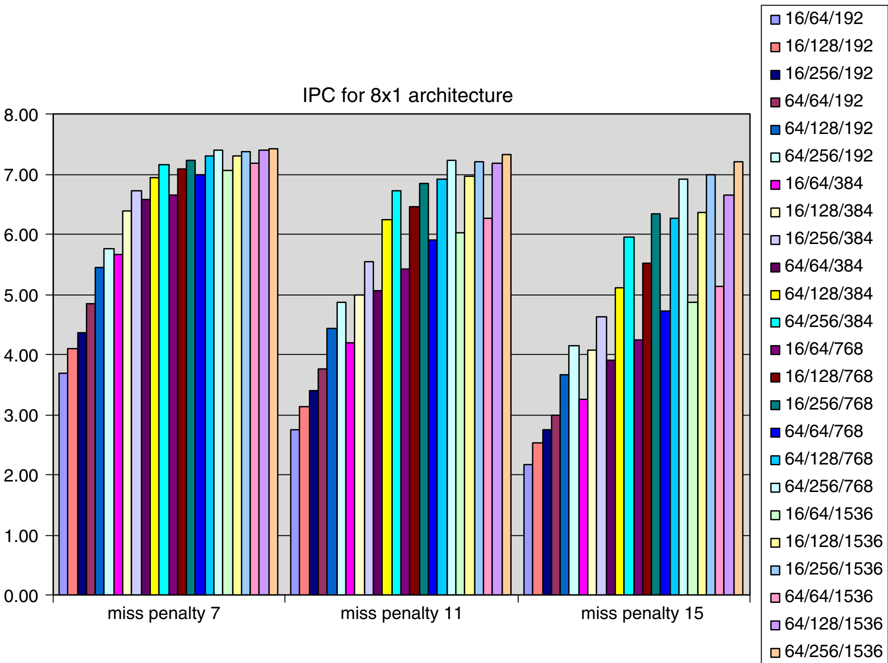

Fig. 14 IPC averages for 8 × 1 architecture

than would be the case with other wide issue designs, that also use a von Neumann main memory architecture.

The IPC figures also demonstrate that using more than one scheduler, i.e. having more than one process in primary mode execution, does not improve the performance: for the 4 × 4 configuration the performance is slightly worse than the 4 × 1 configuration, while for 4 × 2, the performance is essentially identical. Having more than 1 thread in

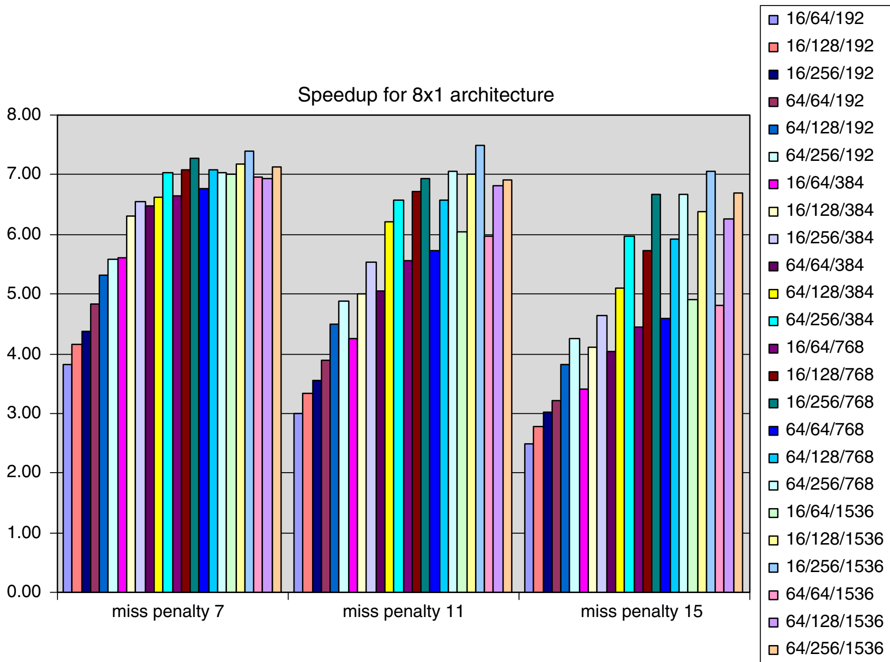

Fig. 15 Speedups for 8×1 architecture

primary mode increases the contention on the bus to the L2 I-cache, and reduces the resources available to the VLIW engine.

From consideration of the results for 64 and 128 kB total D-cache capacity, a small number of experiments have been run with 256 kB total D-cache capacity using the 4×1 configuration and with 192, 384 and 768 kB total VLIW cache capacity. The IPC results for these experiments are shown in Fig. 6 as the last three results of each group. With the extra D-cache resources, 64 kB per thread, the IPC results are at least equivalent to an architecture with twice the VLIW cache resources, i.e. the 64/256/768 result are very similar to the 64/128/1536, the 64/256/192 to the 64/256/384. The relative improvement in IPC increases with miss penalty, so that with a miss penalty of 49 the 64/256/192 architecture has nearly as good an IPC average as the 64/128/1536 architecture for the same miss penalty. The 64/256/192 architecture results are less than 10% different from the best achieved for lower D-cache, but higher VLIW cache, sizes for the same miss penalty. This improvement in the 192 kB VLIW cache performance is startling: going from configuration 16/64/192 to 64/256/192 almost doubles the average IPC for miss penalties 15 and 49, and gives a 33% improvement for miss penalty 7. With the 192 kB total VLIW cache capacity (48 kB per thread), there is an increased use of primary mode and thus extra fetches to the L2 memory, which suffer extra delays over the defined miss penalty because of conflicts with the D-cache miss traffic. The increased total D-cache capacity reduces the D-cache bus traffic, and as a side effect reduces conflict delays on the L2 instruction fetches, so that VLIW

blocks are built more rapidly reducing the times that threads stay in primary mode, and increasing the VLIW mode activity.

The speedup results for the 4-thread architectures in Figs. 9–11 are much more consistent within a group and decrease more slowly with miss penalty. This is because the speedup is the ratio of the SSP execution time for the test suite to that of the mDTSVLIW for the same total I-cache and total D-cache capacities and miss penalties. Thus as the mDTSVLIW performance changes with changed cache capacities and miss penalties, so does the SSP's performance. Within the figures it can be seen that the 16 kB I-cache results are generally better than those for the 64 kB I-cache (opposite to the IPC results), which indicates that the SSP is suffering increased fetch misses from the small I-cache.

The IPC results for the 2×1 architecture (Fig. 12) results are very flat with slight increases related to increasing cache totals: the results demonstrate that IPC is not resource limited but is limited by the available number of instructions for issue. The speedup results in Fig. 13 tell the same story: the increased variation arising mostly from the variation in the SSP performance with different configurations. The 2×1 architecture would benefit from increases in the density of instructions in the blocks perhaps from increasing the scheduling width to get more parallelism.

Figures 14 and 15 show the results for an 8×1 configuration. To get results with a reasonable period with eight processes active, the group of eight processes used in other experiments have been run twice so that 16 processes are executed to completion, although making comparison with the 2×1 and 4×1 experiments less valid. The IPC average is measured when the process load becomes less than 8, once nine processes have completed, while the speedup is the time to execute all 16 processes by the SSP divided by the time taken to execute them by the mDTSVLIW. The experiments show a marked increase in both IPC and speedup as more resources are available showing that for smaller caches sizes the active processes are competing for limited resources. As more resources are made available, the available parallelism begins to have an impact. This last is clearly seen in the IPC results with a miss penalty of 7, where there is little performance improvement moving from the 64/256/384 configuration (average IPC of 7.16) to the larger VLIW cache sizes. Again as noted previously, the total D-cache size has a major impact upon performance along with the VLIW cache size: with a miss penalty of 7, the IPC average for the 64/64/768 configuration of 6.99 is worse than that of the 64/256/384 one, while that for the 64/64/1536 configuration of 7.19 is essentially the same. The IPC results for miss penalty 11 are similar to those of miss penalty 7, except that with 8 processes a 64 kB total D-cache is a major constraint. Performance with such a D-cache size is reduced by 15% or more over an identical configuration but with a 256 kB D-cache: the 64/64/384 IPC for miss penalty 11 is 25% less than that of the 64/256/384 result. Unlike the miss penalty 7 results, there is continued improvement as the total VLIW cache size increases above 384 kB, although not above 768 kB, but this improvement is negated if the D-cache is too small. The IPC results for miss penalty 15 again demonstrate the impact of too small a D-cache which overwhelms the impact of the VLIW cache size at larger sizes. The speed-up results of Fig. 15 tell the same story, but indicate that there is only marginal value going beyond a total VLIW cache size of 768 kB, while a 384 kB size gives useful results.

Table 3 Comparison of some average IPC results from the 4 × 1 and 8 × 1 experiments

Cache sizes	Average IPC for 8 × 1 experiments			Average IPC for 4 × 1 experiments		
	Miss penalty 7	Miss penalty 11	Miss penalty 15	Miss penalty 7	Miss penalty 11	Miss penalty 15
64/256/768	7.4	7.2	6.9	7.1	6.9	6.6
64/256/384	7.2	6.7	6.0	7.0	6.7	6.5
64/256/192	5.8	4.9	4.2	6.7	6.4	6.1

Table 3 compares average IPC result for 3 cache configurations for the 4 × 1 and 8 × 1 experiments, and shows that having 8 processes available gives only a small increase in IPC for the larger VLIW caches: thus the 64/256/768 configuration IPC averages are better for 8 × 1, but for smaller sizes this is not so, particularly for the 64/256/192 configurations where performance drops by 16% or more. With eight processes there is much more competition for cache space and yet there is only a small increase in the utilisation of the available processing power: 4% for the 64/256/768 configurations. There are too many processes competing for too few cache and processor resources.

The speedup results demonstrate the same result. The 2 × 1 experiments show a speed-up of just below five across all configurations with little restrictions apparent. The 4 × 1 configurations give increased performance with a speed-up of 6 with miss penalty 7 across the same configurations of the 2 × 1 experiments, with resource constraints becoming apparent at larger miss penalties. For the 8 × 1, cache size constraints have a major impact and speed-ups greater than that attained by the 4 × 1 experiments are only achieved for the larger cache configurations.

9 Conclusions and Further Work

It is possible with a single scheduling unit working on only one scalar process at a time and with sufficient D-cache and VLIW cache capacity, to provide enough instructions to keep the PEs of a SMT architecture busy, provided there are sufficient threads and processes available. The increase in PE activity does not heavily impact upon the instruction fetch rate from primary memory and I-cache as do other SMT designs. There are indications that increasing the D-cache size can offset the need for very large VLIW caches: the results for the 4 thread, single primary architecture with 64 kB of D-cache per thread, 48 kB of VLIW cache per thread, and just one 64 kB I-cache gave results within at most 10% of the best results with much larger VLIW caches (384, 768 and 1536 kB caches) for the same miss penalty. Results also suggest that it may be counter-productive to have too many processes active at any time as this may reduce performance by putting too much pressure on the available cache resources, particularly with the smaller cache sizes. It should be noted that the architecture does not depend on speculative execution or branch prediction logic for its performance.

In the context of the mDTSVLIW architecture, the number of VLIW instructions executed per cycle average shows that wastage (horizontal and vertical) and stalls due

to memory access latencies can be greatly reduced by having a small number of threads (as other studies have shown), but without a large increase in the instruction fetch rate from I-cache and main memory. There is an increase in complexity in going from the DTSVLIW to the mDTSVLIW in terms of the separate register sets, and separate data and VLIW caches. The last two can be merged into single larger data and VLIW caches, which would probably improve utilization [7], but the need for separate resister sets will limit the number of threads, so that improvements to the design to improve the block instruction densities are warranted, particularly as the VLIW cache utilization is only around 60%. Our experiments indicate that further work is well warranted to more fully develop and investigate this architecture. In particular to examine: a wider range of benchmarks and benchmark sets, in particular moving to the SPECint 2006 test suite; variations to the VLIW cache; and decoupling the scheduling width from the width of the VLIW engine.

References

1. Olukotun, K., Hammond, L.: The Future of Microprocessors. ACM Queue, pp. 27–34, September 2005
2. Ungerer, T., Robic, B., Silc, J.: Multithreaded processors. Comput. J. **45**(3), 320–348 (2002)
3. Schlansker, M., Rau, B.: EPIC: Explicitly parallel instruction processing. IEEE Computer **33**, 37–45 (2000)
4. Ozer, E., Conte, M.: High-performance and low-cost dual-thread VLIW processor using weld architectural paradigm. IEEE Trans. Parallel Distribut. Syst. **16**(12), 1132–1142 (2005)
5. Özer, E., Conte, T.M., Sharma, S.: Weld: a multithreading technique towards latency-tolerant VLIW processors. In: Proceedings of the 8th International Conference on High Performance Computing–HiPC 2001, Lecture Notes in Computer Science 2228, pp. 192–203, December 2001
6. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: maximizing on-chip parallelism. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, Assoc. Comput. Mach., pp. 392–403 (1995)
7. Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L., Tullsen, D.M.: Simultaneous multithreading: a platform for next-generation processors. IEEE Micro. **17**(5), 12–19 (1997)
8. Rau, B.R.: Dynamically scheduled VLIW processors. In: Proceedings of the 26th Annual International Symposium on Microarchitecture, pp. 80–92. Austin, Texas (1993)
9. Spadini, F., Fahs, B., Patel, S., Lumetta, S.S.: Improving quasi-dynamic schedules through region slip. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. ACM International Conference Proceeding Series, vol. 37, pp. 149–158. San Francisco, California (2003)
10. Nair, R., Hopkins, M.E.: Exploiting instruction level parallelism in processors by caching scheduled groups. In: Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 13–25 (1997)
11. De Souza, A.F., Rounce, P.A.: Dynamically trace scheduled VLIW architectures. In: Proceedings of the High-performance Computing and Networking 1998–HPCN'98, Lecture Notes in Computer Science 1401, pp. 993–995, April 1998
12. De Souza, A.F.: Integer performance evaluation of the dynamically trace scheduled VLIW architecture. Ph.D. thesis, Department of Computer Science, University College London, University of London (1999)
13. De Souza, A.F.: Dynamically scheduling VLIW instructions. J. Parallel Distribut. Comput. **60**(12), 1480–1511 (2000)
14. De Souza, A.F.: Integer performance via block Compaction. In: Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing, pp. 98–105 (2001)
15. Santana, S.C., De Souza, A.F., Rounce, P.A.: A comparative analysis between EPIC static instruction scheduling and DTSVLIW dynamic instruction scheduling. In: Proceedings of the ICS 03 Workshop

on Exploring the Trace Space for Dynamic Optimization Techniques, International Conference on Supercomputing, San Francisco, ACM SIGARCH, June 22–26, 2003
16. Rounce, P.A., De Souza, A.F.: The mDTSVLIW: a multi-threaded trace-based VLIW architecture, sbac-pad. In: 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'06), pp. 63–72 (2006)
17. Fisher, J.A.: The VLIW machine: a multiprocessor for compiling scientific code. IEEE Computer **17**(7), 45–53 (1984)
18. Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., Lavery, D.M.: The superblock: an effective technique for VLIW and superscalar compilation. J. Supercomput. **7**, 229–248 (1993)
19. Sun Microsystems: The Sparc Architecture Manual—Version 7. Sun Microsystems, Inc. (1987)