Boosting Single-thread Performance in Multi-core Systems through Fine-Grain Multi-Threading

Carlos Madriles, Pedro López, Josep M. Codina, Enric Gibert, Fernando Latorre,
Alejandro Martínez, Raúl Martínez and Antonio González

Intel Barcelona Research Center, Intel Labs
Universitat Politècnica de Catalunya, Barcelona (Spain)

{carlos.madriles.gimeno, pedro.lopez, josep.m.codina, enric.gibert.codina, fernando.latorre,
alejandrox.martinez, raulx.martinez, antonio.gonzalez}@intel.com

ABSTRACT

Industry has shifted towards multi-core designs as we have hit the memory and power walls. However, single thread performance remains of paramount importance since some applications have limited thread-level parallelism (TLP), and even a small part with limited TLP impose important constraints to the global performance, as explained by Amdahl's law.

In this paper we propose a novel approach for leveraging multiple cores to improve single-thread performance in a multi-core design. The proposed technique features a set of novel hardware mechanisms that support the execution of threads generated at compile time. These threads result from a fine-grain speculative decomposition of the original application and they are executed under a modified multi-core system that includes: (1) mechanisms to support multiple versions; (2) mechanisms to detect violations among threads; (3) mechanisms to reconstruct the original sequential order; and (4) mechanisms to checkpoint the architectural state and recovery to handle misspeculations.

The proposed scheme outperforms previous hardware-only schemes to implement the idea of combining cores for executing single-thread applications in a multi-core design by more than 10% on average on Spec2006 for all configurations. Moreover, single-thread performance is improved by 41% on average when the proposed scheme is used on a Tiny Core, and up to 2.6x for some selected applications.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures, D.3.4 [**Programming Languages**]: Processors – *compilers, code generation, optimization*.

General Terms

Performance, Design.

Keywords

Speculative multithreading; Core-fusion; thread-level parallelism; single-thread performance; multicore; automatic parallelization.

ISCA'09, June 20–24, 2009, Austin, Texas, USA.

1. INTRODUCTION

Single-threaded processors have shown significant performance improvements during the last decades by exploiting instruction level parallelism (ILP). However, this kind of parallelism is sometimes difficult to exploit requiring complex hardware structures that lead to prohibitive power consumption and design complexity. In this scenario, chip multiprocessors (CMPs) have emerged as a promising alternative in order to provide further performance improvements under a reasonable power budget.

CMP processors comprise multiple cores where parallel workloads are executed. They exploit what is called thread level parallelism (TLP). However, the design of a CMP able to efficiently exploit TLP and ILP is not straight-forward. For instance, multi-core architectures based on simple cores are very effective to exploit TLP but their performance is compromised for lowly threaded applications. By contrast, multi-core architectures based on big cores usually have very few of them because of area and power constraints. They have limited TLP capabilities but are more effective to exploit ILP.

Core Fusion [11] is an alternative that tries to get the best of both worlds. Core Fusion schemes use small cores to exploit highly threaded workloads, while dealing with lowly threaded scenarios by combining the computational capabilities of several cores. In order to achieve the effect of combining cores instructions are dynamically distributed to cores. Therefore, Core Fusion relies on parallelizing techniques but it is limited to the exploitation of ILP inside an instruction window of a typical size. It is well known that the parallelism available in a given instruction window is quite dependent on the size of this window.

In this paper we propose *Anaphase*, a hardware/software co-designed threading scheme that leverages multiple cores to execute a single-threaded code. This scheme is based on a novel speculative multithreading technique that decomposes single-threaded applications in a fine-grain fashion. In this approach, the compiler is responsible for distributing instructions to cores whereas the hardware includes special components to support this execution model.

When decomposing an application into speculative threads, the independence among threads is not guaranteed and memory accesses are not performed in sequential order. The hardware is responsible for reconstructing the original program order and detecting memory dependence violations among threads. Moreover, the hardware supports different versions of the memory state for each core. Finally, checkpointing and recovery mechanisms are implemented through special hardware support.

In this paper we focus on the hardware for *Anaphase*, and the main contributions of this paper are as follows:

- A novel fine-grain threading-based approach to exploit multiple cores for boosting single-thread performance. We present results that show that this approach significantly speeds up single-threaded code, and it outperforms previous hardware-only schemes such as Core Fusion.
- A cost-effective hardware support to implement *Anaphase* on top of a conventional multi-core system. In particular, a novel hardware component named *Inter-Core Memory Coherency Module* (*ICMC*) is proposed. The *ICMC* updates the memory state in program order, detects memory violations, and implements checkpointing and recovery in case of misspeculation.

The rest of the paper is organized as follow. In Section 2, previous work related to *Anaphase* is discussed. In Section 3 the *Anaphase* scheme is described. In Section 4, the hardware support for *Anaphase* is described. Finally, the evaluation is presented in Section 5, future work in Section 6, and we conclude in Section 7.

2. RELATED WORK

2.1 Multi-core designs

A significant amount of research efforts have been devoted to come up with appropriate design points to deal with ILP and TLP. These include alternatives comprising either a large number of tiny cores [3], a limited number of big cores [20], or heterogeneous designs [13]. Some researchers propose adaptive architectures and reconfigurable hardware where the characteristics of the architecture dynamically adapt to the parallelism of the applications. As far as we know, the techniques more closely related to our proposal are those that pursue the idea of combining or fusing cores [11][17][27].

Core Fusion can be seen as a natural evolution of clustered microarchitectures [4][6]. In a clustered microarchitecture, instructions are distributed among the clusters statically at compile time, or dynamically through a steering logic. Instead, Core Fusion involves the task of distributing instructions in cores rather than clusters. Thus, starting from a single-threaded code, Core Fusion decomposes a dynamic stream of instructions into different hardware contexts. Therefore, Core Fusion exploits the concept of non-speculative multithreading execution where instructions are distributed in a fine-grain fashion. However, the parallelism exploited by core fusion techniques is limited to ILP. In this paper, we show that *Anaphase* achieves higher performance than Core Fusion techniques by exploiting TLP, in addition to ILP.

2.2 Threading Schemes

Traditional speculative multithreading schemes decompose sequential codes into large chunks of consecutive instructions [2][5][7][8][12][18][21][24][25][26]. Such coarse grain decomposition may constraint the benefits of this paradigm. This is particularly true when facing hard to parallelize codes, where coarse grain decomposition may introduce too many dependences among threads. This may end up limiting the parallelism in this codes and harming performance. Instead, *Anaphase* parallelizes applications at instruction granularity, which provides more flexibility and thus it has more potential to further exploit TLP than previous schemes.

Moreover, previous hardware proposals for speculative multithreading [1][5][9][10][16][22] do not support the execution of speculative threads at the finer granularity proposed in this paper.

3. ANAPHASE

The proposed *Anaphase* scheme decomposes a sequential application into speculative threads (SpMT threads) at compile time. SpMT threads are generated for those regions that cover most of the execution time of the application. Although *Anaphase* can decompose a region into any number of speculative threads, in this paper we limit our study to partitioning each region into two threads. In this section we first describe the speculative threads considered in this model and its execution model and, then the compiler algorithms for generating them.

3.1 Threads

The main feature of the proposed threading scheme is that the application is shred into speculative threads at instruction granularity. An example of such fine-grain decomposition is shown in Figure 1. Figure 1 (a) depicts the static control flow graph (CFG) of a loop and a possible dynamic execution of it consisting of the basic block stream {A, B, D, A, C, D}, while Figure 1 (b) shows a possible fine-grain decomposition into speculative threads.

Inter-thread dependences might arise between speculative threads. These dependences occur when a value produced in one speculative thread is required by another thread. Inter-thread dependences are detected at compile time analyzing the code and using profile information. Since not all actual dependences can be identified by the profiler (i.e. memory dependences), the resulting threads are speculative and may sometimes execute wrong. The hardware is responsible for identifying when the execution is wrong and act accordingly.

For all inter-thread dependences identified at compile time, appropriate code is generated in the speculative threads to handle them. In particular, one of the following techniques is applied: (i) the dependence is satisfied by an explicit communication; or (ii) the dependence is satisfied by a pre-computation slice (p-slice), that is the subset of instructions needed to generate the consumed datum. Instructions included in a p-slice may need to be assigned to more than one thread. Therefore, speculative threads may contain replicated instructions, as is the case of instruction D1 in Figure 1.

Finally, another feature of the proposed scheme is that each speculative thread is self-contained from the point of view of the control flow. This means that each thread has all the branches it needs to resolve its own execution. Note that in order to accomplish it those branches that affect the execution of the instructions of a thread need to be placed on the same thread. In case a branch needs to be placed in more than one thread it is replicated. This is also handled by the compiler when threads are generated.

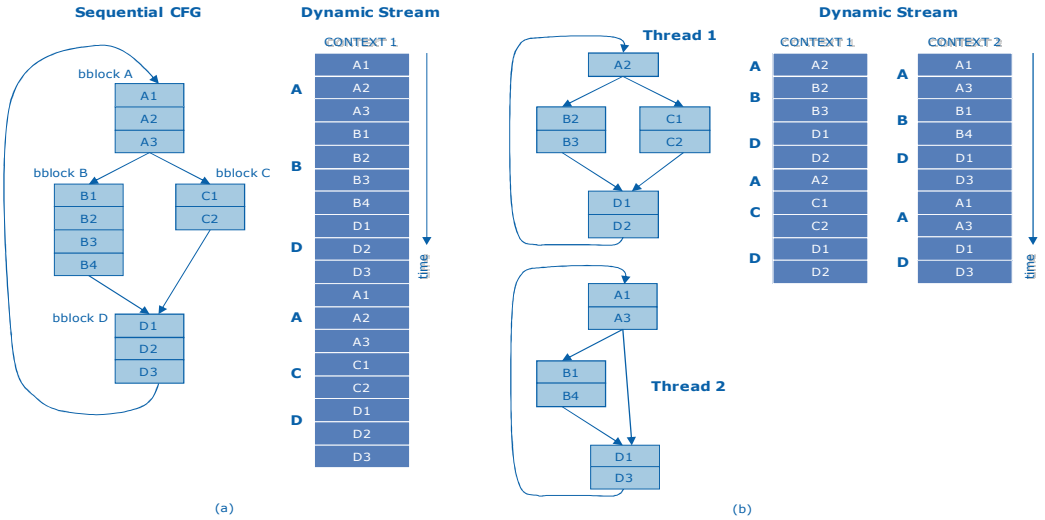

Figure 1. Conceptual view of the fine-grain decomposition into speculative threads.

3.2 Execution Model

The compiler detects that a particular region B is suitable for applying speculative multithreading. Hence it decomposes B into two speculative threads that are mapped somewhere else in the application code. We refer to this version of B as the optimized version.

A spawn instruction is inserted in the original code before entering region B. This instruction creates a new thread, and both, the spawner and the spawnee speculative threads, start executing the optimized version of the code. When both threads complete, they synchronize, the speculative state becomes non-speculative and execution resumes on single thread.

Two speculative threads synchronize every time an inter-thread dependence must be satisfied by an explicit communication. However, communications imply synchronization only on the consumer, since the producer puts the produced datum into a buffer in memory.

On the other hand, checkpointing is performed by hardware at the places decided by the compiler through CKP instructions. This instruction marks the place where the register checkpoint can be taken. In this paper, CKP instructions are inserted at the beginning of any loop belonging to optimize regions. Then in case of violations, exceptions and interrupts the speculative threads are squashed, and the execution jumps to a recovery code generated at compile time, which in our case is the original sequential execution.

3.3 Decomposition Algorithm

Speculative threads are generated at compile time. The compiler is responsible for: (1) profiling the application, (2) analyzing the code and detecting the most convenient regions of code for parallelization, and finally, (3) decomposing the selected region into speculative threads.

Anaphase decomposes selected regions using a multi-level graph partitioning [14]. This algorithm consists of two main steps: coarsening and refinement. The coarsening step creates a first partition of instructions among speculative threads. Then, the initial partition is refined in the refinement step by moving some instructions from one thread to the other.

3.3.1 Coarsening

The coarsening step receives the Data Dependence Graph (DDG) with profiling information indicating the number of occurrences of each instruction and dependence. Then, it iteratively reduces the DDG by collapsing pairs of nodes into supernodes until the final graph has as many supernodes as threads, describing a first partition of instructions to threads.

The coarsening algorithm gives the highest priority to the fusion of those instructions belonging to the critical path. In case of a tie, it gives priority to those instructions that have larger number of common ancestors since we have measured that this heuristic provides the greater benefits to fuse instructions. By contrast, the algorithm promotes workload balance among threads giving very low priority to the fusion of nodes that do not depend on each other (directly or indirectly). Finally, memory level parallelism (MLP) is promoted giving very low priority to the fusion of delinquent loads [7] and their consumers. Loads with a miss rate higher than 10% in the L2 cache during profiling are considered as delinquent

3.3.2 Refinement

The second step of the multi-level graph partitioning is the refinement process. This step traverses the different supernodes created during the coarsening step and tries to refine the partition by moving supernodes to other threads and estimating their benefits based on an evolution of the traditional K-L algorithm [15]. The benefits are estimated by analyzing the execution time of the partition obtained after the movement.

The execution time of a partition is estimated using a 20K dynamic instruction stream of the region obtained through profiling. Execution time is then computed, as the number of cycles of the longest thread, using a simple performance model that takes into account dependences, issue width resources and the size of the ROB (Reorder Buffer) of the target out-of-order core.

The studied partition in each refinement includes all necessary branches in each thread to compute its control path, as well as all

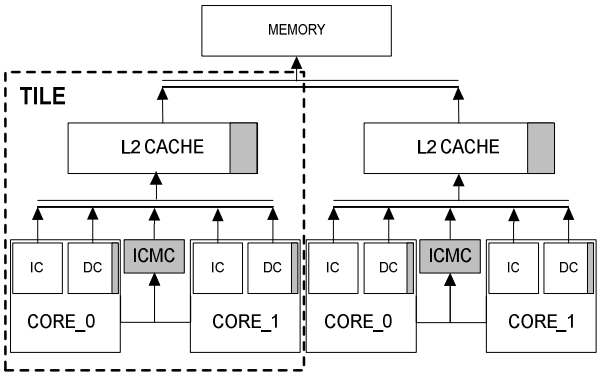

Figure 2. Multicore architecure overview.

required communications and p-slices. Given an inter-thread dependence, it is explicitly communicated if the amount of replicated dynamic instructions estimated to satisfy the dependence locally exceeds a threshold. Otherwise, the p-slice is built in the destination thread.

4. ANAPHASE ARCHITECTURE

In this paper we consider a multi-core x86 architecture [20] divided in tiles as shown in Figure 2. Every core executes instructions out-of-order. Every tile implements two cores with private first level write-through data cache and instruction cache. The first level data cache includes a state-of-the-art stream hardware prefetcher. These caches are connected to a shared copy-back L2 cache through a split transactional bus. The L2 cache is connected through another interconnection network to main memory and to the rest of the tiles.

Tiles have two different operation modes: single-core mode and cooperative mode. These cores execute conventional threads when the tile is in single-core mode and they execute speculative threads (one in each core) from the same decomposed application when the tile is in cooperative mode. When two *Anaphase* threads start running the optimized code and, the spawn instruction is executed, the cores transition from single to cooperative-core mode.

When two speculative threads are running on a tile in cooperation-mode, synchronization among them occurs when an inter-thread dependence must be satisfied by an explicit communication. However, communications imply synchronization only on the consumer side, since the producer puts the produced datum when retires into a buffer in memory. Then, communications are handled through the regular memory hierarchy.

On the other hand, every tile implements a hardware component called *Inter-Core Memory Coherency Module* (*ICMC*) that is in charge of controlling the activity of the cores inside the tile when they execute in cooperative mode. This piece of logic should interfere very little with the cores. Hence, the cores fetch, execute and retire instructions from the speculative threads in a decoupled fashion most of the time. Then, a subset of the instructions is sent to the *ICMC* after they retire in order to perform the validation of the execution. The set of instructions considered by the *ICMC* is limited to memory and some control instructions. We refer to those instructions as ordering instructions.

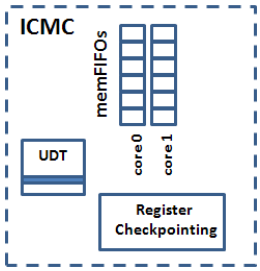

Figure 3. *ICMC* main structures.

The *ICMC* receives the ordering instructions and it handles them through the three structures shown in Figure 3. These components, together with some bit extensions included in the memory hierarchy provide the hardware support for *Anaphase*:

- The *ICMC* sorts ordering instructions in order to: (1) make changes made by the multi-threaded application visible to the other tiles as if it would have been executed sequentially; and (2) detect memory dependence violations among the threads running on the cores of the tile. For the purpose of reconstructing the memory sequential order the *ICMC* implements one FIFO queue (*memFIFO*) per core as shown in Figure 3. These queues keep ordering instructions when they retired from the associated core as described in Section 4.1.
- The *ICMC* and the extended memory hierarchy inside a tile allow each core running in a cooperative mode to update its own memory state, while still committing the same state that the original sequential execution will produce. This is accomplished as described in Section 4.2 by: (1) allowing different versions of the same line in multiple L1's; and (2) avoiding speculative updates to propagate outside the tile.
- On the other hand, the proposed scheme requires some form of register checkpointing to roll back the state to a correct state when a misspeculation is found. Frequent checkpoints should be taken in order to keep the penalty due to a misspeculation small, without increasing too much the overhead to create them. The *ICMC* implements a novel scheme described in Section 4.3 that can take frequent checkpoints (every few hundreds of instructions) of the architectural register state while allows a core running in a cooperative mode to retire instructions, reclaim execution resources and keep doing forward progress even when other cores are stalled.

We describe the aforementioned tasks in more detail on the following sections.

4.1 Reconstructing Memory Sequential Order

When a tile is executing in cooperative mode, the *ICMC* is in charge of reconstructing the original sequential order of memory instructions that have been arbitrarily assigned to the speculative threads. This order allows detecting memory violations and updating memory correctly.

The sequential order is reconstructed using certain marks called *Program Order Pointer* (*POP*) bits. *POP* bits are included by the compiler in ordering instructions. The *POP* bit of every ordering instruction indicates the speculative thread (e.g. 0 or 1) where the

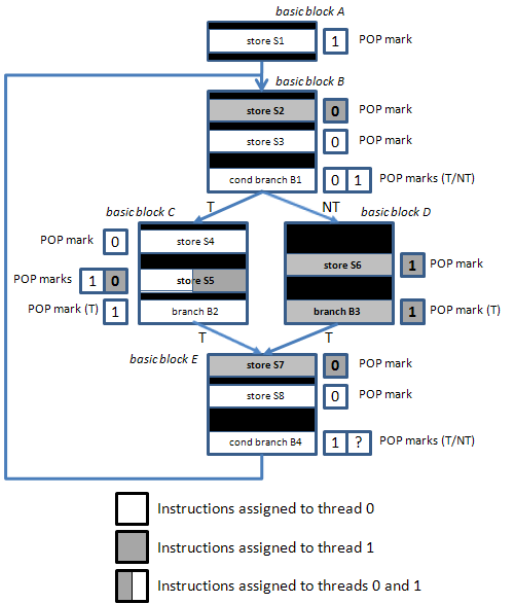

Figure 4. Example of how *POP* pointers are assigned to ordering instructions. In this example "?" is used when there is no information about the following instructions.

next instruction in the original sequential code was assigned. In case of conditional branches, two *POP* bits are included; one is used when the branch is taken and the other when the branch is not taken. An example of how these *POP* bits are assigned to instructions is shown in Figure 4. For example, load L1 has a *POP* bit of 0 because the next ordering instruction in sequential order (branch B1) is assigned to speculative thread 0.

The order between two replicated instructions is not important as long as the order with respect to the rest of the instructions is guaranteed. Finally, indirect branches can have many destinations. This scenario can be handled by forcing the first ordering instruction of all known destinations to be assigned to the same thread. Alternatively nops are used as ordering instructions in those destination paths where the first instruction fits better in a different thread.

The *ICMC* is in charge of committing all instructions in the original program order using the *POP* bits. In particular, when a core retires an ordering instruction, the instruction is stored in the *memFIFO* associated to the core. Then, the *ICMC* process and removes the instructions from the *memFIFOs* based on the *POP* bits. The value of the *POP* bit of the last committed instruction identifies the head of the *memFIFO* where the next instruction to commit resides. We have measured that a switch between threads is performed each 1.7 instructions on average. This clearly show the fine-grain granularity of the *Anaphase* approach.

Note that instructions are committed by the *ICMC* when they become the oldest instructions in the system in original sequential order. Therefore, this is the order in which store operations can update the shared cache levels and be visible outside the tile.

From now on, we say that an instruction *retires* when it becomes the oldest instruction in a core and do the retirement. By contrast, we say that an instruction *globally commits*, or *commits* for short,

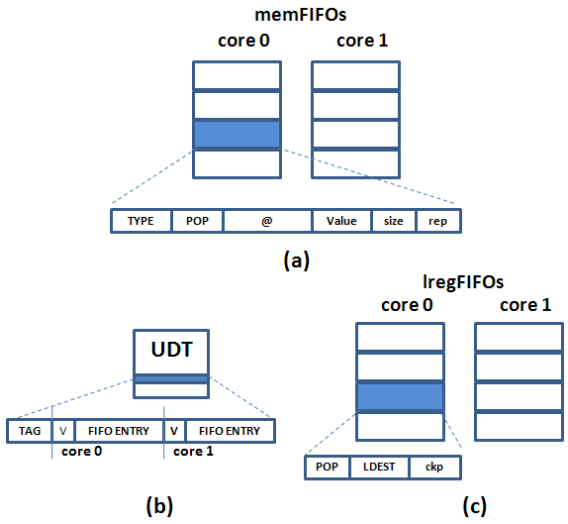

Figure 5. Contents of the main structures in the *ICMC*.

when the instruction is processed by the *ICMC* because is the oldest in the tile.

The contents of the entries in the *memFIFOs* is detailed in Figure 5 (a): 2 TYPE bits that identify the type of instruction (ld, st, branch, ckp); 1 *POP* bit generated by the compiler; 64 bits for memory address; 32 bits for the value in case of a store; 2 bits to describe the size of the memory access; and 1 bit to mark replicated (*rep*) instructions. Replicated instructions are marked to avoid the *ICMC* to check for dependence violation.

Note that *memFIFOs* allow each core to fetch, execute and retire instructions independently at core-level. The only synchronization happens when a core prevents the other core to retire instructions. A core may eventually fill up its *memFIFO* and stall until its retired instructions can leave the *memFIFO*. This situation occurs when the next instruction to be globally committed comes from a different core and this instruction has not retired yet.

4.2 Memory State Management

In Figure 6 the changes introduced in the memory hierarchy from the point of view of a single tile are shown. The main components and changes involved in managing the memory state are highlighted in grey. In the following sections the modifications in the L1 cache, L2 cache and the use of the *UDT* are detailed.

4.2.1 L1 Cache

The L1 data caches do not invalidate other L1 caches in cooperative mode when a line is updated: each L1 cache may have a different version of the same datum. These caches are extended with a versioned bit (*V*) per line. The *V* bit of a line in one core is set when a store instruction executes in that core and updates that line similar to [19].

Speculative updates to the L1 are not propagated (written-through) to the shared L2 cache. Store operations are sent to the *ICMC* which is in charge of updating the L2 cache in the original order when they globally commit.

When a line with its *V* bit set is replaced from the L1, its contents are discarded. When the cores transition from cooperative to

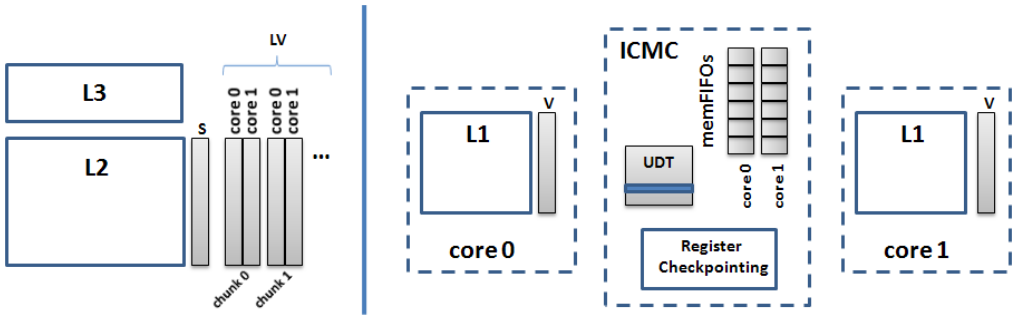

Figure 6. The extented memory hierarchy of a tile.

single-core mode, all the L1 lines with the *V* bit set are invalidated since the correct data resides in the L2 and the *ICMC*.

4.2.2 L2 Cache

The shared L2 cache is extended with a speculative bit (*S*) per line and a set of two last version bits (*LV*) per chunk of information. A chunk is the granularity at which memory disambiguation between the two speculative threads (and hence, memory violations) are detected. In this work, we consider byte granularity.

When a store commits, it updates the corresponding L2 line and sets its *S* bit to 1. Such *S* bit describes that the line has been modified since the last checkpoint. Once a new checkpoint is taken, the *S* bits are cleared. In case of a misspeculation, the threads are rolled back and the lines with an *S* bit set are invalidated. Hence, when a non-speculative dirty line is to be updated by a speculative store, the line must be written back to the next memory level in order to have a valid non-speculative version of the line somewhere in the memory hierarchy. Since speculative state cannot go beyond the L2 cache, an eviction from the L2 of a line that is marked as speculative (*S*) implies rolling back to the previous checkpoint to resume executing the original application.

On the other hand, the *LV* bits indicate the core that has the latest version of a particular chunk. When a store commits, it sets the *LV* bits of the modified chunks belonging to that core to one and resets the rest. If a store is tagged as replicated (executed by both cores), both cores will have the latest copy. In this case, the *LV* bits are set to 11. Upon a global commit of a load, these bits are checked to see whether the core that executed the load was the core having the latest version of the data. If the *LV* bit representing the core that executed the load is 0 and the bit for the other core is 1, a violation is detected and the threads are squashed. This is so because as each core fetches, executes and retires instructions independently and the L1 caches also work decoupled from each other, the system can only guarantee that a load will read the right value if this was generated in the same core.

4.2.3 UDT

The *Update Description Table* (*UDT)* is a table that describes the L2 lines that are going to be updated by store instructions located in the *memFIFO* queues. The purpose of the *UDT* is to delay any fill from the shared L2 cache to the L1 cache as long as there are still some stores pending to update that line. This way we avoid filling an L1 with a stale line from the L2. In particular, a fill to the L1 of a given core is delayed until there are no more pending stores in the *memFIFOs* for that particular core (there is no any entry in the *UDT* for the line tag).

Note that there is no need to wait for stores from other cores that access the same line, since in case of a memory dependence the *LV* bits will already detect it, and in case that the two cores access different parts of the same line, the *ICMC* will properly merge the updates at the L2.

An *UDT* entry as shown in Figure 5 (b) has: the tag identifying the L2 line, plus a valid bit attached to a *memFIFO* entry id for each core. The *memFIFO* entry id is the entry number of the last store that updates that line. This field is updated every time a store is appended to a *memFIFO*. If a store writes a line without an entry in the *UDT* then it allocates a new entry. By contrast, if a committed store is pointed by the *memFIFO* entry id then its valid bit is set to false; and if both valid bits are false then the entry is removed from the *UDT*.

4.3 Register State Management

In order to allow cores running in cooperative mode to work asynchronously, the *ICMC* is in charge of merging and building the register architectural state. Therefore, a core does not have to generate the complete architectural state. Instead, this can be partially computed by multiple cores.

Figure 7 shows a conceptual view of the checkpointing mechanism. This mechanism conceptually creates a ROB where instructions are stored in the order they should be globally committed. However, since the threads execute asynchronously, the entries in this conceptual ROB are not allocated sequentially. Instead we have areas where we do not know either how many nor the kind of instructions to be allocated there. This situation may happen if for instance the core 0 is executing a region of code that should be committed after the instructions executed from core 1. In Figure 7, *GRetire_C* points to the last instruction retired by core C. As it can be seen, core 0 goes ahead of core 1 so that there are gaps (shown as shaded regions) between *Gretire_0* and *Gretire_1*.

Checkpoints taken by the core that retires the youngest instructions in the system are always partial checkpoints. We cannot guarantee that this core actually produces a whole architectural state. By contrast, checkpoints taken by the core that does not retire the youngest instruction in the system are complete checkpoints because it knows the instructions older than the checkpoint that the other core has executed. Therefore, it knows where each of the architectural values resides at that point.

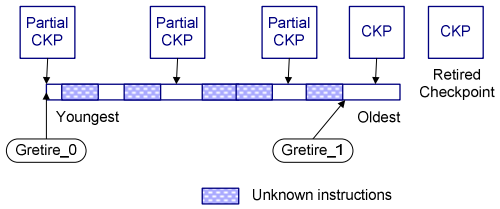

Figure 7. Conceptual view of register checkpointing.

The reason why a core takes periodic checkpoints even when they are partial, as core 0 in the example, is because all physical registers that are not pointed by these partial checkpoints are reclaimed. This feature allows this core to make forward progress with little increase on the pressure over its register file. Moreover, when the other core reaches the checkpoint, core 1 in the example, it is guaranteed that the registers containing the values produced by core 0 that belong to the architectural state at this point have not been reclaimed so that we can build the complete checkpoint with the information from core 1. On the other hand, registers being allocated in core 0 that did not belong to the checkpoint because they were overwritten by core 1 can also be released.

Note that the core that goes ahead is not always the same. This role changes depending on the decomposition of instructions among threads. Therefore, the role of complete checkpointing is moving from one core to the other.

At a given time, a complete checkpoint has pointers to the physical registers in the register files (either in core 0 or 1) where the value resides for each logical register. A checkpoint can be released and its physical registers reclaimed when all instruction have been globally committed and a younger checkpoint becomes complete.

A checkpoint is taken when a *CKP* instruction inserted by the compiler is found, and at least a minimum number of dynamic instructions have been globally committed since the last checkpoint (*CKP_DIST_CTE*). This logic is shown in Figure 8. This *CKP* instruction has the IP of the recovery code which is stored along with the checkpoint, so that when an interrupt or data misspeculation occurs, the values pointed by the previous checkpoint are copied to the core that will resume the execution of the application. In this paper, we consider that this copy is done by hardware. However, it can also be done by software as the beginning of a service routine.

A checkpoint includes the IP of the instruction where the checkpoint was created, the IP of the rollback code, and an entry for each logical register in the architecture. Each of these entries have: the physical register (*PDest*) where the last value produced prior to the checkpoint resides for that particular logical register; the overwritten bit (*O*) which is set to 1 if the *PDest* identifier differs from the *PDest* in the previous checkpoint; and the remote bit (*R*) which is set to 1 if the architectural state the logical register resides in another core.

The components included in the *ICMC* for handling these checkpoints comprise:

- One FIFO queue (*lregFIFO*) per core where all retired instructions that writes a logical register allocate an entry. Each entry as shown in Figure 5 (c) consists of: 1-bit field named ckp that is set to 1 in case there is an architectural state checkpoint associated to this entry; the identifier of the logical register written by the instruction (*LDest*); and the *POP* bit to identify which thread contains the next instruction in program order.

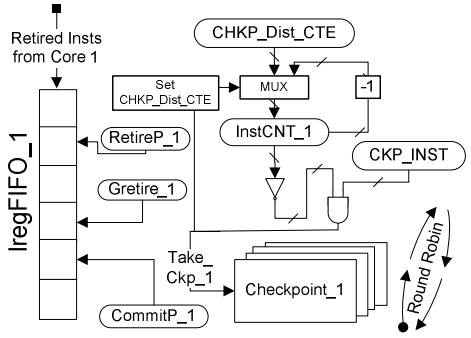

Figure 8. Register checkpointing mechanism.

- A set of pointers per *lregFIFO*: (1) a *RetireP* pointer to the first unused entry of the *lregFIFO*, where new retired instructions allocate an entry; (2) a *CommitP* pointer to the oldest allocated entry in the *lregFIFO* which is used to deallocate the *lregFIFO* entries in order; (3) a *Gretire* pointer to the last entry in the *lregFIFO* we visited in order to build a complete checkpoint.
- A pool of checkpoint tables per *lregFIFO*. The number of these tables defines the maximum number of checkpoints we can have in-flight. Each pool of checkpoints works as a FIFO queue where checkpoints are allocated and reclaimed in order.

Every time a core retires an instruction that produces a new architectural register value, a new entry is allocated in the corresponding *lregFIFO*. Then, it reads the entry in the active checkpoint for the logical register it overwrites. In case the *O* bit is set, the *PDest* identifier stored in the entry is reclaimed. Then, the *O* bit is set and the *R* bit unset. Finally, we update the *PDest* field with the physical register allocated by the retired instruction. Once we found the starting point of a new checkpoint, we reset all *O* bits in the new active checkpoint.

When *GRetire* pointer does not match *RetireP*, we do not have to take any action, because we are the youngest instruction in the system. Otherwise, if the *GRetire* pointer matches the *RetireP* pointer means that this instruction is not the youngest instruction and so we are performing complete checkpoints. For that, we check the *POP* bit and in case it points to other core C, we use the *GRetire* pointer of C to walk over its *lregFIFO* until we find an entry with a *POP* pointer pointing to us again. For every entry we visit, we read the *LDest* value and update our active checkpoint: in case the *O* bit is set, we reclaim the physical register identifier written in *PDest*. Then, in any case we reset the *O* bit, set the *R* bit and update the *PDest*. When an entry with the ckp bit set to 1 is visited, the partial checkpoint with the information of our active checkpoint is completed. This merging involves reclaiming all *PDest* which in the partial checkpoint has the *O* bit set and the *R* bit in our active checkpoint is reset. Then, we update our active checkpoint resetting the *O* bit of these entries.

4.4 Hardware Complexity

We have used CACTI 5.3 [23] to measure the area required by our proposal in a contemporary CMP. For this study we have considered a processor with only one tile based on the parameters shown in Table 1 (*Medium* core), which is similar to an Intel® Core™ 2 Duo processor. On 65 nm technology, the studied base processor has an area of 143 mm^2. We have measured that our proposal, the *ICMC* and the extensions to L1 and L2 caches, increases the area up to 153.4 mm^2 (7.2% increase).

5. EXPERIMENTAL EVALUATION

5.1 Framework

The *Anaphase* speculative thread decomposition scheme has been implemented on top of the Intel® production compiler (*icc*). For evaluating the proposed technique, we have selected the SPEC2006 benchmark suite compiled with *icc –O3*. In total 12 SpecFp and 12 SpecInt benchmarks have been optimized with *Anaphase* using the train input set for profiling information.

Representative loops of the train execution have been selected to be speculative parallelized with *Anaphase* based on 20M instruction traces generated with the PinPoint tool. In these traces, those outer loops that cover more than 500K dynamic instructions have been selected for thread decomposition.

For each of the selected loops, the *Anaphase* thread decomposition technique generates a partition for different replication thresholds, as described in section 3. Once all the partitions are generated, the *Anaphase* scheme selects the best partition for each loop and generates the optimized binary.

The performance of the optimized benchmarks has been evaluated through a detailed cycle accurate simulator using the ref input set. The simulator models the x86 CMP architecture described in section 4. For this evaluation only one tile has been used. Two cores have been considered to represent different CMP scenarios. One out-of-order core, called *Medium*, that is similar to current CMP cores [20], and another called *Tiny*, where main structures have been reduced to half the size, which represents a likely core on a many-core environment.

The main parameters that we have considered are shown in Table 1. The shadowed fields on the table are per core. Regarding the *ICMC* structures, our studies show that 1024 entries per *mem/lreg*

Table 1. Architecture configuration parameters.

Parameter	*Tiny* Core	*Medium* Core
Fetch, Retire and Issue width	2	4
Pipeline stages	7	7
Branch Predictor GShare history bits/table entries	12/4096	12/4096
Sizes ROB/Issue Queue/MOB	48/16/24	96/32/48
Miss Status Holding Registers per core	8	16
L1 and ICache size/ ways/ line size/ latency	16KB/4/32B/2	32KB/4/32B/2
L2 size/ ways/ line size/ latency (round trip)	4MB/8/32B/32	4MB/8/32B/32
Memory latency	328	328
Explicit communication penalty	32	32
Overhead spawn / commit (cycles)	64 / 64	64 / 64
Mem Fifos / Ireg Fifos Size	512	1024
UDT Size	32	64

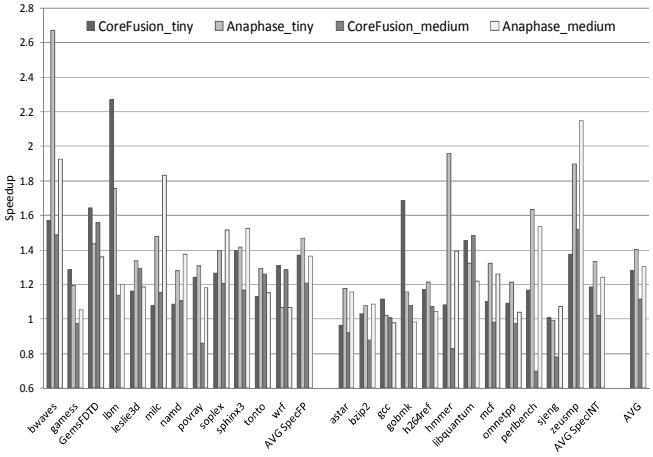

Figure 9. *Anaphase* performance compared to Core Fusion.

FIFO and 64 entries in the *UDT* for the *Medium* core gives us almost the maximum performance we can get with the threading scheme we are evaluating.

In order to perform the performance simulations, for each studied benchmark, we have randomly selected traces of 100M instructions of the ref execution beginning with the head of each of its optimized loops. Results for the benchmark are then reported as the addition of all the simulated traces. On average, about 10 traces have been generated per benchmark. We have measured that on average the optimized loops found in these traces cover more than 90% of the whole ref execution.

5.2 Results

Figure 9 shows the performance of the *Anaphase* scheme. We have compared the execution of *Anaphase* with another scheme to boost single thread performance on CMPs based on dynamic non-speculative fine-grain thread decomposition. This scheme closely models the Core Fusion proposal [11]. Performance for both schemes is reported as speedup over execution on a single core. For this comparison, both *Tiny* and *Medium* core configurations have been evaluated.

As can be seen, *Anaphase* clearly outperforms the Core Fusion scheme on both core configurations. An average speedup of 41% and 31% is observed for the *Tiny* and *Medium* core respectively, whereas Core Fusion achieves 28% and 12%. Note that on a CMP with small cores, where the amount of ILP each core is able to exploit is limited by resources, Core Fusion performs significantly better than on a CMP with bigger cores. On the other hand, *Anaphase* performs well on both CM P environments since it better exploits different sources of parallelism: ILP, TLP, and MLP. Regarding TLP, we have measured that for these benchmarks *Anaphase* is able to speculate on a window comprised of several thousands and even a few millions of instructions. This is measured as the distance between the youngest instruction and the most senior instruction (not yet globally retired) in the system. Regarding MLP, we have observed that exploiting this parallelism is key for some benchmarks like bwaves, lbm, milc, hmmer, and zeusmp. As can be seen in Figure 9, *Anaphase* significantly benefits from exploiting more MLP thanks to its fine-grain decomposition and the delinquent load heuristic. On the other hand, differences between the train and ref input sets affect the profiling information used by *Anaphase* and

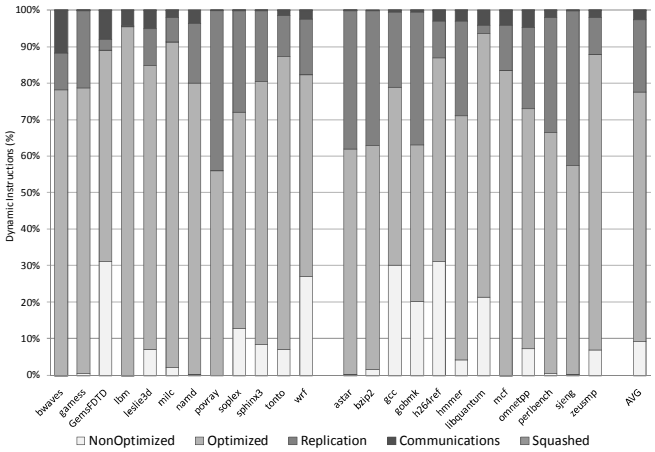

Figure 10. *Anaphase* dynamic instruction breakdown.

may cause that available MLP remains to be exploited. This effect can be observed in the lbm benchmark, where Core Fusion decomposition ends up in a better distribution of load misses.

Other features that may affect *Anaphase* performance compared to Core Fusion are inter-thread communications and the amount of replication due to control instructions. Regarding communications, although we have observed that for some benchmarks inter-thread communication latency is up to a few thousand cycles on average, the *Anaphase* decomposition scheme is able to adapt to it. On the other hand, we have verified that for some benchmarks, like libquantum, Core Fusion benefits a lot from having a dedicated communication mechanism with only 2 cycles latency. In addition, Core Fusion requires a tight synchronization in the front-end in order to steer instructions among cores in sequential order. *Anaphase* allows asynchronous execution of each of the threads and thus no modifications on the front-end at the expenses of having to replicate some control instructions. We have observed that the impact on performance of this control replication is about 5% on average, and for some benchmarks like gcc and gobmk this is the cause why *Anaphase* performs worse than Core Fusion.

Finally, in some benchmarks where Core Fusion performs better than *Anaphase*, like gemsFDTD, wrf, gcc, gobmk, h264ref, and libquantum, *Anaphase* suffers from low coverage of the optimized regions.

Figure 10 shows the breakdown of dynamic executed instructions for the different benchmarks optimized with *Anaphase*. As expected, the number of instructions in optimized regions is very high in almost all the benchmarks. However, as previously pointed out, for some benchmarks like gemsFDTD, wrf, gcc, gobmk, h264ref, and libquantum, the non optimized code represents more than 20% of the dynamic execution. This low coverage is mainly caused by inaccuracies of the profiling information to select the loops to be optimized.

As can be seen in the Figure 10, the amount of additional instructions needed to solve inter-thread dependences (replicated instructions and explicit communications) represent just about 22% of the dynamic instructions on average. However, for some benchmarks like povray, astar, bzip2, gobmk, and sjeng, replication represents up to 40% of dynamic instructions. Although, this large amount of replication code does not imply a slowdown in performance, it may imply an increase in energy. One important thing to notice is that in *Anaphase* p-slices are conservative and do not include any speculative optimization. Previous work has shown that through speculative optimizations p-slices can be significantly reduced with little impact on accuracy [8]. On the other hand, explicit communications, only account for 3% of additional instructions on average. This short amount of extra instructions has proven to be a very effective technique to handle inter-thread dependences. We have verified that with a scheme that do not allow explicit communications, the performance speedup of *Anaphase* for the studied benchmarks drops until 20% on average.

Thanks to the checkpointing support, the *Anaphase* scheme is also very effective on reducing the amount of work that is wasted due to squashed regions. For all benchmarks the number of instructions wasted due to squashes represents less than 1%. It is worth to point out that on average 25% of the optimized regions turn out to be squashed due to memory misspeculations or evictions of speculative lines on the MLC. For some benchmarks, like bwaves and lbm, the percentage of squashed regions is close to 95%. We have measured that on average more than 50% of the squashes are due to evictions of speculative lines on the MLC. This strengthens the importance of having a fine-grain checkpointing mechanism as the one proposed for *Anaphase*. Our evaluations show that in order to take a checkpoint every 200 original instructions on average, less than 8 live checkpoints are required for our system.

6. FUTURE WORK

Our future work can be decomposed into three big areas. First, we would like to improve the overall coverage of optimized regions. Increasing coverage is a simple mechanism to increase performance further. On a second set of future actions, we would like to study the implications of implementing *Anaphase* on a dynamic optimizer, and the interactions with the OS. Finally we will explore how *Anaphase* scales when fine-grain speculative multithreading is applied to more than two threads.

7. CONCLUSIONS

In this paper we have presented *Anaphase* fine-grain threading-based approach to exploit multiple cores to boost single-thread performance. The proposed technique features a cost-effective hardware support for the execution of *Anaphase* threads generated at compile time. In particular, a novel hardware component named *Inter-Core Memory Coherency Module* is proposed which updates the memory state in program order, detects memory violations, and implements checkpointing and recovery mechanisms.

Results reported in this paper strongly validate the effectiveness of *Anaphase* for boosting single-thread performance, resulting from exploiting ILP, TLP and MLP, with a high accuracy, and low overheads. In addition, results shown in this paper demonstrate that *Anaphase* fine-grain decomposition of applications into speculative threads provides an appropriate mechanism for combining the execution of cores and improving single-thread performance in a multi-core design. In particular, when *Anaphase* is used on a CMP with tiny cores single-thread performance of Spec2006 applications is improved by 41% on average, and up to 2.6x for some selected applications.

Finally, we have shown that *Anaphase* outperforms previous hardware-only schemes to implement Core Fusion by more than 10% on average on Spec2006 for all configurations.

8. ACKNOWLEDGMENTS

This work has been partially supported by the Spanish Ministry of Science and Innovation under contract TIN 2007-61763 and the Generalitat de Catalunya under grant 2005SGR00950. We thank the reviewers for their helpful and constructive comments.

9. REFERENCES

[1] H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor", in Proc. of the 31st Int. Symp. on Microarchitecture, 1998

[2] S. Balakrishnan, G. Sohi, "Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs", in Proc. of the Int. Symp. on Computer Architecture, pp. 302-313, 2006

[3] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing", in Proc. of the 27th Int. Symp. on Computer Architecture, pp. 282-293, June 2000

[4] R. Canal, J.-M. Parcerisa, and A. Gonzalez, "A Cost-effective Clustered Architecture". in Int. Conf. on Parallel Architectures and Compilation Techniques, pp 160–168, Newport Beach, CA, October 1999

[5] M. Cintra, J.F. Martinez and J. Torrellas, "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems", in Proc. of the 27th Int. Symp. on Computer Architecture, 2000

[6] J. D. Collins and D. M. Tullsen, "Clustered Multithreaded Architectures - Pursuing Both Ipc and Cycle Time", in Int. Parallel and Distributed Processing Symp., April 2004

[7] J. D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y-F. Lee, D. Lavery and J.P. Shen,"Speculative Precomputation: Long Range Prefetching of Delinquent Loads", in Proc. of the 28th Int. Symp. on Computer Architecture, 2001

[8] C. García, C. Madriles, J. Sánchez, P. Marcuello, A. González, D. Tullsen, "Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices", in Procs. of the Conf. on Programming Language Design and Implementation, 2005

[9] S. Gopal, T.N. Vijaykumar, J.E. Smith and G.S. Sohi, "Speculative Versioning Cache", in Proc. of the 4th Int. Symp. on High Performance Computer Architecture, 1998

[10] L. Hammond, M. Willey and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor", in Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 1998

[11] E. Ipek, M. Kirman, N. Kirman, and J.F. Martinez, "Core fusion: Accommodating Software Diversity in Chip Multiprocessors", in Proc. of the Int. Symp. on Computer Architecture, 2007

[12] T. Johnson, R. Eigenmann, and T. Vijaykumar, "Min-Cut Program Decomposition for Thread-Level Speculation", in Procs. of Conf. on Programming Language Design and Implementation, 2004

[13] J. A. Kahle , M. N. Day , H. P. Hofstee , C. R. Johns , T. R. Maeurer , and D. Shippy, "Introduction to the Cell Multiprocessor", IBM Journal of Research and Development, v.49 n.4/5, p.589-604, July 2005

[14] G. Karypis, and V. Kumar, "Analysis of Multilevel Graph Partitioning", in Procs. of the 7th Supercomputing, 1995

[15] B. Kernighan, and S. Lin, "An Efficient Heuristic Procedure for Partitioning of Electrical Circuits", in Bell System Technical Journal, 1970

[16] V. Krishnan and J. Torrellas, "Hardware and Software Support for Speculative Execution of Sequential binaries on a Chip-Multiprocessor", in Int. Conf. on Supercomputing, pp. 85-92, 1998

[17] F. Latorre, J. Gonzalez, and A. Gonzalez, "Back-end Assignment Schemes for Clustered Multithreaded Processors", in Intl. Conf. on Supercomputing, pp 316–325, Malo, France, June–July 2004

[18] P. Marcuello, and A. González, "Thread-Spawning Schemes for Speculative Multithreaded Architectures", in Procs. of the Symp. on High Performance Computer Architectures, 2002

[19] J.F. Martinez, J. Renau, M.C. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Recycling in Out-of-order Microprocessors", in Procs. of the Int. Symp. on Microarchitecture, November 2002

[20] A. Mendelson, J, Mandelblat, S. Gochman, A. Shemer, R. Chabukswar, E. Niemeyer, A. Kumar, "CMP Implementation in Systems Based on the Intel® Core™ Duo Processor", in Intel Technology Journal, Volume 10, Issue 2, 2006

[21] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita, "Pinot: Speculative Muti-threading Processor Architecture Exploiting Parallelism over a wide Range of Granularities", in Proc. of the 38th Int. Symp. on Microarchitecture, 2005

[22] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas, "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization", in Proc. of the 28th Int. Symp. on Computer Architecture, 2001

[23] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. P. Jouppi, "CACTI 5.1", Technical Report HPL-2008-20, HP Labs.

[24] N. Vachharajani, R. Rangan, E. Raman, M. Bridges, G. Ottoni, and D. August, "Speculative Decoupled Software Pipelining", in Procs. of the Conference on Parallel Architecture and Compilation Techniques, pp. 49-59, 2007

[25] C.B. Zilles and G.S. Sohi, "Execution-Based Prediction Using Speculative Slices", in Proc. of the 28th Int. Symp. on Computer Architecture, 2001

[26] C.B. Zilles and G.S. Sohi, "Master/Slave Speculative Parallelization", in Proc. of the 35th Int. Symp. on Microarchitecture, 2002

[27] H. Zhong, S. A. Lieberman, and S. A. Mahlke, "Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications". In Int. Symp. on High-Performance Computer Architecture, 2007