A Simple Latency Tolerant Processor

Satyanarayana Nekkalapu, Haitham Akkary[1], Komal Jothi, Renjith Retnamma, Xiaoyu Song
[1]*Electrical and Computer Engineering* *Electrical and Computer Engineering*
American University of Beirut *Portland State University*
[1]*hakkary@acm.org, {cha_nu, komalj, renjith, song}@ece.pdx.edu*

Abstract—**The advent of multi-core processors and the emergence of new parallel applications that take advantage of such processors pose difficult challenges to designers. With relatively constant die sizes, limited on chip cache, and scarce pin bandwidth, more cores on chip reduces the amount of available cache and bus bandwidth per core, therefore exacerbating the memory wall problem [24]. How can a designer build a processor that provides a core with good single-thread performance in the presence of long latency cache misses, while enabling as many of these cores to be placed on the same die for high throughput.**

Conventional latency tolerant architectures that use out-of-order superscalar execution have become too complex and power hungry for the multi-core era. Instead, we present a simple, non-blocking architecture that achieves memory latency tolerance without requiring complex out-of-order execution hardware or large, cycle-critical and power hungry structures, such as dynamic schedulers, fully associative load and store queues, and reorder buffers. The non-blocking property of this architecture provides tolerance to hundreds of cycles of cache miss latency on a simple in-order issue core, thus allowing many more such cores to be integrated on the same die than is possible with conventional out-of-order superscalar architecture.

I. Introduction

Increased integration on a single chip has led to the current generation of multi-core processors having a few cores per chip. Future improvement in process technology will eventually allow packing many more cores on the same die than is possible today [12]. This creates a difficult dilemma for processor designers: how to balance single-thread performance of a single core with system throughput critical for parallel applications that will emerge to exploit the many-core processors of the future. Due to the growing gap between processor cycle time and memory access latency on one hand, and between execution throughput of many integrated cores and memory access bandwidth of a single chip on the other hand, processor core pipelines will increasingly stall waiting for data in the event of cache misses to memory. Limited on chip cache area, reduced cache capacity per core, and the increase in application cache foot prints as applications scale up with the number of cores, will only make cache miss stalls more problematic.

Continual Flow Pipelines (CFP) [22] has been proposed as a processor core architecture that can sustain a very large number of in-flight instructions without requiring cycle-critical hardware structures of an out-of-order superscalar to scale up. By allowing a processor to continue processing instructions even in the presence of long latency cache misses to memory, CFP achieves memory latency tolerance of very large instruction window processors without actually building very large instruction buffers and register files.

CFP treats miss independent and miss dependent instructions differently. Independent instructions are executed and retired quickly while miss-dependent instructions are moved out of the pipeline into a Slice Data Buffer (SDB) where they wait until the miss data request to memory is processed. When the miss data is loaded into the on-chip cache, the miss-dependent program slice re-issues again for execution from the slice data buffer. The miss-dependent slice carries its ready input data with it in the SDB and forms a complete self contained program slice. The results of the dependent slice and the previously executed miss-independent instructions are automatically integrated, when the miss-dependent slice re-issues, via incremental updates of the register rename map table with the slice live-out register mappings. CFP rolls back execution to register checkpoints taken at low-confidence branches [1] to recover from slice exceptions and mispredicted branches.

Although the CFP architecture in [22] provides memory latency tolerance without increasing the circuit complexity and the size of cycle-critical hardware structures, it achieves its goals at the expense of significant additional hardware. The architecture requires new logic for processing and re-issuing the miss dependent slice. In addition to the Slice Data Buffer, CFP requires new "Slice Rename Filter" and "Slice Remapping" units [22]. This additional logic on top of the significant complexity of the out-of-order execution pipeline consumes large silicon area. The cost may be acceptable for single-thread performance, but comes at the expense of the number of cores and the total multi-core execution throughput.

We propose in this paper a non-blocking in-order Continual Flow pipeline that provides latency tolerance to hundreds of cycles of memory access time, but avoids the logic complexity and the hardware cost of the original CFP proposal. We believe that our simple core design is more suitable for future multi-core chips with 10s or 100s of integrated cores targeting very high throughput applications.

A. *Paper Contributions*

This paper makes the following contributions:

- It proposes and evaluates simple in-order continual flow pipeline architecture. The non-blocking property of in-order CFP provides memory latency tolerance while keeping the core complexity low, thus providing an attractive low power, small core architecture for future many-core processors.

- The simplicity of the in-order pipeline reduces the penalty of CFP miss-dependent slice re-issue. This property widens the applicability of the latency tolerant CFP technique to medium latency events such as lower level cache misses that hit in the on-chip higher level caches.
- The paper describes a fast result integration algorithm performed using masked flash copy performed locally within the register file. This algorithm further reduces the penalty of CFP dependent slice execution.

Section II describes our in-order CFP architecture. Section III outlines the simulation methodology and machine configuration. Section IV presents a performance analysis of the architecture. Section V discusses related work and we conclude the paper in section VI.

II. In-Order Continual Flow Architecture

The processor core handles instructions dependent on a miss differently from miss independent instructions. On a load cache miss, a first register checkpoint is taken. The load and its dependent instructions drain out of the pipeline and are stored in a Slice Data Buffer, freeing the pipeline for miss independent instructions to execute. The miss dependent slice instructions are stored with their input register values, in program order, in the SDB outside the pipeline. Because these slice instructions do not tie the pipeline staging latches, the processor achieves a continual flow property and can look far ahead for useful miss-independent instructions to execute while the data miss is outstanding.

When the miss data returns, a second register checkpoint is taken and execution switches to the slice buffer instructions. When all the slice buffer instructions execute, their results are merged with the independent instruction results from the second checkpoint. The two checkpoints are discarded and execution resumes normally without having to go back to execute independent instructions again. Fig. 1 shows a block diagram of our in-order CFP microarchitecture.

We now describe various details of this microarchitecture.

A. *Independent Instruction Execution and Dependent Slice Construction*

We call execution in this phase CFP execution. This phase starts when a cache miss occurs and the first register checkpoint is taken. Miss-dependent instructions in this execution phase are identified by propagating poison bits from producer to consumer instructions. We extend each register with one poison bit. On a load cache miss, a pseudo writeback occurs that sets the destination register poison bit and marks the destination register as ready. Instructions that read poisoned registers are miss-dependent instructions. When they issue, they read their source registers including the poison bits. At writeback, they set the poison and the ready bits of their destination register, and write their opcodes and source registers in the SDB. Instructions without poisoned sources execute normally and clear their destination register poison bits at writeback. They do not go into the SDB.

If multiple cache misses occur during slice construction, the destination register of a new miss is poisoned as well, and the instructions from all slices are written into the SDB in program order and not data flow order. Instructions from different slices therefore are often stored in an interleaved manner.

CFP execution phase ends when the data for the first miss in the SDB returns.

B. *Dependent Slice Execution*

This phase starts when the data for the first miss in the SDB returns and the second checkpoint taken. We call execution in this phase SDB execution. When this phase starts, fetch and execution switch to the slice data buffer. Instructions in the SDB may have input operands stored with them from the time these values were produced during CFP execution phase. The slice buffer instructions also have input operands that were poisoned. The poisoned input operands are outputs of other instructions in the slice data buffer and are computed during the slice execution and propagated normally through the register file or its bypass network.

When SDB execution encounters a cache miss, the execution stalls until the miss is processed. Although the CFP proposal in [22] allows switching back to CFP execution in the event of a miss during SDB execution, we choose in our implementation the simplicity of stalling SDB execution until the miss data returns instead of switching back to CFP execution. By ensuring that once the SDB execution starts it is completed for all the instructions in the buffer, we simplify register file checkpoints and result integration of CFP and SDB execution with little performance loss.

To understand the performance impact of our decision to simplify handling SDB misses, consider the case when there are multiple misses in the SDB. These misses are usually independent. When the first miss returns and SDB execution starts, a subsequent miss would have been issued to memory during CFP execution and would either be in processing, or have already been completed (since latency to DRAM varies depending on whether an address hits or misses an open DRAM page). CFP helps performance in this case by issuing the independent requests to memory concurrently, therefore increasing memory level parallelism.

Another case is when the address of a second miss is dependent on a previous miss. In this case, the second miss

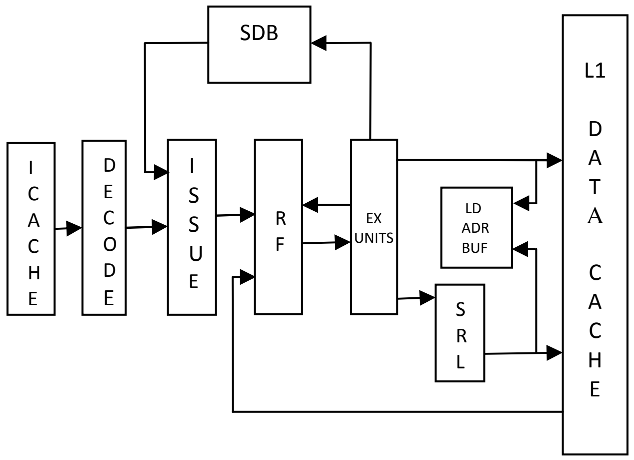

Fig. 1. Block Diagram of In-Order CFP Architecture

will be poisoned and the request to DRAM cannot be issued until SDB execution. This serialization of dependent cache misses does not hurt overall performance in a significant way since this case is less common.

The SDB execution phase ends when all instructions in the SDB are fetched and issued for execution.

C. *Checkpoints*

We use a flash copy of the register file for checkpointing. In one cycle all register values are shifted into a backup latch within the register cell. Fig. 2 shows a register file cell with two checkpoints and flash copy support. When CHKPT_CLK is asserted, a backup copy of the bit is shifted into a local edge-triggered latch. The register file bit value can be restored from a checkpoint by asserting RSTR_CLK.

D. *Independent and Dependent Slice Result Integration*

Result integration is a special checkpoint copy-and-restore sequence. At the end of CFP execution phase, a second checkpoint of the register file is taken including the poison bits. The register file is then used for SDB execution. In the absence of miss dependent slice exceptions and branch mispredictions, a restore cycle is performed at the end of SDB execution from the second checkpoint. However, not all registers are restored. As shown in Fig. 2, only the non-poisoned registers are restored by using the poison bits to gate the clock of the restore operation.

E. *Handling Slice Exceptions and Mispredicted Branches*

Exceptions and branch mispredictions encountered during CFP execution are handled normally. On the other hand, miss-dependent exceptions and mispredicted branches are detected when miss-dependent instructions execute from the SDB. For recovery, execution is rolled back to the first checkpoint taken at the load cache miss instruction. This rollback and re-execution from the checkpoint does not increase power or reduce CFP performance significantly since exceptions are rare and the majority of mispredicted

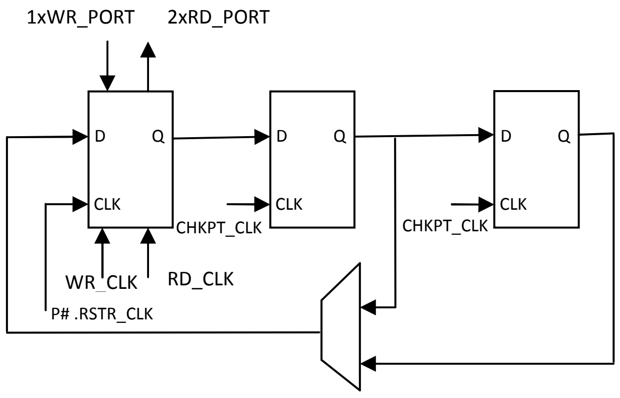

Fig. 2. Register File Cell with Checkpoint Store

branches in the shadow of a cache miss are miss-independent (see [22] and Table 3) and are therefore handled during CFP execution normally without rollback to the checkpoint.

F. *Memory Ordering*

To maintain proper memory ordering of loads and stores from the CFP and the SDB execution phases, we use a Store Redo Log (SRL) technique [11]. During CFP execution, stores are written speculatively into the first level cache and CFP loads can read data from these stores by accessing the cache. The stores are also written in the SRL buffer (see Fig. 1). When execution switches to SDB, the speculative stores in the cache are cleared, and then redone (and committed) in program order with the SDB stores. We use memory dependence prediction [16] to predict loads that depend on SDB stores and we poison these loads during CFP execution. In case the memory dependence predictor misses a load to store dependency, we roll back execution to the first checkpoint taken at the load miss. For this we use, as in [11], a set associative load address buffer to snoop committed internal processor stores. We also use this buffer to snoop stores from other threads and processors for memory consistency.

Since our pipeline is in-order, we maintain correct memory ordering without the complexity of a 1st level fully associative store queue or the need for forwarding from the SRL buffer as required in original CFP. This reduces the memory ordering and the SRL hardware significantly and adds another advantage to our in-order CFP implementation.

III. Simulation Methodology

To evaluate our in-order CFP microarchitecture proposal, we use PTLsim simulation infrastructure [25]. PTLSIM simulates x86 code after converting complex instructions into RISC-like micro-ops (uops), a technique used in Intel processors [18]. Due to the lack of representative many-core benchmarks and because we are mainly interested in this work in evaluating the latency tolerance and performance of in-order CFP in the presence of frequent cache misses, we use a representative subset of Spec2000 benchmarks and a

Table 1. Baseline Machine Configuration

Pipeline	8 stage, 1-wide, in-order
L1 Data cache	16KB , 4-way, 2 cycles, 64-byte line
L1 Ins. cache	16 KB, 4-way, 2 cycles, 64-byte line
L2 cache	64 KB,4-way, 12 cycles, 64-byte line
L3 cache	256KB, 8-way, 50 cycles, 64-byte line
L3 to memory latency	350 cycles
Branch predictor	Combined bimodal and gshare 64K each Meta, bimodal, gshare 4K BTB

hypothetical machine configuration we project from next generation Intel Multicore architecture (Nehalem) [10].

Since our simulator models a single core processor and assuming the on-chip cache is shared equally among cores in our hypothetical multicore processor, we use the Nehalem configuration to estimate the cache size that would be available for each core if all the Nehalem out-of-order cores were replaced by in-order CFP cores. We conservatively assume that it is possible to replace the 4-wide out-of-order Nehalem core with four single-issue in-order CFP cores and derive the cache sizes shown in the baseline machine configuration in Table 1. All simulations were done using this machine configuration for 80 million instructions from each benchmark after skipping the initialization phase.

The benefits of in-order CFP comes from the application dataflow characteristics and the pipeline organization. Although we have used x86 code in our study, we expect similar results from in-order CFP on the same benchmarks if run on a processor that uses different ISA, such as RISC.

IV. RESULTS AND ANALYSIS

Section A discusses in-order CFP performance when applied to L2 and L3 misses. Section B analyzes performance results. Section C discusses various execution statistics.

A. *In-Order CFP Performance*

Fig. 3 shows percent speedup of in-order CFP over the baseline machine configuration when applied to L2 and L3 cache misses. Speedup over baseline varies from 3% on Gzip to 62% on Mgrid when CFP execution is applied to L3 cache misses. CFP execution when applied to L2 cache misses as well improves performance further with speedups varying from 8% on Crafty to 85% on Mgrid. Performance difference between benchmarks is a result of variations in cache miss rates. The benchmarks in the graph (and other Spec2000 benchmarks not shown) display consistent performance gains with CFP benefiting benchmarks with high cache miss rates more.

Notice the effectiveness of in-order CFP architecture in handling, not only off chip latencies on L3 misses, but also on chip latencies for loads that miss the L2 cache but hit the on chip L3 cache, as indicated by the significant improvement in performance when CFP execution is applied to L2 cache misses. This result is particularly interesting to future many-core processor designers as on chip latencies increase with further chip integration.

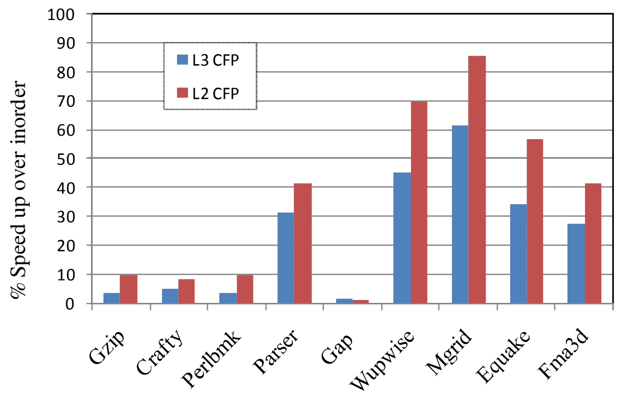

Fig. 3. L2 and L3 In-Order CFP Performance

Finally, we point out that even with the performance benefits of in-order CFP, conventional out-of-order superscalar execution performs better than our simple in-order CFP architecture. However, such performance comes at a significant cost in power and area. The objective of our study is to present a low power, latency-tolerant core architecture as another option for future many-core processors running highly parallel applications. Our objective is not to propose a higher performance substitute for conventional out-of-order superscalar cores which are still, and may always be, the best high-performance architecture for conventional hard to parallelize single-thread applications.

B. *Performance Analysis*

Fig. 4 shows the instruction-window size distribution for in-order CFP when applied to L3 cache misses. In-order CFP is able to achieve look-ahead execution of hundreds of micro-ops for a significant fraction of the execution time. For example on Mgrid, the look-ahead CFP execution distance is more than 128 micro-ops for 30% of the execution time. This is very good look-ahead execution distance for a simple in-order pipeline.

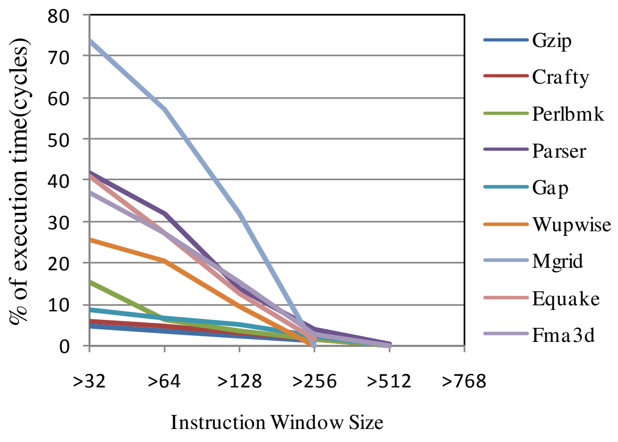

Fig. 4. Look-Ahead Execution Distance

Table 2. Percent of Look-Ahead Execution Independent of L3 Miss.

Gzip	Crafty	Perlbmk	Parser	Gap	Wupwise	Mgrid	Equake	Fma3d
54%	76%	63%	53%	96%	51%	77%	58%	70%

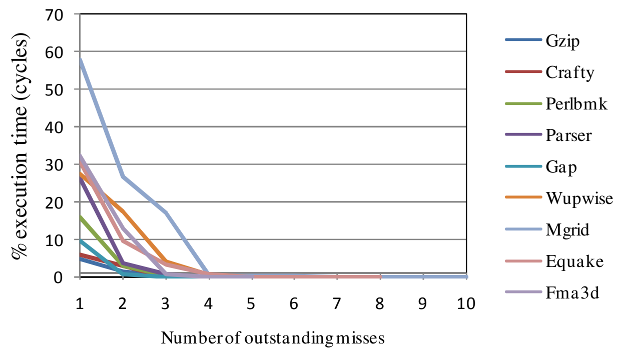

Fig. 5. Memory Level Parallelism with In-Order CFP.

There are two factors that contribute to the in-order CFP performance gains 1) its tolerance to long memory latencies and 2) its ability to overlap the processing of multiple cache misses thus exposing memory level parallelism. The architecture ability to tolerate long memory latencies can be measured by the fraction of look-ahead execution after a miss that is independent of the miss data. Table 2 shows a significant percentage of the look-ahead execution to be independent of the miss data, thus keeping the processor busy while a miss is processed. This is consistent with observations made by others in [13].

Fig. 5 shows the outstanding L3 cache miss distribution. As can be seen from the graphs, in-order CFP increases the memory level parallelism significantly over the baseline machine which has memory level parallelism of one. Since in the baseline in-order machine the pipeline stalls on a miss, at most one miss can be processed at a time. Compare this with Mgrid running on in-order CFP. From Fig. 5, during 18% of the total execution time of Mgrid, there are 3 outstanding miss requests to memory processed simultaneously.

C. *Various Execution Statistics*

We present various execution statistics in Table 3. Most interesting are the cache miss rates in columns 6 and 7, which correlate very well to the observed performance. Also interesting is column 2 which reports the average occupancy of the SDB when in use. The SDB can be implemented as a single ported FIFO structure using SRAM and should not significantly increase the in-order CFP core size.

Column 3 has the percent of total micro-ops that enter the SDB. These micro-ops flow through the pipeline twice (CFP and SDB execution phases) but represent a small fraction of the total micro-ops, thus limiting the power wasted by SDB execution.

Table 3. In-Order CFP Execution Statistics

Bench mark	Avg. SDB occupancy	SDB to total inst %	Branch misp per 1000 uops	SDB Branch misp per 1000 uops	Number of cache misses per 1000 uops	
					L2	L3
Gzip	60	0.90	6.15	2.51	3.7	0.41
Crafty	36	0.71	4.31	1.54	1.6	0.37
Perlbmk	39	1.87	2.22	0.49	1.0	0.70
Parser	50	7.49	7.79	4.10	6.6	3.90
Gap	10	0.32	5.71	0.13	0.5	0.07
Wupwise	60	3.60	0.49	0.19	4.4	4.30
Mgrid	22	9.61	0.93	0.56	5.9	5.84
Equake	34	6.21	3.4	0.53	3.4	2.31
Fma3d	66	8.61	0.28	0.24	3.0	2.82

Columns 4 and 5 show the branch misprediction rates. It is good for in-order CFP execution that the SDB branch misprediction rates are small compared to overall branch misprediction rates. This is because SDB branch mispredictions are costly as they are detected late after a cache miss is serviced and require rolling back execution to the load miss checkpoint.

V. Related Work

Proposals for latency tolerant microarchitectures include the Waiting Instruction Buffer [14], Virtual ROBs [8], Cherry [15], Checkpoint Processing and Recovery [1], Kilo Instruction Window Processors [7], Continual Flow Pipelines [22] and Out of Order Commit Processors [6]. All these proposals are based on out of order core microarchitecture and are more suitable for maximizing performance of conventional single-thread applications running on multicore processors with few cores. In-order CFP is more appropriate with its simple small core design for future many core architectures running very high throughput parallel applications.

Runahead execution has been proposed as a method to increase memory level parallelism on in-order processors [9] without having to build complex out-of-order execution pipeline, and on out-of-order cores [17] without having to build large reorder buffers. In runahead execution, the processor state is checkpointed at a long latency miss operation. Execution continues speculatively past the miss and prefetches data. When the miss data returns, runahead execution terminates, the execution pipeline is flushed, the checkpoint is restored, and execution restarts from the load miss instruction. Except for the prefetching effect of runahead, all work performed during runahead is discarded. In-order CFP does not discard execution. Instead, it seamlessly integrates CFP and SDB execution. Therefore, in-order CFP provides latency tolerance to memory for better energy-efficient performance and is more suitable for future power limited many-core processors.

Thread-based pre-execution methods have been proposed where either additional code [4][20] or a small subset of the program (*e.g.*, a backward slice of a cache miss) [19][26] is pre-executed on idle threads of a processor prior to encountering the miss. The idea is to proactively execute a slice leading to the miss to prefetch the miss data ahead of time. Unlike thread-based pre-execution, in-order CFP execution continues on the same blocked thread and does not require or waste another thread.

Flea-Flicker [2][3] microarchitecture has been proposed where-in a program executes on two in-order back-end pipelines coupled by a queue. An advance pipeline executes independent instructions without stalling on long latency cache misses while deferring dependent instructions. A backup pipeline executes instructions deferred in the advance pipeline and merge with results from the advance pipeline stored in a queue. A similar microarchitecture was also proposed for a minimal speculative multithreading architecture in [21]. A key difference between flea-flicker and in-order CFP is the result integration method and the coupling queue. Flea-flicker stores all instructions and results from the advance pipeline in the queue and merge results sequentially during backup pipeline execution. In contrast, in-order CFP stores only the dependent instructions in a smaller SDB queue with their inputs and automatically merges results using a fast one-cycle operation.

Sun Microsystems implemented for high throughput computing a chip multithreading processor with scout threads [5]. On a cache miss, a checkpoint is taken and a scout thread performs run-ahead execution. A very recent paper from Sun [23] adds that the architecture defers dependent instructions into a buffer and executes the deferred instructions from the checkpoint after the miss data returns, merging the results into the scout thread future file. This scout thread architecture seems to have many similarities with in-order CFP, but [23] presents very little detail of the multithreading and checkpoint hardware, the register file or the merge into the future file method for us to fully compare with our in-order CFP proposal.

VI. Conclusion

We have shown that it is possible to design a simple, small core that provides tolerance to cache miss latencies of hundreds of cycles. This core is very promising as a building block for future many-core processors with their limited cache capacity and pin bandwidth per core. Our simulations show up to 85% performance gain on programs that frequently miss the cache.

Even though we have used single-thread benchmarks to test the latency tolerance of our design and its performance benefits, we believe that our results will hold or may even be better for highly parallel applications running on future many-core processors. We expect future parallel applications to exert tremendous pressure on the cache and bus bandwidth. Latency tolerance at low chip area and power cost can help mitigate this pressure.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. *MICRO-36,* December 2003.

[2] R. D. Barnes, E. M. Nystrom, J. W. Sios, S. J. Patel, N. Navarro, and W. W. Hwu. Beating In-Order Stalls with “Flea Flicker” Two-Pass Pipelining. *MICRO-36,* December 2003.

[3] R. D. Barnes, S. Ryoo, W. W. Hwu. “Flea Flicker” Multi-Pass Pipelining: An Alternative to the High power Out-of-Order Offence. *MICRO-38,* November 2005.

[4] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Multithreading (SSMT). *ISCA-26*, May 1999.

[5] S. Chaudhry, P. Caprioli, S. Yip, M. Tremblay. “High-performance Throughput Computing.” *IEEE Micro*, vol. 25, no. 3, pp. 32-45, May 2005.

[6] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-Order Commit Processors. *HPCA-10*, February 2004.

[7] A. Cristal, O. J. Santana, F. Gazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero. Kilo-Instruction Processors: Overcoming the Memory Wall. In *IEEE MICRO, vol. 25, no. 3,* May/June 2005.

[8] A. Cristal, M. Valero, J. Llosa, and A. Gonzalez. Large Virtual ROBs by Processor Checkpointing. *Tech. Report, UPC-DAC-2002-39, Department of Computer Science, Barcelona, Spain*, July 2002.

[9] J. Dundas and T. Mudge. Improving data cache performance by preexecuting instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing*, June 1997.

[10] First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem). Intel Corporation White Paper. http://www.intel.com/pressroom/archive/reference/whitepaper_nehalem.pdf

[11] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan and K. Lai. Scalable Load and Store Processing in Latency Tolerant Processors. *ISCA-32,* June 2005.

[12] J. Held, J. Bautista, and S. Koehl. From a Few Cores to Many: A Tera-Scale Computing Research Review. Intel Research White Paper. http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf.

[13] T. Karkhanis and J. E. Smith. A Day in the Life of a Data Cache Miss. In *Workshop on Memory Performance Issues*, June 2002.

[14] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. *ISCA-29*, May 2002.

[15] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. *MICRO-35*, November 2002.

[16] A.Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronization of Data Dependences. *ISCA-24,* June 1997.

[17] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. *HPCA-9*, February 2003.

[18] D. B. Papworth. Tuning the Pentium Pro Microarchitecture. In *IEEE MICRO, vol. 16, no. 2,* April 1996.

[19] A. Roth and G. S. Sohi. Speculative Data-Driven Multi-Threading. *HPCA-7*, January 2001.

[20] Y. Song and M. Dubois, Assisted Execution. University of Southern California, Technical Report #CENG 98-25, Department of EE-Systems, October 1998.

[21] S. T. Srinivasan, H. Akkary, T. Holman, and K. Lai. A Minimal Dual-Core Speculative Multithreading Architecture. In *Proceedings of the 22nd IEEE International Conference on Computer Design,* October 2004.

[22] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. *ISCA-11,* October 2004.

[23] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *Proceedings of IEEE International Solid-State Circuits Conference,* February 2008.

[24] W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGArch Computer Architecture News,* 23(1):20-24, March 1995.

[25] X86 Cycle Accurate Processor Simulation Design Infrastructure. http://www.ptlsim.org/

[26] C. B. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. *ISCA-28*, June 2001.