

A Fine-Grain Multithreading Superscalar Architecture

Mat Loikkanen and Nader Bagherzadeh
Department of Electrical and Computer
Engineering
University of California, Irvine
loik, nader@ece.uci.edu

Abstract

In this study we show that fine-grain multithreading is an effective way to increase instruction-level parallelism and hide the latencies of long-latency operations in a superscalar processor. The effects of long-latency operations, such as remote memory references, cache-misses, and multi-cycle floating-point calculations, are detrimental to performance since such operations typically cause a stall. Even superscalar processors, that are capable of performing various operations in parallel, are vulnerable. A fine-grain multithreading paradigm and unique multithreaded superscalar architecture is presented. Simulation results show significant speedup over single-threaded superscalar execution.

1: Introduction

Multithreading continues to be an active area of research in computer science and engineering. In a general sense, multithreading is the logical separation of a task into independent *threads* whose work may be done separate from one another, with limited interaction or synchronization between threads. A thread may be anything from a light-weight process in the operating system domain to a short sequence of machine instructions in a program. It may also be meant to denote a static entity such as the instructions that make up a program, or it may be a dynamic activation of the program. In the work presented here, a thread specifies instruction-sequence-level parallelism. Multithreading is defined to be the execution of a multi-threaded program on a processor that is able to exploit the existence of multiple threads in increasing performance.

The execution of multiple threads, or contexts, on a multithreading processor has been considered as a means for hiding the latency of long operations [1, 8] and for

feeding more independent instructions to a processor with lookahead/out-of-order capabilities and, thus, exposing more instruction level parallelism [2, 3, 8]. The negative effect that a long-latency instruction can have on the performance of a processor is clear: processing generally cannot continue until the operation has completed; processor throughput and overall performance suffers

Domain decomposition multithreading for a superscalar architecture has been investigated, and the benefits of this program partitioning based on data parallelism are presented in [3]. This work also uncovered significant stalling of the fetch phase due to long latency floating-point operations.

A dynamically-scheduled, functional-thread execution model and processor architecture with on-chip thread management is presented in this paper. Threads are used to present more instruction level parallelism to the processor in the form of independent streams of instructions, and to mitigate the effect of long-latency instructions by allowing for the hiding of such latency with thread suspension and scheduling. Hardware support for multithreading is added to a base superscalar processor. Long-latency instructions are decoupled from the main pipeline to execute independently in a functional unit so that instructions from other threads may pass through the pipe uninhibited.

Though the multithreading concepts and general thread scheduling methods presented in this paper are not necessarily new, the combining of these and the incorporating of them into the multithreading superscalar processor model described herein is. We present a unique multithreading superscalar processor architecture.

2: The Superscalar Architecture

The base superscalar architecture for this research is derived from and is very similar to that of an on-going research project at the University of California, Irvine, named the Superscalar Digital Signal Processor (SDSP) [10] (see Figure 1). The superscalar employs a central instruction window into which instructions are fetched, and from which they are scheduled to multiple, independent functional units, potentially out-of-order. Stalls prevent instruction results at the bottom of the buffer from being committed to the register file, and new instructions from being fetched.

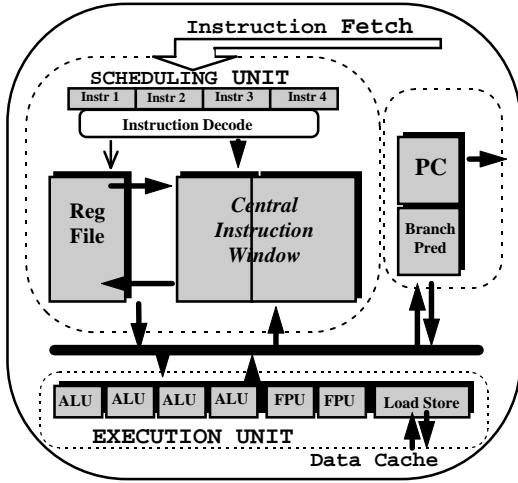


Figure 1 Base Superscalar (SDSP) Block Diagram

3: Threads

A thread is a machine code sequence that is explicitly specified as distinct from other threads of the same program; thread creation and synchronizing instructions are used. These instructions not only partition the code into threads but also specify parallelism and serve to direct multithreaded control flow and synchronization. Threads are concurrent and independent: no data dependencies exist between them. A thread is static in the form of a sequence of instructions as seen in an assembly listing of the code, and becomes a dynamic entity as the processor executes instructions of the thread.

The *fork* instruction is used to spawn a new thread of control in the program. The spawned thread begins at a new instruction address (PC) while the parent falls through to the next instruction after the *fork*. The *join* instruction is used to synchronize or merge multiple threads into a single continuing thread. The last thread to execute the *join* instruction continues execution with the following instruction while all others die ([4]).

Thread Register Allocation and Remapping. A register usage and allocation scheme that dynamically partitions the register file and uses runtime register remapping was devised (see also [9]). This partitioning allows threads to execute the same code, and assumes a general purpose register set of 32 registers, but may be scaled to other sizes. Register numbers specified in machine instructions are logical registers which are mapped, during decode, to physical registers according to a thread's Register Relocation Map (RRM) [4]--see

Figure 2. Actually only a subset of the registers are remapped, as others serve as registers common between threads (e.g. for synchronization).

The user-visible register set consists of thirty-two 32-bit general purpose registers; the processor actually has 64 physical registers. The compiler, or programmer, allocates/uses from the logical register set, and the processor re-maps these registers to its physical register set.

The RRM of the initial thread is always zero, and subsequent threads that are forked or continue past a *join* receive their RRM from the *fork* or *join* instruction. Table 1 lists the possible register set partitionings and associated RRM values. For example, if two threads are executing in parallel, they both utilize logical registers 1 through 15 which are remapped to physical registers 1 through 15 for one of the threads and 33 through 47 for the other.

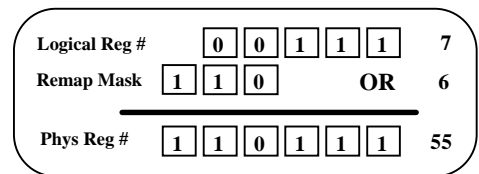


Figure 2 Register Remapping Example

Table 1 Register Set Partitioning

Threads	Logical Regs	Register Remap Values					
		T#0	T#1	T#2	T#3	T#4	T#5
1	1 - 15	0	n/a	n/a	n/a	n/a	n/a
2 to 3	1 - 15	0	4	6	n/a	n/a	n/a
4 to 6	1 - 7	0	1	4	5	6	7

4: The Multithreading Architecture

Hardware enhancements to the base superscalar, and thread data and control flow are detailed in this section (further details can be found in [4]). Latency hiding without stalling is accomplished by suspending a thread. The instruction is removed from the main pipeline of the processor (central instruction window) and placed into a buffer associated with the functional unit doing the computation.

The block diagram in Figure 3 illustrates the flow of control and data as they relate to thread scheduling. The Thread Attribute Store (TAS) keeps information about thread activations, including the program counter of a thread and its ready state. The TAS feeds the Fetch Unit

with the instruction address of a ready thread. Context switching is accomplished with round-robin fetching from different threads on a cycle-by-cycle basis. If the decode phase sees a long-latency instruction, its thread in the TAS is suspended. When the long-latency instruction is issued, the entry data for the instruction is transferred from the central buffer into the functional unit's Thread Suspending Instruction Buffer (TSIB). If the instruction completes before it reaches the bottom, its result is written back into the central buffer, as with normal instructions. However, if the instruction entry reaches the bottom of the central buffer before completion, a completion signal is sent to its TSIB. In this case the result is committed directly from the TSIB to the register file. The TAS is also informed when a long-latency instruction is retired so that its thread may be activated.

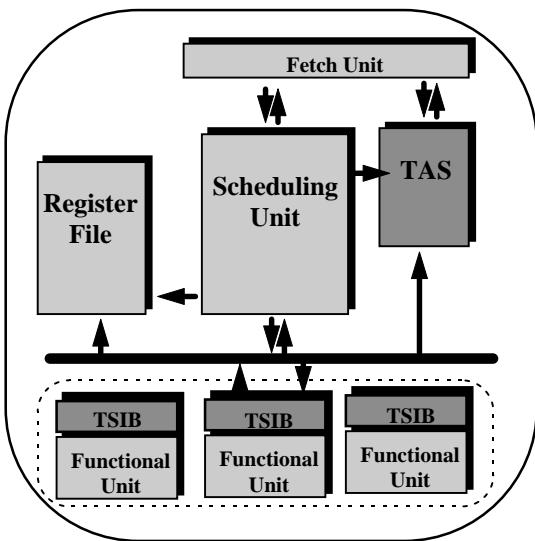


Figure 3 Thread Data and Control Flow

Thread Suspending Instructions (TSI). Certain instructions are classified as Thread Suspending Instructions (TSIs) and are recognized as such when decoded. These include remote loads, division, and certain floating-point instructions. Initial simulations showed that instructions with latencies of less than approximately ten cycles should not cause thread suspension. In fact, suspending on these shorter long-latency instructions caused decreasing performance on occasion.

The Thread Attribute Store (TAS) contains entries for thread activations ready to execute and those which have been suspended. An entry is added to the TAS when a *fork* instruction is executed, an entry is suspended

because of a TSI, an entry is activated when a TSI completes, and an entry is invalidated when a thread dies by executing a *join*. For our simulations, the TAS contained a maximum of six thread entries, which placed an upper-bound on the number of concurrent threads.

Thread Suspending Instruction Buffers (TSIB). A TSIB exists for each functional unit which executes TSIs. A TSIB is similar to a reservation station except that an instruction does not enter a TSIB until it is ready to execute. TSIBs are the key behind eliminating stalls: instructions are removed from the main pipeline, allowing continued flow.

Scheduling Unit and Pipeline. In addition to the standard base superscalar functions of the Scheduling Unit ([4]), the multithreaded version performs some specialized functions. It important to note that a single-threaded program executes just as fast (number of cycles) on the multithreaded superscalar as on the base superscalar.

The initial PC of the program belongs to thread 0 and is loaded into TAS entry 0, while subsequent thread activations are born and die as *fork* and *join* instructions are executed. A round-robin algorithm is used by the TAS to accomplish cycle-by-cycle "context switching." This has been shown to be desirable because of its simplicity and its effectiveness [2]. The superscalar scheduling unit is fed with instructions, that, by virtue of being from different, independent threads are free of dependencies that normally limit out-of-order issue/execution.

Instruction Fetch. The TAS supplies the Fetch Unit with a Thread ID and the address from which to fetch a block of instructions. The Fetch Unit returns the next sequential address to the TAS, who then stores it into the thread's PC register. If the cache signals the Fetch Unit that the request cannot be satisfied and that a line-fill has started, the TAS is told to suspend the thread, and the PC and Thread ID are buffered in the Fetch Unit's Cache Miss Buffer for the duration of the line-fill. The cache-controller signals the Fetch Unit when the line-fill completes, which in-turn signals the TAS to re-activate the thread.

Instruction Decode. As in the base superscalar, up to four instructions are decoded each cycle: tags are created and register operand lookup is performed. Register remapping is also done. If one of the four instructions is a TSI then the following is done: 1) The TAS is informed that the corresponding thread should be suspended. (The TAS will not suspend the thread if it is the only ready thread.) 2) The Fetch Unit is told to cancel the current fetch if that fetch happens to be for the same thread for

which the TSI was decoded. 3) Instructions following the TSI in the same block of four are invalidated.

Instruction Issue. Instruction Issue occurs essentially the same as in the base superscalar as far as the choosing of instructions to issue to available functional units. The TAS itself is a functional unit as it executes up to one *fork* or *join* instruction per cycle.

Instruction Execution. Instruction execution is, for the most part, the same as in the base superscalar.

Data Cache-Miss. When a load instruction misses the data cache (Dcache) it effectively becomes a long-latency instruction even though it was not initially recognized as such in the Decode phase. Dealing with Dcache misses is different than dealing with Lcache misses with respect to hiding line-fill latency since a Dcache-miss is not recognized until the execute phase of the pipeline. At most, only a few instructions must be discarded in a scalar processor [8], equal to the number of pipeline phases that precede the phase in which the Dcache-miss is recognized. However, many more instructions would have to be invalidated in a four-way superscalar, and thus more work thrown away which has already been done. This is because up to four instructions are fetched each cycle, and instructions may be buffered in the central window for several cycles before being issued to functional units. It is not feasible to invalidate all instructions of the thread that came after the Dcache-missing load instruction since this represents potentially many cycles worth of work that has already been done. Thus there is no reason to suspend a thread upon a Dcache-miss. Simulations did, however, show that multithreading still mitigates the effects of Dcache misses.

Remote Loads and Stores. Remote loads are issued into a Load Queue of TSIBs of the Load Unit. Remote stores are buffered in the Store Queue in the same manner as local stores (to assure in-order completion). When a remote load instruction is issued to the Load Unit, the instruction is added to the Load Queue and the request is sent out to the network. Completion of the remote load instruction occurs when the requested data arrives. (The thread was suspended when the instruction was decoded.)

Instruction Commit and Write-Back. The completion of a TSI is relatively complex (see [4] for details). In-order commit of instructions must be maintained. A TSI cannot simply always commit to the register file directly from the TSIB since instructions which preceded it might still be in the central buffer, and must be committed before the TSI. It is also not feasible to keep a TSI in a TSIB until that instruction reaches the

bottom of the buffer since this would effectively keep the functional unit busy and unavailable until that time.

Branch Handling and Mispredict Recovery. Branch prediction occurs normally except that the PC affected is that of the branch instruction's thread. The invalidation of instructions because of a mispredict happens only to those instructions in the Scheduling Unit that are from the same thread as the mis-predicted branch instruction. TSIBs also "monitor" invalidation signals.

5: Results and Analysis

The Processor Simulator. The program to be executed is loaded and relocated into memory, and data areas, including the runtime stack, are allocated (see [4]). The simulator then fetches and "executes" the program instructions. The processor state (i.e. program counter and register file) and data memory state are kept and modified dynamically. The simulator is quite detailed with respect to the Scheduling Unit and the Thread Attribute Store while the functional units are black boxes with latencies. The benchmark programs were annotated with fork and join language extensions recognized by the compiler. Such annotations were made to partition obviously independent program code. See [4] for details.

Figure 4 shows the execution cycle counts of the benchmarks, comparing the single-threaded (ST) run against the multithreaded (MT) run. The cycle counts were normalized to the common denominator of one million cycles for the single-threaded runs. The results show an average multithreading speedup of 1.51.

An interesting phenomenon occurs with the execution of the *wc* which has no thread suspending instructions (no remote-load instructions, no divide) and had a very high single-threaded instructions per cycle (IPC) count of approximately 3.0. Recall that we are processing on a 4-way superscalar. The essentially negligible performance increase obtained from multithreading can be attributed to this high single-threaded IPC, and shows that for multithreading to be of benefit, there must be room for improvement in the single-threaded version of the program. *li* and *di* represent the ideal situation for latency hiding: the latencies of divides and remote loads were almost completely hidden.

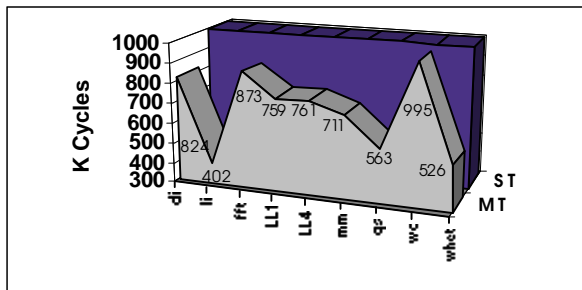


Figure 4 MT Performance: Perfect Caches

During other simulations, the latency of remote loads were varied from 5 to 50 cycles to see what effect this change has on performance. It was shown that in the ideal case, as the remote load latency increased so did the speedup of multithreading over single-threaded execution ([4]). At some point, however, the speedup would stabilize and then begin to decrease since the amount of independent latency hiding code is, of course, finite.

A fair amount of hardware support was added to specifically handle thread suspension and for removing thread suspending instructions from the pipeline into the TSIB. Figure 5 shows the difference between multithreading results obtained with and without suspending. The point of reference is the normalized result of running multithreaded programs with the processor in no-suspend mode. The results in the background represent multithreading speedup obtained with thread suspension (MT) over no-suspend multithreading (Mtns), and the results in the foreground represents single-threaded (ST) performance. Overall, it is clear that suspension functionality of the processor is worth the added cost as multithreading would otherwise not offer nearly as much benefit.

Hiding Instruction Cache Miss Latencies. The cache modeled was direct-mapped, non-blocking, had a line size of 4 instructions, and line-fill was done on read misses only. It was found that multithreading does in fact hide latencies of Icache misses, and offers substantial speedup over single-threaded execution. In general, it could be said that speedups increase as the size of the Icache is decreased because more cache misses occur and thus more opportunity for its latency hiding exists, though lack of code spatial-locality causes thrashing when the size becomes too small [4].

Figure 6 shows the improved performance of multithreading with an Icache large enough to contain the entire program once initially loaded. Appreciable improvement is seen even with this cold-start simulation test.

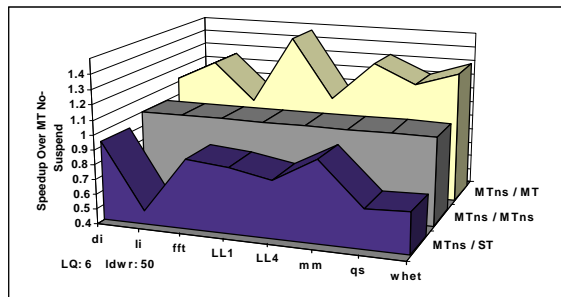


Figure 5 Multithreading Suspend Results

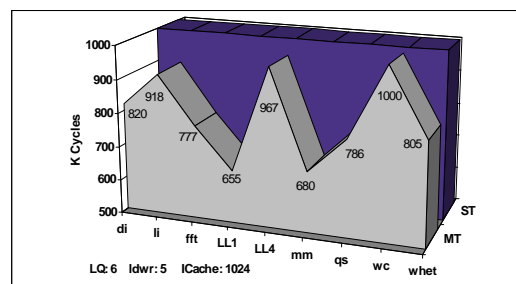


Figure 6 MT Performance: Icache Cold-Start

Data Cache Miss Latencies. Recall, the latency of a Dcache miss is not hidden but rather its effects are mitigated with multithreading. Figure 7 shows results. Other simulation results also showed that suspension logic for Dcache-misses was not feasible (recall that thread suspension is not done on a Dcache miss).

Full Model. A final simulation was done which tested the overall performance of multithreading with the processor parameterized with full thread scheduling and suspension, and with 8K instruction and data caches. Full model simulations were done to see if the modeling of the various latency hiding features would conflict with one another in a way as to decrease multithreading performance. Not only did this not occur, but speedup increased quite dramatically for certain programs. (See [4].)

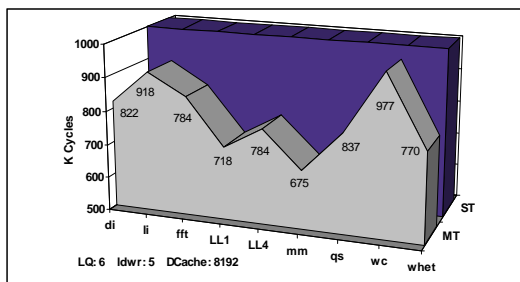


Figure 7 MT Performance: Dcache 8192

Table 2 Multithreading Performance: Full Model

Full Model Performance (ST, MT, Speedup - K cycle times)								
di	li	fft	LL1	LL4	mm	qs	wc	whet
603	398	524	3969	5932	308	619	694	689
496	160	407	3011	1763	221	372	694	353
1.22	2.49	1.29	1.32	3.36	1.39	1.66	1.01	1.95

6: Conclusions

The multithreading paradigm and unique multithreading superscalar processor architecture presented in this work can be decisively said to significantly improve performance over single-threaded superscalar execution. Simulation results showed that gains through multithreading increased as the opportunity for latency hiding increased (e.g. as the latency of remote-load instructions increased). The performance advantage of multithreading is, however, closely tied to the availability of independent instructions from different threads. When a thread is suspended, there needs to be a enough instructions from other threads to fill the void. It was also realized that if a program has very good single-threaded performance (stalls infrequently), multithreading offers less of a benefit.

References

- [1] Boothe, B.; Ranade, A. "Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors," *19th Annual International Symposium on Computer Architecture*, pp. 214-223, May 1992.
- [2] Gulati, M. and Bagherzadeh, N "Performance Study of a Multithreaded Superscalar Microprocessor", *Proceeding of the Second International Symposium on High-Performance Computer Architecture*, pp. 291-301, February 1996.

[3] Laudon, J., Gupta, A., Horowitz, M. "Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations", *Conference on Architectural Support for Programming Languages and Operating Systems*, 1995.

[4] Loikkanen, M. "A Fine-Grain Multithreading Superscalar Architecture," M.S. Thesis, University of California, Irvine, 1995.

[5] Papadopoulos, G.M., Traub, K.R., "Multithreading: a revisionist view of dataflow architectures," *18th Annual International Symposium on Computer Architecture*, pp. 342-351, May 1991.

[6] Smith, B.J. "A Pipelined, Shared Resource MIMD Computer," (HEP), *Proceedings of the 1978 International Conference on Parallel Processing*, pp. 6-8, 1978.

[7] Thekkath, R., Eggers, S.J. "Impact of Sharing-Based Thread Placement on Multithreaded Architectures," *21st Annual International Symposium on Computer Architecture*, pp. 176-186, 1994.

[8] Tullsen, D.M., Eggers, S.J., Levy, H.M. "Simultaneous Multithreading: Maximizing On-chip Parallelism," *22nd Annual International Symposium on Computer Architecture*, June 1995.

[9] Waldspurger, C.A., Wehl, W.E. "Register Relocation: Flexible Contexts for Multithreading," *Computer Architecture News*, Vol.21 No.2, May 1993.

[10] Wallace, S.D., Bagherzadeh, N. "Performance issues of a superscalar microprocessor," *Microprocessors and Microsystems*, April 1995.