Performance Study of a Multithreaded Superscalar Microprocessor

Manu Gulati

NexGen Inc.
1623 Buckeye Drive
Milpitas, CA 95035

Nader Bagherzadeh

Electrical and Computer Engineering
University of California at Irvine
Irvine, CA 92714

Abstract

This paper describes a technique for improving the performance of a superscalar processor through multithreading. The technique exploits the instruction-level parallelism available both inside *each individual stream, and* across *streams. The former is exploited through out-of-order execution of instructions within a stream, and the latter through execution of instructions from different streams simultaneously. Aspects of multithreaded superscalar design, such as fetch policy, cache performance, instruction scheduling, and functional unit utilization are studied. We analyze performance based on the simulation of a superscalar architecture and show that it is possible to provide support for multiple streams with minimal extra hardware, yet achieving significant performance gain (20 - 55%) across a range of benchmarks.*

1 Introduction

In an effort to improve performance, computer designers make use of parallelism at various levels – from coarse-grain parallelism, at the program or application level, to very fine-grain parallelism, at the level of individual instructions. In most cases, extraction of parallelism is the responsibility of software, either being specified explicitly in the programming language or discovered by the compiler. Superscalar processors, on the other hand, accomplish this through hardware techniques. As compared to traditional scalar processors, superscalars require a wider pipeline. In order to generate correct results and extract as much parallelism as possible, they may also use sophisticated techniques like out-of-order execution [8], register renaming [11] and speculative execution [6]. The hardware in a superscalar processor is responsible for keeping track of dependencies, determining when an instruction is to be executed, and on which functional unit.

The processor's own finite resources place a limit on its maximum achievable performance. The parallelism present in the program being executed (henceforth referred to as *workload*) is an equally important factor. With present VLSI technology, it is possible to provide sufficient hardware to take advantage of short periods of high parallelism. Very often, however, the processor is unable to execute the maximum number of instructions, owing to true dependencies on both short and long latency operations, and mispredicted control transfer operations. This paper explores the effectiveness of multithreading in creating a workload with greater parallelism. Rather than expend hardware to meet peak requirements, it is used for support of multiple threads, and applications programmed accordingly.

The methodology of multithreading is also addressed in this paper. The processor must have fetch and issue mechanisms that allow it to execute the different threads with equal or different priorities, as desired. Benefits of superscalar techniques like speculative execution and register renaming should be available to all threads. On-chip resources like registers and the cache have to be shared by all threads in an effective way. Section 3 deals with architectural support for multithreading.

This paper uses a particular superscalar processor, called SDSP [12], as an example of an architecture that could benefit from these techniques. SDSP is a pipelined RISC processor with the ability to fetch and decode four instructions per cycle. The SDSP has been designed for integer processing, and therefore does not possess floating point (FP) computation units. The difference between the superscalar model we use and the SDSP is only that we employ these units, since our benchmarks contain floating point computations. A discussion of the SDSP architecture follows in Section 2. The emphasis is on techniques used for exploiting the parallelism available in workloads. Section 4 briefly describes the simulation methods and benchmarks used. Different performance factors and design issues are dealt with in Section 5. Section 6

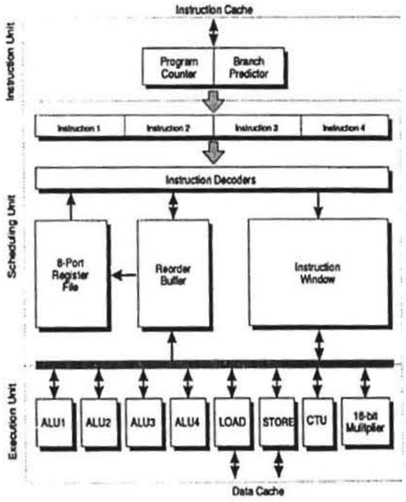

Figure 1: **Organization of essential components in SDSP**

draws conclusions and presents alternatives for further improving performance. Throughout this paper, the terms "instruction stream" and "thread" are used interchangeably, and refer to one segment of a parallel program.

2 The SDSP architecture

The Superscalar Digital Signal Processor (SDSP) is being developed as part of an on-going research project, with full considerations towards VLSI implementation. Figure 1 shows the architectural organization of the processor. Its key components are: the Instruction Unit (IU), decoders, the Scheduling Unit (SU) and the Execution Unit (EU). The basic dataflow through these units is as follows: the IU fetches a block of four contiguous instructions (each instruction is 32-bits long) from instruction memory. A hardware branch predictor, with a 2-bit prediction algorithm [5], allows speculative fetching. For each valid fetched instruction, the decoder decodes opcodes and fetches operands, if available. It then places the decoded instructions into the instruction window (IW) part of the scheduling unit, where they remain until they are scheduled for execution. When an instruction is free of dependencies and a functional unit is available, it is executed in the execution unit (EU). Upon completion of execution, results are written back to the scheduling unit. This may cause other instructions that were awaiting these results to become ready for execution. A brief description of the key units follows; a more thorough explanation can be found in [12, 13].

2.1 Decoder

The decoder is responsible for the following tasks:

- decoding opcodes and fetching operand values/tags,
- assigning new renaming tags to destination registers, and
- placing decoded instructions along with all values and tags into the scheduling unit.

The required value or tag for a source operand is obtained by performing an associative lookup on all entries in the reorder buffer (RB) in the SU, with its register number as the key. If more than one entry matches, value/tag from the most recent occurrence is used. If there is no match in the SU, its value is obtained from the register file.

2.2 Scheduling unit

The reorder buffer, instruction window and dynamic scheduling logic make up the scheduling unit of the SDSP. An entry is made in the SU for each and every valid instruction decoded. The SU is maintained in a FIFO manner, with newly decoded instructions being entered at the top. Place is made for these by shifting the contents of the entire SU down by one block (four instructions for SDSP), and shifting out the oldest block of instructions from the bottom of the SU. Computation results from the shifted out instructions are written to the register file. This operation is known as result commit, since it updates the in-order state of the machine to a new point of execution. If the results of the bottom entries are not yet available, they cannot be committed, and SU entries cannot be shifted down. Consequently, no new entries can be made. This occurrence is known as a scheduling unit *stall*.

The dynamic scheduling algorithm is simply "oldest first". For a scheduling unit with FIFO ordering, this implies that instructions closer to the bottom have higher priority. The scheduling logic analyzes instructions from bottom to top, issuing the ones that are ready for execution.

3 Running multiple threads

While multithreading might provide more parallelism, it comes at a cost. First, there is added hardware to support multiple threads. This includes extra storage to hold the state (e.g. contents of registers) of several threads, and the extra complexity in dynamically scheduling instructions. Second, wasted cycles result from synchronization and communication delays between threads. These are inherent in the parallel programming model and cannot be avoided. Third, a context switch penalty (in terms of wasted cycles) is incurred in switching execution from one thread to another. Fourth, the *total* number of instructions executed increases, because of the overhead of creating multiple threads. Finally, there is a loss in the locality of both data and instruction accesses to memory. Lower locality has a negative effect on performance owing to the reduced effectiveness of the cache.

In light of the above observations, it would be interesting to see the effects of multithreading on the operation of a superscalar processor. The methodology for multithreading must address the above considerations, and keep the following two goals in mind [1]:

1. To have a low cost of switching contexts, since this is simply an overhead.
2. To have good single-thread performance, so that applications with low parallelism, and inherently sequential code like critical sections can execute efficiently.

One way of executing several threads on a processor is to load the state of a particular thread from memory, execute that thread, and switch contexts by storing back its new state to memory before loading that of another one. This is the technique used by many multithreaded processors [1, 3, 2]. The "state" of a thread in a superscalar processor consists of the following: its set of registers, program counter, reorder buffer, instruction window, and store buffer. Saving and restoring all this information at every context switch constitutes an enormous overhead. Therefore, a different approach to multithreading is taken. All threads stay resident on the processor at all times, and instructions from different threads are executed simultaneously. All resources on the chip, viz. the register file, reorder buffer, instruction window, store buffer, and renaming hardware are shared by the threads. The manner in which the register file is shared is determined by the compiler. Register allocation is thus static, and in the results presented in this paper, all threads are allotted equal numbers of registers. Different static distributions *can* be made by the compiler, if it is capable of determining an appropriate distribution. The reason for selecting an equal distribution is its simplicity in the compiler as well as the hardware. Sharing of the remaining resources among different threads changes dynamically and is directly dependent on the fetch policy used by the processor.

The remainder of this section describes how exactly multiple threads are supported on the SDSP. Two goals are kept in mind: to keep hardware complexity at a minimum, and not to increase the cycle time of the machine by an amount that would undo any benefits of the modifications. Unless stated otherwise, the letter N refers to the number of simultaneously executing threads.

3.1 Instruction unit

The IU of the SDSP has to be enhanced to manage the control flow for all executing threads. There are N program counters, instead of one. A block of four contiguous instructions is fetched in one cycle, as before. Instructions fetched in one cycle all belong to the same thread, but fetching in different cycles is done from different streams. Branch prediction with multiple threads is addressed in the next section.

3.2 Decoder and scheduling unit

The scheduling unit has an extra field that holds the thread ID (TID) of the decoded instruction. The remaining fields stay the same. In addition to the tasks mentioned earlier, the decoder performs the following:

- assigns the correct TID value in the extra field in the SU,
- modifies the associative lookup for fetching operand values/tags to succeed only if the thread number *and* the register number match those of the instruction being decoded.

The decoder still assigns a unique tag to each and every valid instruction decoded, irrespective of the thread. It also places the decoded block at the top of the SU, as before. The tag issued to a particular instruction is different not only from all others issued to instructions of the same thread, but from *all* other tags in use, irrespective of thread number. That is, the renaming hardware continues to allocate tags as if all instructions belonged to the same thread, and does not reuse one until its previous occurrence is no longer in use.

3.3 Instruction scheduling

The improvement in performance that a superscalar can achieve depends on its ability to issue instructions out-of-order [6]. Preliminary results on the VLSI implementation of the SDSP processor indicate that SU design is critical in meeting timing constraints of the processor [13]. It is therefore important that multithreading not add to the existing complexity of the SU.

Once instructions are decoded and placed into the SU, the scheduling logic does not have to concern itself with the thread that an instruction belongs to. This is because all dependencies have been expressed in terms of matching tag values between the instructions. Once source operand values are known, and functional units available, the instruction can execute. This approach to multithreading has some distinct advantages. First, the issue logic need not be changed from its original design. Second, *all* independent instructions, whether

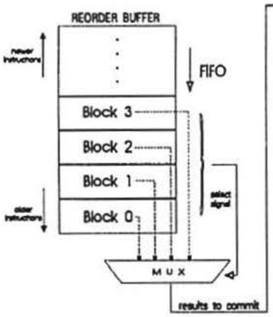

Figure 2: **Reorder buffer with the ability to commit results from one of four bottom slots.**

they belong to the same thread or different threads, are identified, thereby exploiting as much ILP as possible. This approach also allows all threads to be scheduled with the same priority. If different priorities are to be allotted, the fetch policy of the processor can be adapted to favor or discriminate against the particular thread(s).

3.4 Instruction execution

Other than the case of control-transfer (CT) operations, instruction execution and Functional Unit (FU) design is the same with multithreading as in a single-threaded processor. In the SDSP, the detection of a mispredicted CT operation results in all entries above it in the SU being discarded. With multithreading, the discarding is done selectively; only instructions belonging to the *same* thread as the mispredicted instruction are discarded. Thus, all entries above the mispredicted one, *and* with a matching thread ID are discarded.

3.5 Write back and result commit

When a functional unit completes a computation, the result is written back to the SU. The entry with the same tag number as that of the result gets updated. Uniqueness of tags guarantees that matching tag numbers will produce correct results, and the TID can be ignored. The Write Back stage of the pipeline is thus exactly the same as for single-threaded execution.

In a processor that utilizes a reorder buffer, only instructions in the lower-most block (of size 4 instructions in the SDSP) may write resluts into the register file, provided there are no exceptions associated with them. This is necessary for exception recovery when speculative execution is done.

For a multithreaded processor, the situation is quite different. Figure 2 shows a picture of what the reorder buffer and instruction window might look like at some point in execution. Assume that block #0 is *not* ready to commit its results at this stage but block #1 *is*. If the instructions in block #1 belong to a thread different from the ones in block #0, then the former are allowed to write their results to the register file. If they *do* belong to the same thread, result commit is not permitted for block #1. If block #1 is unable to commit its results, block #2 is examined. Block #2 will commit its results only if they are ready *and* the instructions belong to a thread that is different from *both* block #0 and block #1. In this way, any number of blocks can be examined, limited by: (a) the complexity the designer is willing to bear to have this feature, and (b) timing constraints of the result commit stage, since the process of making a decision causes a delay. The ability to commit results from some block other than the lower-most one will be referred to as Flexible Result Commit. The reorder buffer used in our simulations allows us the flexibility to examine four blocks of instructions for result commit.

Though the techniques described are specific to SDSP's style of instruction scheduling, they can easily be extended to other schemes that achieve out-of-order execution. If the processor used a mechanism like the CDC 6600 scoreboard or Tomasulo's algorithm [11], for instance, the only change needed would be that the TID would no longer be ignored, but used along with the register number for tracking dependencies between instructions. The techniques described here are also just as applicable if reservations stations were used instead of an instruction window.

4 Simulation methods

A total of eleven benchmarks, all written in C, have been simulated. Each one is compiled, assembled and linked into a single object module, using software tools for the SDSP processor. The object module is then loaded into an instruction-level simulator, which simulates the benchmark as accurately as possible by maintaining the values of the register set (including PC), reorder buffer, instruction window and store buffer on a cycle-by-cycle basis.

The simulator is scalable and reconfigurable in terms of the hardware configuration it represents. There are a total of 128 registers, which are shared by the threads. Allocation of registers is static, being done at compile-time. In our case, the 128 registers are distributed equally among all threads. This allocation is easy to implement, and is also consistent with our model of parallel programming, in which all threads execute the same piece of code on different items of data. This method of programming, known as *homo-*

Type of FU	Default no.	Other no.	Latency
Integer ALU	4	6	1
Integer Multiply	1	2	2
Integer Divide	1	2	15
Load	1	2	1
Store Unit	1	2	1
Control Transfer	1	1	1
FP Add	1	2	3
FP Multiply	1	2	6
FP Divide	1	2	40

Table 1: **Functional unit configuration. Latency is in cycles.**

geneous multitasking, makes use of the data parallelism in an application. A number of problems can be solved using this style of parallel programming: like Sieve, Laplace, Water, MP3D, Matrix multiply and many of the Livermore loops. These constitute the set of benchmark programs used for this research. Water and MP3D have been obtained from the SPLASH suite [10] of benchmarks, Sieve and Laplace have been written by Robert Boothe [3], and Matrix (multiply) has been written by the authors. Six Livermore loops were chosen, which exhibit varying amounts of data parallelism, and of different granularity. They all have different characteristics that make them suitable or unsuitable for multithreading to varying degrees. Livermore loops are identified by numbers. They will be referred to as "*LL #n*". LL1, LL2, LL3, LL4, LL7 and LL22 have been simulated.

4.1 Default configuration

Unless stated otherwise, all applications are programmed to run with four parallel threads, which execute simultaneously on the processor. The register file is shared equally by the streams, irrespective of the latter's number. The compiler for the SDSP was modified to produce code for a register set of different sizes for this purpose. In all simulations, a constant fetch bandwidth of four instructions per cycle is assumed. A 2-bit algorithm for branch prediction is used, as described in [5]. Since the code executed by all streams is the same, only one BTB is maintained, regardless of the number of threads. Branch instructions of all threads update the same history after execution. While this may seem too simplistic, it yielded prediction accuracies upwards of 80% for all applications.

The result commit policy is Flexible Result Commit. Dependence analysis is done through the renaming tags allocated during the decode stage. A 4-way set associative 8KB data cache with a line size of 16 bytes and a perfect LRU replacement algorithm is accurately simulated. This is the default model, a direct-mapped cache being the other one used, when so stated. An 8-entry store buffer exists between the cache and the SU for all simulations. The scheduling unit, consisting of combined reorder buffer and instruction window, has 32 entries. The word "entry" refers to an individual instruction. Thus, 8 blocks can be accommodated. The default functional unit configuration is listed in Table 1. The SU can issue up to 8 instructions for execution per cycle, as the functional units may write eight different results back to the SU in any cycle. The middle column of Table 2 summarizes this configuration. The right-hand side column lists non-default values that were used in some of the simulations.

5 Performance analysis

It is essential to establish a base case of superscalar operation at the outset, to serve as a point of reference. This will be provided by the execution of non-multithreaded (single-threaded) code of the eleven benchmarks on a processor with a configuration as described in Section 2. In addition to the functional units of the SDSP, we will assume one floating point (FP) unit each for addition, multiplication and division. For convenience, results are presented in two groups: Group I contains all the simulated Livermore loop benchmarks, and Group II contains the remaining benchmarks: Laplace, MP3D, Matrix, Sieve and Water.

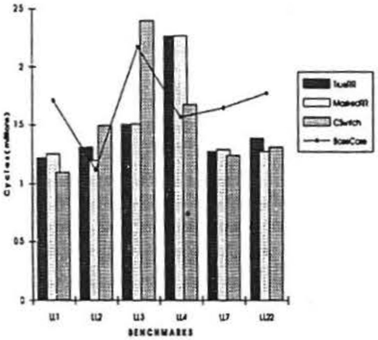

Figure 3: **Cycles (millions) of execution of Livermore loops for different different fetch policies.**

5.1 Fetch policy

Traditionally, the issue of context switch policy on a multithreaded processor has received great attention [1, 4, 9, 2, 3]. For a processor with lookahead, the instruction window creates a time gap between the fetch and execute stages. Therefore, it is more appropriate to talk about its fetch policy. Three different mechanisms are compared.

Feature	Default Value	Others
Number of threads	4	6, 5, 3, 2 or 1
Fetch Bandwidth	4 instructions/cycle	—
Branch Prediction	2-bit hardware predictor	—
Result Commit from	Bottom 4 blocks of RB	Lower-most block only
Register Renaming	Full renaming	1-bit scoreboarding
Bypassing of results	Have bypassing	No bypassing
Data Cache	8K, 4-way set associative, line size = 16 bytes, LRU replacement algorithm	Direct-mapped cache of 8K
Instruction cache	Perfect cache (100% hits)	—
Store Buffer depth	8 entries	—
Depth of Sched. Unit	32 entries	64, 48, or 16 entries
Functional Units	See Table 1	Table 1
Writes to RBIW/cycle	Eight	—
Insns Issued/cycle	Eight	—

Table 2: **Hardware Configuration**

The simplest of these, *True Round Robin*, allocates one fetch cycle to all threads, from 0 through *N-1*, in cyclic order. This is implemented via a modulo N (N = number of threads) binary counter. At every clock tick, the thread with ID equal to the value of the counter is allowed to fetch a block of instructions. The counter is advanced on every clock tick, irrespective of the state of execution (running or waiting on an event) of the threads. This results in cycle-by-cycle interleaving of instruction blocks placed into the instruction window. This policy will be referred to as True Round Robin (True RR) fetch, since each thread gets a turn to fetch once every N cycles. All multithreaded simulations described in this paper use True RR by default.

The second mechanism, known as *Masked Round Robin*, is similar to True RR, with the difference that one or more threads can be skipped in the process of circular selection. If thread #2 were temporarily suspended, say because of a synchronization delay, and $N = 4$, the order of fetching would be ...0,1,3,0,1,3,... Once the synchronization attempt is successful for thread #2, fetching would resume for it. In true round robin, the order would always be ...0,1,2,3,0,1,2,3..., irrespective of the state of any thread. The benefit of this scheme is that threads with low parallelism can be skipped, allowing other threads to take their place in the SU. This policy will be referred to as *Masked Round Robin*, since threads may be "masked" out from the selection process from time to time.

The difficulty with this approach is in determining when to exclude a particular thread. It requires a reliable means of knowing when the execution rate of a particular thread is likely to be low. The criterion that was used is as follows: every time a thread fails to commit its results from the lower-most block in the reorder buffer, fetching for that thread is suspended until the commit *does* take place. The effectiveness of approach depends on the latency of the operation that fails to commit results. The longer the latency, the more beneficial excluding that thread is likely to prove.

Both of the above schemes are based on cycle-by-cycle interleaving of blocks of instruction blocks from different threads. The alternative, called *Conditional Switch*, is to continue fetching from the same thread until there is an indication of its rate of execution becoming low. Several multithreaded processors follow this policy [1, 3, 2]. However, there is an essential difference between processors with and without lookahead in this regard. When fetching for a thread is stopped in the former type of processor, instruction execution continues as long as there are fetched blocks in the SU. A scalar processor simply completes the ones already initiated in the pipeline.

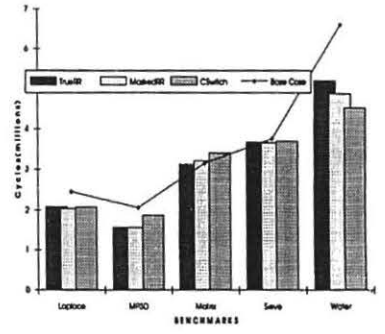

Figure 4: **Cycles (millions) of execution of Group II benchmarks (Laplace, MP3D, Matrix, Sieve and Water) for different fetch policies.**

As in the case of scalar processors, the problem of determining when to switch arises. Ideally, the decision to switch threads should be made as close to the fetch stage as possible. The earliest this can be done is during decode. Upon detecting certain instructions, the decoder sends a switch signal to the fetch mechanism. The fetch mechanism ceases fetching for the currently active thread and switches to another one. The instructions that can trigger a context switch are:

- integer divide,
- floating point multiply or divide,
- a synchronization primitive.
- a long-latency I/O operation.

Cache misses do not belong to this list, because the decision to switch is being made at the decode stage.

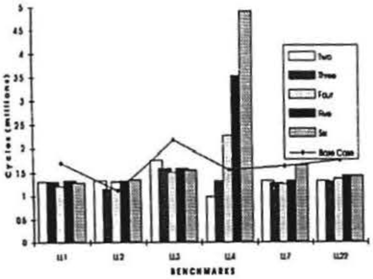

Figure 5: **Cycles of execution of Livermore loops for different numbers of threads in execution (1-6).**

Figures 3 and 4 show the cycles of execution for Group I and II benchmarks respectively, with the three different fetch policies. The base case is also shown for sake of comparison. Each benchmark is compiled to run with four parallel threads, which is the default number, as per Table 2. All other hardware features correspond to the configuration in the same table. "LL#n" refers to Livermore loop #n. Performance-wise, True RR and Masked RR emerge as about equivalent. While Masked RR has distinct advantages, it has the drawback of sometimes masking threads out unnecessarily. Threads may get masked owing to short-latency operations, and if this occurs frequently, it would result in a sparsely occupied SU. Conditional switch, which has been included for sake of comparison, has similar performance. This implies that the latencies of operations that trigger a context switch for this policy are not a bottleneck in the processor's execution rate. Of the three policies, True Round Robin is the easiest to implement.

5.2 Number of threads

Figures 5 and 6 present the results of execution of the benchmarks with 1, 2, 3, 4, 5, and 6 threads. We shall use the term "peak improvement" of a benchmark to refer to its maximum improvement among all multithreaded simulations, i.e the maximum observed value of speedup among 2, 3, 4, 5, or 6 threads. The terms "speedup" and "improvement in performance" are synonymous, and are computed according to the formula:

$$Speedup = (Mt_{perf} - St_{pef})/St_{perf}$$

where Mt_{perf} and St_{perf} refer to multithreaded and single-threaded performance, respectively. Performance is defined as the reciprocal of number of cycles.

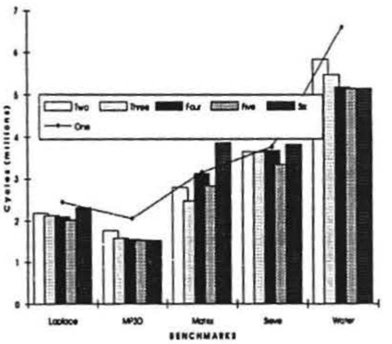

Figure 6: **Cycles of execution of Group II benchmarks for different numbers of threads in execution (1-6).**

Observed values of peak improvement for the simulated benchmarks lie between -8.6% to 57.2%, relative to the base case. A negative value indicates poorer performance than the base case (only LL2 consistently yielded a negative value, whereas LL4 yielded a negative value with a large number of threads). The average peak improvement for the Livermore loops was 33.1%. The remainder of the benchmarks grouped together showed an average peak improvement of 24.5%.

Measuring performance by number of threads, for Livermore loops on the average, an improvement of 25.26% over single-threaded execution was achieved with three threads. For six threads, there was a *deterioration* by 18.15%.

In Figure 5, LL4 can be seen to behave notably different from the others. This is because of a dependency that exists across loop iterations in this benchmark. Explicit synchronization primitives have to be inserted to guarantee correct execution. Its performance improves as number of threads is decreased, indicating the lower cost of synchronization. By the same token, it would be expected to produce the best results for single-threaded execution, which is not the case either. The negative effects of this synchronization are outdone by the greater parallelism when three or less threads are used.

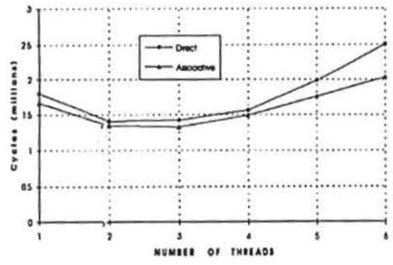

Figure 7: **Average execution cycles (in millions) of Livermore loops with direct and associative caches for different numbers of threads (1-6).**

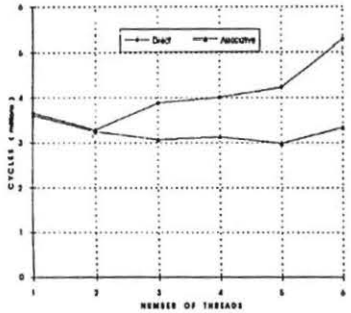

Figure 8: **Average execution cycles (in millions) with direct and associative caches for Laplace, MP3D, Matrix, Sieve and Water, with varying numbers of threads (1-6).**

5.3 Associative vs direct cache

To allow sharing, the cache could either be partitioned among threads, or a uniform cache can be shared by all threads. In the partitioned case, the space available to any one thread is small, since its accesses are limited to a section of the total cache. If the cache is uniform (not partitioned), the entire cache space is available to all threads. There is a high probability of contention, however, as each thread would try to establish its working set in the cache. Either way, the effectiveness of the cache is reduced. We picked a uniform cache for our study because it is simpler to design. Moreover, as the number of threads being executed changes, the sizes of partitions of the latter would also have to be changed. This problem does not arise in uniform caches.

There is another aspect to cache behavior in a multithreaded processor, one that can go in its favor. Consider the execution of threads that have small working sets, small enough that the working sets of all of them can be accommodated at the same time. In such a case, the total number of accesses to the cache is higher, even though the hit rate may remain the same. This is particularly true for a cache that is capable of servicing a cache miss at one location, while providing data to the processor from other locations at the same time. By doing so, only the thread that experiences the cache miss slows down, while the processor continues to execute the other threads. However, this requires a cache with higher design complexity. The simulations described in this paper assume that the cache is capable of servicing *one* line refill while simultaneously providing data. A second miss renders the cache incapable of servicing data requests, and requires that the missing lines be refilled. We use a 4-way set associative cache, with an LRU replacement policy.

Figures 7 and 8 show the performance of benchmarks with different numbers of threads in execution, with direct and associative caches. The hit rates observed for these simulations are listed in Table 3. As the number of threads increases, hit rate is seen to improve, and then fall. The reason for this is that the working sets of most threads can be accommodated as long as the number of threads is not too large, but beyond a certain point too many threads contend for the same locations in the cache, causing misses to occur more frequently. This effect is more pronounced for the Livermore loop benchmarks as compared to the others. The former are insignificant in the amount of code they contain, and exhibit great data locality. As a result, working sets are small. The others contain significant amounts of code, and their access patterns are less regular.

Threads	Benchmarks	Cache Hit Rate	
		Direct	Assoc.
6	Group I	66.33	86.83
6	Group II	60.20	91.50
5	Group I	72.83	86.33
5	Group II	66.40	94.40
4	Group I	74.50	80.00
4	Group II	73.60	91.17
3	Group I	70.33	80.00
3	Group II	77.20	89.33
2	Group I	71.33	77.67
2	Group II	93.46	93.77
1	Group I	65.83	72.33
1	Group II	90.44	93.33

Table 3: **Average hit rates for direct and 4-way set associative caches when simulated for different numbers of threads.**

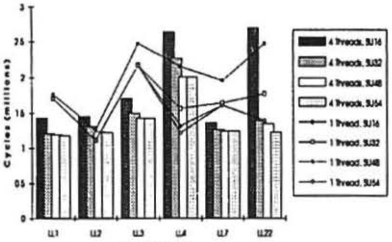

Figure 9: **Performance of Livermore loops for scheduling units with 64, 48, 32 and 16 entries.**

On the whole, performance is better with an associative cache than with a direct one. This is only to be expected, but note that the difference in performance keeps increasing steadily as the number of threads is increased. This is because there is greater contention with many threads accessing the cache than a single one. Also note that cache performance has a direct bearing on overall performance, as can be seen in the correlation between cache hit rate (Table 3) and cycles of execution (Figures 7 and 8).

5.4 Depth of scheduling unit

The lookahead capability of a processor [7] is determined by the size of its instruction window. Figures 9 and 10 show how varying the SU Depth affects performance. The difference in multithreaded and single-threaded performance reduces as a deeper SU is used. A deep SU helps in finding more independent instructions, making multithreading less useful. There is a significant increase in performance between 16 and 32 entry SU's. The difference between 32 and 48 entry cases is much less, and negligible for the next increment of 16. For some programs, a greater SU depth results in lower performance. The execution of Matrix for instance, is slower by 0.59% with a 64-entry SU as compared to a 48-entry SU. Two factors contribute to this behavior: first, branch prediction statistics are updated only when an instruction is shifted out of the SU during result commit, and delayed updating might cause other branch instructions to be mispredicted. The second contributor is the restricted load/store policy. Since an instruction stays in the store buffer until its entry in the SU is shifted out, other stores, and consequently loads, could be prevented from being issued.

5.5 Functional units

The choice of numbers of functional units to employ depends on the available instruction-level parallelism and on the hardware cost of a functional unit. For single-threaded execution, the column *Default no.* of

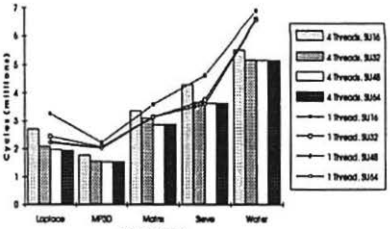

Figure 10: **Performance of Group II benchmarks for scheduling units with 64, 48, 32 and 16 entries.**

Table 1 lists a configuration of functional units that has been found to be suitable [12]. Column *Other no* of the same table lists a more enhanced configuration of functional units. This will be referred to simply as the "enhanced" configuration. Simulation results are shown in Figures 11 and 12. A "++" indicates that the enhanced configuration of functional units has been used for that simulation. With the default numbers of FU's, performance with multithreading is better by 11.60% for the Livermore loops. Group II benchmarks yield an improvement of 15.27%. With the enhanced configuration, the speedup for the Livermore loops over the single-threaded case with this configuration is 24.85%. For the remaining benchmarks, the equivalent speedup is 17.42%. Thus, for both groups of benchmarks, the relative speedup over single-threaded execution is greater with the enhanced configuration than with the default configuration. The improvement shown by the Livermore loops is greater, owing to their computation-intensive nature. To get an idea of the relative usefulness of each of the functional units, Table 4 lists the *percentage* of total execution cycles that the extra functional units were made use of, averaged over all benchmarks. These results argue strongly in favor of a second load unit, and a floating point multiplier, though the latter is more useful to the compute-intensive Group I benchmarks.

5.6 Result commit from multiple blocks

Figures 13 and 14 show the usefulness of committing results from a block other than the lower-most one with multithreading. For the Livermore loops, performance was better by an average of 34.61% when result commits took place from multiple (four) blocks. Without this ability, scheduling unit stalls occur with greater frequency. (See Figure 2). Group II benchmarks showed an improvement of of 11.21% with this feature. The point of reference of single-threaded execution was not used here, since, as explained in Section 3, doing this with only one thread is not feasible.

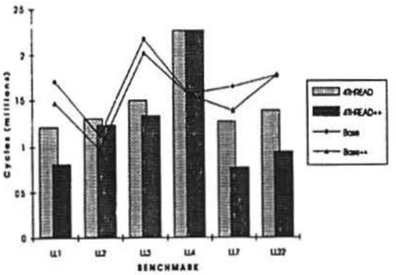

Figure 11: **Comparison of execution cycles for different numbers of functional units (Livermore Loops). "++" indicates the enhanced configuration.**

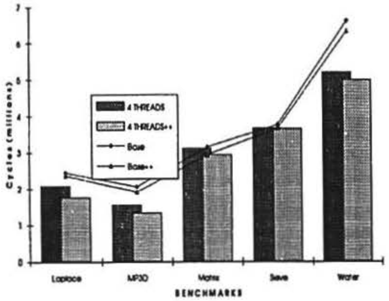

Figure 12: **Comparison of execution cycles of Group II benchmarks for different configurations of functional units. "++" refers to the enhanced configuration.**

Benchmarks	Extra Unit	% Cycles Used
Group I	Integer ALU	1.630
Group II	Integer ALU	3.516
Group I	Integer ALU	0.748
Group II	Integer ALU	0.972
Group I	Load Unit	8.482
Group II	Load Unit	13.006
Group I	Store Unit	0.000
Group II	Store Unit	0.000
Group I	Integer Multiply	1.377
Group II	Integer Multiply	3.352
Group I	Integer Divide	0.017
Group II	Integer Divide	0.134
Group I	FP Add	10.115
Group II	FP Add	4.616
Group I	FP Multiply	30.140
Group II	FP Multiply	7.754
Group I	FP Divide	10.563
Group II	FP Divide	0.164

Table 4: **Average usage of extra functional units as a percentage of total cycles.**

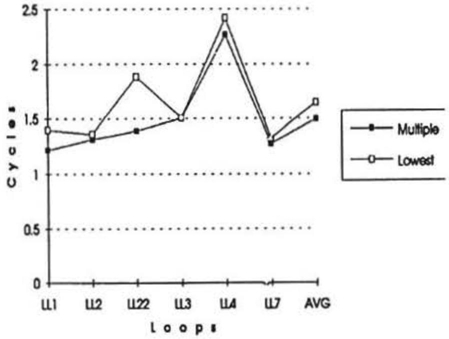

Figure 13: **Comparison of execution cycles with the reorder buffer committing results from a single and multiple (four) blocks, for Group I benchmarks.**

6 Conclusion

This paper showed how multithreading can be used for creating larger amounts of Instruction-level Parallelism in the workload. Based on simulations on our superscalar architecture, we observed a speedup of 20 to 55% for most benchmarks. For the extra cost of hardware incurred, this is a significant improvement. The caveat, of course, is that there is a cost associated with multithreading, which can reduce overall performance. An example of this is LL2, which showed consistently poorer performance compared to the base case in Section 5.2. In the final analysis, it is the characteristics of the program itself (one of which *is* its instruction parallelism) that determine how beneficial multithreading will be.

One of the considerations in devising a multithreading technique was minimizing the hardware overhead. Of the key components of the SDSP, the instruction unit and the logic for result commit are the only areas affected. The instruction unit has to keep track of several PC's instead of just one. This involves differentiating between the mispredicted operations of different threads, and providing a mechanism to sequence among the various PC's for instruction fetch. As results in Section 5.2 indicate, a modulo *N* binary counter is sufficient for this purpose (to implement the True Round Robin fetch policy).

The change to the result commit stage involves writing results from one of four different locations to the register file, as opposed to a single one. Minor changes are also required in the decode stage, handling of mispredicted control transfer instructions and access to the register file. The remaining components – register file, reorder buffer, instruction window, func-

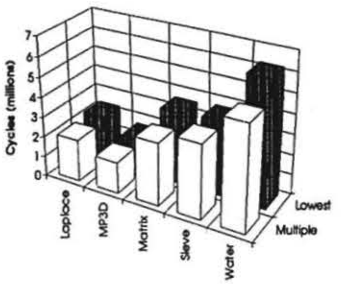

Figure 14: **Comparison of execution cycles with the reorder buffer committing results from a single and multiple (four) blocks, for Group II benchmarks.**

tional units and instruction issue logic are virtually unchanged. An additional field for thread ID of the instruction is the only change in the scheduling unit.

6.1 Scope for improvement

Several alternatives may be considered in trying to improve performance further, some of which are:

1. Employ more cache ports and functional units, especially the scarce ones.

2. Align instructions in memory in such a way that control transfer operations lie at the end of a fetched block, and branch targets at the beginning of a block. That way, all of the fetched instructions in a block will be valid.

3. Use a judicious fetch policy, that slows down fetching for a thread in a region of low execution rate.

4. Use software scheduling to eliminate unnecessary delays owing to synchronization.

The final alternative, that of software scheduling and code rearrangement, can have a great impact on performance. One of the limitations of the multithreading paradigm that has been used in this study is that the code the threads execute is the same. The software scheduling algorithm is static, which has its own limitations. However, even with static scheduling, one can write parallel code for an application in more than one way. In many cases, it may be possible to reduce the synchronization overhead by rearranging code and dividing tasks judiciously.

References

[1] Anant Agarwal. "Performance tradeoffs in multithreaded processors,". *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Poretrfield, and B. Smith. "The Tera Computer System,". In *Proceedings of International Conference on System Science*, pages 1–6, June 1990.

[3] Bob Boothe and Abhiram Ranade. "Improved multithreading techniques for hiding communication latency in multiprovessors,". In *Proceedings of the* 19^{th} *International Symposium on Computer Architecture*, pages 214–223, May 1992.

[4] R. H. Halstead Jr and T. Fujita. "A multithreaded processor architecture for parallel symbolic computing,". In *Proceedings of the* 15^{th} *International Symposium on Computer Architecture*, pages 443–451, June 1988.

[5] John Hennessey and David Patterson. *"Computer Architecture: A Quantitative Approach,"*. Morgan Kaufmann Publishers, Inc., San Fransisco, California, second edition, 1996.

[6] Mike Johnson. *"Superscalar Microprocessor Design,"*. Prentice Hall, Englewood Cliffs, 1991.

[7] R. M. Keller. "Look-ahead processors,". *Computing Surveys*, 7(4):177–195, December 1975.

[8] IBM Microelectronics and Motorola Inc. *"PowerPC 603 RISC Microprocessor User's Manual,"*, 1994.

[9] Rishiyur S. Nikhil and Arvind. "Can dataflow subsume von Nuemann computing?,". In *Proceedings of the* 16^{th} *International Symposium on Computer Architecture*, pages 262–272, 1989.

[10] J.P. Singh, Wolfe-Dietrich, and Anoop Gupta. "SPLASH: Stanford Parallel Applications for Shared-Memory,". Technical Report CSL-TR-92-526, Stanford University, Computer Systems Laboratory, Stanford University, CA 94305, 1992.

[11] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". *IBM Journal*, 11:25–33, January 1967.

[12] Steven Wallace and Nader Bagherzadeh. "Performance Issues of a Superscalar Microprocessor,". In *Proceedings of the 1994 International Conference on Parallel Processing*, volume 1, pages 293–297, August 1994.

[13] Steven Wallace, Nirav Dagli, and Nader Bagherzadeh. "Design and Implementation of a 100 MHz Reorder Buffer,". In *37h Midwest Sympsium on Circuits and Systems*, August 1994.