LAR-CC: Large Atomic Regions with Conditional Commits

Edson Borin
Institute of Computing
University of Campinas[1]

Youfeng Wu
Programming Systems Lab.
Intel Labs

Mauricio Breternitz Jr.
Advanced Software and Analytics
Technology Group - AMD[1]

Cheng Wang
Programming Systems Lab.
Intel Labs

***Abstract*—HW/SW Co-designed systems rely on dynamic binary translation and optimizations for efficient execution of binary code. Due to memory ordering properties and other architectural constraints, most binary optimizations are applied to regions of code that are atomically executed. To ensure that the underlying hardware has enough speculative resources to execute the whole atomic region, these systems typically form short atomic regions, with only 20 to 30 instructions. However, the shorter is the atomic region the smaller is the scope for optimizations. We present LAR-CC, a novel technique that enables HW/SW co-designed systems to optimize large atomic regions and dynamically fit them into the available speculative hardware resources by means of conditional commits. The LAR-CC technique consists of two major components: 1) conditional branch instructions to conditionally skip commit operations; 2) code transformations that replace commit operations by conditional commits and enable optimizations to be applied on the large atomic regions. Our experiments show that LAR-CC can effectively achieve dynamic atomic region sizes larger than 1000 instructions, providing sufficiently large scope to apply many advanced optimizations on HW/SW co-designed systems.**

I. Introduction

HW/SW co-design is a promising approach to design power efficient processors [1], [3], [11], [12], [18], [19], [20]. It efficiently supports a source ISA, e.g., the popular X86, via a runtime dynamic translation software that translates and optimizes the source binary for execution on the native implementation ISA. This not only allows the processor designers to design innovative new microarchitectures without concern for backward compatibility, but also leverages powerful runtime software, guided by dynamic runtime information, to optimize the programs for significant performance improvement.

This design approach usually relies on specialized hardware support to enable efficient execution of the translated code. Self-modifying code detection [7], memory alias detection [11], indirect branch execution [10] and atomic execution [12] are examples of mechanisms that rely on hardware support to enable the efficient execution of translated code. Atomic execution, in particular, is an important mechanism that allows the translator to perform aggressive speculative optimizations, such as dead store elimination and scheduling, without concern with infrequent corner cases, like exceptions or memory ordering issues. The optimized code is speculatively executed inside an atomic region and, if any corner cases happens, the system safely rolls back, discarding the speculative computation, and executes a more conservative version of the code (e.g., via interpretation).

Atomic execution typically employs hardware support to buffer the data produced by the speculative execution until the execution reaches the end of the region, when the speculative state is committed and converted into architectural state. In this way, the amount of code that can be executed atomically is a function of the size of the speculative buffers and the translator must be careful to ensure that the region of code optimized is small enough to fit into the speculative buffers. However, as we discuss below, building small regions of code may reduce the optimizations scope, and consequently the optimization opportunities. Therefore, the translator must build regions as large as possible, to increase the optimization opportunities, but small enough to fit into the speculative buffers.

Selecting the right atomic region size is a non-trivial task because it is hard to predict the amount of speculative resources necessary to buffer the speculative data produced by a given region. Notice that, the region code may contain a complex control flow structure, including if-then-else statements and loops, making it hard to predict the amount of speculative data that may be produced during the execution. Moreover, speculative buffers like speculative caches allow the re-use of buffer entries by speculative data (e.g., multiple speculative stores may write to the same cache line), making it hard to predict how many entries will be required to hold the speculative data. As we discuss in Section IV, even a state-of-the-art HW/SW co-designed system, such as the Transmeta Efficeon [12], employs conservative heuristics to limit the size of the atomic regions, ensuring that the underlying hardware can buffer the speculative data produced by the regions.

We propose a novel technique that allows the translator to form and optimize large atomic regions by means of conditional commits and dynamically expand the atomic region size to fit into the available speculative hardware resources. As we show in the experimental results, the Large Atomic Region with Conditional Commit, or LAR-CC, effectively achieve atomic region sizes larger than 1000 instructions.

The contributions of the paper can be summarized as follows.

- We propose LAR-CC, a novel technique which enables the optimization of large atomic regions of code and dynamically selects the atomic region size to fit into the speculative buffers.

[1]Work done while at the Programming Systems Lab. - Intel Labs.

- We propose a simple and effective heuristic to predict when the conditional commits should commit the speculative state.
- We demonstrate the potential of the LAR-CC technique by implementing and evaluating it on a state-of-the-art HW/SW co-designed infrastructure that models the Transmeta Efficeon processor [12]. We show that the dynamic atomic region size can be improved by 3X on average, and achieves more than 1000 instructions for loops that cover half of the loops execution.

The paper is organized as follows. Section II discusses the code optimization scope on HW/SW co-designed systems and the atomic region size limitation imposed by the atomic execution hardware support. Section III describes the LAR-CC technique and how it increases atomic regions sizes. Section IV presents the experimental infrastructure and shows the experimental results. Section V discusses the related work and Section VI concludes the paper.

II. Code optimization scope on HW/SW Co-designed Systems

The key component of a high performing HW/SW co-designed system is the binary translation module. It translates the source binary to the implementation ISA, and optimizes the code for the specific microarchitecture. The binary translation usually operates in multiple phases, or gears, as described by Dehnert *et al.* [7] and Borin *et al.* [3]. In the first phase, the source binary is interpreted or quickly translated with simple or no optimizations for native execution with low compilation overhead. Most of the cold code is rarely executed and will go only through the first phase of execution. Once a region of code is identified as hot, or frequently executed, the subsequent phases are initiated and the hot code is re-translated. In these phases, the binary optimizer (BO) performs more extensive optimizations on the region in the hope that the frequent execution of the optimized code will pay off the optimizations overhead and improve the overall performance. As we discuss below, larger regions of code unveil more optimization opportunities and can potentially improve program performance significantly. Consequently, the binary translation software may try to form larger regions of code to enable more optimization opportunities and further performance improvements.

A region can be loosely defined as a collection of instructions from the source ISA. Depending on the system, the set of instructions in the same region may form a linear trace [1], [18], [19], [20], a super-block, a hyperblock, a directed acyclic control-flow graph (CFG) or even a generic CFG, with loops and function calls [3], [7]. Typically, the region defines the scope in which a HW/SW co-designed system apply optimizations. In this sense, the larger the region is the bigger the scope and the more optimization opportunities exist.

In order to overcome the constraints imposed by the source ISA architectural features, like the memory ordering model and precise exception requirement, the HW/SW co-designed systems usually rely on atomic execution support to safely execute aggressively optimized regions of code. In this model, the BO extensively optimizes the regions under some assumptions and, if any of the assumptions are violated during the execution of the optimized code, the system discards the computed data and re-executes the region, using a less aggressive version of the code or even interpretation. If none of the assumptions are violated, the changes are committed to the architectural state and the execution proceeds. Figure 1 shows an example, where a simple region of x86 code is translated and aggressively optimized under some assumptions. In this example, the optimizer removed the first instruction of the translated code because the third instruction performs a store on the same address, which overwrites the data stored by the first instruction. One of the assumptions here is that no exceptions will occur between the execution of the first and the third instructions. This assumption is important because the x86 architecture implements precise exceptions and, in case the second instruction (a load) raises an exception, the data stored by the first instruction is required to be present in the memory. In case the assumption is violated, i.e., when an exception happens on the second instruction, the atomic execution support allows the system to safely discard the data computed by the region and re-execute a more conservative version of the code.

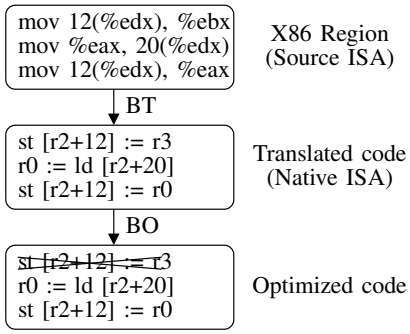

Fig. 1. Region of x86 code translated and optimized.

In many HW/SW co-designed systems, the whole region is executed atomically [1], [3], [18], [19], [20]. These systems perform checkpoints and store the computed data on speculative buffers until they are committed at the end of the region execution or discarded at a rollback operation (e.g., due to an exception). On the one hand, this approach simplifies the optimizer, enabling it to optimize the whole region without concern with memory order and exceptions, on the other hand, it limits the maximum size of the region to the size of the speculative buffers, since the whole region execution must be stored in speculative buffers until the execution commits on the region exit.

Some HW/SW co-designed systems, like Transmeta Efficeon [12] and Crusoe [11], use a more flexible approach, which builds large regions of code and allows the execution to commit from time to time, before leaving the region, allowing the whole region execution to be larger than the size of the

speculative buffers. Figure 2 shows an example of a large region that commits from time to time to avoid running out of speculative resources. This region commits the speculative state at every loop iteration to avoid executing too many instructions between commits. Notice that the static size of the region is small, just a few static instructions, but the dynamic size may be very large, since the loop at block B2 may execute thousands or even millions of iterations.

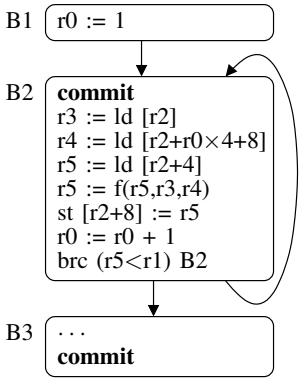

Fig. 2. Large region with multiple commits to avoid running out of speculative resources.

Since the region at Figure 2 performs multiple commits, it is composed of multiple atomic segments. Block B1 is executed atomically until the commit at block B2. Block B2 is executed atomically until the commit at B2, and blocks B2 and B3 may also be executed atomically until the commit at B3. The issue with this approach is that the commit at the loop header requires the architectural state to be precise at that point, which may inhibit some optimizations across loop iterations. Figure 3 shows an example where the code of Figure 2 is incorrectly optimized across the commit. The optimizer performed loop invariant code motion (LICM) by moving two load instructions from the loop body (B2) to the pre-header (B1) and partial dead store elimination (PDSE) by moving the store operation from the loop body (B2) to block B3. Notice that the store on address r2+8 is not executed inside the loop anymore and, after the first iteration, the commit at the block B2 may not generate a precise architectural state. This would violate the very basic assumption that the architectural state is precise at every commit. Another problem is the memory ordering model. Notice that the LICM optimization modified the order in which the loads are executed and, eventual modifications on these memory locations by other cores may cause the execution to be inconsistent with the architectural memory model.

One could argue that the loop at block B2 could be unrolled to increase the size of the atomic region, but this approach still does not enable optimizations across loop iterations (like software pipelining) and the costs associated with code expansion, such as increased pressure on the instruction cache and increased optimization overhead, may hinder it's benefits. Furthermore, the unroll has to be conservative to make sure the execution of the resulting loop body won't run out of speculative resources. In the next section we introduce the conditional commit mechanism and show how it can aggressively enlarge atomic regions and allow us to safely optimize loops across iterations while keeping the atomic execution within the limit of the speculative buffers.

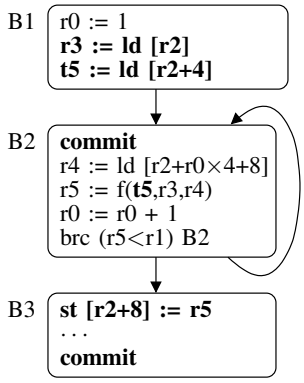

Fig. 3. Invalid optimizations across atomicity boundaries.

III. Large Atomic Regions with Conditional Commits

The large atomic regions with conditional commit (LARCC) technique consists of two major components: 1) conditional branch instructions to conditionally skip commit operations; 2) code transformations that replace commit operations by conditional commits and enable optimizations to be applied on larger atomic regions.

The first conditional branch instruction is called *Branch To Skip*, or BTS for short, and performs in a similar fashion as a regular conditional branch instruction. The BTS takes a predicate and a target address as arguments and it branches to the target address if two conditions are met:

1) The predicate is true; and
2) The system has enough speculative resources to continue executing speculative code;

The key idea behind the BTS is that the target address should be taken if the system has spare speculative buffer entries to continue executing speculative computations without performing a commit. Figure 4 shows an example where the region of code is transformed to perform conditional commits. The transformation applied is similar to strip-mining [23], but in this case the commit operation is moved to the outer loop and the test in the inner loop is performed by a BTS instruction, instead of the regular conditional branch instruction (brc). During the execution, the BTS branch may be taken many consecutive times, executing multiple iterations of the inner loop without any commit. Whenever the predicate is false or the system is about to run out of resources, the BTS instruction falls through and the execution may either take the path B2→B2$_{brc}$→B3 (in case the predicate is false), which will commit at B3 and exit the region, or the path B2→B2$_{brc}$→B2$_{cmt}$, which will commit at B2$_{cmt}$ and continue the execution at the inner loop B2. Notice that the resulting

inner loop became a large atomic region because it could execute multiple iterations, potentially up to thousands of instructions, without performing any commit operation.

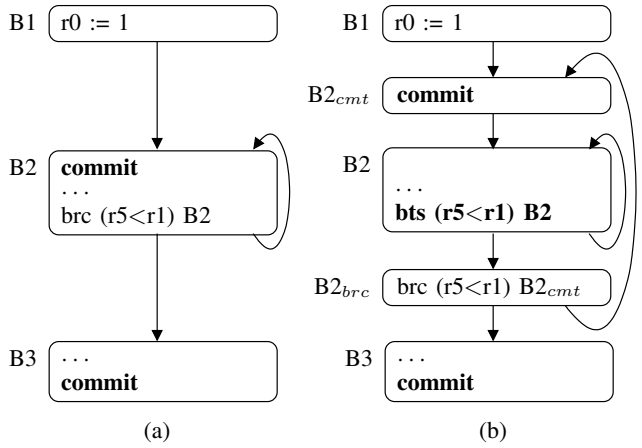

Fig. 4. Region of code before (a) and after (b) the conditional commit transformation.

The BTS instruction must be able to predict if the system will have enough speculative resources to skip the commit and execute the next loop iteration successfully. If the prediction is incorrect, the system may run out of speculative resources. In this case, the system can still safely proceed by rolling the state back to the last commit point executed and re-executing the region using a more conservative code, e.g., using interpretation or even the same code with a more conservative skip prediction mechanism. Even though the system is able to continue, this scenario may be very costly because the rollback operation may discard a large amount of speculative work, increasing the energy consumption and affecting the performance. Therefore, it is imperative that the skip prediction mechanism be intelligent enough to avoid running out of speculative resources. In Section IV we propose and evaluate an effective skip prediction mechanism.

A. Conditional Commit Transformations

We propose two code transformations to take advantage of conditional commits. The first one is specific for loops and, as we showed in Figure 4, is able to remove commit operations from the loop body, enabling optimizations across loop iterations. The second one is more generic and can be used on any commit operation.

Algorithm 1 shows the conditional commit transformation for loops with commit operations on the loop header. Lines 1 to 3 create the outer loop header block and transfer the commit operation to it. Lines 4 to 7 move the original loop's input edges to the outer loop. The remaining of the algorithm processes the back edges source blocks, called loop tails. In this step, each loop tail is split into two blocks, $LTAIL$ and $LTAIL_{brc}$ (e.g., B2 and B2$_{brc}$ on Figure 4). The $LTAIL$ will contain a BTS branch, enabling the code to directly branch to the inner loop header and skip the commit operation. The $LTAIL_{brc}$ will contain the regular conditional branch allowing the execution to either leave the loop by falling through or continue by branching back to the outer loop header and committing.

Algorithm 1: Conditional Commit Transformation for Loops

Input: Loop Header Block (LH), $Back\ Edges\ Set$
Output: Transformed Loop

1 Create the outer loop header block: LH_{cmt}
2 Create an edge from LH_{cmt} to LH
3 Move the commit operation from LH to LH_{cmt}
4 **foreach** *edge E targeting LH* $\notin$ *Back Edges Set* **do**
5 $\quad PreHeader \leftarrow E.source$
6 $\quad$ Remove E
7 $\quad$ Create a new edge ($PreHeader$, LH_{cmt})
8 **foreach** *edge E* $\in$ *Back Edges Set* **do**
9 $\quad$ Let $LTAIL$ be the block $E.source$, and $BRC(p)$ be the branch at $LTAIL$ with predicate p
10 $\quad$ Create a new block $LTAIL_{brc}$
11 $\quad$ Copy $BRC(p)$ from $LTAIL$ to $LTAIL_{brc}$
12 $\quad$ Change the $BRC(p)$ at $LTAIL$ into a $BTS(p)$
13 $\quad$ Create an edge from $LTAIL_{brc}$ to $LTAIL$
14 $\quad$ Let FT_E be the fall-through edge of $LTAIL$
15 $\quad$ Change $FT\_E.source$ to $LTAIL_{brc}$
16 $\quad$ Create a back edge from $LTAIL_{brc}$ to LH_{cmt}

For simplicity, the presented algorithm assumes that the back edges are the branch taken paths. For back edges that correspond to fall-through paths, we may either transform them by inverting the predicate of the branch or by using the second conditional commit instruction: the *Branch To Commit* (BTC). The BTC instruction is similar to the BTS, but in this case, it branches if the system does not have enough speculative resources to continue executing speculative code. Figure 5 (a) shows the region of Figure 2 after the conditional commit transformation and Figure 5 (b) shows the resulting code after LICM and PDSE optimizations. Notice that, in this case, the optimizer was able to move code out of the inner loop without crossing any atomicity boundaries.

The generic conditional commit transformation can be applied to other commit operations and is depicted in Figure 6. In this transformation, a basic block B1 containing a commit operation is first split in two blocks: B1' and B1" as shown in Figure 6 (b). B1' contains all the instructions before the commit and the commit operation and B1" contains all the instructions after the commit. Finally, a new block (B1$_{cmt}$) with a commit operation is created and the original commit, at B1', is replaced by a Branch To Commit (BTC) operation as shown in Figure 6 (c). The new code will fall through and skip the execution of B1$_{cmt}$ if the system has enough speculative resources to continue the speculative execution, otherwise, the target of the BTC instruction will be taken and the speculative state will be committed at B1$_{cmt}$. Since the

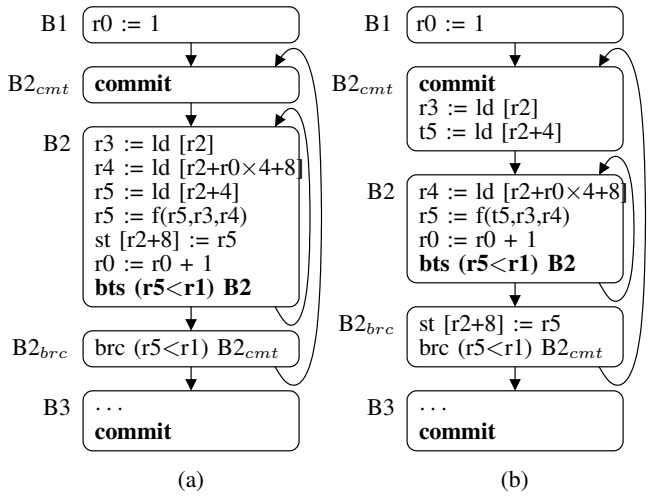

Fig. 5. Region after the conditional commit transformation (a) and after LICM and PDSE optimizations (b).

transformation is straightforward, we omit the pseudo-code for this transformation.

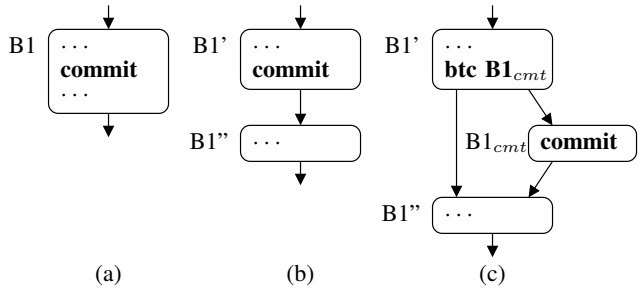

Fig. 6. Generic conditional commit transformation. (a) Original block; (b) Block split at commit; (c) Commit replaced by conditional commit (BTC).

Figure 7 shows an example where the generic conditional commit transformation enables new optimizations. In this example, the store performed at block B1 is killed by the store at B3. However, the store at B1 cannot be removed because it is alive at the commit in B2, which requires the architectural state to be precise. The store at B1 becomes partially dead after the conditional commit transformation (it is dead at path B1→B2→B3), and can be optimized as in Figure 7 (c).

Next section presents the infrastructure used, the skip prediction mechanism and the experimental results.

IV. Experimental Results

In this section we describe the experimental infrastructure and the results achieved with the LAR-CC technique. We first introduce the experimental framework and show the potentials for atomic region execution using an perfect skip predictor. Finally, we propose and evaluate a heuristic to predict conditional commits.

We implemented the LAR-CC technique on top of an infrastructure that models the Transmeta Efficeon [12], a state-of-the-art HW/SW co-designed system that executes x86 code

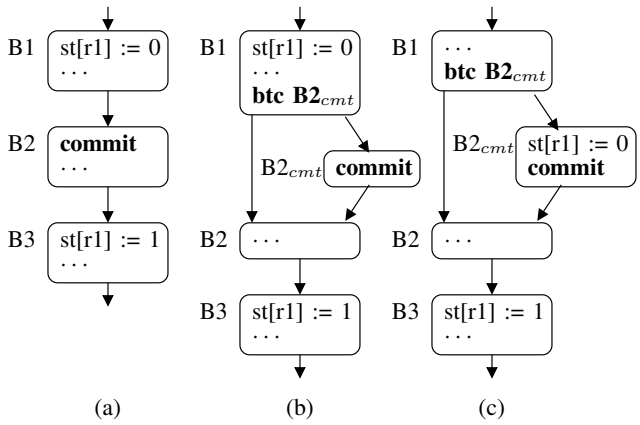

Fig. 7. (a) Original code; (b) conditional commit transformation; (c) code after partial dead store elimination.

on top of a VLIW instruction set via software emulation. The Efficeon hardware consists of an in-order VLIW processor capable of executing up to seven operations per cycle. The hardware also contains special support to accelerate the execution of dynamically generated code. The special hardware support includes mechanisms for efficient atomic execution, like fast register checkpoints and memory buffering via a shadow register file and a speculative cache, and primitives for dynamic memory alias detection, which enables aggressive optimizations on memory operations, like dead store elimination and instruction scheduling.

The Efficeon software, called Transmeta Code Morphing System [7], [16], or CMS for short, is a runtime system that executes on top of the Efficeon processor and is able to provide high-performance execution of x86 binaries via interpretation and dynamic binary translation. Whenever the system is started, CMS is loaded and initialized. After this, the CMS interpreter starts executing and profiling x86 instructions from the x86 boot sequence. If an x86 instruction is frequently executed, the CMS compiler (a.k.a. translator) selects a region of x86 code, starting at the frequently executed instruction, translates the x86 code and optimize it for native execution on the Efficeon processor. The translated code is executed until the control flow reaches x86 code that has not been translated yet. In this case, the CMS interpreter takes over and the execution via interpretation continues until a new x86 instruction becomes frequently executed and a new region is translated, or until the execution reaches x86 code that is already translated, in which case the execution is switched back to translated code. The CMS also monitors the execution of translated code and re-translates the code with more aggressive optimizations if the code turns out to be very hot. In fact, the CMS employs a four gears (stages) execution mechanism [12]: in the first gear the cold code is executed via interpretation. Whenever the code becomes frequently executed, the second gear takes place and the CMS translates and optimizes the code using light optimizations to minimize compilation time. Gear 3 starts when a translated code becomes very hot. In this

case, the CMS compiler re-compiles the same region of code applying aggressive optimizations. Gear 4 starts when the code becomes even hotter. This last gear is characterized by even more aggressive optimizations and the eventual combination of multiple regions into a single larger region, increasing the optimization scope. We refer the reader to previous work [7], [11], [12], [16] for details on the CMS translator and the gearing system.

The Efficeon hardware includes a shadow register file that is used for creating register checkpoints quickly. Whenever a commit operation is executed, the speculative register state, stored at the regular register file, is copied into the shadow register file. If a rollback operation is executed, the state is copied from the shadow register file back to the regular register file, recovering the state saved at the last commit operation. Different from the registers approach, the memory contents are not entirely saved at commits. Instead, the results of memory operations are tagged as speculative and buffered at the speculative cache during execution. When a commit operation is executed all the speculative data buffered is turned into non-speculative by a mechanism that efficiently clears all the speculative tags in one cycle. In case a rollback operation is performed, the data tagged as speculative is tagged as invalid and discarded.

The register checkpoint mechanism does not impose any limitations on the number of instructions executed by an atomic region. Notice that a checkpoint is saved at the beginning of an atomic region and it does not grow with the executions of more instructions. The speculative memory buffering mechanism, on the contrary, may run out of resources during the execution. The speculative updates are buffered in a 64 KB, 8-way, 32B line speculative L1 data cache together with a 1 KB, fully-associative, 32B line victim cache. Entries evicted from the data cache are buffered by the victim cache and later drained to the second level cache (L2), however, speculative data is not allowed to be drained from the victim cache into the L2 cache. Therefore, whenever the victim cache becomes full with speculative data, the system must roll back.

The regions selected by the CMS translator are generic CFG, with loops and function calls. The translator converts a region of x86 code into an intermediate representation (IR) and introduces commit operations at selected points to prevent rollbacks caused by lack of speculative resources. For example, the translator inserts commit operations in the middle of regions that contain too many static instructions and at loop headers, ensuring that loops will not execute multiple iterations without committing the speculative state.

The great majority of commits in middle of the translations in our experiments were introduced on loop headers. As discussed before, the commits at the loop headers may prevent the system from running out of resources, but they limit the size of atomic regions and limit optimizations across loop iterations. Therefore, we focus our analysis on conditional commit transformations for loops.

We implemented the LAR-CC loop transformation into the CMS translator and added the BTS and BTC instructions into an Efficeon simulator. We used a pre-release version of CMS (CMS 7.0) and a functional simulator that models the speculative L1 data cache and the victim cache. We simulate 230 traces extracted from 23 SPEC CPU 2000 benchmarks. Each trace is executed in two phases. The first phase executes 1 billion x86 instructions to warm-up the translation code cache and the speculative data cache, while the second phase executes 100 million x86 instructions to collect the statistics.

We evaluate the impact that commit operations have on the size of atomic regions using two metrics:

- *Average atomic loop size*: the number of x86 instructions executed by loops divided by the number of commit operations executed by loops. If the execution commits once at every loop iteration, this metric is equivalent to the average dynamic loop body size.
- *Average dynamic loop size*: the number of x86 instructions executed by loops divided by the number of times the execution entered loops. Assuming the system always commit before entering (or leaving) a loop, this number provides an upper bound for average atomic loop sizes.

The LAR-CC technique enables the execution to skip commits at some of the loop iterations and increases the average atomic loop size. The average dynamic loop size would be equivalent to the average atomic loop size if the execution is able to skip all the commits executed after the loop back edges.

Figure 8 shows the dynamic loop size and the atomic loop sizes with and without commit skips plotted against loop coverage for loops in the Spec 2000 applications. For the skip data we used a perfect skip predictor that is able to precisely predict if the system will run out of resources in the next iteration, always making the correct skip/commit prediction. On average, the loops compiled by CMS cover 52% of the execution and have an average atomic loop size (without skips) of 26.4 x86 instructions. The LAR-CC mechanism is able to bring the average atomic loop size up to 79 x86 instructions, which is very close to the dynamic loop size and 3 times larger than the original size. The curves at Figure 8 show that, even if we select only a few of the largest loops (at around 3% of execution coverage), the average atomic loop size without skips is still limited to 166 instructions. With the same coverage, the conditional commit mechanism is able to achieve an average atomic loop size of 28K instructions, while the average dynamic loop size is above 456K instructions. If we select enough loops to cover 30% of the execution (covering more than half of the hot loops execution), the average atomic loop size with conditional commit is above 1K instructions, significantly better than the original average atomic loop size of 60 instructions. Clearly, the 64KB speculative cache is capable of accommodating large atomic region sizes, exceeding thousands of instructions.

Figure 8 also shows that there are two classes of loops: loops with large trip counts, which can benefit from conditional commit to increase atomic size significantly, and loops with short trip counts, which could execute all the loop iterations without any commit. To achieve the large atomic loop sizes in the first

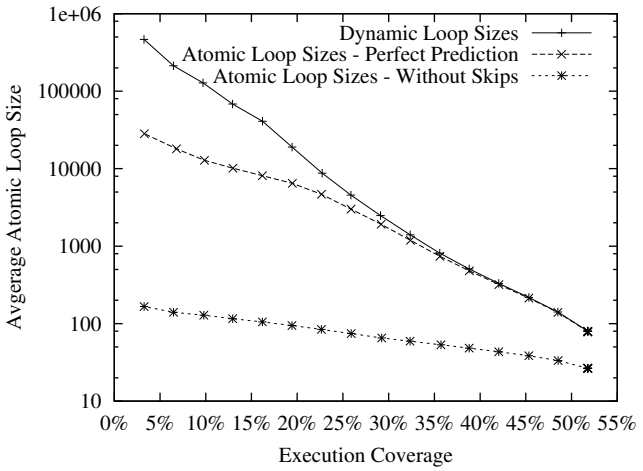

Fig. 8. Dynamic loop size and atomic loop sizes for Spec 2000 benchmarks.

class of loops we need a good skip prediction mechanism to prevent the system from running out of resources, as there is a big gap between the atomic loop size and the dynamic loop size in these loops.

In the next section we propose a skip prediction mechanism and evaluate it using our infrastructure.

A. Skip Prediction Mechanism

As described before, the BTS and BTC instructions must predict if the system will have enough speculative resources to skip the current commit and continue executing until the next commit operation. A good skip prediction technique needs to have the following two pieces of knowledge: 1) how much speculative resources available the hardware still has, and 2) how much speculative resources the execution may need until the next conditional commit instruction.

The Transmeta Efficeon system initially stores the data produced by speculative memory operations on the L1 speculative data cache. Both speculative and non-speculative cache lines evicted from the speculative data cache are moved to the victim cache (VC). Non-speculative cache lines may later be drained from the VC to the L2 cache. However, the system cannot drain speculative data to the L2 cache, therefore, the VC accumulates speculative cache lines evicted from the L1 data cache until the execution commits or rolls back. In our experiments we observed that whenever the data cache becomes saturated with too much speculative data, it starts evicting speculative data to the victim cache. Since the victim cache is not allowed to drain the speculative data to the second level cache, it quickly fills up with speculative data. Intuitively, a high amount of speculative data in the victim cache is a good indication that the system is running out of speculative resources. Based on this insight, we implemented a simple heuristic that monitors the victim cache usage. The heuristic, called *VC Delta*, is executed at every conditional commit branch and computes a growth rate (delta) by subtracting the current amount of speculative entries at the VC by the amount of speculative entries seen when the previous conditional commit instruction was executed. The delta is added to the current VC speculative usage to estimate the VC usage at the next conditional commit instruction. If the estimated VC usage is bigger than a given threshold (16 in our experiments), the current conditional commit branch takes the commit path, otherwise, it takes the skip path.

Figure 9 shows the average atomic loop sizes when using the proposed VC Delta skip predictor and the perfect skip predictor. Notice that, the average atomic loop size achieved by the VC Delta predictor is very similar to the one achieved by the perfect skip predictor.

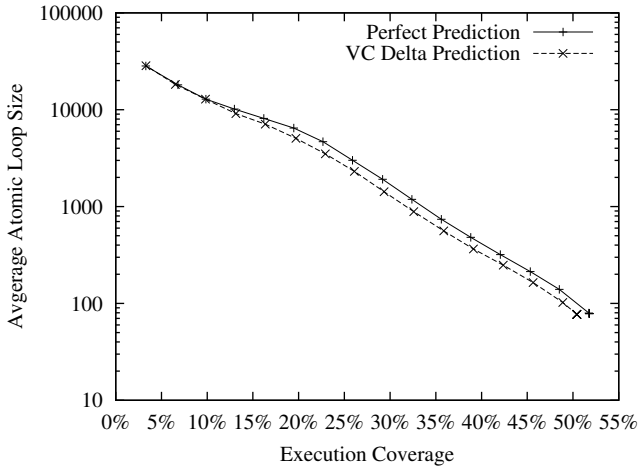

Fig. 9. Average atomic loop sizes with perfect prediction and VC Delta skip prediction.

CMS has a mechanism that re-compiles regions that roll back twice due to lack of speculative resources. To avoid future rollbacks on the same loops, we do not apply the conditional commit to regions that are re-compiled due to aggressive skip prediction. The overall loop execution coverage when using the VC Delta heuristic is slightly smaller than the perfect prediction because the data does not include the loops from regions that were re-compiled without conditional commit. The small reduction of loop coverage on the graphic also indicates that only loops with low coverage were discarded due to re-compilation.

The overhead caused by a rollback is twofold: 1) the cost of rolling the state back to the previous checkpoint, including eventual re-compilations and bookkeeping, and 2) the time/energy spent on the speculative computation being discarded. The amounts of rollbacks measured in our experiments with and without the conditional commit technique are very similar. However, we noticed a larger amount of work discarded on rollbacks when using the conditional commit technique. As expected, the larger the atomic region, the larger the amount of speculative work discarded on eventual rollbacks. In our experiments, most of the rollbacks happened before the atomic region grew large. However, some rollbacks caused by x86 interruptions happened when the executing atomic region

was already very large, causing lots of speculative work to be discarded due to the rollback. To fix this problem, we created a mechanism that delays handling the interruption until the next commit, and forces the conditional commit instructions to take the commit path whenever an interruption is pending. Since the number of instructions between two conditional commits is small, this mechanism won't delay interrupts for long. The mechanism is only used on regions with conditional commits and is triggered by a conditional commit instruction.

Figure 10 shows the amount of speculative work discarded per rollback cause when executing without the conditional commit, with the perfect prediction and with the VC Delta heuristic. Despite the differences, all the techniques discard a very small amount of speculative work. The perfect prediction approach discards the equivalent of 0.0025% of the total work committed during the benchmarks execution, while the VC Delta heuristic discards 0.0054% and the execution without conditional commit discards 0.0078%.

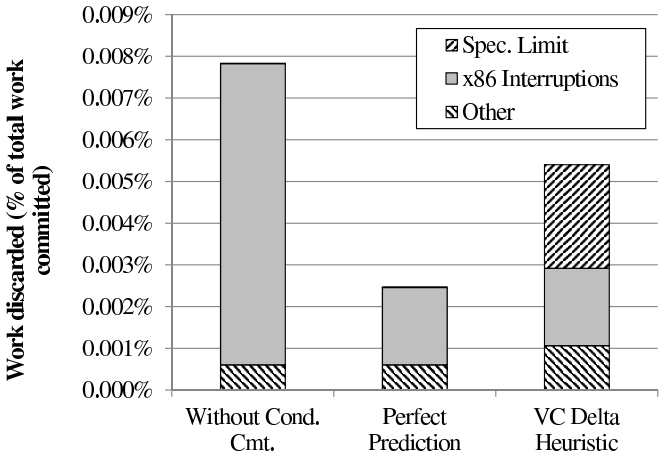

Fig. 10. Amount of work discarded per rollback cause.

The amount of work discarded due to the lack of speculative resources (Spec. Limit) when executing without the conditional commit is very small because the CMS translator usually produces small atomic regions. The perfect prediction approach also discards little work because it is always able to predict when the system will run out of resources and commit the execution of large regions. The VC Delta, on the contrary, discarded more speculative work because the heuristic was not able to predict that the system would run out of speculative resources on regions. The perfect prediction and the VC Delta heuristic approaches discarded less work on x86 interruptions due to the mechanism that delays x86 interruptions (the mechanism is not used on the experiments without the conditional commit).

Figure 10 also shows that there is a small increase in the amount of work discarded due to other causes (Other) when executing with the VC Delta heuristic. This happens because the atomic regions are larger, causing more work to be discarded on the rollbacks. The perfect prediction approach is able to predict all the rollbacks, including rollbacks not related to the lack of speculative resources, therefore, it does not increase the amount of work discarded due to the other causes.

V. Related Work

Many HW/SW co-designed systems have leveraged atomic execution to support speculative optimizations. PARROT [1], [19] and rePLay [18], [20] use atomic hardware support to execute optimized traces of code. However, these systems use internal pipeline buffers to hold the speculative data and the optimization scope is constrained to simple traces, without conditional if-then-else or loop constructs.

Transmeta Crusoe [11] employed a register checkpoint mechanism and a gated store buffer to store the speculative data. The gated store buffer is very limited in size and does not allow the execution of large atomic regions. Transmeta Efficeon [12], the successor of Crusoe, included a 64KB L1 speculative data cache and a 32 entry victim cache, capable of holding a larger amount of speculative data. Even with the larger cache, the translator conservatively inserted commit operations in the middle of the region (e.g., loop headers) to prevent the execution from running out of resources, reducing the size of the atomic regions.

In TAO [3], the authors proposed a technique that uses two-level atomicity support to enable optimizations on larger regions of code. The proposed technique is similar to LAR-CC in the sense that it allows the speculative optimization and execution of large atomic regions. However, instead of predicting if the system will run out of resources, the TAO system performs speculative checkpoints for each loop iteration and executes until it runs out of resources. Whenever the system runs out of resources, the execution performs a short rollback to the last speculative checkpoint (e.g., discarding the data produced by the last loop iteration) and commits the state. LAR-CC prevents the overhead caused by these short rollbacks by predicting if the system will run out of resources. Moreover, LAR-CC is simpler and does not require a Two-level atomicity hardware support.

Neelakantam *et al.* [16] explored the atomic execution mechanism to transform atomic regions by removing infrequent side-exits. The transformation does not increase the atomic region size, but it is able to unveil more optimization opportunities and it achieves an average of 3% performance improvement on a real Transmeta Efficeon [12] system. The transformation did not require any new hardware support, which enabled the authors to implement and measure the technique on a real system.

Atomic execution was also explored at high level languages to improve performance. Chen *et al.* [6] proposed atomicity support in the static compiler to optimize and parallelize C/C++ program aggressively. Neelakantam *et al.* also used hardware atomicity support to optimize Java programs in a JVM [17] and reported 10-15% average speedup.

User level dynamic binary translators and optimizers, such as IA32-EL [2], StarDBT [22], HDTrans [21], DynamoRIO [4], and Pin [14], optimize and execute user level

binaries from a source ISA on a target ISA. To the best of our knowledge these systems do not leverage atomic execution and cannot perform aggressive optimizations.

Thread-level speculations (TLS) relies on atomic execution for automatic code parallelization [5], [8], [13], [15]. TLS speculatively runs potentially conflicting regions of code in parallel. Whenever a dependency violation is detected at runtime, the conflicting threads are rolled back to the beginning of the region using the hardware atomicity support.

Transactional memories (TM) [9] also rely on atomicity hardware support for efficient execution of transactions. In order to use the atomicity hardware support, the amount of speculative data produced by the transaction must be small enough to fit into the speculative buffers. However, normal transactions cannot be committed in the middle in case the system is about to run out of speculative resources. Closed nested transaction may commit inner transactions, but the inner transaction boundary is specified by users, independent of hardware resources, and not managed by runtime. Because of atomicity feature, a transactional memory can also be leveraged to support atomic optimizations [17].

VI. Conclusions

Many HW/SW co-designed systems rely on atomicity support to speculatively execute aggressively optimized regions of code [1], [3], [11], [12], [18], [19], [20]. During the execution, the system stores the speculative data in speculative buffers. If the region produces too much speculative data and the system does not have enough speculative buffers to hold the data, the execution is rolled back, discarding the speculative data produced, and the system proceeds by executing a more conservative version of the code (e.g., by using interpretation).

In many cases it is hard to predict the amount of speculative resources needed by a region of code and, in order to ensure that regions will be able to execute to completion without running out of speculative resources, typical systems limit the size of the code inside the atomic regions. This practice effectively reduces the occurrence of rollbacks due to lack of speculative resources, however, the reduced code size also reduces the scope for optimizations. In this work, we proposed the Large Atomic Region with Conditional Commit, or LAR-CC, a novel technique that allows HW/SW co-designed systems to form and optimize large atomic regions by means of conditional commits that can be dynamically adjusted to fit into the available speculative hardware resources.

We implement LAR-CC inside a state-of-the-art HW/SW co-designed system and showed that it can increase the atomic region size significantly. For example, it enlarges the atomic loop size by $\tilde{3}$X over the original size, and effectively achieves atomic region sizes larger than 1000 instructions for loops that cover half of the loop execution.

This work can be expanded in many ways. We experimented with LAR-CC in a system that uses a speculative data cache and a victim cache to hold speculative data. In the future, we may try the LAR-CC technique with different cache configurations and with other mechanisms of speculation.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and discussions. We also appreciate the support provided by Ali-Reza Adl-Tabatabai at the Programming System Laboratory at Intel and by the Computer Systems Laboratory at the University of Campinas.

References

[1] Almog, Y., Rosner, R., Schwartz, N., and Schmorak, A. Specialized Dynamic Optimizations for High-Performance Energy-Efficient Microarchitecture. In Proceedings of the international symposium on code generation and optimization (CGO'04), Palo Alto, CA, 2004.

[2] Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skalesky, A., Wang, and Y., Zemach, Y. IA-32 Execution Layer: A Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems. In Proceedings of the 36th international symposium on microarchitecture (MICRO'03). San Diego, CA, 2003.

[3] Borin, E., Wu, Y., Wang, C. Liu, W., Breternitz Jr., M., Hu, S., Natanzon, E., Rotem, S., and Rosner, R. TAO: Two-level Atomicity for Dynamic Binary Optimizations. In Proceedings of the international symposium on code generation and optimization (CGO'10), Toronto, Canada, 2010.

[4] Bruening, D. L. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D thesis, Massachusetts Institute of Technology, 2004.

[5] Chen, M. K. and Olukotun, K. The Jrpm System for Dynamically Parallelizing Java Programs. In Proceedings of the 30th annual international symposium on computer architecture (ISCA'03). San Diego, CA, 2003.

[6] Chen, L-L. and Wu, Y. Aggressive Compiler Optimization and Parallelization with Thread-Level Speculation. In Proceedings of international conference on parallel processing (ICPP'03). Kaohsiung, Taiwan, 2003.

[7] Dehnert, J. C, Grant, B., Banning, J. P., Johnson, R., Kistler, T, Klaiber, A., and Mattson, J. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In Proceedings of the international symposium on code generation and optimization (CGO'03). San Francisco, CA, 2003.

[8] Du, Z.-H., Lim, C.-C., Li, X.-F., Yang, C., Zhao, Q., and Ngai, T.-F. A cost-driven compilation framework for speculative parallelization of sequential programs. In Proceedings of the ACM SIGPLAN 2004 conference on programming language design and implementation (PLDI'04). Washington, DC, 2004.

[9] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In Proceedings of the 20th annual international symposium on computer architecture (ISCA '93). New York, NY, 1993.

[10] Kim, H-S. and Smith, J. Hardware Support for Control Transfers in Code Caches. In proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03). Washington, DC, 2003.

[11] Klaiber, A. The Technology Behind Crusoe Processors. Transmeta white Paper, Jan. 2000.
http://www.charmed.com/PDF/CrusoeTechnologyWhitePaper_1-19-00.pdf

[12] Krewell, K. Transmeta Gets More Efficeon. Microprocessor report. v.17, October, 2003

[13] Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., and Torrellas, J. POSH: a TLS compiler that exploits program structure. In Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP'06). New York, NY, 2006.

[14] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney G., Wallace, S., Reddi, V., and Hazelwood K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation (PLDI'05). New York, NY, 2005.

[15] Luo, Y., Packirisamy, V., Hsu, W.-C., Zhai, A., Mungre, N., and Tarkas, A. Dynamic performance tuning for speculative threads. In Proceedings of the 36th annual international symposium on computer architecture (ISCA'09). Austin, TX, 2009.

[16] Neelakantam, N., Ditzel, D., and Zilles, C. A real system evaluation of hardware atomicity for software speculation. In Proceedings of the 15th international conference on architectural support for programming languages and operating systems (ASPLOS'10). Pittsburgh, PA, USA.

[17] Neelakantam, N., Rajwar, R., Srinivas, S., Srinivasan, U., and Zilles, C. B. Hardware atomicity for reliable software speculation. In Proceedings of the 34th annual international symposium on computer architecture (ISCA'07). San Diego, CA, 2007.

[18] Patel, S. J. and Lumetta, S. S. rePLay: A Hardware Framework for Dynamic Optimization. IEEE Transactions on Computers.50, 6 (Jun. 2001), 590-608.

[19] Rosner, R., Almog, Y., Moffie, M., Schwartz, N., and Mendelson, A. Power Awareness through Selective Dynamically Optimized Frames. In Proceedings of the 31st annual international symposium on computer architecture (ISCA'04). Mnchen, Germany, 2004.

[20] Slechta, B., Crowe, D., Fahs, B., Fertig, M., Muthler, G., Quek, J., Spadini, F., Patel, S. J., and Lumetta, S. S. Dynamic Optimization of Micro-Operations. In Proceedings of the 9th international symposium on high-performance computer Architecture (HPCA'03), Washington, DC, 2003.

[21] Sridhar, S., Shapiro, J. S., Northup, E., and Bungale, P. HDTrans: An Open Source, Low-Level Dynamic Instrumentation System. In Proceedings of the 2nd international conference on virtual execution environments (VEE'06), Ottawa, Canada, 2006.

[22] Wang, C., Hu, S., Kim, H-S., Nair, S. R., Breternitz Jr., M., Ying, Z., and Wu, Y. StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In Proceedings of Asia-pacific computer systems architecture conference, 2007.

[23] Wolfe, M. More iteration space tiling. In Proceedings of the Supercomputing 89, New York, NY, 1989.