Speculative Thread Execution in a Multithreaded Dataflow Architecture

Wentong Li, Krishna Kavi, Afrin Naz and Phil Sweany
Department of Computer Science and Engineering
P. O. Box 311366, University of North Texas, Denton, TX 76203.
email: {wl, kavi, an0042, sweany}@cse.unt.edu

Abstract

Instruction Level Parallelism (ILP) in modern Superscalar and VLIW processors is achieved using out-of-order execution, branch predictions, value predictions, and speculative executions of instructions. These techniques are not scalable. This has led to multithreading and multi-core systems. However, such processors require compilers to automatically extract thread level or task level parallelism. Loop carried dependencies and aliases caused by complex array subscripts and pointer data types limit compilers' ability to parallelize code. Hardware support for thread-level speculation (TLS) allows compilers to more aggressively parallelize programs using speculative thread execution, since hardware will enforce correct order of execution. In this paper, we show how thread-level speculation can be implemented within the context of our Scheduled Dataflow architecture and provide preliminary performance analysis.

1 Introduction

Superscalar and VLIW architectures are the main architectural models used in commercial processors. These models rely on out-of-order execution, branch prediction, speculative execution of instructions, value prediction, and trace caches to achieve high degrees of instruction level parallelism (ILP). But researchers have shown that simply adding more functional units, or dynamic scheduling of instruction are approaching diminishing returns in terms of further improving single processor performance [1]. This has led to an increased interest in architectures that support concurrent execution of multiple threads. There are now commercially available products like the Intel hyper-threading architecture that implements the SMT (simultaneous multithreading) [6], and Intel DUO architecture which implements a dual core CMP (chip multiprocessor) on a single chip. These architectures are derived from the superscalar model. The complexity of the underlying superscalar architecture makes it harder to scale the clock frequency to improve performance for these designs.

We have been exploring hybrid von Neumann – dataflow models of execution for the past decade. Our SDF (scheduled dataflow) architecture uses dataflow-like non-blocking threads, but uses the conventional von Neumann execution model for instructions within a thread [4]. We have shown that SDF outperforms superscalar and VLIW based models [5]. We have also shown that SDF compares favorably with SMT-like architectures. We believe that higher Instructions Per Cycle (IPC) can only be achieved when multiple threads, either from a single program, or from a workload are executed on multi-core processors with large numbers of functional units. Compiler technology is improving in automatic extraction of thread-level parallelism from programs written in imperative languages like C. However, compile time analyses are conservative in order to guarantee program correctness. Compilers can more aggressively extract parallelism if hardware checks are in place to guarantee program correctness. Architecture researchers are exploring one such hardware mechanism: support for speculative thread execution [7, 8, 9, 10]. In this paper, we propose speculative thread execution within the context of our SDF. We provide preliminary data on the performance gains resulting from our thread-level speculation.

2 Related Research

Thread-level speculation is a technique that enables compilers to optimistically parallelize applications despite ambiguous data or control dependencies. Most parallel compilers extract parallelism by spawning multiple loop iterations if the compiler can clearly identify loop-carried dependencies and resolve aliases due to array subscripts or pointers. Thread-level speculation (TLS) hardware will enforce dynamic data and control dependency checks. Once a violation is detected, the system will squash the results of speculative threads and restart the computation. Marcuello et al. [8] proposed a multithreaded micro-architecture that supports speculative thread execution within a single processor. This architecture dynamically spawns speculative threads. It contains multiple instruction queues, register sets, and a very complicated multi-value cache to support speculative execution of threads. Zhang et al. [9] proposed a scheme that supports speculative thread execution in large scale distributed shared memory (DSM) systems relying

on cache coherence protocols. Steffan et al. [10] proposed an architecture that supports TLS execution both within a CMP core and across distributed nodes in DSMs. This design is based on conventional architecture, but needs very extensive support from the operating system. The design is based on cache coherence protocols. However, the published literature does not provide details on the implementation. Our design needs a small amount of extra hardware to implement speculation in the context of SDF architecture. It can be fully integrated with the data cache coherence controller to ensure the correctness of execution across multiple clusters of SDF cores.

3 SDF Architecture Review

SDF architecture differs from other multithreaded architectures in two ways: i) our programming paradigm is based on non-blocking threads, and ii) we decouple all memory accesses from execution pipeline. Data is preloaded into an enabled thread's register context prior to its scheduling on the execution pipeline. After a thread completes execution, the results are post-stored from its registers into memory (or frames of awaiting threads). We use two separate processing elements: SP performs preload and post-store while the Execution Processor (EP) performs actual computations of a thread. A third processing element is responsible for scheduling threads and moving them between EPs and SPs. The continuation of a thread is represented by a four-tuple – <FP, IP, RS, SC>; FP is the Frame Pointer (where thread input values are stored), IP is the Instruction Pointer (which points to the thread code), RS is a Register Set (a dynamically allocated register set), and SC is a Synchronization Count (the number of values needed to enable that thread). The continuation fully and uniquely identifies a thread.

Unlike Superscalar, our architecture performs no (dynamic) out-of-order execution of instructions of a thread and thus eliminates the need for complex instruction issue and retiring hardware.

4 Thread-Level Speculation Schema for the SDF Architecture

For the original SDF architecture, our compiler generated sequential threads when ambiguous dependencies existed among threads, to guarantee correct execution. With some hardware to support speculative execution of threads and committing results only when the speculation is verified, a complier can more aggressively create threads to execute concurrently.

4.1 The SDF Architecture Supported by the Schema

Our TLS schema not only supports speculative execution within a single node of SDF cluster consisting of multiple EPs and SPs, but also supports speculation in SDF clusters using distributed shared memory (DSM) protocols. Our design is derived from a variation of the invalidation based MESI protocol [11]. We add extra hardware in each node to maintain intra-node coherence.

4.2 States in Our Design

In our schema, an invalidate message will be generated by a node to acquire exclusive ownership of data stored in a cache line before updating the cache. Three states, Excusive (E), Shared (S), and Invalid (I), are needed to maintain the inter-node coherence. We add two more states: speculative read of an exclusive data (SpR.Ex) and speculative read of a shared data (SpR.Sh). We can distinguish the states easily by adding an extra S (Speculative read) bit to each cache line. Table 1 shows the encoding of the cache line states.

Table 1: Encoding of Cache Line States

State	SpRead	Valid	Dirty(Exclusive)
I	X	0	X
E/M	0	1	1
S	0	1	0
SpR.Ex	1	1	1
SpR.Sh	1	1	0

4.3 Hardware Design for Our Schema

In the new architecture, a (speculative) thread is defined by a new continuation consisting of a 7-tuple – <FP, IP, RS, SC, EPN, RIP, ABI>. The first four elements are the same as the original continuations in SDF (see Section 3). The added elements are: epoch number (EPN), re-try instruction pointer (RIP) and address-buffer ID (ABI). For any TLS schema, an execution order of threads must be defined based on the program order. We use epoch numbers (EPN) for this purpose. Speculative threads must commit in the order of their epoch numbers. RIP defines the instruction at which a failed speculative thread must start its retry. ABI defines the buffer ID that is used to store the addresses of speculatively-read data. For the non-speculative thread, the three new fields will all be set to zero. We use a separate queue for speculative threads to control the order of their commits. Figure 1 shows the

overall design of our new architecture.

For the controller (Thread Schedule Unit) to distinguish between speculative and non-speculative threads, it only needs to test the epoch field of the continuation - any continuation that has a non-zero epoch number is a speculative thread. Speculative threads commit strictly in the order of the epoch numbers. The commit control maintains the epoch number of the next thread that can commit based on the program order and will test the epoch number of a continuation that is ready for commit. If these numbers are the same and no data access violations are found in the reorder buffer associated with the thread, the commit controller will schedule the thread for commit. If there is a violation, the commit controller sets the IP of that continuation to RIP and places the thread back in the preload queue for re-execution. At this time, the thread becomes non-speculative.

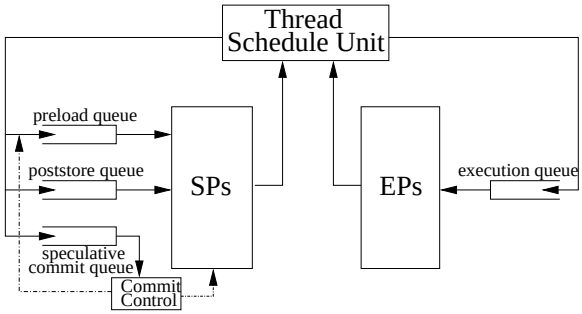


Figure 1: Overall Design

In order to achieve address matches in parallel for the speculative threads, we add a few small fully-associative buffers to record the addresses of data that is speculatively accessed by a thread. Data addresses are used as indices into these buffers. The address buffer ID (ABI) is assigned when a new continuation for a speculative thread is created. When a speculative read request is issued by a thread, the address of the data being read is stored in the associated address buffer assigned to the thread and the entry is set to valid. When a speculatively read data is subsequently written by a non-speculative thread, the corresponding entries in the address buffers are invalidated, and this will prevent a speculative thread from committing. To determine whether a continuation is valid or not, we only need a simple OR- tree, to logically-or all entries of the buffer assigned to that thread. The block diagram of address buffer for a 4-SP node is shown in Figure 2. This design allows invaliding a speculatively-read data in all threads simultaneously. It also allows different threads to add different addresses into their buffers. When an invalidate request comes from the bus or a write request comes from inside the node, the data cache controller will change the cache line states, and the speculative controller will search the address buffer to invalidate appropriate entries.

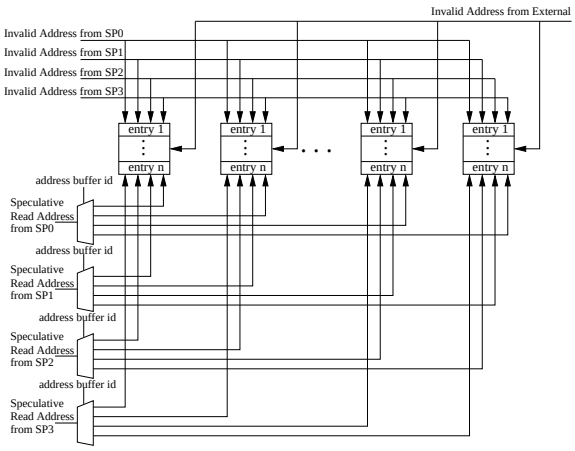


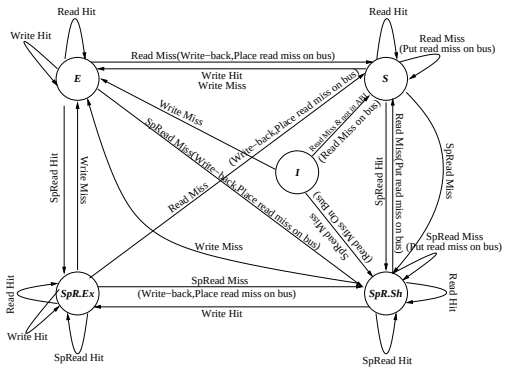
Figure 2: Address Buffer Block Diagram

Threads in SDF architecture are fine-grained and thus the number of data items read speculatively will be small. We believe that by limiting the number of speculative reads by a thread, we can improve the probability that a speculative thread does not fail. Unlike other TLS models, since our threads are non-blocking, we allow threads to complete execution even if some of their speculatively read data is invalidated. The results of a thread (in post-store) are not committed unless all speculative reads remain valid at the time the thread is ready for commit. An invalid speculation will force the thread to retry using RIP pointer.

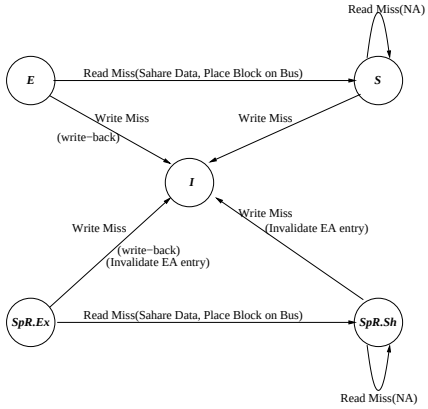
4.4 States Transition Diagram

In our design, a speculative thread cannot write any results to data cache. Once a speculative thread is allowed to commit, it starts to write its result into cache (performed by SP in post-store phase).

Figure 3 shows the state transition diagrams for tracking data reads and writes by speculative and non-speculative threads. Figure 3(a) shows the cache line state transitions due to requests from a node within a cluster (i.e., intra-node). The key idea is that every speculative read will change the cache line states to speculative and also allocates an entry in the corresponding ABI buffer and every (non-speculative) write will invalidate the entries in the ABI buffer. Figure 3(b) shows the cache line state transitions due to the memory bus activities (i.e., inter-node transactions). The write miss message from the bus will invalidate cache line and corresponding ABI entries. Due to the page limits, we will not be able explain each of the state transitions in detail. Most transactions are similar to MESI type cache coherency protocols.



(a) Request from Internal



(b) Request from Bus

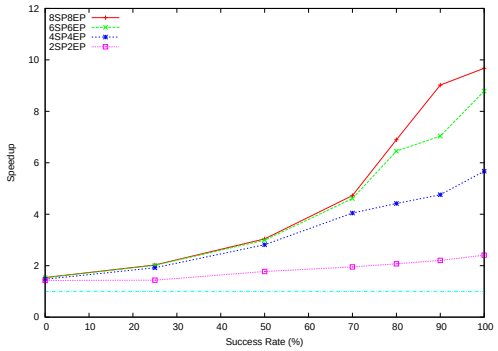
Figure 3: State Transition Diagrams

4.5 Instruction Set Architecture Extension

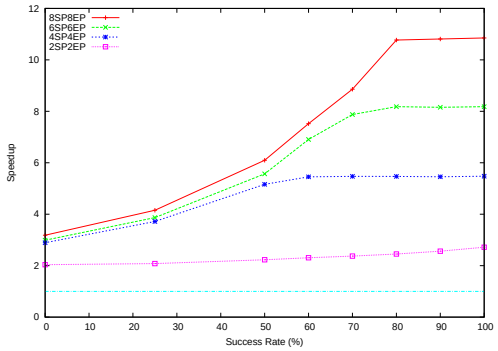
We added 3 new instructions to the SDF for thread-level speculation. The first instruction is for speculatively spawning a thread. This instruction will request the system to assign an epoch number and an ABI for the new continuation. The second instruction is for speculatively reading data, which will cause the addition of an entry into the address buffer associated with that continuation. It should be noted that not all reads of a speculative thread are speculative reads. A compiler can resolve most data dependencies and use speculative reads only when static analyses cannot determine memory ordering. It should also be noted that when a speculative thread is invalidated, the retry needs only to re-read speculatively-read data. The third instruction is for committing a speculative thread. This instruction places the speculative thread continuation into the speculative thread commit queue.

5 Experiments and Results

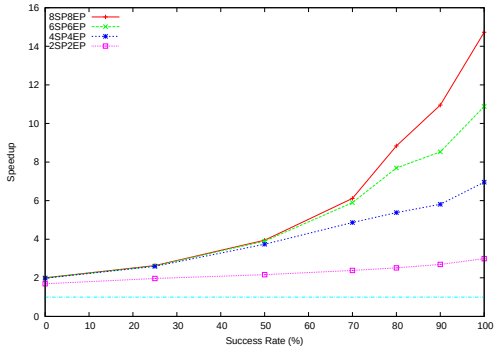
We extended our SDF simulator with this speculative thread execution schema. This simulator performs cycle-by-cycle functional simulation of SDF instructions. We used synthetic benchmarks to evaluate the performance of our design.



(a) SP:EP 33%:66%



(b) SP:EP 66%:33%



(c) SP:EP 50%:50%

Figure 4: Performance Model of TLS Schema

5.1 Synthetic Benchmark Results

We created benchmarks that execute a loop containing variable number of instructions. We controlled the amount time a thread spends at SPs and EPs by controlling the number of LOADs and STOREs (for SP load) and computational instructions (for EP load). We parallelized these benchmarks with speculation. We now show the results of our evaluation.

Figure 4(a) shows the performance of a program that will spend 33% of the time in SPs and 67% of time in EPs.

Figure 4(b) shows the performance for threads with 67% of SP time, 33% of EP time, and Figure 4(c) shows the performance for a 50%-50% EP-SP loads. All benchmarks are tested using different success rates of speculation. We configured our simulator with different number of functional units: 8SPs-8EPs, 6SPs-6EPs, 4SPs-4EPs, and 2SPs-2EPs to evaluate the scalability of our thread-level speculation.

Since our SDF performs well when the SPs and EPs have balanced load, we would expect best performance for the case when we started with a balanced load (Figure 4(c)), and when the success of speculation is very high (closer to 100%). However, even in this case, as the speculation success drops (and is closer to zero), the load on EPs increase because failed threads will have to re-execute their computations. As stated previously, a failed thread only needs to re-read (or pre-load) the data items that were read speculatively and data from a thread are post-stored only once when the thread speculation is validated. Thus a failed speculation will disproportionately add to EP load. For the case shown in Figure 4(b), with a smaller EP loads, we obtain higher speed-ups (compared Figures 4(a) or 4(c)) at lower success rates of speculation, since EPs are not heavily utilized in this workload. For the 33%-66% work load in Figure 4a, even a close to100% success rate will not lead to good performance gains on SDF, because EP is overloaded to start with, and the speculative execution adds to the load of EPs.

Figure 5 below shows the ratio of instructions executed by EPs and SPs as the fail rate increases. With the increase of the fail rate, EPs will be more heavily loaded. Since more instructions must be executed per thread (on retry on a mis-speculation), the performance drops as the success rate drops. The balance of the workload between EP and SP will affect the scalability of Of speculation. With SP33:EP66 load, the EP is heavily loaded and it shows the worst performance with speculation, compared to the other two workloads. Figure 6 shows the increase in the workload, normalized to the case when the speculation success is 100%. Since most of the re-try work is done by the EP, when EP is less heavily loaded than SP (EP/SP work ratio is small) we see smaller increases in the overall work, as the fail rate increases. When the speculation success approaches zero, (indicating strict sequential ordering of threads) the overhead due to retries can actually cause a performance drop in a multithreaded system. The SP66:EP33 workload which has the smallest increases in the overall workload has the best performance when the fail rate above 50%.

We can also see that when the success rate of speculation is below 50%, there are insignificant differences among the performance gains resulting from different number of functional units (number of EPs and SPs). This is because of sequential execution resulting from dependencies (leading to lower speculation successes) and cannot utilize available functional units.

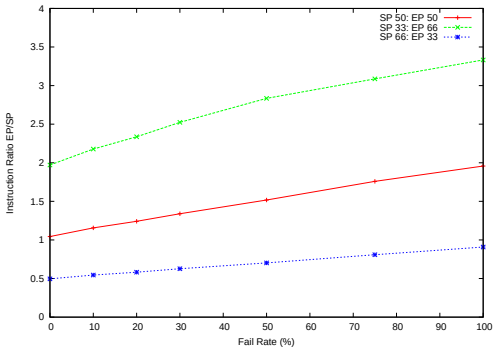


Figure 5: Ratio of Instruction Executed EP/SP

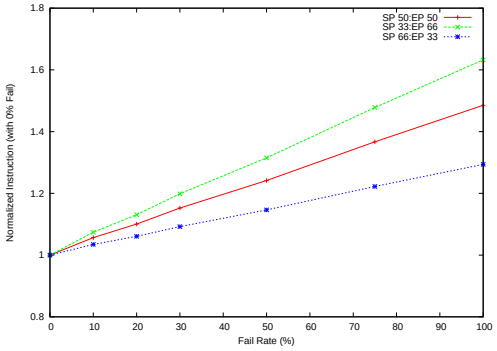


Figure 6: Normalized Instruction Ratio to 0% Fail Rate

From this group of experiments, we can draw the following conclusions. Speculative thread execution can lead to performance gains over a wide range of speculation success probabilities. In all workloads with different load on SPs and EPs, we can obtain at least 2 fold performance gain when the success of speculation is greater than 50%. If the success rate drops below 50%, one should turn off speculative execution to avoid excessive retries that can overload EPs. When the EP load is less than the SP load, we can tolerate higher rates of mis-speculation. When the success rates are below 50%, the performance does not scale well with added SPs and EPs (8SPs-8EPs, 6SPs-6EPs, and 4SPs-4EsP all show similar performance). This suggests that the success of speculation can be used to decide on the number of SPs and EPs needed to achieve optimal performance.

6 Conclusion and Future Work

In this paper, we described hardware support for thread-level speculation (TLS) in the context of our SDF architecture and evaluated the performance of such TLS support by varying workloads, number of functional units and success rates of speculations. With hardware TLS support, compilers can aggressively generate parallel threads

even with unresolved loop-carried dependencies. The TLS support works well with a wide range of speculative thread success rates $(0\% \sim 100\%)$. Our architecture achieves scalable performance with added functional units when the success of speculation is high (greater than 50%). Our hardware TLS support is much simpler than TLS support in traditional architectures that are based on superscalar paradigm. We are currently working to extend the experiments with real benchmarks. We are also planning to explore silent writes [13] for buffering writes from speculative threads to further improve performance by retiring speculative threads early.

References

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger. "Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures", *27th International Symposium on Computer Architecture (ISCA)*, pp.248-259, June 2000.

[2] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C.K. Kim, D. Burger, S.W. Keckler, and C.R Moore. "Exploiting ILP, TLP, and DLP Using Polymorphism in the TRIPS Architecture", *30th International Symposium on Computer Architecture (ISCA)*, pp. 422-433, June 2003.

[3] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. "WaveScalar", *In Proceedings of the 36th International Symposium on Microarchitecture(MICRO)*, pp291-302, December 2003.

[4] K.M. Kavi. H.S. Kim and A.R. Hurson. "Scheduled dataflow architecture: A synchronous execution paradigm for dataflow", *IASTED Journal of Computers and Applications*, pp114-124, Vol. 21, No. 3 (Oct. 1999).

[5] K.M. Kavi, R. Giorgi and J. Arul. "Scheduled Dataflow: Execution paradigm, architecture and performance evolution", *IEEE Transactions on Computers*, pp 834-846, Vol. 50, No. 8, August 2001.

[6] D.M. Tullsen, S.J. Eggers, H.M. Levy, and J.L. Lo, "Simultaneous multithreading: Maximizing on-chip parallelism", *In International. Symposium on Computer Architecture (ISCA)*, pp. 392-403, June 1995.

[7] M. Franklin and G.S. Sohi. "ABR: A Hardware Mechanism for Dynamic Reordering of Memory References", *IEEE Transactions on Computers*, Vol. 50, No. 5, May 1996.

[8] P. Marcuello, A. Gonzalez and J.Tubella. "Speculative Multithreaded Processors", *In proceeding of the International Conference on Supercomputing*, pages 77-84, July 1998.

[9] Y. Zhang, L. Rauchwerger, and J. Torrelas. "Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors", *In 5th International Symposium on High-Performance Computer Architecture (HPCA)*, pp.135-141, January 1999.

[10] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry, "A Scalable Approach to Thread-Level Speculation", *27th International Symposium on Computer Architecture (ISCA)*, pp.1-12, June 2000.

[11] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach (3rd edition)*, ,2003.

[12] A.R. Hurson, J.T. Lim, K.M. Kavi, and B. Lee, "Parallelization of DOALL and DOACROSS Loops – A Survey", *Advances in Computers*, pp.53-103, Vol. 45, 1997

[13] K.M. Lepak and M.H. Lipasi, "On the Value Locality of Store Instructions", *27th International Symposium on Computer Architecture (ISCA)*, pp.182-191, June 2000.