Scalable Cache Memory Design for Large-Scale SMT Architectures

Muhamed F. Mudawar

Computer Science Department
The American University in Cairo

mudawwar@aucegypt.edu

Abstract. The cache hierarchy design in existing SMT and superscalar processors is optimized for latency, but not for bandwidth. The size of the L1 data cache did not scale over the past decade. Instead, larger unified L2 and L3 caches were introduced. This cache hierarchy has a high overhead due to the principle of containment. It also has a complex design to maintain cache coherence across all levels. Furthermore, this cache hierarchy is not suitable for future large-scale SMT processors, which will demand high bandwidth instruction and data caches with a large number of ports.

This paper suggests the elimination of the cache hierarchy and replacing it with one-level caches for instruction and data. Multiple instruction caches can be used in parallel to scale the instruction fetch bandwidth and the overall cache capacity. A one-level data cache can be split into a number of block-interleaved cache banks to serve multiple memory requests in parallel. An interconnect is used to connect the data cache ports to the different cache banks, thus increasing the data cache access time. This paper shows that large-scale SMTs can tolerate long data cache hit times. It also shows that small line buffers can enhance the performance and reduce the required number of ports to the banked data cache memory.

1 Introduction

Simultaneous multithreading (SMT) is a latency-tolerant processor architecture that enables multiple threads to simultaneously share the processor resources, effectively converting thread-level parallelism to instruction-level parallelism [9, 14, 15]. SMT improves the utilization of shared resources, such as register files, functional units, and caches, as it extracts ILP from multiple threads. SMT can also better tolerate pipeline and memory latencies, coping with the deeper pipelines, branch mispredictions, and the longer cache miss penalties. Some manufacturers have introduced their versions of SMT processors. Examples include the 2-context Intel Pentium 4 [3, 7] and the proposed 4-context Alpha 21464.

To implement higher-context and super-wide SMT processors, however, a number of challenges have to be addressed. These challenges include dynamic instruction scheduling, the shared register file, the shared cache hierarchy, and the degree of sharing or partitioning of hardware resources. This article addresses the problem of the shared cache hierarchy.

Current SMT processors use small L1 instruction and data caches. For example, the hyper-threaded Intel Pentium 4 uses a 16K L1 data cache. A thread that regularly sweeps through the L1 data cache will evict data needed by the other thread as shown in [13]. This negative interference will become more serious as the number of threads increases. The size of L1 data cache did not scale over the past decade. It was kept small to match the increasingly higher clock frequencies and to optimize the hit access time. Larger unified L2 and now L3 caches are introduced to increase the overall cache capacity and to optimize the memory access time. Figure 1 shows the cache hierarchy of a typical small scale 4-to-6 issue superscalar or SMT architecture. Two load/store ports are used for the D-Cache.

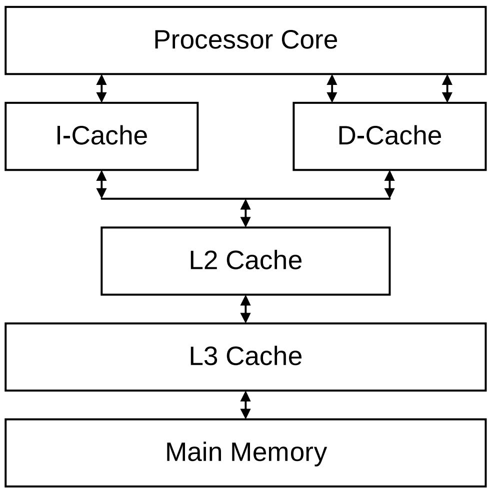

Fig. 1. Cache hierarchy for a typical wide-issue superscalar or a small-scale SMT architecture

Another more serious problem is the demand for higher cache bandwidth. Memory instructions account for about a third of all instructions executed on average. For example, an 8-context 32-issue processor should allow 12 load/store instructions to execute each cycle. This means that the L1 data cache should be designed to have 12 ports. The unified L2 and L3 caches should also support multiple ports to handle the multiple cache misses in parallel. In contrast, the current hyper-threaded Pentium 4 is a small scale SMT processor that supports a dual-ported L1 cache, a single-ported instruction trace cache, and single-ported unified L2 and L3 caches. This cache hierarchy, optimized for latency on a superscalar processor, does not fit into a large-scale SMT processor. It has to be redesigned and optimized for bandwidth rather than for latency. A simple scalable single-level banked cache design can optimize the bandwidth demand of large-scale SMT processors, while slightly increasing the latency of primary data cache access. This design will be described in Section 3. This design will be shown to be very effective for large-scale SMT processors, as increasing the latency of primary data cache access results in a minor degradation on the IPC.

The remainder of this paper is organized as follows. Section 2 discusses some background and related work. Section 3 introduces a scalable SMT architecture with a scalable cache design. Section 4 shows the simulation and performance of the architecture introduced in Section 3.

2 Background and Previous Work

Multiple cache ports can be implemented in one of four ways: ideal multi-porting, time division multiplexing, replication, and multiple independently addressable banks [4, 11, 17]. Ideal multi-porting requires that each cache block be simultaneously addressable by all the cache ports, allowing all the cache ports to operate independently in a given cycle. Ideal multi-porting is considered to be impractical, as the costs of ideal multi-porting will be enormous in terms of area, power consumption, and access time, as the number of ports increases. For example, the 24-ported register file (16-read and 8-write ports) in the proposed 4-context 8-issue Alpha 21464 SMT processor occupies over five times the area of the 64 KB primary cache according to [12]. A banked multi-ported register file is proposed in [12] to reduce the area, access time, and energy consumption. For this reason, ideal multi-porting is never applied to caches and will not be considered further.

Time division multiplexing is a technique that uses time to achieve virtual ports. It is used in the DEC Alpha 21264 [5]. The L1 data cache is referenced twice each cycle, once for each of the clock phases, effectively operating at twice the processor clock speed. Although simple enough, this technique is not scalable for a large number of ports, as it requires the cache to operate at significantly higher clock frequencies than the processor core. Current processors are already operating at significantly high clock frequencies and primary cache access time is already increasing from one to few clock cycles and will continue to increase in the future. Therefore, time division multiplexing is not a feasible solution.

A third possibility for multi-porting is through primary cache replication. Multiple copies will allow multiple loads to go in parallel. However, stores have to be broadcast and replicated to maintain identical copies. An example is the duplicate primary data cache used in the Alpha 21164 [2]. This solution improves the bandwidth of the load instructions. However, it will not improve the bandwidth of stores. Another overhead is the die area required for cache replication.

The fourth known technique to multi-porting is multi-banking. A cache is divided into multiple banks that can be accessed in parallel. Each cache bank is single-ported and can handle a single memory instruction per cycle. A fast interconnect, such as a crossbar, provides parallel access to the cache banks [8]. High bandwidth cache access can result, as long as parallel memory addresses map to different banks.

A simple and effective mapping scheme is to map contiguous memory blocks onto consecutive cache banks. This mapping scheme distributes uniformly the cache blocks. However, cache accesses to the same cache bank cannot proceed in parallel.

One problem of multi-banking is the probability of bank conflicts that arises from consecutive memory references that target the same cache line or the same cache bank. The same-line conflicts are shown to be high due to the inherent spatial locality in memory references, averaging 35% across integer benchmarks and 22% for floating-point benchmarks, according to [11]. These conflicts cannot be eliminated by simply increasing the number of cache banks. However, they can be exploited, using access combining, to improve multi-bank cache access. Access combining [1, 16] is a technique that attempts to combine memory accesses to the same cache line into a single request. Combining requires additional logic in the load/store queue to detect memory addresses targeted to the same cache line that can be combined. However, this additional logic is a small extension, because load/store queues in current architectures already implement a matching logic to detect and resolve memory dependencies.

Line buffering is another technique to avoid same-line conflicts. A line buffer holds cache data inside the processor load/store unit, allowing a same-line load to be satisfied from the line buffer, instead of from the cache [17]. A line buffer also reduces the utilization of the cache ports and the access latency of a multi-cycle multi-ported cache.

A second problem associated with multi-banked caches is the overhead of the interconnect. This unavoidable interconnect increases the cost and the delay of a multi-banked cache. A crossbar can be used for a small number of ports, but a multi-stage interconnect should be used for a larger number of ports. Depending on the interconnect, non-uniform cache bank access [6] may also result, where near cache banks are accessed faster than distant banks.

3 Scalable SMT Architecture

In this section, we propose a scalable SMT architecture that can scale to a large number of contexts. An 8-context SMT architecture is depicted in Figure 2. The most prominent feature of this architecture is the elimination of the cache hierarchy. We only preserve primary instruction and data caches and scale them according to requirement. The cache hierarchy is only an added overhead and a waste of space due to the principle of containment, as all the cache blocks in the L1 instruction and data caches are contained in L2 and L3. The cache hierarchy is also an added complexity. This complexity is required to maintain cache coherence across the different levels. Every store to the primary data cache has to be written through to reach the L2 and L3 caches. Every cache block invalidate in the L3 cache caused, for instance, by a different processor in a multiprocessor, has to be propagated upwards to reach the unified L2 and L1 data cache. Therefore, eliminating the cache hierarchy is desirable. Observe that what is being proposed here is against the current industry trend of increasing the cache hierarchy from 2 to 3 levels. The idea is to turn the second (or third) level cache into a primary data cache, effectively increasing the primary data cache capacity and bandwidth, as long as the processor is capable of tolerating the increased data cache hit time without much affecting the IPC.

3.1 Scalable Front End

To allow the front end to scale, multiple independent instruction caches must be used. Each instruction cache can be shared by a small number of threads (typically 2 to 4). The result is a simplified instruction cache design. Rather than using a single multi-ported instruction cache to fetch multiple instruction blocks from different threads per cycle, multiple single-ported instruction caches are used instead. One advantage is a simplified instruction cache design, which reduces the access time. A second advantage is the increased instruction cache capacity, which can scale with the number of threads, and which can eliminate negative thread interference and some of the capacity misses. For example, if four 128KB instruction caches are used in an 8-context processor then the overall instruction cache capacity is 512KB, eliminating the need for a second level cache. Each i-cache can be designed to have a large number of ways and to use way prediction to reduce cache energy and access time. A third advantage is the increased instruction cache bandwidth, which is also scalable with the number of threads. For example, four instruction cache blocks can be fetched per cycle in Figure 3, instead of a single one. This is essential to enable the IPC to scale. A fourth advantage is the absence of the interconnect in front of the instruction caches. An added interconnect will add more cycles to instruction fetching, which will also increase the branch misprediction penalty. The absence of the interconnect, however, implies that instruction blocks might be replicated in different instruction caches, especially when different threads execute the same instruction stream on different data streams. A simple snooping protocol can detect and forward replicated cache blocks from one instruction cache to another to avoid long memory access.

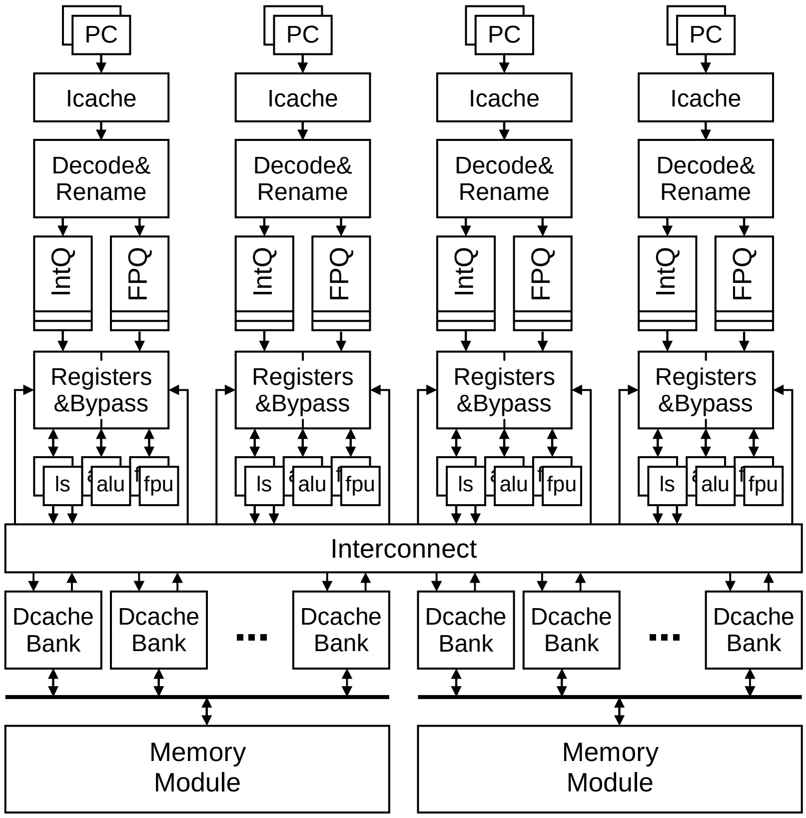

Fig. 2. An 8-Context SMT Processor

3.2 Partitioned Hardware Resources

In addition to instruction caches, many hardware resources are partitioned and replicated as shown in Figure 2. This includes the rename tables, the scheduling queues, the register files, and the functional units. Limited sharing allows few threads (typically two) to share some hardware resources, but hardware partitioning is essential to reduce complexity and to enable scalability.

3.3 Scalable and Shareable Data Cache

For data caches, we split them into multiple banks shareable by all threads. The cache banks are block-interleaved to obtain a uniform distribution. The use of multiple cache banks increases data cache capacity, which eliminates the need for a second or third level cache. For example, a 2MB data cache can be obtained by splitting it into sixteen 128KB banks. Each cache bank can be designed to have a large number of ways and to use way prediction or selective direct mapping [10]. This will increase the capacity of the cache banks and will reduce their access time and energy consumption. Each cache bank is designed to be single ported, which simplifies its implementation. A third advantage is that no cache block replication can occur among the different banks, since cache block interleaving will map a cache block to a unique bank. This eliminates the need to maintain cache coherence among the different banks. A fourth advantage is that the cache banks can use multiple busses to multiple memory modules. In other words, the memory modules will also be cache block interleaved. This will increase main memory bandwidth and will decrease the bus conflicts due to the increased number of cache misses generated by the increased number of cache banks.

An unavoidable price is the overhead of the interconnect, which increases in complexity with the number of ports and the number of cache banks. This interconnect can be a crossbar, a multi-stage network with uniform data cache bank access, or a distributed non-uniform data cache access network. Whatever it might be, the interconnect increases the access delay to the data cache from one to several clock cycles. However, our simulation results indicate that increasing the access delay to the data cache can be tolerated in a large scale SMT processor. In other words, we can trade the increase in threads and the cumulative ILP with the increase in data cache access time. Therefore, this data cache organization is scalable in terms of capacity, bandwidth, and access delay.

3.4 Pipeline Stages for a Load Instruction

The pipeline stages for a typical load instruction are shown in Figure 3. At least 11 pipeline stages are required, starting with instruction fetch, going through decode, rename, and queue, and ending with register write and retirement. The data cache access delay should be at least three cycles, after computing the effective memory address. One or more cycles are used to forward the address from the input ports to the corresponding data cache banks through the interconnection network. One or more cycles for cache bank access, and one or more cycles to forward the data to the corresponding physical destination register.

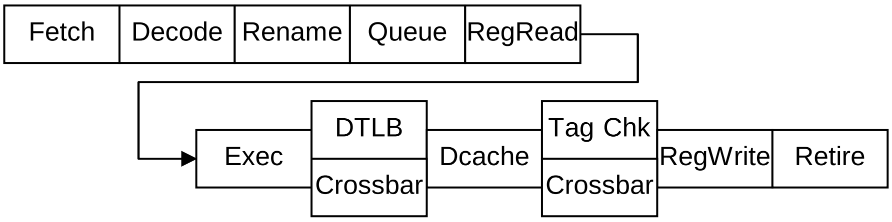

Fig. 3. Pipeline Stages for a Typical Load Instruction

3.5 Data Translation Lookaside Buffers

The data translation lookaside buffers (DTLBs) are searched in parallel while establishing paths through the interconnection network to the corresponding data cache banks. Observe that the cache bank address is NOT part of the virtual page number as shown in Figure 4, and hence virtual address translation and interconnection path establishment can proceed in parallel. The DTLBs are integrated as part of Load/Store units, such that each DTLB is associated with one or at most few threads. This is much better than integrating the DTLBs with the data cache banks, as each bank is shared by all the threads.

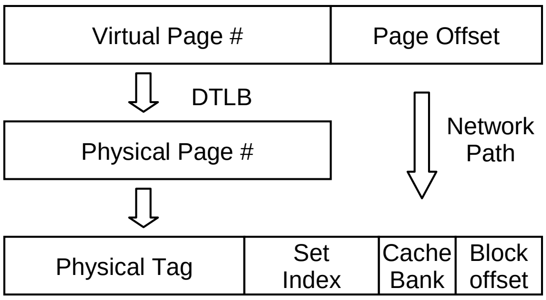

Fig. 4. Network path establishment can be done in parallel with DTLB virtual address translation

Data cache banks are physically tagged, since tag checking is done is a separate cycle. Therefore, the physical tag is not needed while indexing the data cache and can be sent in the next cycle. The set index, on the other hand, should be virtual, to allow it to reach the data cache without waiting for the DTLB. However, virtual indexing can cause aliasing problems. Alternatively, the data cache can be physically indexed but this will add an extra delay cycle, because it will have to wait for the result of the DTLB.

3.6 Cache Line Buffering

A small but important optimization is to have few data cache blocks buffered also in the load/store units. This level zero cache can mainly reduce the conflicts to the same cache block, caused by consecutive load instructions, due to spatial locality of reference. It also improves the data cache access latency. Only few cache lines need be buffered, and a small fully-associative buffer can be established in every load-store unit and for each thread. A hit on a load in a line buffer is served from the buffer, freeing the data cache access port to another memory instruction. A hit on a store updates the line buffer as well as the corresponding data cache bank. A miss on a load is served from a data cache bank, which forwards a copy of the cache block to the requesting line buffer. The address of the allocated line buffer is also recorded in the data cache bank to establish cache coherence. A miss on a store only writes the corresponding data cache bank, but no block transfer takes place.

A cache coherence problem is introduced because of the replication of some cache lines in the line buffers. A simple *one-copy* cache coherence protocol can be defined as follows. Only one copy of a cache block can be forwarded to a line buffer at a time. Therefore, if a request to forward some cache block, already buffered in line buffer *A*, to some other line buffer *B*, then line buffer *A* must be invalidated, as the data cache block is forwarded to line buffer *B*. If a cache block in a data cache bank is invalidated then its corresponding line buffer must also be invalidated.

4 Simulation and Performance

The simulation program was built on top of the Simplescalar simulator using the PISA instruction set. We simulated an 8-context 32-issue SMT processor with four 64KB instruction caches, each shared by 2 threads, and a 12-ported 16-banked data cache shared by all threads. A total of 36 functional units were used: 24 integer ALUs (half of them shared by load-store instructions), 8 FPUs (used for FP add and convert instructions), and 4 units used for all integer and floating-point multiplications and divisions. The scheduling and load-store queues were partitioned. Each thread had a 128-entry scheduling queue and a 64-entry load-store queue. The front end can fetch four instruction blocks (up to 64 instructions) per cycle from four different threads. The simulation parameters are summarized in the following table:

I-Cache	4 independent i-caches are used Each is 128KB, 8-way associative, 64-byte lines, 1 cycle access time

D-Cache	12 ports, 16 banks Each is 128 KB, 8-way associative, 64-byte lines, total capacity: 2MB Access time: 3, 5, 7, 9, and 11 cycles
Line Buffers	None and 8 lines per thread
L2 Unified	None
Memory	100 cycles latency
Issue width	32 instructions per cycle
ALUs	24 units, where 12 are used also to compute effective address of load-store instructions
FPUs	8 fully pipelined, 4 cycle latency for FPadd
Mul, Div, …	4 units 4 cycle latency, pipelined, for int/fp multiply 20 cycles non-pipelined for integer divide 12 cycles non-pipelined for FP divide 24 cycles non-pipelined for FP sqrt
Scheduling Q	128 entries per thread
Load-Store	64 entries per thread

4.1 SPEC 2000 Benchmarks

We chose a subset of eight programs to run as independent threads. These benchmarks were compiled with optimization for the PISA instruction set. The first three belong to the SPECfp2000 benchmarks. The last five belong to SPECint2000. The eight benchmarks were run in parallel until one of them reached the 100 million instruction limit. The total number of instructions executed across all benchmarks exceeded 700 million instructions for a typical simulation run.

- *188.ammp*: Computational Chemistry.
- *183.equake*: Seismic Wave Propagation Simulation.
- *177.mesa*: 3D Graphics Library.
- *176.gcc*: GNU C compiler generating optimized code.
- *197.parser*: Word Processing.
- *255.vortex*: Object-Oriented Database.
- *175.vpr*: FPGA Circuit Placement and Routing.
- *181.mcf*: Combinatorial Optimization.

4.2 Simulation Results of a Single-Level Banked Data Cache

The performance of an 8-context 32-issue SMT under different data cache latencies is shown in Figure 5. The first column shows the performance under an ideal main memory with a 1-cycle latency. The second column shows the performance of a 2MB ideal data cache with 1-cycle latency, which is impossible to realize, especially as the feature size goes below 100 nm and the wire delay becomes a dominant factor. It was shown here just for comparison. The main memory latency was assumed to be 100 cycles. The remaining columns assume a data cache latency of 3, 5, 7, and 9 cycles, which include the latency of the interconnect. No line buffers exist. The overall IPC goes down from 23.07 (ideal memory case) to 20.46 (ideal cache 1-cycle latency and 100-cycle main memory), to 20.19 (3 cycles), 19.71 (5 cycles), 19.10 (7 cycles), and 18.39 (9 cycles).

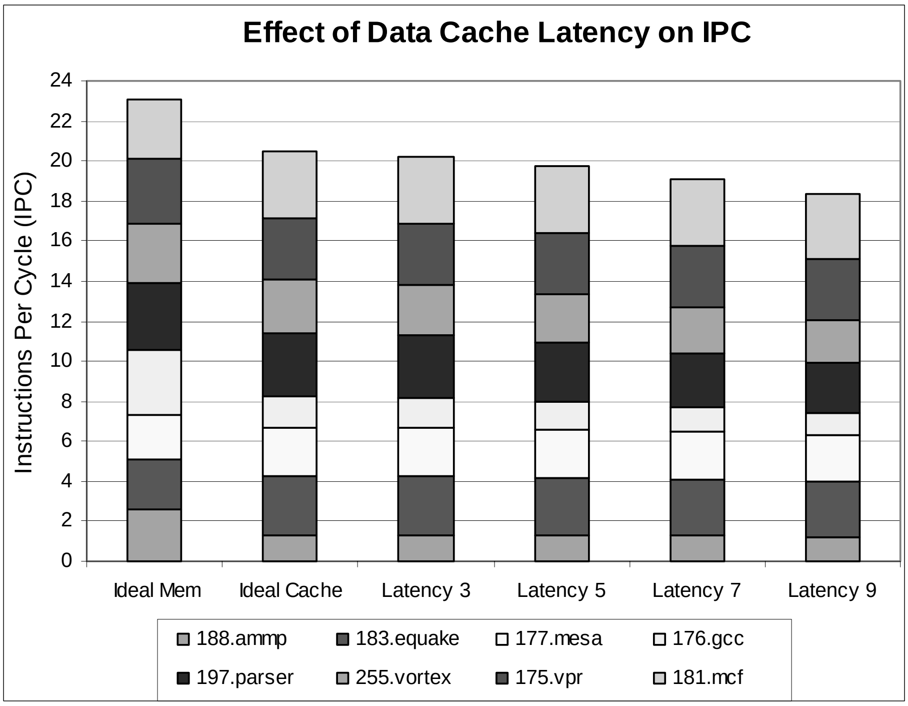

Fig. 5. Performance of an 8-context 32-issue SMT under different cache latencies

4.3 Adding an 8-Line Buffer to Each Thread

A second experiment was conducted to assess the performance of an 8-context 32-issue SMT under the presence of line buffer. A small buffer or level zero cache, consisting of 8 cache lines or blocks was added to each thread. The line buffer was assumed to have a latency of 1 cycle. The data cache latency was assumed to be 7, 9, and 11 cycles. The results are shown in Figure 6. The overall IPC under the presence of an 8-line buffer is 20.04 for a 7-cycle latency data cache, 19.86 for 9-cycle latency, and 19.63 for 11-cycle latency.

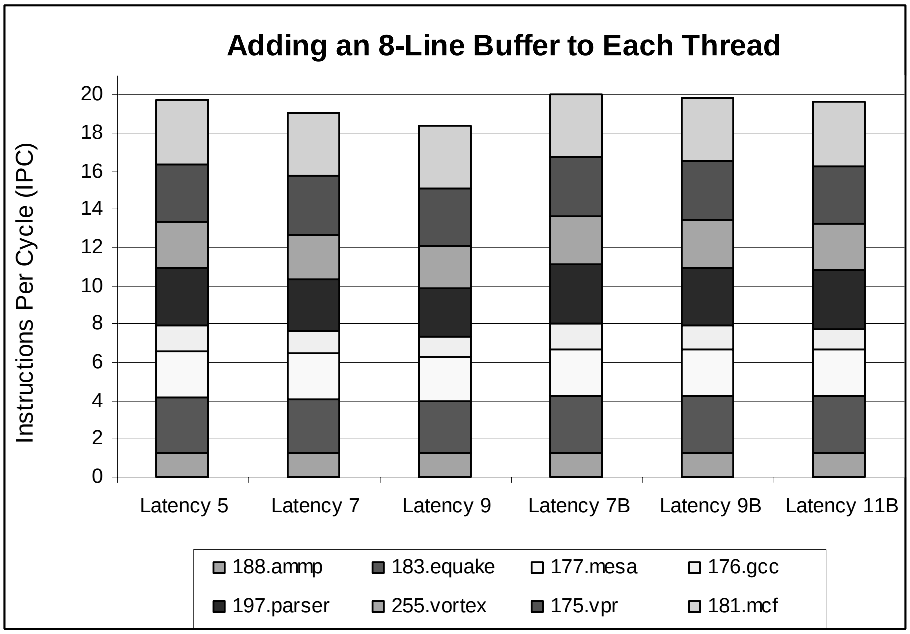

Fig. 6. Performance comparison under the presence of 8-line buffer for each thread

4.4 Discussion

Increasing the data cache latency from the ideal situation of 1 cycle to 3 cycles reduced the IPC by 1.33%. Increasing it from 1 to 5 cycles reduced the IPC by 3.64%, from 1 to 7 cycles reduced the IPC by 6.63%, and from 1 to 9 cycles reduced the IPC by 10.1%. This was the case in the absence of line buffers. It showed that a large-scale SMT processor has the ability to tolerate cache latency due to the presence of ample parallelism among the different threads. While the IPC of few threads were negatively affected by the increased latency, such as *ammp* and *gcc*, others were able to tolerate and sometimes took advantage of the wasted cycles, which slightly improved their IPC.

Adding a small line buffer showed better latency tolerance and improved results. For example, a 7-cycle data cache latency had only 2.03% degradation in IPC from the ideal cache performance under the presence of a line buffer. Similarly, a 9-cycle latency data cache had 2.92% degradation in IPC performance and an 11-cycle latency data cache had 4.04% degradation for the simulated benchmarks.

5 Conclusion

This paper has demonstrated that large primary data caches, with a large number of ports and banks, are well suited for large-scale SMT processors, which can tolerate longer hit times in the data cache. Increasing the number of data cache banks increases the overall capacity, which in turn eliminates the need for unified L2 and L3 caches. Increasing the number of ports increases the bandwidth, but also increases the latency of data cache access and the cost of the interconnect. Having small line buffers (called also level zero caches) with few entries (typically 8 or 16 per thread) does improve the overall IPC and can better tolerate the longer latency of the data cache. The small line buffers have also an added benefit in reducing the number of data cache ports and hence the complexity and latency of the interconnect.

References

1. T. Austin and G. Sohi, "High-Bandwidth Address Translation for Multiple-Issue Processors", *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996, pages 147-157.
2. J. Edmondson et al, "Internal Organization of the Alpha 21164 a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor", *Digital Technical Journal*, Special 10th Anniversary Issue, Vol. 7, No. 1, 1995, pages 119-135.
3. "Hyper-Threading Technology", *Intel Technical Journal*, vol.6, no.1, February 2002.
4. T. Juan, J. Navarro, and O. Temam, "Data Caches for Superscalar Processors", *Proceedings of the 11th International Conference on Supercomputing*, July 1997, pages 60-67.
5. R. Kessler, "The Alpha 21264 Microprocessor", *IEEE Micro*, March-April 1999, pages 24-36.
6. C. Kim, D. Burger, and S. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches", *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002, pages 211-222.
7. D. Koufaty and D. Marr, "Hyperthreading Technology in the Netburst Microarchitecture", *IEEE Micro*, March-April 2003, pages 56-65.

8. L. Li, N. Vijaykrishnan, M. Kandemir, M. J. Irwin and I. Kadayif, “CCC: Crossbar Connected Caches for Reducing Energy Consumption of On-Chip Multiprocessors”, *Proceedings of the Euromicro Symposium on Digital System Design*, September 2003, pages 41-48.
9. J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen, “Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading”, *ACM Transactions on Computer Systems*, vol.15, no.3, August 1997, pages 322-354.
10. M. D. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, and K. Roy, “Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping”, *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001, Austin, Texas, pages 54-65.
11. J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin “On High-Bandwidth Data Cache Design for Multi-Issue Processors”, *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pages 46-56.
12. J. Tseng and K. Asanovic, “Banked Multiported Register Files for High-Frequency Superscalar Microprocessors”, *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003, pages 62-71.
13. D. Tullsen and J. Brown, “Handling Long-Latency Loads in a Simultaneous Multithreading Processor”, *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001, pages 318-327.
14. D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor”, *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996, pages 191-202.
15. D. Tullsen, S. Eggers, H. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism”, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pages 392-403.
16. K. Wilson, K. Olukotun, and M. Rosenblum, “Increasing Cache Port Efficiency for Dynamic Superscalar Processors”, *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996, pages 147-157.
17. K. Wilson and K. Olukotun, “Designing High Bandwidth On-Chip Caches”, *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pages 121-132.