On Reducing Overheads in CMP TLS Integrated Protocols

Radulović, B., Milan; Tomašević, V., Milo

Abstract—*The paper primarily tries to identify the main obstacles for performance and complexity improvements in CMPs (speculative chip multiprocessors) with TLS (thread level speculation). It is focused on an analysis of the integrated speculation and coherence protocols in the state-of-the-art CMPs and identifies four areas where the improvements are promising: hardware overhead, software overhead, bursty traffic, and replacement policy. After an overview of each aspect, some ideas for reducing the identified overheads are outlined. Finally, the paper concludes with a very brief sketch of an innovative proposal which employs the lessons learned the previous analysis.*

Index Terms—*coherence protocols, single chip multiprocessors, thread-level speculation*

1. INTRODUCTION

The rapid technology advances in the past decade resulted in emerging of single chip multiprocessors (CMP). Their wide acceptance depends not only on the efficiency for parallel applications, but also on their ability to execute sequential applications in a cost-effective way. For this purpose, CMPs predominantly employ the thread-level speculation (TLS) technique [1].

The CMP architectures with TLS support can be classified into three main groups. The approaches from the first group are completely oriented towards exploiting speculative parallelism, e.g., Multiscalar (ARB and SVC schemes) [1,2], Multiplex [3], Trace processor [4], SM processor [5], MAJC [6], MP98 [7] and MIT MAP [8]. In these proposals application threads can communicate both through registers and the shared memory. It has been shown that hardware and software support for inter-register communication in these architectures is quite acceptable and effective to provide correct speculative execution. In addition, some of these systems, such as Multiscalar, Multiplex, Trace processor, MIT MAP or SM processor, have sufficient hardware support that enables them to deliver high performance by handling sequential binaries without a need for full recompilation of the source code. However, when these systems run a parallel application or a multiprogrammed workload, large amount of speculative hardware and software support remains unutilized.

The second group includes mostly generic CMP architectures with only minimal support for speculative execution, e.g., Hydra [9], STAMPede [10], and SP [11]. These systems restrict the inter-thread communication to occur through memory only. Studies about impact of communication latency on overall performance of speculative CMPs argued that a fast communication scheme between processor cores may not be required and that inter-thread communication through the memory is fast enough to minimize the performance impact from communication delays [9,10,11,12]. The limitation of inter-thread communication through the memory simplifies the design but the need for source code recompilation is still a disadvantage.

IACOMA [12], Atlas [13] and SpecCMP [14] architectures are the representatives of the third group of CMPs with TLS support. They combine the best features of previous two approaches:

- inter-thread communication both through registers and memory, and operation on sequential binaries without need for source recompilation as in the first group,
- modest hardware support for speculative execution and generic enough CMP architecture as in the second group.

The third group of CMPs with TLS support has simpler design and modest hardware/software support for speculative execution compared to CMPs from the first group.

In speculative CMPs, correct speculation handling is usually integrated into protocols for coherence maintenance [15]. Several kinds of overheads or can be noticed in these protocols such as:

- Hardware overhead,
- Software overhead,
- Bursty traffic on thread commit,
- Inappropriate replacement policy.

Our intention is to propose a speculative CMP architecture similar to the systems from the third group since they have a modest hardware and software support for the speculative execution and, also, with a support for the inter-thread communication both through registers and

Manuscript received December 12, 2006.

Milan Radulović is with T-mobile, Podgorica, Montenegro (e-mail: radulovic_milan@yahoo.com). (contact person)

Milo Tomašević is with School of Electrical Engineering, University of Belgrade, Serbia (e-mail: mvt@etf.bg.ac.yu).

memory. The goal is to alleviate some of these overheads and to improve over existing solutions of the same kind. For that reason, in the following sections a brief analysis of overheads observed in Hydra [9], STAMPede [10], SP [11], IACOMA [12], Atlas [13] and SpecCMP [14] is presented. In addition, SVC scheme of Multiscalar employs an important speculative and, therefore, it is also included in this analysis [2].

2. HARDWARE OVERHEADS

IACOMA, STAMPede, SpecCMP, Multiscalar (SVC scheme) and Hydra assign a number of state bits to L1 data cache lines/words as a part of the hardware support for speculative execution. In IACOMA and STAMPede each private L1-cache word and line, respectively, are augmented with additional 6 bits, while SpecCMP has even 8 additional bits assigned to each L1-cache word. Tags of each data cache 32-byte line in Hydra also include additional bits to record the state necessary for speculation. The first two bits, Modified and Pre-Invalidate, augment the basic cache coherence scheme, while the other two sets of bits, Read-by-Word and Written-by-Word, allow the detection of RAW violation using the write bus mechanism. SVC scheme incurs 6-bit overhead plus a pointer for each L1-cache line. The pointer identifies the processor that has the next copy/version, if any, in the Version Ordering List (VOL) for a particular line [2].

SP uses the memory buffer in the run-time dependence checking. Namely, all of the target store entries, from current and the predecessor threads, are stored in the memory buffer. The memory buffer is used as a write-back buffer for the non-target store data. Each entry in the memory buffer includes the address tag and the data field as well as additional bits and bit fields such as: valid bit, alias count, target store distance vector (N bits) for address and target store distance vector (N bits) for data (N is the number of thread processing units) [11].

Also, all speculative CMP architectures exploit different hardware mechanisms for resolving memory data dependences and buffering of speculative states. This support adds on already present hardware overheads.

The disambiguating mechanism in IACOMA is implemented through the memory disambiguation table (MDT) and a related logic that checks for data dependence violations. The MDT is analogous to a directory in a shared-memory multiprocessor. It is a centralized approach that keeps its entries on cache line basis, while the information is maintained on per-word basis. It augments overall bit overhead since each word in MDT has Load and Store bit for each processor. Also, it is possible that the MDT runs out of entries causing a stall of speculative thread while trying to insert a new entry in MDT. By using the values from MDT, the additional Check-on-store logic determines whether any successor thread has performed an unsafe load causing memory dependence violation [12].

The hardware overhead in SpecCMP is caused by complexity of control logic required for operations of the cache controller: ownership probing and data transfer for data forwarding between processor cores, violation detection and state transition. Since the management of speculative state is performed on per-word basis, the increase in area overhead of applied cache directory is significant in comparison to original MSI cache directory. The estimation of the delay of logic on critical paths and additional area overhead caused by added state bits in L1 caches has shown that area overhead occupies more than half of the total delay for many protocol operations. However, the delay caused by accessing and comparing cache tags is higher than the area overhead. The critical path latency is increased by 11% when protocol operations are performed in parallel with tag comparison [14].

The Hydra employs a set of write buffers, rather than the L1-caches, to hold the speculative writes until they can be safely committed into the L2 cache. Hence, the shared L2 cache is guaranteed to hold the non-speculative data only. One write buffer is assigned to each speculative thread. In case when a write buffer has to be drained to L2 cache, the processor core sends the message to the buffer controller to initiate the procedure. There are more sets of buffers than processor cores in Hydra in order to allow continuity of speculative execution when those buffers drain their contents into the shared L2 cache. Although inclusion of write buffers simplifies the protocol, they may become full and stall the speculative threads. Also, a coprocessor is assigned to each processor core to control the thread sequencing The coprocessor has several hardware mechanisms to support speculation and to simplify cache coherence scheme, but it incurs the additional area overhead. It has several control registers, a set of duplicate L2-cache buffer tags, a state machine that tracks the current thread sequence and the interrupt logic to initiate software handlers [9].

The centralized logic called the Version Control Logic (VCL) is applied in SVC scheme as a hardware support for speculation. Each cache line includes a pointer that identifies the processor core that has the next version in the Version Ordering List (VOL) for that line. The VCL uses the bus request, the program order among the tasks and the VOL for appropriate response for each L1 cache when cache misses issue a bus request. Hence, each cache line is updated based on its initial state, the bus request and the VCL response.

The Atlas is a CMP that engages aggressive

speculation techniques to enable the dynamic parallelization of sequential binaries. Thread speculation and data value/control prediction are combined to enable a processor to execute dependent threads in parallel. This architecture is critically dependent on performance of applied sophisticated data value/control predictor. It is implemented together with the global predictor with hardware structures added to each processor core or with modification of already existed hardware support inside each processor core. The inclusion of this mechanism resulted in an area as well as a run-time overhead [13].

3. SOFTWARE OVERHEADS

Hydra uses a coprocessor as a hardware/software interface to control the thread sequencing in the system. These simple "speculation coprocessors" consist of several control registers, a set of duplicate secondary cache buffer tags, a state machine to track the current thread sequencing, and an interrupt logic that can start software handlers to control thread sequencing if necessary. Also, Hydra requires source recompilation, which is a serious problem when source code is not available. The register-level coherence is also handled by a software support, which incurs additional time penalty [9].

Both STAMPede and SP require source recompilation to extract thread-level parallelism. STAMPede uses software speculation handlers and sophisticated compiler technology to support speculative execution [10]. SP architecture heavily relies on compiler to identify speculative threads and to generate an efficient threaded code. It successfully applies both classical and innovative complier techniques for program analysis and transformation in order to exploit more parallelism in programs [11].

4. BURSTY TRAFFIC ON THREAD COMMIT

IACOMA L1 caches work in a restricted write-back mode during the speculative execution and they are not allowed to displace modified lines. However, when a speculative thread acquires non-speculative status, modified lines can be displaced from L1 caches and they switch to write-through mode. When a thread completes and before it commits, any remaining modified cache line is flushed to memory causing the bursty traffic on interconnect. This may increase the time to commit the thread [12]. The same situation appears also in SpecCMP [14].

The STAMPede also performs writes of non-speculative contents on thread completion [10], which results in same problems as in SpecCMP and IACOMA. SP's thread pipelining execution model drains data from memory buffer to the L2 cache during the write-back stage causing bursty traffic [11]. The same action is performed in Atlas on non-speculative thread completion. The write buffer associated to each CPU has to be flushed to the shared L2 cache by broadcasting the data values out to L2 cache and all speculative nodes causing the bursty traffic on an interconnect as well as an increase of thread commit time [12].

5. REPLACEMENT POLICY

The IACOMA speculative protocol allows only non-speculative thread to displace an updated cache word from L1 cache, while any other speculative thread stalls on that occasion. To keep system simple, IACOMA did not include hardware support that allows the committed lines to remain in L1 cache after a new speculative thread starts on a processor [12].

The SpecCMP temporarily holds the speculative data in L1 cache of a speculative thread. When a speculative thread acquires non-speculative status it is allowed to store data to the shared L2 cache and to replace the cache word found either in shared or modified state [14].

A cache line cannot be evicted from the Speculative Data Cache of a processor core in Atlas while that processor runs a speculative thread. In case when a cache line has to be replaced and only if cache lines with active speculation bits are available for replacement, the corresponding speculative thread must stall until it becomes non-speculative. The active speculation bits added to each cache line enable the detection of data dependency violation. If the speculative cache lines are evicted, the processor would not be able to track data dependences anymore and the speculative execution fails [13].

If a speculative thread tries to evict a cache line with the read bits set, the corresponding processor core in Hydra is stalled until the thread becomes either the head thread or is restarted. However, a small victim buffer is added to data cache in each processor core to prevent its stalling until the victim cache is full. The write buffers are added between each processor core and shared L2 cache to collect all writes made by processors during speculative execution. The buffer is drained to the shared L2 cache only when a speculative thread acquires the non-speculative status. They may fill up during speculative execution, so the corresponding threads will be stalled (unless they are restarted) until they become the non-speculative [9].

The STAMPede generates a flush each time a speculative write accesses a dirty cache line. This action is performed in order to ensure that only up-to-date copy of the given cache line is not corrupted with the speculative write. When an epoch tries to replace a speculative cache line from the L1 cache during speculative execution, it is treated as a dependence violation. Two schemes are possible: first, that allows the epoch to proceed and to signal the violation, and

second, that suspends the epoch until it becomes non-speculative and, then, allows the replacement of given cache line [10].

SP uses memory buffers to save store addresses and data for run-time data dependence checking, as well as to save uncommitted produced data. The memory buffer has a fixed size and when it overflows the corresponding speculative thread must be stalled until all of its predecessors are completed. When it becomes the head thread, it obtains non-speculative status and it is allowed to resume the execution. Then, the data from the memory buffer can be saved into the next memory level [11].

6. IMPROVEMENT AVENUES

Based on lessons learned from previous brief analysis, some avenues for an improvement in each of four areas can be summarized:

1. Hardware overhead can be reduced by using L1 data caches to buffer speculative data until they are safe to be saved in the shared L2 cache, and by decreasing the number of additional bits for each cache word in L1 data caches as a support for resolving inter-thread data dependence violations. Also, it would implement simpler hardware mechanisms for thread control and sequencing in each processor core.

2. Software overhead can be reduced if the source recompilation is avoided during partitioning the sequential programs into threads. Avoiding speculation handlers also leads to reducing the software overhead.

3. The bursty traffic during thread commit incurred in most of speculative CMPs causes a delay in issuing a new thread to the core which executed the committed thread. Thread commits can be made more efficient by retaining the modified data in L1 caches until their replacement or intended modification. That way, the delay in issuing a new thread caused by bursty traffic on thread commits can be neglected.

4. Replacement policies applied in current CMP architectures are based on classical temporal history of either cache lines or words that are candidates for replacement and they do not care about their current coherence/speculation states. A new replacement policy should first consider the cache word memory state instead of temporal history of use. However, among the cache words that are in the same memory state the replacement policy can follow classical FIFO, LRU or random policy as a secondary criterion.

7. BRIEF SKETCH OF THE PROPOSAL

With the experience gained from the previous analysis a proposal is made which attempts to achieve a cost-effective solution by removing or reducing the overheads observed in HW/SW support for speculation in current CMP architectures along the recognized improvement avenues. This section very briefly describes the basic elements of the proposal.

1. CMP architecture

Underlying speculative CMP architecture is similar to IACOMA [13], Atlas [14] and SpecCMP [15] and consists of four processor cores, each with private L1 data and instruction caches connected by a snoopy shared bus, while all cores share the unified L2 on-chip cache (Fig. 1). This architecture supports the inter-processor communication both through registers and memory. The hardware support for register communication includes a snoopy shared bus for transferring register values between cores and local scoreboards for keeping the status of data in registers. A hardware mechanism to support thread-level memory speculation is integrated into each processing unit's cache controller.

2. Speculation support

The speculative threads are limited to inner-most loop iterations. The status of non-speculative thread moves from one thread to its immediate successor and so on after a thread completes. It is presumed that threads strictly commit in order to keep the sequential semantics. The thread has to wait to reach the non-speculative status before it can be retired and a new thread to be initiated on the same processor.

The sequential application starts execution on one core, and then, when it reaches a loop entry point the multiple threads are spawned on other cores. Hence, there is always one non-speculative thread, while all its successors are speculative threads. After the last iteration is completed, any iteration that was speculatively spawned after the last one is squashed.

The speculative state is kept in private L1 caches while the shared L2 cache keeps the sequential state (Fig. 1). Speculative data can be committed to the shared L2 cache only after thread becomes a non-speculative. Speculative threads get the required data from appropriate producer thread. In addition, when a successor speculative thread reads a cache word before a predecessor thread writes to the same cache word, the data dependence violation (RAW hazard) is detected. Then, the thread that caused the violation as well as all its successors, that might use data produced by violating thread, has to be squashed and restarted.

3. Thread identification and manipulation

The thread identification in a sequential application is achieved in software during a compilation For this purpose the modified binary annotator based on IACOMA annotation tool is used [16, 17]. The modifications of the IACOMA binary annotator are related to the classification

of writes to the loop-live registers. Those writes can be divided into non-final writes (NFW), final writes (FW) [16], and possibly final writes (PFW) (in enhanced version) [17]. NFW is one that is definitely followed by at least one write to the same register, while FW is definitely the last one. The PFW is one that might be either final or non-final depending on the control flow of the current iteration. The annotator extracts multiple threads from sequential binaries without the need to recompile the source code.

4. Register-level protocols

Two versions of register-level protocol are proposed Snoopy Inter-Register Communication (SIC) protocol [16], and its extended version SIC (ESIC) [17]. The register communication between threads in both of them can be producer-initiated and consumer-initiated.

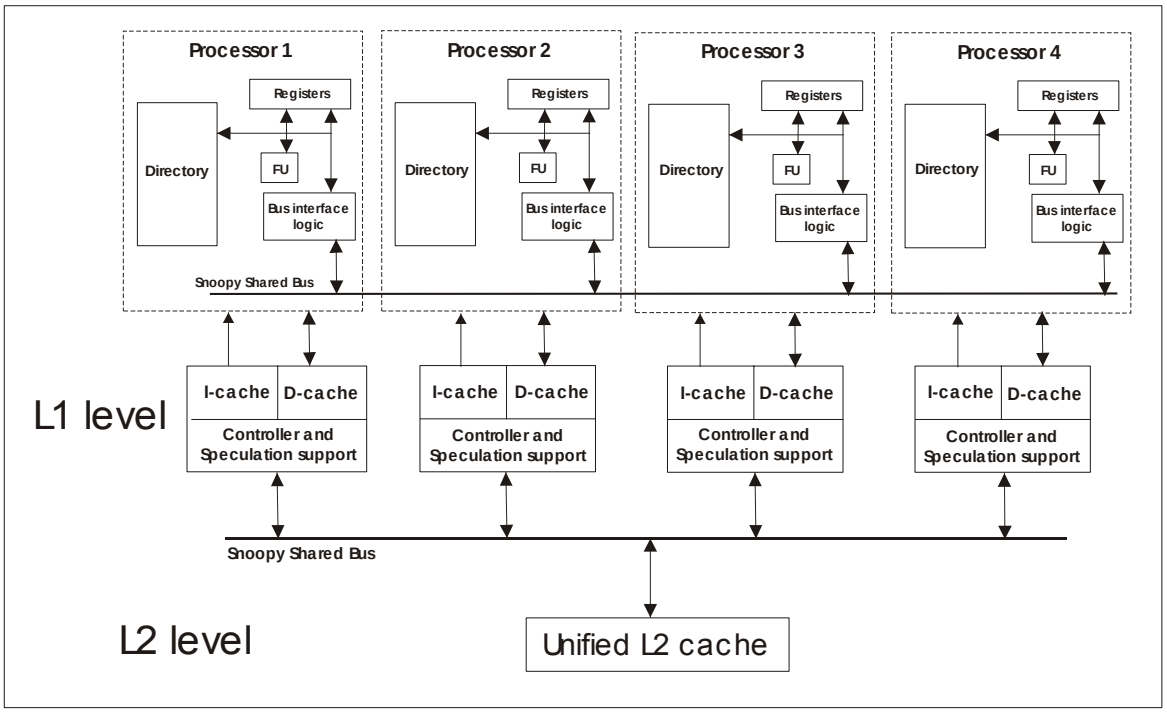

Fig. 1. CMP architecture

A loop-live register value in the SIC protocol can be found in one of the following states: Invalid (INV), Valid-Unsafe (VU), Valid-Safe (VS), and Last Copy (LC). Other registers can be either in Invalid (INV) or Valid-Safe (VS) states. A read miss for a loop-live register causes a consumer-initiated inter-thread communication. All possible suppliers, i.e., predecessors (non-speculative thread or/and earlier speculative threads) participate in distributed bus arbitration which chooses the latest predecessor. If there is no supplier available at the moment, the consumer thread blocks. FW request to a loop-live register incurs the producer-initiated communication by sending the value to the immediate successor thread. If the successor was blocked waiting for this particular value, it continues the execution.

In handling of read misses to a non-looplive registers Read Snarfing is employed in order to decrease a number of misses and to validate the other registers more quickly. Last copy problem is managed by means of a separate state.

However, the SIC baseline scheme does not resolve the problem of consumer thread blocking when requested register value has not been produced yet. Another proposal, the enhanced SIC protocol (ESIC), aims to obtain further performance gain preventing a consumer thread blocking in SIC scheme by aggressively and speculatively forwarding of possibly final register values. Although the ESIC protocol introduces two additional states, Valid-Possibly-Safe (VPS) and Valid-Possibly-Safe-Forwarded (VPSF), hardware and software support for this proposal are very slightly more demanding.

5. Memory-level protocols

The memory communication, buffering of speculative data and data dependence detection in this architecture is performed through L1 write-back data caches with added support for speculative execution. Compared to register dependences it is very difficult to identify memory dependences from the binaries. Therefore, we decided to include a hardware support that is fully responsible for identification and managing of memory dependencies as well as for recovering the execution. The proposal for speculative execution respects the strict sequential semantics both in thread retiring, and a new thread initiation.

This proposal for cache coherence protocol

with TLS support relies on snoopy cache coherence principles [15]. The baseline scheme modifies the snooping bus-based cache coherence protocol to support speculative versioning for proposed CMP architecture.

The proposed integrated coherence and speculation protocol is called Speculation Integrated with Snoopy Coherence (SISC) protocol. The memory state is managed on a per-word basis since previous work has shown that memory state management on a per-line basis could result in false dependence detection that causes the unnecessary threads squash and worse performance [2, 13, 15].

Three variants of speculative cache coherence protocol are proposed: first, a write-invalidate SISC protocol as the baseline scheme, second, an enhanced SISC variant, and, third, a write-update based SISC protocol.

The baseline scheme of SISC write-invalidate protocol uses 11 states to keep track of data status. It enables consumer-initiated inter-thread communication only. The correct value is supplied by most recent speculative thread, non-speculative thread or shared L2 cache.

The committed versions in baseline SISC protocol can remain in the caches after the thread that created the data has been committed. This way, the possible bursty bus traffic that may increase the time to commit the thread and delay initialization of a new thread to that processor is avoided. A modified committed cache word is written back to L2 cache when it is accessed next time either on local thread read/write request or on remote request(s) from other thread(s). The cache words found in stale, shared or modified committed state remains in the same state when a speculative thread reaches non-speculative status. Then, the thread can be retired and a new thread initiated on the same core.

Following the analogy with register-level protocols, since the baseline SISC protocol does not support producer initiated communication on memory level, it is extended with new state Shared-Unsafe (SU) in order to support non-demand inter-thread communication on write hits and misses. If a cache word is either in shared or stale state on a non-speculative write, or in any shared or speculatively modified state on a speculative write, it is forwarded to successor threads on non-demand basis.

Third version of the cache coherence protocol with TLS support in proposed CMP architecture is based on write-update protocol. It uses 9 states and an additional stale bit for each cache word in L1 cache. The inter-thread memory communication in this version can be both producer-initiated and consumer-initiated. In SISC write-update protocol whenever a shared cache word is written to the corresponding L1 cache, its value is updated in the L1 caches of all successor threads that also have a given cache word in shared state.

In all three variants of SISC protocol, an improved replacement policy is employed. It considers the cache word memory state as primary criterion for eviction instead of usual temporal history of reference. The first candidates for replacement are the words in non-speculative states, because the replacement of a speculative word would causes a speculative thread squash and restart. It allows a speculative thread to evict from its L1 cache the modified committed cache words that are only up-to-date copies and that have been speculatively read or written. Also, clean words that have not been yet speculatively either read or write can be replaced. Hence, the processor stall will be avoided and the speculative execution will continue. However, among the clean cache words that are in the same state standard FIFO, LRU or random policy can be considered as the secondary criterion.

8. CONCLUSION

Current CMPs with TLS support have been analyzed for possible improvements. It resulted in identification of overheads in four areas: hardware complexity, software support, bursty traffic on thread commit and replacement policy. Based on the results of the previous analysis, a proposal of CMP architecture with protocols for register-level and memory-level communication is made which tries to reduce those overheads.

This proposal decreases the directory overhead by using fewer bits both in register and memory protocols. Also, SISC protocols use the distributed arbitration for finding the appropriate supplier thread with modest hardware support. The software support doesn't include speculation handlers nor requires source recompilation. SISC protocols retain the non-speculative data in L1 caches on thread commits over time until their replacement in order to minimize the delay in issuing new threads. They also implement an improved replacement policy that reduces thread squashing and stalling. Future evaluation study should reveal the quantitative performance effects of proposed optimizations.

REFERENCES

[1] Sohi, G.S., Breach, S., Vijaykumar, T.N., "Multiscalar Processors", *Proc. of 22nd ISCA*, May 1992, pp. 414-425.

[2] Gopal, S., Vijaykumar, T., Smith, J. E., Sohi, G. S., "Speculative Versioning Cache", *Proc. of the 4th HPCA*, February 1998.

[3] Ooi, C.L., Kim, S.W., Park, L., Eigenmann, R., Falsafi, B., Vijaykumar, T.N., *"Multiplex: Unifying Conventional and Speculative Thread-Level Parallelism on a Chip-Multiprocessor"*, ICS-15, June 2001.

[4] Rotenberg, E., et al, "Trace Processors", *Micro-30*, December 1997, pp. 68-74,.

[5] P.Marcuello, A. Gonzalez, "Control and data dependence speculation in multithreaded processors", *Proc. of the 4th HPCA*, February 1998.

[6] Tremblay, M., “MAJC™: An Architecture for the New Millennium”, *Proc. Hot Chips 11*, Aug. 1999, pp. 275-288.
[7] Matsushita, S., et al., “Merlot: A Single-Chip Tightly Coupled Four-Way Multi-Thread Processor”, *Proc. Cool Chips III Symp.,* Apr. 2000, pp. 63-74.
[8] Keckler, S., et al., “Exploiting Fine-Grain Thread Level Parallelism on the MIT ALU Processor”, *Proc. of 25th ISCA*, ACM press, Jun.-Jul. 1998, pp. 306 -317.
[9] Hammond, L., et al., “The Stanford Hydra CMP”, *IEEE Micro*, 2000, pp. 71-84.
[10] Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C., “A Scalable Approach to Thread-Level Speculation”, *ISCA 2000*, June 2000.
[11] Tsai, J.Y., Huang, J., Amlo, C., Lilja, D. J., Yew, P-C., “The Superthreaded Processor Architecture”, *IEEE Trans. on Computers*, September 1999. pp. 881-902
[12] Krishnan, V., Torrellas, J., “A Chip-Multiprocessor Architecture with Speculative Multithreading.” *IEEE Trans. on Computers,* September 1999, pp. 866-880.
[13] Codrescu, L., Scott Wills, D., “Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications”, *1999 IEEE International Conference on Computer Design,* October 1999.
[14] Yanagawa, Y., et al., “Complexity Analysis of a Cache Controller for Speculative Multithreading Chip Multiprocessors”, *HiPC - International Conference on High Performance Computing*, December 2003.
[15] Tomašević, M., Milutinović, V., The Cache Coherence Problem in Shared Memory Multiprocessors: Hardware Solutions, *IEEE Computer Society Press*, 1993.
[16] Radulović, M., Tomašević, M., “A Proposal for Register-level Communication in a Speculative Chip Multiprocessor“, *XLIX ETRAN Conference,* June 2005.
[17] Radulović, M., Tomašević, M., “An Aggressive Register-level Communication in a Speculative Chip Multiprocessor”, *IEEE EUROCON2005 Conference,* November 2005.