A Non-blocking Multithreaded Architecture with Support for Speculative Threads

Krishna Kavi[1], Wentong Li[1], and Ali Hurson[2]

[1] University of North Texas
{kavi,wl}@cse.unt.edu
[2] Missouri University of Science and Technology
hurson@mst.edu

Abstract. In this paper we provide both a qualitative and a quantitative evaluation of a decoupled multithreaded architecture that uses non-blocking threads. Our architecture is based on simple in-order pipelines and complete decoupling of memory accesses from execution pipelines. We extend the architecture to support thread level speculation using snooping cache coherency protocols. We evaluate the performance gains from speculations by varying the number of load/store instructions compared to computational instructions, miss speculation rates and the degree of thread level speculation. Our architecture presents a viable alternative to complex superscalar and super-speculative CPUs.

Keywords: Multithreaded Architectures, Cache Coherency, Thread Level Speculation, Decoupled Architecture.

1 Introduction

Superscalar and VLIW architectures are the main architectural models used in commercial processors. These models allow for more than one instruction to be issued on every cycle. Modern processors expend large amounts of silicon area and transistor budgets to achieve higher levels of performance with techniques such as out-of-order execution, branch and value prediction and speculative instruction execution. It has been shown that these techniques 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 processing, and multicore or chip multiprocessors (CMP) systems. The complexity of the underlying superscalar architecture makes it harder to scale the clock frequency for these designs.

It appears that the dataflow computing paradigm is back in vogue, as an alternative to superscalar models, as can be seen from recent architectural proposals including TRIPS [3, 4] and Wavescalar [5]. However, implementing dataflow model at instruction level (such as token driven models) requires complex hardware for communicating operands among instructions. In contrast, our architecture uses dataflow like synchronization at the thread-level, while using control flow semantics within a thread. This approach minimizes instruction level communication, but permits for scalable implementations. Our architecture should be also be contrasted with Wavescalar [5] that uses a complex memory-ordering scheme that involves tagging each memory transaction with a predecessor and successor memory access. We use epoch numbers with threads and extend cache coherency protocols to achieve proper memory ordering.

Our architecture differs from other multithreaded architectures in two ways: i) our threads are based on dataflow paradigm, and ii) we completely decouple all memory accesses from execution pipeline. The underlying non-blocking thread model permits for clean separation of memory accesses from execution (which is very difficult to coordinate in other programming models). Data is pre-loaded 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. The execution engine relies on control-flow like sequencing of instructions, but our architecture performs no (dynamic) out-of-order execution and thus eliminates the need for complex instruction issue and retiring hardware. These hardware savings may be utilized to include either more processing units on a chip or more register sets to increase the degree of multithreading. Moreover, it was stated that a significant power is expended by instruction issue logic, and the power consumption increases quadratically with the size of the instruction issue width [6], and thus our architecture should be more energy efficient since we perform in-order instruction issue.

We are able to perform some quantitative evaluation of our architecture using hand-coded programs. Our goal here is to provide both a quantitative (albeit limited in scope) and a qualitative evaluation of our innovative architecture. In this paper we extend our architecture to support speculative execution of threads using epoch numbers and provide some preliminary quantitative analysis.

1.1 Related Research

Compilers extract parallelism by spawning multiple loop iterations concurrently, and with hardware support for thread-level speculation (TLS) that enforces dynamic data and control dependency checks, compilers can more aggressively exploit thread level concurrency. Marcuello et. al., [7] proposed a multithread micro-architecture that supports speculative thread execution within a single processor. This architecture contains multiple instruction queues, register sets, and a very complicated multi-value cache to support speculative execution of threads. Zhang et. al., [8] proposed a scheme that supports speculative thread execution in large scale distributed shared memory (DSM) systems relying on cache coherence protocols. Steffan et. al., [9] proposed an architecture that supports TLS execution both within a CMP core and large scale 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, but 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.

2 Scheduled Dataflow Architecture

A processing element in our scheduled dataflow architecture (SDF) is composed of three components: Synchronization Processor (SP), Execution Processor (EP) and thread schedule unit. Each thread is uniquely represented by a continuation <FP, IP,

RS, SC>, where 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 context), and SC is the synchronization count (the number of inputs needed to enable the thread). The synchronization count is decremented when a thread receives its inputs, and the thread is scheduled on SP when the count becomes zero. SP is responsible for pre-loading data needed by the thread into its context (i.e., registers), and post-storing results from a completed thread into memory or *frames* of destination threads. The EP performs thread computations, including integer and floating point arithmetic operations, and spawns new threads. A more general implementation can include multiple EPs and SPs to execute threads from either a single task or independent tasks. Multiple SPs and EPs can be configured into multiple clusters. Inter-cluster communications will be achieved through shared memory.

An Example. To understand the decoupled, scheduled dataflow concept, consider one iteration of the innermost loop of matrix multiplication: c[i,j] = c[i,j] + a[i,k]*b[k,j]. Our SDF code is shown in Figure 1. In this example we assume that all necessary base addresses and indexes for the arrays are stored in the thread's frame. The thread is enabled after it receives all inputs in its frame, and a register context is allocated.

```
Preload:    LOAD      RFP|2, R2       # base of a into R2
            LOAD      RFP|3, R3       # index a[i,k] into R3
            LOAD      RFP|4, R4       # base of b into R4
            LOAD      RFP|5, R5       # index b[k,j] into R5
            LOAD      RFP|6, R6       # base of c into R6
            LOAD      RFP|7, R7       # index c[i,j] into R7
            IFETCH    R2, R3, R8      # fetch a[i,k] to R8
            IFETCH    R4, R5, R9      # fetch b[k,j] to R9
            IFETCH    R6, R7, R10     # fetch c[i,j] to R10
            FORKEP    body            # transfer to EP
            STOP

body:       MULTD     R8,R9 R11       #a[i,k]*b[k,j] in R11
            ADDD      R10,R11, R10    # c[i,j] + a[i,k]*b[k,j] in R10
            FORKSP    poststore       #transfer to SP
            STOP

poststore:  ISTORE    R6,R7, R10      #save c[i,j]
            STOP
```

Fig. 1. A SDF Code Example

SP executes the *preload* portion of the code to transfer data into the registers allocated for the thread. The *body* portion of the code is executed by the EP performing necessary computations while the *poststore* portion is completed by the SP to store results into either the frames of other threads (and possibly enabling them) or the I-structure [10]. I-structure access instructions (IFETCH and ISTORE) need a base and an index into the array and these values are contained in a pair of registers. Note that only SP accesses data caches (frame cache and I-structure cache) while EP only accesses thread registers. A thread can move between EP and SP as needed to fetch or store data from/to registers (FORKSP and FORKEP serve this purpose and they take 4 cycles). Although not shown in this example, SP can perform index and address computations since each SP is provided with an integer arithmetic unit. Unlike token driven models, our instructions (for example MULTD) are provided with a pair of store locations (in our example R8 and R9) for input operands so that the instructions need not be executed immediately when the second operand arrives (as is the case in token driven models). Our instructions are "*scheduled*" like control flow architectures using program counters. Our instruction driven approach eliminates the need for complex communications to exchange tokens among processing elements. We simplified this example to illustrate the general structure of SDF code. In general, techniques such as loop unrolling can be used to increase the size of the loop body, and multiple threads can be created to execute loop iterations in parallel.

3 Thread-Level Speculation Schema for the SDF Architecture

For the non-speculative SDF architecture, if there is an ambiguous RAW (true dependence) that cannot be resolved at compile time, the compiler generates sequential threads to guarantee correct execution using I-structure [10] semantics. This will reduce the performance of programs. However, with hardware support for speculative execution of threads and committing results only when the speculation is verified, a complier can more aggressively create concurrent threads.

3.1 SDF Architecture Supported by the Schema

Our TLS schema not only supports speculative execution within a single SDF cluster consisting of multiple EPs and SPs, but also supports speculation among SDF clusters using distributed shared memory (DSM). Our design is derived from a variation of the invalidation based MESI protocol [13]. By applying the MESI protocol, we can enforce coherence of data caches on different nodes in a DSM system. We add extra hardware in each node to maintain intra-node coherence.

3.2 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. In addition to the 3 states of MESI protocol (Excusive (E), Shared (S), and Invalid (I)), we add two more states: speculative read of an exclusive data (SpREx) and speculative read of a shared data (SpR.Sh)[1]. 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 states.

Table 1. Encoding of Cache Line States

	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

[1] We do not permit speculative writes.

3.3 Hardware Design of Our Schema

In the new architecture, a (speculative) thread is defined by a new continuation -- <FP, IP, RS, SC, EPN, RIP, ABI >. The first four elements are the same as the original continuations in SDF (see Section 2). The added elements are the epoch number (EPN), retry instruction pointer (RIP) and an 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[2] 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 add a separate queue for speculative threads to control the order of their commits. Figure 2 shows the overall design of our new architecture .

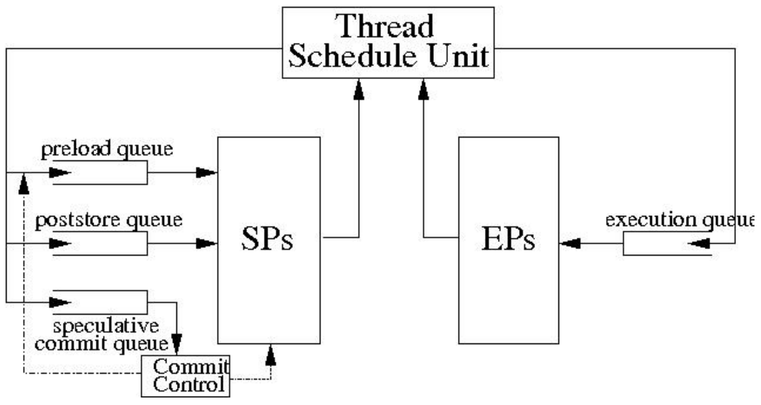


Fig. 2. Overall Design

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 to see if it is equal zero (as stated previously, a non-speculative thread's EPN is set to zero and any continuation that has a non-zero epoch number is a speculative thread). 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 (i.e, schedule the thread on SP for post-store). If there is a violation, the commit controller sets the IP of that continuation to RIP and places it back in the preload queue for re-execution. At this time, the thread becomes non- speculative.

We use a few small fully-associative buffers to record the addresses of data that are speculatively accessed by speculative threads. Data addresses are used as indices into these buffers. The small fully associative buffers can be implemented using an

[2] In our architecture, a thread commits its results to memory by executing the post-store part of its code.

associative cache where the number of sets represents the maximum number of speculative threads and the associativity represents the maximum number of speculative data that can be read by a thread. For example, a 64 set 4-way associative cache can support 64 speculative threads, with 4 speculative address entries per thread. 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 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, preventing speculative threads from committing. 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.

Threads in SDF architecture are fine-grained and thus the number of data items read speculatively will be small. By limiting the number of data items read speculatively, the probability that a speculative thread successfully completes can be improved. For example, if p is the probability that a speculatively read data will be invalidated, then the probability that a thread with n speculatively read data items will successfully complete is given by $(1-p)^n$. With 4 to 8 speculative reads per thread and 16 speculative threads, we only need 64 to 128 entries in the address buffers. Because our threads are non-blocking, we allow threads to complete execution even if some of the speculatively read data is invalidated. This eliminates complex mechanisms to abort threads, but may cause wasted execution of additional instructions of speculative threads.

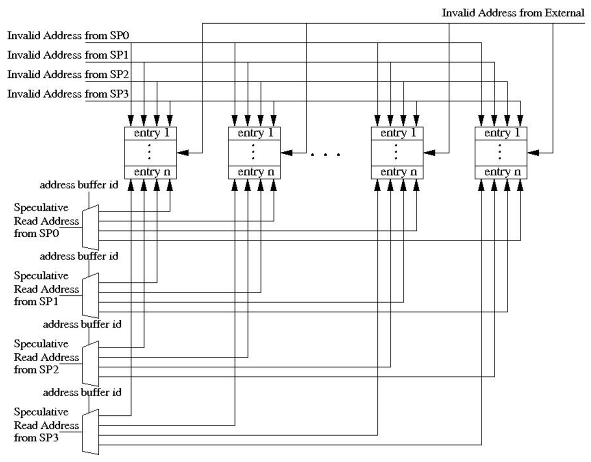
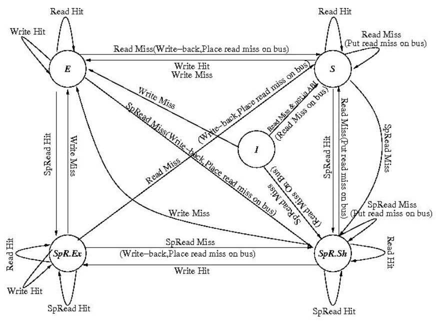


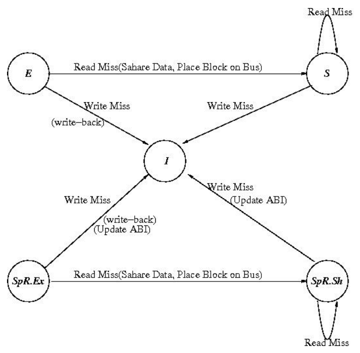
Fig. 3. Address Buffer Block Diagram

3.4 States Transition Diagram

A speculative thread cannot write any results to data cache. The results of a thread (during post-store) are not committed unless all speculative reads remain valid at the time the thread is ready for commit (in the order of epoch numbers). An invalid speculation will force the thread to retry using RIP pointer.



a. Request from a Node



b. Request from Bus

Fig. 4. State Transition Diagrams

Figure 3 shows the state transition diagrams for tracking data reads and writes by speculative and non-speculative threads. Figure 3a 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 state 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 3b shows the cache line state transitions due to 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 explain these diagrams in detail, but they are similar to MESI type cache coherency protocols.

3.5 Instruction Set Architecture Extension

We added three new instructions to SDF instruction set for thread-level speculation support. 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.

3.6 Experiment and Results

We extend our SDF simulator with this speculative thread execution schema. This simulator performs cycle-by-cycle functional simulation of SDF instructions.

3.6.1 Synthetic Benchmark Results[3]

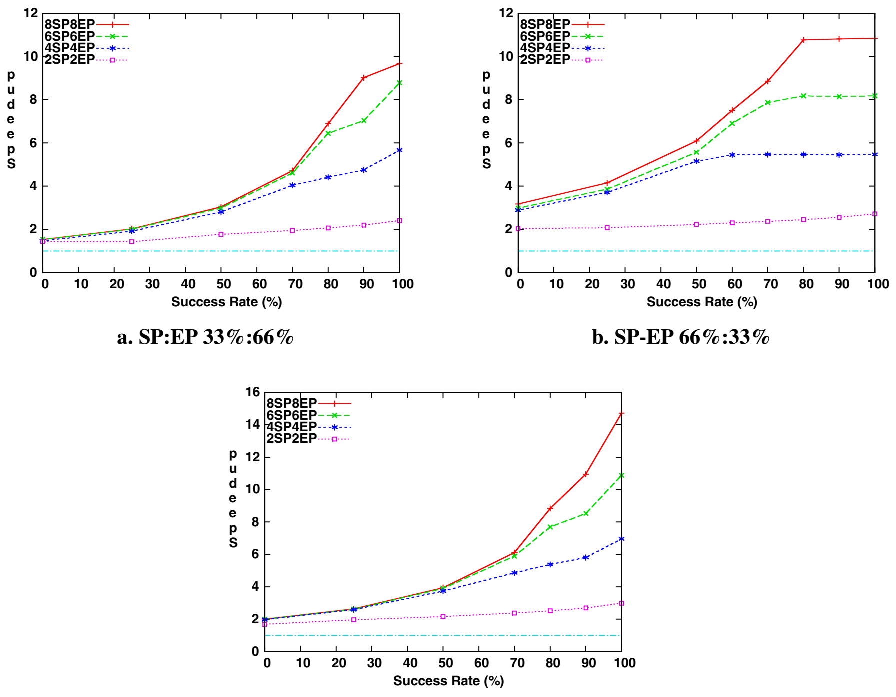
We created benchmarks that execute a loop containing variable number of instructions. We control the amount time a thread spends at SPs and EPs by controlling the number of LOADS and STORES (workload on SP) and computational instructions (workload on EP). Then we use the TLS to parallelize these benchmarks. We test this group of benchmarks both in term of the scalability and the success rate of the speculative threads.

Figure 4a shows the performance of a program that spends 33% of the time at SPs and 67% of time at EPs, when executed without speculation. Figure 4b shows the performance for programs with 67% SP workload, 33% EP workload, while Figure 4c shows the data for programs with 50% SP and EP workloads (if executed non-speculatively). All programs are tested using different speculation success rates. We show data with different number of functional units: 8SPs-8EPs, 6SPs-6EPs, 4SPs-4EPs, and 2SPs-2EPs.

Since our SDF performs well when the SPs and EPs have balanced load (and achieve optimal overlap of threads executing at EPs and SPs), we would expect best performance for the case shown in Figure 4c and when the success of speculation is very high (closer to 100%). However, even if we started with a balanced load, 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 the data items that were read speculatively and data from a thread are post-stored only when the thread speculation is validated. Thus a failed speculation will disproportionately add to EP workload. For the case shown in Figure 4b, with a smaller EP workload, we obtain higher speed-ups (compared Figures 4a or 4c) even at lower success rates of speculation, since EPs are not heavily utilized in this workload. For the 33%-66% SP-EP workload in Figure 4a, even a very high success rates will not lead to high performance gains on SDF, because EP is overloaded to start with, and the mis-speculative will add to the load of EPs.

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. 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 workload is less than the SP workload, we can tolerate higher rates of mis-speculation. Finally, 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.

[3] These are actual programs written for SDF and run on our simulator. We controlled the number of Load/Store instructions, and controlled which speculative threads successfully commit (post-store) their results.



a. SP:EP 33%:66%

b. SP-EP 66%:33%

C. SP:EP 50%:50%

Fig. 4. Performances Model of TLS Schema

3.6.2 Real Benchmarks

To further test our design, we selected a set of real benchmarks. We hand-coded these benchmarks using SDF assembly language. This group of benchmarks includes: Livermore loops 2 and 3; two major functions from compress() and decompress() (from 129.compress) and four loops chosen from 132.ijpeg. Table 2 shows the detailed description of the benchmarks. We code these benchmarks in two forms: one without speculation, where all the threads are executed linearly, and the other with speculation. In the speculative execution, earlier iterations (or threads with lower epoch numbers) generate speculative threads for later iterations (or threads with higher epoch numbers).

Table 2. Selected Benchmarks

Suite	Application	Selected Loops
Livermore		Loop2
Loops		Loop3
SPEC 95	129.Compress95	Compress.c:480 while loop
		Compress.c:706 while loop
	132.ijpeg	Jccolor.c:138 for loop
		Jcdectmgr.c:214 for loop
		Jidctint.c:171 for loop
		Jidctint.c:276 for loop

We evaluated performance gains using different number of SPs and EPs and the results are shown in Figure 5. The speculative execution does achieve higher speed-ups - between 30% and 158% for 2SP-2EP configuration and between 60% and 200% speedup for 4SP4EP configuration. To compare our results with those of [9], we use the parallel coverage parameter defined in [9]. Using 4SP4EP configuration to compare with their 4 tightly coupled, single threaded superscalar pipeline processors, for compress95 we achieve a speedup of 1.94 compared to 1.27 achieved by [9]; and a speedup of 2.98 for ijpeg compared to 1.94 achieved by [9].

Another finding from Figure 5 is that our performance does not scale well after 4SP4EP configuration. This is because of the way we generated threads – we generate very limited number of speculative threads, since each iteration only generates one new speculative thread. However with an optimizing compiler, it will be possible to generate as many speculative threads as needed to fully utilize available processing and functional units.

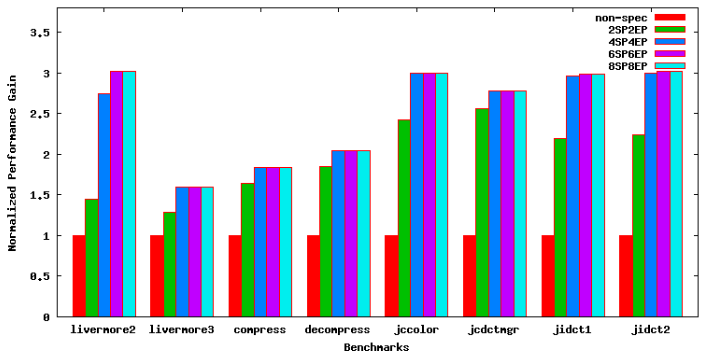


Fig. 5. Performance gains normalized to non-speculative implementation

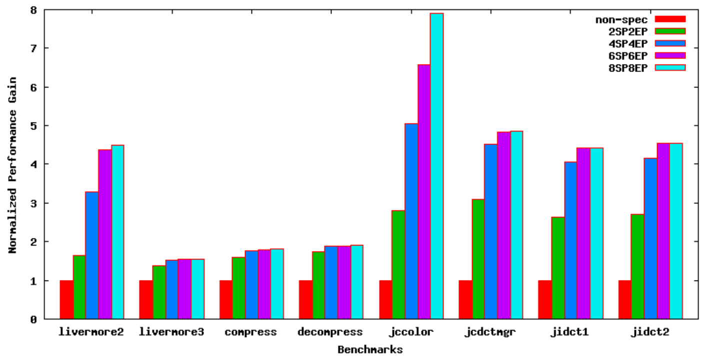


Fig. 6. Performance gains normalized to non-speculative implementation

We repeated our experiments with the same benchmarks but using a control thread that spawning multiple speculative threads at a time. For livermore loops, the control thread spawns 10 iterations a time, and for the compress95 and the jpeg, the control thread spawns 8 iterations a time. The results are shown in Figure 6. For most cases, this approach does show better scalability with added functional units. Livermore loop 3 and compress are the exceptions. For these applications, the mis-speculation is very high and since on mis-speculation all threads become non-speculative (executing sequentially) the available concurrency is reduced. It should be noted, however, our approach does lead to higher speedups than those reported in [9].

4 Summary and Conclusions

Our goal here is to provide a qualitative and a quantitative evaluation of an innovative non-blocking multithreaded architecture that decouples all memory access from execution pipeline. Our quantitative evaluations are limited to hand-coded benchmarks. At this time, we do not have a compiler, but we hope that we will be able find support to design and implement an optimizing compiler for our architecture. An optimizing compiler is needed to take full advantage of SDF features.

In previous sections we have shown that SDF can achieve scalable performance that is comparable or better than Simplescalar, VLIW and SMT architectural paradigms. We also have shown that thread level speculation on SDF can lead to speedups that are better than or comparable to other speculative execution models. In addition, SDF offers several qualitative advantages over existing architectural paradigms.

Separating PEs into SPs and EPs has distinct advantages. One can tailor the number of SP and EP units included in a single "computation cluster" to maximize performance of experimentally determined computation needs. The number and types of functional units (viz., integer and floating point arithmetic units) within these processing elements can also be varied. The EPs and SPs can easily be run at different clock speeds, providing power savings. Such control is easier to implement in our system than proposed globally asynchronous, locally synchronous (GALS) designs that contain multiple clock domains (MCD's) ([15], [16]). And by keeping the EP and SP pipelines extremely simple with no out-of-order instruction execution, we can address power constraints, provide additional computing power by including multiple simple SP and EP clusters on a chip, or more register sets.

SDF uses non-blocking threads, leading to non-preemptive scheduling of threads. Although real-time systems often use pre-emptive scheduling to meet required reactive times, non-preemptive scheduling is more efficient, particularly for soft real-time applications and applications designed for multithreaded systems, since the non-preemptive model reduces the overhead needed for switching among tasks (or threads) [17]. The decoupled memory of SDF implies that each thread goes through at least 3 scheduling points: *preload* when the thread's inputs (and I-structure data) are transferred to its registers at an SP, *execute* when the thread performs its computation at an EP, and *poststore* when the thread transfers results from its registers to memory or other threads at an SP. Each of these scheduling points allows us to determine which thread should be scheduled. Such fine-grained real-time scheduling is not possible with other thread models. The non-preemptive execution is applicable even to speculative threads, thus simplifying the management of thread-level speculation. All threads are allowed to complete but only those threads that can commit are allowed to complete post-store portions of their code.

References

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

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

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

[4] Burger, D., et al.: Scaling to the end of silicon with EDGE architectures. IEEE Computer, 44–55 (July 2004)

[5] Swanson, S., Michelson, K., Schwerin, A., Oskin, M.: WaveScalar. In: Proceedings of the 36th International Symposium on Microarchitecture(MICRO), December 2003, pp. 291–302 (2003)

[6] Onder, S., Gupta, R.: Superscalar execution with direct data forwarding. In: Proc of the International Conference on Parallel Architectures and Compiler Technologies, Paris, October 1998, pp. 130–135 (1998)

[7] Marcuello, P., Gonzalez, A., Tubella, J.: Speculative Multithreaded Processors. In: Proceeding of the International Conference on Supercomputing, July 1998, pp. 77–84 (1998)

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

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

[10] Arvind, Nikhil, R.S., Pingali, K.K.: Istructures: Data-structures for parallel computing. ACM Transactions on Programming Languages and Systems 4(11), 598–632 (1989)

[11] Burger, D., Austin, T.M.: The SimpleScalar Tool Set Version 2.0, Tech Rept. #1342, Department of Computer Science, University of Wisconsin, Madison, WI

[12] Terada, H., Miyata, S., Iwata, M.: DDMP's: Self-timed Super-pipelined Data-driven Multimedia Processor. Proceedings of the IEEE, 282–296 (February 1999)

[13] Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 3rd edn. (2003)

[14] Hurson, A.R., Lim, J.T., Kavi, K.M., Lee, B.: Parallelization of DOALL and DOACROSS Loops – A Survey. Advances in Computers 45, 53–103 (1997)

[15] Magklis, G., et al.: Dynamic Frequency and Voltage Scaling for a Multiple Clock Domain Microprocessor. IEEE Micro, 62–69 (November/December 2003)

[16] Semeraro, G., et al.: Dynamic frequency and voltage control for multiple clock domain microarchitecture. In: Proc. of International symposium on microarchitecture (MICRO-35), pp. 356–370 (2002)

[17] Jain, R., Hughes, C.J., Adve, S.V.: Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In: Proceedings of the 23rd IEEE International Real-Time Systems Symposium (December 2002)